Yesterday, My Program Worked. Today, It Does Not. Why?

Andreas Zeller

Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, D-94032 Passau, Germany
zeller@acm.org

Abstract. Imagine some program and a number of changes. If none of these changes is applied ("yesterday"), the program works. If all changes are applied ("today"), the program does not work. Which change is responsible for the failure? We present an efficient algorithm that determines the minimal set of failure-inducing changes. Our *delta debugging* prototype tracked down a single failure-inducing change from 178,000 changed GDB lines within a few hours.

1 A True Story

The GDB people have done it again. The new release 4.17 of the GNU debugger [6] brings several new features, languages, and platforms, but for some reason, it no longer integrates properly with my graphical front-end DDD [10]: the arguments specified within DDD are not passed to the debugged program. Something has changed within GDB such that it no longer works for me. Something? Between the 4.16 and 4.17 releases, no less than 178,000 lines have changed. How can I isolate the change that caused the failure and make GDB work again?

The GDB example is an instance of the "worked yesterday, not today" problem: after applying a set of changes, the program no longer works as it should. In finding the cause of the regression, the *differences* between the old and the new configuration (that is, the changes applied) can provide a good starting point. We call this technique *delta debugging*—determining the causes for program behavior by looking at the differences (the *deltas*).

Delta debugging works the better the *smaller* the differences are. Unfortunately, already one programmer can produce so many changes in a day such that the differences are too large for a human to trace—let alone differences between entire releases. In general, conventional debugging strategies lead to faster results.

However, delta debugging becomes an alternative when the differences can be *narrowed down automatically.* Ness and Ngo [5] present a method used at Cray research for compiler development. Their so-called *regression containment* is activated when the automated regression test fails. The method takes ordered changes from a configuration management archive and applies the changes, one after the other, to a configuration until its regression test fails. This narrows the search space from a set of changes to a single change, which can be isolated temporarily in order to continue development on a working configuration.

Regression containment is an effective delta debugging technique in some settings, including the one at Cray research. But there are several scenarios where linear search is not sufficient:

Interference. There may be not one single change responsible for a failure, but a *combination of several changes:* each individual change works fine on its own, but applying the entire set causes a failure. This frequently happens when merging the products of parallel development—and causes enormous debugging work.

Inconsistency. In parallel development, there may be *inconsistent configurations*—combinations of changes that do not result in a testable program. Such configurations must be identified and handled properly.

Granularity. A single logical change may affect several hundred or even thousand lines of code, but only a few lines may be responsible for the failure. Thus, one needs facilities to *break changes into smaller chunks*—a problem which becomes evident in the GDB example.

In this paper, we present automated delta debugging techniques that generalize regression containment such that interference, inconsistencies, and granularity problems are dealt with in an effective and practical manner. In particular, our dd^+ algorithm

- detects arbitrary interferences of changes in linear time
- detects individual failure-inducing changes in logarithmic time
- handles inconsistencies effectively to support fine-granular changes.

We begin with a few definitions required to present the basic *dd* algorithm. We show how its extension dd^+ handles inconsistencies from fine-granular changes. Two real-life case studies using our WYNOT prototype[1] highlight the practical issues; in particular, we reveal how the GDB failure was eventually resolved automatically. We close with discussions of future and related work, where we recommend delta debugging as standard operating procedure after any failing regression test.

2 Configurations, Tests, and Failures

We first discuss what we mean by configurations, tests, and failures. Our view of a *configuration* is the broadest possible:

Definition 1 (Configuration). *Let* $C = \{\Delta_1, \Delta_2, \ldots, \Delta_n\}$ *be the set of all possible changes* Δ_i. *A change set* $c \subseteq C$ *is called a* configuration.

A configuration is constructed by applying changes to a *baseline.*

Definition 2 (Baseline). *An empty configuration* $c = \emptyset$ *is called a* baseline.

Note that we do not impose any constraints on how changes may be combined; in particular, we do not assume that changes are ordered. Thus, in the worst case, there are 2^n possible configurations for n changes.

To determine whether a failure occurs in a configuration, we assume a *testing function.* According to the POSIX 1003.3 standard for testing frameworks [3], we distinguish three outcomes:

[1] WYNOT = "Worked Yesterday, NOt Today"

- The test succeeds (PASS, written here as ✔)
- The test has produced the failure it was indented to capture (FAIL, ✘)
- The test produced indeterminate results (UNRESOLVED, ?).[2]

Definition 3 (Test). *The function test* : $2^{\mathcal{C}} \to \{✘, ✔, ?\}$ *determines for a configuration* $c \in \mathcal{C}$ *whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).*

In practice, *test* would construct the configuration from the given changes, run a regression test on it and return the test outcome.[3]

Let us now model our initial scenario. We have some configuration "yesterday" that works fine and some configuration "today" that fails. For simplicity, we only consider the changes present "today", but not "yesterday". Thus, we model the "yesterday" configuration as baseline and the "today" configuration as set of all possible changes.

Axiom 1 (Worked yesterday, not today). *test*$(\emptyset)$ = ✔ *("yesterday") and test*$(\mathcal{C})$ = ✘ *("today") hold.*

What do we mean by changes that cause a failure? We are looking for a specific change set—those changes that make the program fail by including them in a configuration. We call such changes *failure-inducing.*

Definition 4 (Failure-inducing change set). *A change set* $c \subseteq \mathcal{C}$ *is* failure-inducing *if*

$$\forall c' \left(c \subseteq c' \subseteq \mathcal{C} \to test(c') \neq ✔\right)$$

holds.

The set of all changes $\mathcal{C}$ is failure-inducing by definition. However, we are more interested in finding the *minimal* failure-inducing subset of $\mathcal{C}$, such that removing any of the changes will make the program work again:

Definition 5 (Minimal failure-inducing set). *A failure-inducing change set* $B \subseteq \mathcal{C}$ *is* minimal *if*

$$\forall c \subset B \left(test(c) \neq ✘\right)$$

holds.

And exactly *this* is our goal: *For a configuration* $\mathcal{C}$*, to find a minimal failure-inducing change set.*

3 Configuration Properties

If every change combination produced arbitrary test results, we would have no choice but to test all 2^n configurations. In practice, this is almost never the case. Instead, configurations fulfill one or more specific *properties* that allow us to devise much more efficient search algorithms.

[2] POSIX 1003.3 also lists UNTESTED and UNSUPPORTED outcomes, which are of no relevance here.

[3] A single test case may take time. Recompilation and re-execution of a program may be a matter of several minutes, if not hours. This time can be considerably reduced by smart recompilation techniques [7] or caching derived objects [4].

The first useful property is *monotony*: once a change causes a failure, any configuration that includes this change fails as well.

Definition 6 (Monotony). *A configuration $\mathcal{C}$ is* monotone *if*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) = ✘ \rightarrow \forall c' \supseteq c\ (test(c') \neq ✔)\big) \tag{1}$$

holds.

Why is monotony so useful? Because once we know a change set does *not* cause a failure, so do all subsets:

Corollary 1. *Let $\mathcal{C}$ be a monotone configuration. Then,*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) = ✔ \rightarrow \forall\, c' \subseteq c(test(c') \neq ✘)\big) \tag{2}$$

holds.

Proof. By contradiction. For all configurations $c \subseteq \mathcal{C}$ with $test(c) = ✔$, assume that $\exists c' \subseteq c\big(test(c') = ✘\big)$ holds. Then, definition 6 implies $test(c) \neq ✔$, which is not the case.

Another useful property is *unambiguity*: a failure is caused by only one change set (and not independently by two disjoint ones). This is mostly a matter of economy: once we have detected a failure-inducing change set, we do not want to search the complement for more failure-inducing change sets.

Definition 7 (Unambiguity). *A configuration $\mathcal{C}$ is* unambiguous *if*

$$\forall c_1, c_2 \subseteq \mathcal{C}\ \big(test(c_1) = ✘ \wedge test(c_2) = ✘ \rightarrow test(c_1 \cap c_2) \neq ✔\big) \tag{3}$$

holds.

The third useful property is *consistency*: every subset of a configuration returns an determinate test result. This means that applying any combination of changes results in a testable configuration.

Definition 8 (Consistency). *A configuration $\mathcal{C}$ is* consistent *if*

$$\forall c \subseteq \mathcal{C}\ \big(test(c) \neq\ ?\big)$$

holds.

If a configuration does not fulfill a specific property, there are chances that one of its *subsets* fulfills them. This is the basic idea of the *divide-and-conquer* algorithms presented below.

4 Finding Failure-Inducing Changes

For presentation purposes, we begin with the simplest case: a configuration c that is monotone, unambiguous, and consistent. (These constraints will be relaxed bit by bit in the following sections.) For such a configuration, we can design an efficient algorithm based on *binary search* to find a minimal set of failure-inducing changes. If c contains only one change, this change is failure-inducing by definition. Otherwise, we *partition* c into two subsets c_1 and c_2 and test each of them. This gives us three possible outcomes:

Found in c_1. The test of c_1 fails—c_1 contains a failure-inducing change.
Found in c_2. The test of c_2 fails—c_2 contains a failure-inducing change.
Interference. Both tests pass. Since we know that testing $c = c_1 \cup c_2$ fails, the failure must be induced by the combination of some change set in c_1 and some change set in c_2.

In the first two cases, we can simply continue the search in the failing subset, as illustrated in Table 1. Each line of the diagram shows a configuration. A number i stands for an included change Δ_i; a dot stands for an excluded change. Change 7 is the one that causes the failure—and it is found in just a few steps.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✘	
3	c_1	.	.	.	.	5	6	.	.	✔	
4	c_2	.	.	.	.	.	.	7	8	✘	
5	c_1	.	.	.	.	.	.	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

Table 1. Searching a single failure-inducing change

But what happens in case of interference? In this case, we must search in *both halves*—with all changes in the other half remaining applied, respectively. This variant is illustrated in Table 2. The failure occurs only if the two changes 3 and 6 are applied together. Step 3 illustrates how changes 5–7 remain applied while searching through 1–4; in step 6, changes 1–4 remain applied while searching in 5–7.[4]

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✔	
3	c_1	1	2	.	.	5	6	7	8	✔	
4	c_2	.	.	3	4	5	6	7	8	✘	
5	c_1	.	.	3	.	5	6	7	8	✘	3 is found
6	c_1	1	2	3	4	5	6	.	.	✘	
7	c_1	1	2	3	4	5	.	.	.	✔	6 is found
Result		.	.	3	.	.	6	.	.		

Table 2. Searching two failure-inducing changes

We can now formalize the search algorithm. The function *dd*(c) returns all failure-inducing changes in c; we use a set r to denote the changes that remain applied.

[4] Delta debugging is not restricted to programs alone. On this LaTeX document, 14 iterations of manual delta debugging had to be applied until Table 2 eventually re-appeared on the same page as its reference.

Algorithm 1 (Automated delta debugging). The *automated delta debugging algorithm $dd(c)$* is

$$
\begin{aligned}
dd(c) &= dd_2(c, \emptyset) \quad \text{where} \\
dd_2(c, r) &= \text{let } c_1, c_2 \subseteq c \text{ with } c_1 \cup c_2 = c, c_1 \cap c_2 = \emptyset, |c_1| \approx |c_2| \approx |c|/2 \\
&\text{in } \begin{cases}
c & \text{if } |c| = 1 \text{ (“found”)} \\
dd_2(c_1, r) & \text{else if } test(c_1 \cup r) = \text{✘} \text{ (“in } c_1\text{”)} \\
dd_2(c_2, r) & \text{else if } test(c_2 \cup r) = \text{✘} \text{ (“in } c_2\text{”)} \\
dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise (“interference”)}
\end{cases}
\end{aligned}
$$

The recursion invariant (and thus precondition) for dd_2 is $test(r) = \text{✔} \wedge test(c \cup r) = \text{✘}$.

The basic properties of *dd* are discussed and proven in [9]. In particular, we show that $dd(c)$ returns a minimal set of failure-inducing changes in c if c is monotone, unambiguous, and consistent.

Since *dd* is a divide-and-conquer algorithm with constant time requirement at each invocation, *dd*'s time complexity is at worst linear. This is illustrated in Table 3, where only the combination of *all* changes is failure-inducing, and where *dd* requires less than two tests per change to find them. If there is only one failure-inducing change to be found, *dd* even has logarithmic complexity, as illustrated in Table 1.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	✔	
3	c_1	1	2	.	.	5	6	7	8	✔	
4	c_2	.	.	3	4	5	6	7	8	✔	
5	c_1	1	.	3	4	5	6	7	8	✔	2 is found
6	c_2	.	2	3	4	5	6	7	8	✔	1 is found
7	c_1	1	2	3	.	5	6	7	8	✔	4 is found
8	c_2	1	2	.	4	5	6	7	8	✔	3 is found
9	c_1	1	2	3	4	5	6	.	.	✔	
10	c_2	1	2	3	4	.	.	7	8	✔	
11	c_1	1	2	3	4	5	.	7	8	✔	6 is found
12	c_2	1	2	3	4	.	6	7	8	✔	5 is found
13	c_1	1	2	3	4	5	6	7	.	✔	8 is found
14	c_2	1	2	3	4	5	6	.	8	✔	7 is found
Result		1	2	3	4	5	6	7	8		

Table 3. Searching eight failure-inducing changes

Let us now recall the properties *dd* requires from configurations: monotony, unambiguity, and consistency. How does *dd* behave when c is *not* monotone or when it is ambiguous? In case of interference, *dd* still returns a failure-inducing change set, although it may not be minimal. But maybe surprisingly, a single failure-inducing change (and hence a minimal set) is found even for non-monotone or ambiguous configurations:

- If a configuration is ambiguous, multiple failure-inducing changes may occur; *dd* returns one of them. (After undoing this change set, re-run *dd* to find the next one.)

- If a configuration is not monotone, then we can devise "undoing" changes that, when applied to a previously failing configuration c, cause c to pass the test again. But still, today's configuration is failing; hence, there must be *another* failure-inducing change that is not undone and that can be found by *dd*.

5 Handling Inconsistency

The most important practical problem in delta debugging is *inconsistent configurations*. When combining changes in an arbitrary way, such as done by *dd*, it is likely that several resulting configurations are inconsistent—the outcome of the test cannot be determined. Here are some of the reasons why this may happen:

Integration failure. A change cannot be applied. It may require earlier changes that are not included in the configuration. It may also be in conflict with another change and a third conflict-resolving change is missing.

Construction failure. Although all changes can be applied, the resulting program has syntactical or semantical errors, such that construction fails.

Execution failure. The program does not execute correctly; the test outcome is unresolved.

Since it is improbable that all configurations tested by *dd* have been checked for inconsistencies beforehand, tests may well outcome unresolved during a *dd* run. Thus, *dd* must be extended to deal with inconsistent configurations.

Let us begin with the worst case: after splitting up c into subsets, all tests are unresolved—ignorance is complete. How we increase our chances to get a resolved test? We know two configurations that are consistent: $\emptyset$ ("yesterday") and C ("today"). By applying *less* changes to "yesterday's" configuration, we increase the chances that the resulting configuration is consistent—the difference to "yesterday" is smaller. Likewise, we can remove less changes from "today's" configuration and decrease the difference to "today".

In order to apply less changes, we can partition c into a *larger number of subsets*. The more subsets we have, the smaller they are, and the bigger are our chances to get a consistent configuration—until each subset contains only one change, which gives us the best chance to get a consistent configuration. The disadvantage, of course, is that more subsets means more testing.

To extend the basic *dd* algorithm to work on an arbitrary number n of subsets $c_1, \ldots, c_n$, we must distinguish the following cases:

Found. If testing any c_i fails, then c_i contains a failure-inducing subset. This is just as in *dd*.

Interference. If testing any c_i passes and its *complement* $\bar{c}_i$ passes as well, then the change sets c_i and $\bar{c}_i$ form an interference, just as in *dd*.

Preference. If testing any c_i is unresolved, and testing $\bar{c}_i$ passes, then c_i contains a failure-inducing subset and is *preferred*. In the following test cases, $\bar{c}_i$ must remain applied to promote consistency.

As a preference example, consider Table 4. In Step 1, testing c_1 turns out unresolved, but its complement $\bar{c}_1 = c_2$ passes the test in Step 2. Consequently, c_2 cannot contain a bug-inducing change set, but c_1 can—possibly in interference with c_2, which is why c_2 remains applied in the following test cases.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	?	Testing c_1, c_2
2	c_2	.	.	.	.	5	6	7	8	✔	$\Rightarrow$ Prefer c_1
3	c_1	1	2	.	.	5	6	7	8	...	

Table 4. Preference

Try again. In all other cases, we repeat the process with $2n$ subsets—resulting with twice as many tests, but increased chances for consistency.

As a "try again" example, consider Table 5. Change 8 is failure-inducing, and changes 2, 3 and 7 imply each other—that is, they only can be applied as a whole. Note how the test is repeated first with $n = 2$, then with $n = 4$ subsets.

Step	c_i	Configuration								*test*	
1	$c_1 = \bar{c}_2$	1	2	3	4	.	.	.	.	?	Testing c_1, c_2
2	$c_2 = \bar{c}_1$	.	.	.	.	5	6	7	8	?	$\Rightarrow$ Try again
3	c_1	1	2	.	.	.	.	.	.	?	Testing $c_1, \ldots, c_4$
4	c_2	.	.	3	4	.	.	.	.	?	
5	c_3	.	.	.	.	5	6	.	.	✔	
6	c_4	.	.	.	.	.	.	7	8	?	
7	$\bar{c}_1$	.	.	3	4	5	6	7	8	?	Testing complements
8	$\bar{c}_2$	1	2	.	.	5	6	7	8	?	
9	$\bar{c}_3$	1	2	3	4	.	.	7	8	✘	
10	$\bar{c}_4$	1	2	3	4	5	6	.	.	?	$\Rightarrow$ Try again

Table 5. Searching failure-inducing changes with inconsistencies

In each new run, we can do a little *optimizing*: all c_i that passed the test can be excluded from c, since they cannot be failure-inducing. Likewise, all c_i whose complements $\bar{c}_i$ failed the test can remain applied in following tests. In our example, this applies to changes 5 and 6, such that we can continue with $n = 6$ subsets. After testing each change individually, we finally find the failure-inducing change, as shown in Table 6.

Step	c_i	Configuration								*test*	
11	c_1	1	.	.	.	5	6	.	.	✔	Testing $c_1, \ldots, c_6$
12	c_2	.	2	.	.	5	6	.	.	?	
13	c_3	.	.	3	.	5	6	.	.	?	
14	c_4	.	.	.	4	5	6	.	.	✔	
15	c_5	.	.	.	.	5	6	7	.	?	
16	c_6	.	.	.	.	5	6	.	8	✘	8 is found
Result		.	.	.	.	.	.	.	8		

Table 6. Searching failure-inducing changes with inconsistencies (continued)

Note that at this stage, changes 1, 4, 5 and 6 have already been identified as *not* failure-inducing, since their respective tests passed. If the failure had not been induced by change 8, but by 2, 3, or 7, we would have found it simply by excluding all other changes.

To summarize, here is the formal definition of the extended dd^+ algorithm:

Algorithm 2 (Delta debugging with unresolved test cases).
The *extended delta debugging algorithm* $dd^+(c)$ is

$$dd^+(c) = dd_3(c, \emptyset, 2) \quad \text{where}$$

$$dd_3(c, r, n) =$$

let $c_1, \ldots, c_n \subseteq c$ such that $\bigcup c_i = c$, all c_i are pairwise disjoint, and $\forall c_i \; (|c_i| \approx |c|/n)$;

let $\bar{c}_i = c - (c_i \cup r)$, $t_i = test(c_i \cup r)$, $\bar{t}_i = test(\bar{c}_i \cup r)$,

$c' = c \cap \bigcap\{\bar{c}_i \mid \bar{t}_i = \text{✘}\}$, $r' = r \cup \bigcup\{c_i \mid t_i = \text{✔}\}$, $n' = \min(|c'|, 2n)$,

$d_i = dd_3(c_i, \bar{c}_i \cup r, 2)$, and $\bar{d}_i = dd_3(\bar{c}_i, c_i \cup r, 2)$

$$\text{in} \begin{cases} c & \text{if } |c| = 1 \text{ ("found")} \\ dd_3(c_i, r, 2) & \text{else if } t_i = \text{✘} \text{ for some } i \text{ ("found in } c_i\text{")} \\ d_i \cup \bar{d}_i & \text{else if } t_i = \text{✔} \wedge \bar{t}_i = \text{✔} \text{ for some } i \text{ ("interference")} \\ d_i & \text{else if } t_i = \text{?} \wedge \bar{t}_i = \text{✔} \text{ for some } i \text{ ("preference")} \\ dd_3(c', r', n') & \text{else if } n < |c| \text{ ("try again")} \\ c' & \text{otherwise ("nothing left")} \end{cases}$$

The recursion invariant for dd_3 is $test(r) \neq \text{✘} \wedge test(c \cup r) \neq \text{✔} \wedge n \leq |c|$.

Apart its extensions for unresolved test cases, the dd_3 function is identical to dd_2 with an initial value of $n = 2$. Like *dd*, dd^+ has linear time complexity (but requires twice as many tests).

Eventually, dd^+ finds a minimal set of failure-inducing changes, provided that they are *safe*—that is, they can either be applied to the baseline or removed from today's configuration without causing an inconsistency. If this condition is not met, the set returned by dd^+ may not be minimal, depending on the nature of inconsistencies encountered. But at least, all changes that are safe and not failure-inducing are guaranteed to be excluded.[5]

6 Avoiding Inconsistency

In practice, we can significantly reduce the risk of inconsistencies by relying on specific *knowledge* about the nature of the changes. There are two ways to influence the dd^+ algorithm:

[5] True minimality can only be achieved by testing all 2^n configurations. Consider a hypothetic set of changes where only three configurations are consistent: yesterday's, today's, and one arbitrary configuration. Only by trying all combinations can we find this third configuration; inconsistency has no specific properties like monotony that allow for more effective methods.

Grouping Related Changes. Reconsider the changes 2, 3, and 7 of Table 5. If we had some indication that the changes imply each other, we could keep them in a common subset as long as possible, thereby reducing the number of unresolved test cases. To determine whether changes are related, one can use

- *process criteria,* such as common change dates or sources,
- *location criteria,* such as the affected file or directory,
- *lexical criteria,* such as common referencing of identifiers,
- *syntactic criteria,* such as common syntactic entities (functions, modules) affected by the change,
- *semantic criteria,* such as common program statements affected by the changed control flow or changed data flow.

For instance, it may prove useful to group changes together that all affect a specific function (*syntactic criteria*) or that occurred at a common date (*process criteria*).

Predicting Test Outcomes. If we have *evidence* that specific configurations will be inconsistent, we can *predict* their test outcomes as unresolved instead of carrying out the test. In Table 5, if we knew about the implications, then only 5 out of 16 tests would actually be carried out.

Predicting test outcomes is especially useful if we can impose an *ordering* on the changes. Consider Table 7, where each change Δ_i implies all "earlier" changes $\Delta_1, \ldots, \Delta_{i-1}$. Given this knowledge, we can predict the test outcomes of steps 2 and 4; only three tests would actually carried out to find the failure-inducing change.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✔	
2	c_2	.	.	.	.	5	6	7	8	(?)	predicted outcome
3	c_1	1	2	3	4	5	6	.	.	✔	
4	c_2	1	2	3	4	.	.	7	8	(?)	predicted outcome
5	c_1	1	2	3	4	5	6	7	.	✘	7 is found
Result		.	.	.	.	.	.	7	.		

Table 7. Searching failure-inducing changes in a total order

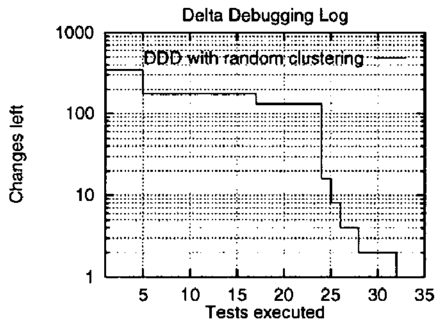
We see that when changes can be ordered, predicting test outcomes makes dd^+ act like a binary search algorithm.

Both grouping and predicting will be used in two case studies, presented below.

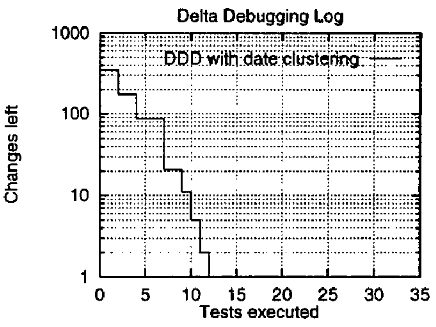
7 First Case Study: DDD 3.1.2 Dumps Core

DDD 3.1.2, released in December, 1998, exhibited a nasty behavioral change: When invoked with a the name of a non-existing file, DDD 3.1.2 dumped core, while its predecessor DDD 3.1.1 simply gave an error message. We wanted to find the cause of this failure by using WYNOT.

The DDD configuration management archive lists 116 logical changes between the 3.1.1 and 3.1.2 releases. These changes were split into 344 textual changes to the DDD source.



(a) with random clustering



(b) with date clustering

Table 8. Searching a failure-inducing change in DDD

In a first attempt, we ignored any knowledge about the nature or ordering of the changes; changes were ordered and partitioned at random. Table 8(a) shows the result of the resulting WYNOT run. After test #4, WYNOT has reduced the number of remaining changes to 172. The next tests turn out unresolved, so WYNOT gradually increases the number of subsets; at test #16, WYNOT starts using 8 subsets, each containing 16 changes. At test #23, the 7th subset fails, and only its 16 changes remain. Eventually, test #31 determines the failure-inducing change.

We then wanted to know whether knowledge from the configuration management archive would improve performance. We used the following *process criteria:*

1. Changes were grouped according to the date they were applied.
2. Each change implied all earlier changes. If a configuration would not satisfy this requirement, its test outcome would be predicted as unresolved.

As shown in Table 8(b), this resulted in a binary search with very few inconsistencies. After only 12 test runs and 58 minutes[6], the failure-inducing change was found:

```
diff -r1.30 -r1.30.4.1 ddd/gdbinit.C
295,296c296
<   string classpath =
<       getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : ".";
---
>   string classpath = source_view->class_path();
```

When called with an argument that is not a file name, DDD 3.1.1 checks whether it is a Java class; so DDD consults its environment for the class lookup path. As an "improvement", DDD 3.1.2 uses a dedicated method for this purpose. Unfortunately, the source_view pointer used is initialized only later, resulting in a core dump. This problem has been fixed in the current DDD release.

8 Second Case Study: GDB 4.17 Does Not Integrate

Let us now face greater challenges. As motivated in Section 1, we wanted to track down a failure in 178,000 changed GDB lines. In contrast to the DDD setting from

[6] All times were measured on a Linux PC with a 200 MHz AMD K6 processor.

Section 7, we had no configuration management archive from which to take ordered logical changes.

The 178,000 lines were automatically grouped into 8721 textual changes in the GDB source, with any two textual changes separated by at least two unchanged lines ("context"). The average reconstruction time after applying a change turned out to be 370 seconds. This means that we could run 233 tests in 24 hours or 8721 changes individually in 37 days.

Again, we first ignored any knowledge about the nature of the changes. The result of this WYNOT run is shown in Table 9(a). Most of the first 457 tests turn out unresolved, so WYNOT gradually increases the number of subsets, reducing the number of remaining changes. At test #458, each subset contains only 36 changes, and it is one of these subsets that turns out to be failure-inducing. After this breakthrough, the remaining 12 tests determine a single failure-inducing change.

Running the 470 tests still took 48 hours. Once more, we decided to improve performance. Since process criteria were not available, we used *location criteria* and *lexical criteria* to group changes:

1. At top-level, changes were grouped according to directories. This was motivated by the observation that several GDB directories contain a separate library whose interface remains more or less consistent across changes.
2. Within one directory, changes were grouped according to common files. The idea was to identify compilation units whose interface was consistent with both "yesterday's" and "today's" version.
3. Within a file, changes were grouped according to common usage of identifiers. This way, we could keep changes together that operated on common variables or functions.

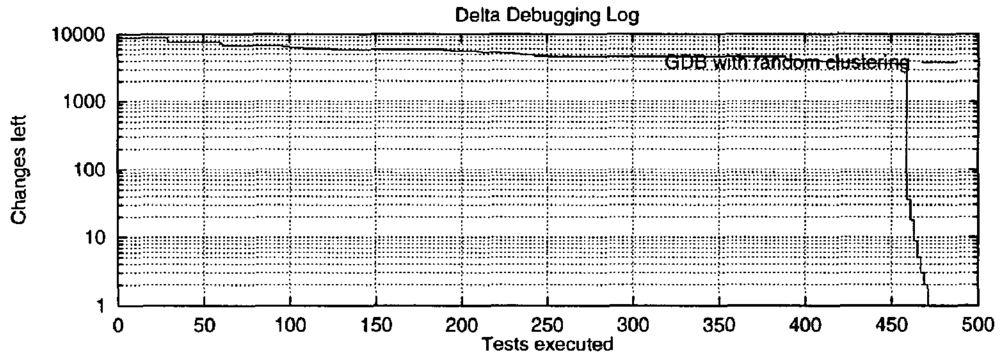
Finally, we added a *failure resolution loop:* After a failing construction, WYNOT scans the error messages for identifiers, adds all changes that reference these identifiers and tries again. This is repeated until construction is possible, or until there are no more changes to add.

The result of this WYNOT run is shown in Table 9(b). At first, WYNOT split the changes according to their directories. After 9 tests with various directory combinations, WYNOT has a breakthrough: the failure-inducing change is to be found in one specific directory. Only 2547 changes are left.

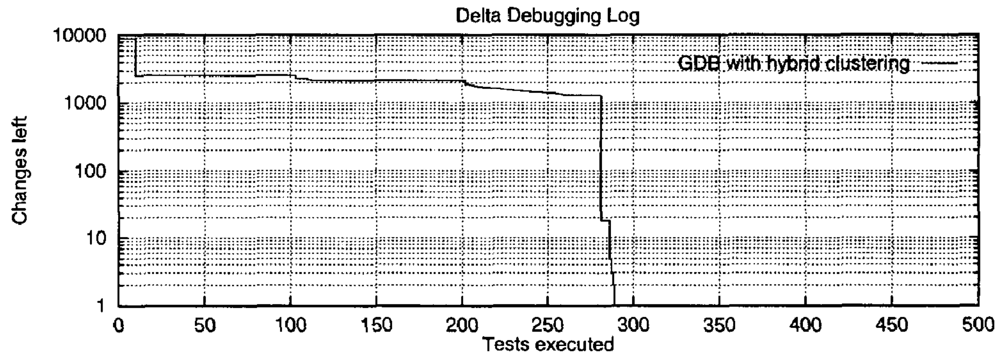
A long period without significant success follows; WYNOT partitions changes into an increasing number of subsets. The second breakthrough occurs at test #280, where each subset contains only 18 changes and where WYNOT narrows down the number of changes to a subset of two files only. The end comes at test #289, after a total of 20 hours. We see that the lexical criteria reduced the number of tests by 38% and the total running time by more than 50%.

In both cases, WYNOT broke down the 178,000 lines down to the same one-line change line that, being applied, causes DDD to malfunction:

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```



(a) with random clustering



(b) with hybrid clustering

Table 9. Searching a failure-inducing change in GDB

This change in a string constant from `arguments` to `argument list` was responsible for GDB 4.17 not interoperating with DDD. Given the command `show args`, GDB 4.16 replies

```
Arguments to give program being debugged when it is started is "a b c"
```

but GDB 4.17 issues a slightly different (and grammatically correct) text:

```
Argument list to give program being debugged when it is started is "a b c"
```

which could not be parsed by DDD! To solve the problem here and now, we simply reversed the GDB change; eventually, DDD was upgraded to make it work with the new GDB version, too.

9 Related Work

There is only one other work on automated delta debugging we have found: the paper on *regression containment* by Ness and Ngo [5], presented in Section 1.[7] Ness and Ngo use simple linear and binary search to identify a single failure-inducing change. Their goal, however, lies not in debugging, but in *isolating* (i.e. removing) the failure-inducing

[7] Ness and Ngo cite no related work, so we assume they found none either.

change such that development of the product is not delayed by resolving the failure. The existence of a configuration management archive with totally ordered changes is assumed; issues like interference, inconsistencies, granularity, or non-monotony are neither handled nor discussed.

Consequently, the failure-inducing change in GDB from Section 8 would not be found at all since there is no configuration management archive from which to take logical changes; in the DDD setting from Section 7, the logical change would be found, but could not have been broken down into this small chunk.

10 Conclusions and Future Work

Delta debugging resolves regression causes automatically and effectively. If configuration information is available, delta debugging is easy; otherwise, there are effective techniques that indicate change dependencies. Although resource-intensive, delta debugging requires no manual intervention and thus saves valuable developer time.

We recommend that delta debugging be an integrated part of regression testing; each time a regression test fails, a delta debugging program should be started to resolve the regression cause. The algorithms presented in this paper provide successful delta debugging solutions that handle difficult details such as interferences, inconsistencies, and granularity.

Our future work will concentrate on avoiding inconsistencies by exploiting domain knowledge. Most simple configuration management archives enforce that each change implies all earlier changes; we want to use full-fledged constraint systems instead [11]. Another issue is to use *syntactic criteria* in order to group changes by affected functions and modules. The most complicated, but most promising approach are *semantic criteria:* Given a change and a program, we can determine a *slice* of the program where program execution may be altered by applying the change. Such slices have been successfully used for semantics-preserving program integration [2] as well as for determining whether a regression test is required after applying a specific change [1]. The basic idea is to determine two *program dependency graphs* (PDGs)—one for "yesterday's" and one for "today's" configuration. Then, for each change c and each PDG, we determine the forward slice from the nodes affected by c. We can then group changes by the *common nodes* contained in their respective slices; two changes with disjoint slices end up in different partitions.

Besides consistency issues, we want to use *code coverage* tools in order to exclude changes to code that is never executed. The intertwining of changes to construction commands, system models, and actual source code must be handled, possibly by multiversion system models [8]. Further case studies will validate the effectiveness of all these measures, as of delta debugging in general.

Acknowledgments. Carsten Schulz contributed significantly to the current WYNOT implementation. The first delta debugging prototype was implemented by Ulrike Heuer. Jens Krinke, Christian Lindig, Kerstin Reese, Torsten Robschink, Gregor Snelting, and Paul Strooper provided valuable comments on earlier revisions of this paper.

Further information on delta debugging, including the full WYNOT implementation, is available at `http://www.fmi.uni-passau.de/st/wynot/`.

References

1. BINKLEY, D. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering 23*, 8 (Aug. 1997), 498–516.
2. BINKLEY, D., HORWITZ, S., AND REPS, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4*, 1 (Jan. 1995), 3–35.
3. IEEE. *Test Methods for Measuring Conformance to POSIX.* New York, 1991. ANSI/IEEE Standard 1003.3-1991. ISO/IEC Standard 13210-1994.
4. LEBLANG, D. B. The CM challenge: Configuration management that works. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 1, pp. 1–37.
5. NESS, B., AND NGO, V. Regression containment through source code isolation. In *Proceedings of the 21st Annual International Computer Software & Applications Conference (COMPSAC '97)* (Washington, D.C., Aug. 1997), IEEE Computer Society Press, pp. 616–621.
6. STALLMAN, R. M., AND PESCH, R. H. *Debugging with GDB*, 5th ed. Free Software Foundation, Inc., Apr. 1998. Distributed with GDB 4.17.
7. TICHY, W. F. Smart recompilation. *ACM Transactions on Software Engineering and Methodology 8*, 3 (July 1986), 273–291.
8. ZELLER, A. Versioning system models through description logic. In *Proc. 8th Symposium on System Configuration Management* (Brussels, Belgium, July 1998), B. Magnusson, Ed., vol. 1349 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 127–132.
9. ZELLER, A. Yesterday, my program worked. Today, it does not. Why? Computer Science Report 99-01, Technical University of Braunschweig, Germany, Feb. 1999.
10. ZELLER, A., AND LÜTKEHAUS, D. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices 31*, 1 (Jan. 1996), 22–27.
11. ZELLER, A., AND SNELTING, G. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology 6*, 4 (Oct. 1997), 398–441.