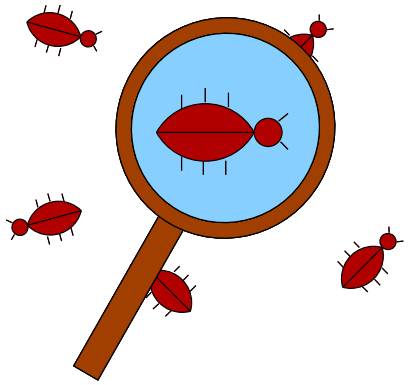
Iterative Delta Debugging (IDD)

Cyrille Artho
Research Center for Information Security (RCIS),
Nat. Inst. of Advanced Industrial Science and Technology (AIST),
Tokyo, Japan
c.artho@aist.go.jp

10/28/2008

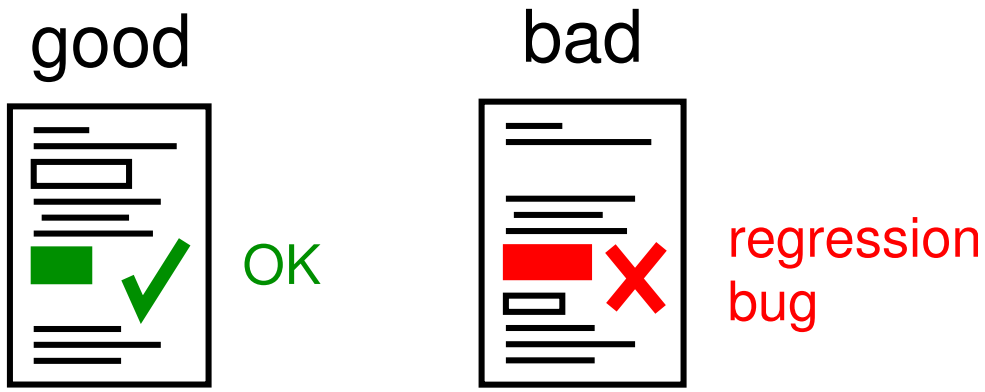
Debugging



- Effort to isolate *fault* in a system.
- Why does the test fail?
  - What part of program contains fault?
  - What part of input provokes failure?
- Manual debugging: inspect test run.
  - Goal: small test case to study.

Delta Debugging [Zeller02]

- Isolate difference between inputs that causes failure.

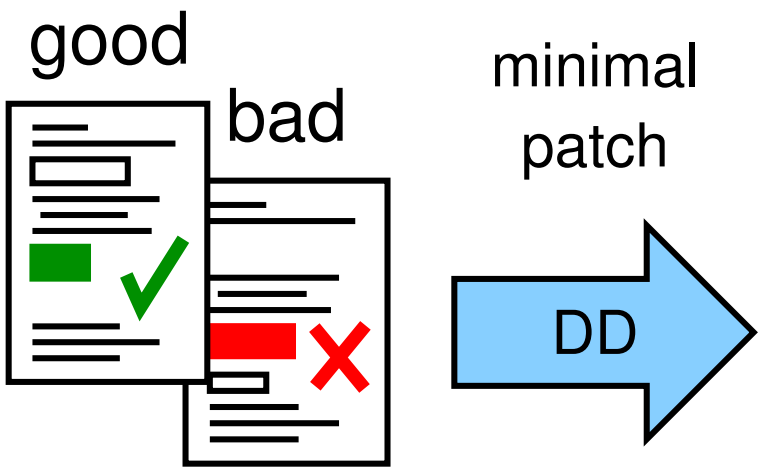


- Idea: (Minimal) difference between inputs = reason of failure.
- Can also be applied to program source code!

Outline

1. Delta Debugging (DD).
2. When and how to iterate DD.
3. Problems with DD on programs.
4. Experiments.
5. Conclusion.

DD on program code



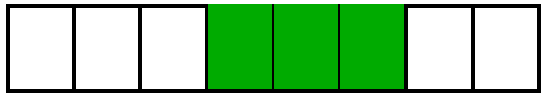
minimal patch

DD

```
234,240d234
< // this code was removed
< // (not relevant for bug)
345,370c365
< // old, correct code
< ...
> // regression bug (new)
> ...
400a398,405
> // new feature (no bug)
```

- Use patch between good and bad version.
- Try to generate a version that is as close to „bad” as possible. . . while still passing the test.

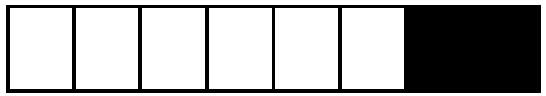
How DD's state space bisection works



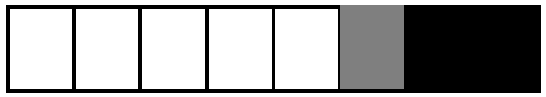
"good" part of the change — green

entire change set disabled — fail

second half of change disabled — fail



last two bits disabled — pass

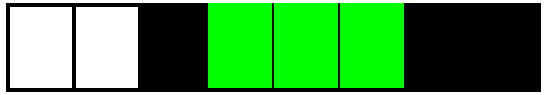


bit 5 disabled — fail

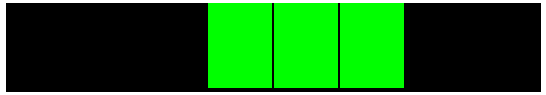
bit 4 disabled — fail

first half of change set disabled — fail

. . .

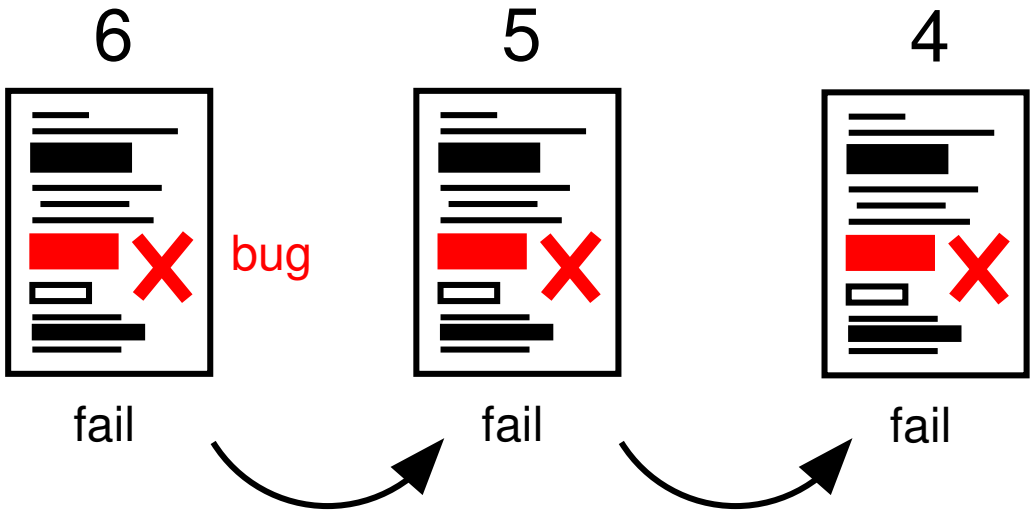


after two more fails — pass



final iteration — pass

What if no „good” version is known?

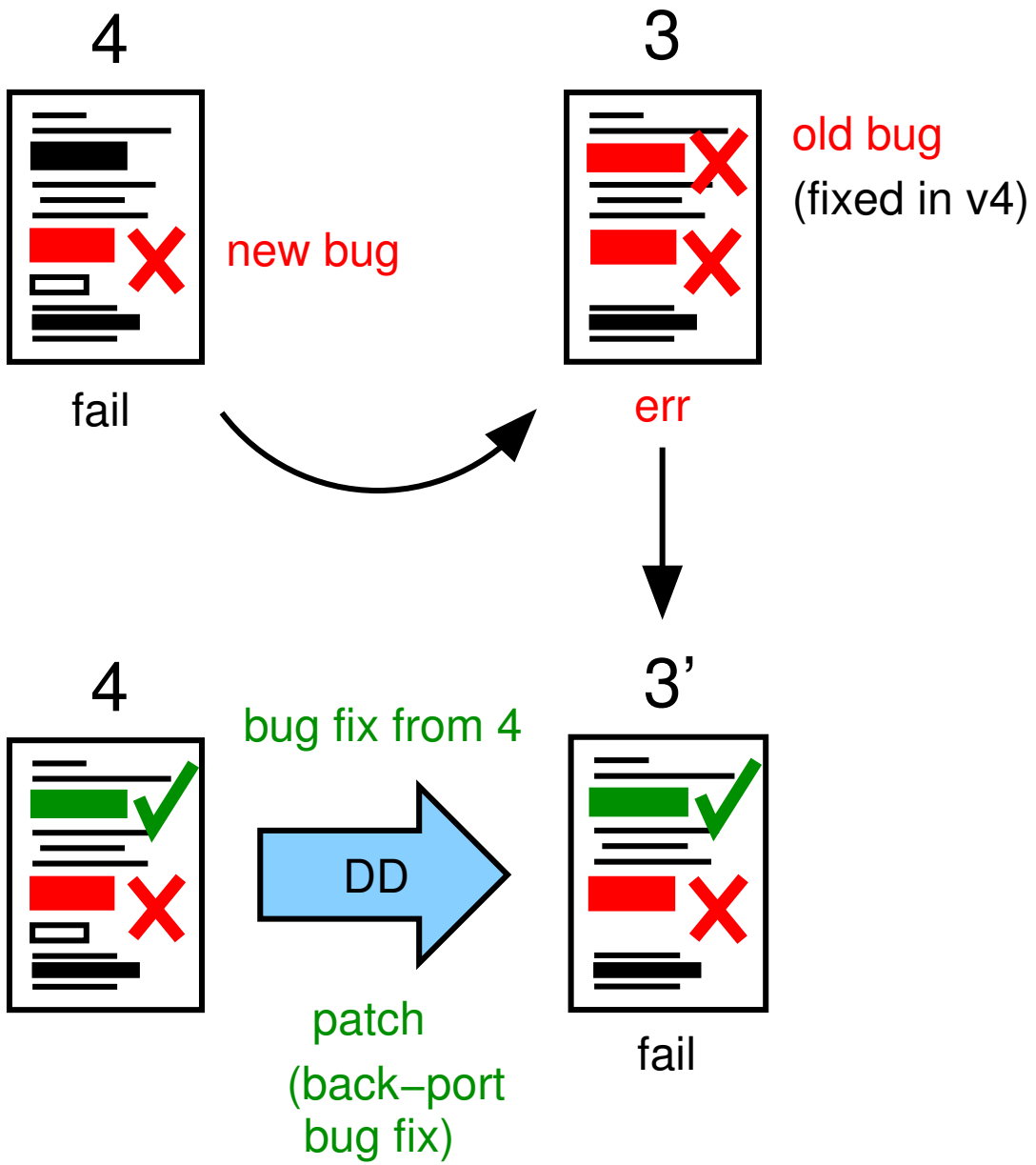


- Regression bug: New version contains defect.
- Assumption (or knowledge): older version can handle this case.
- Which version works is not known.

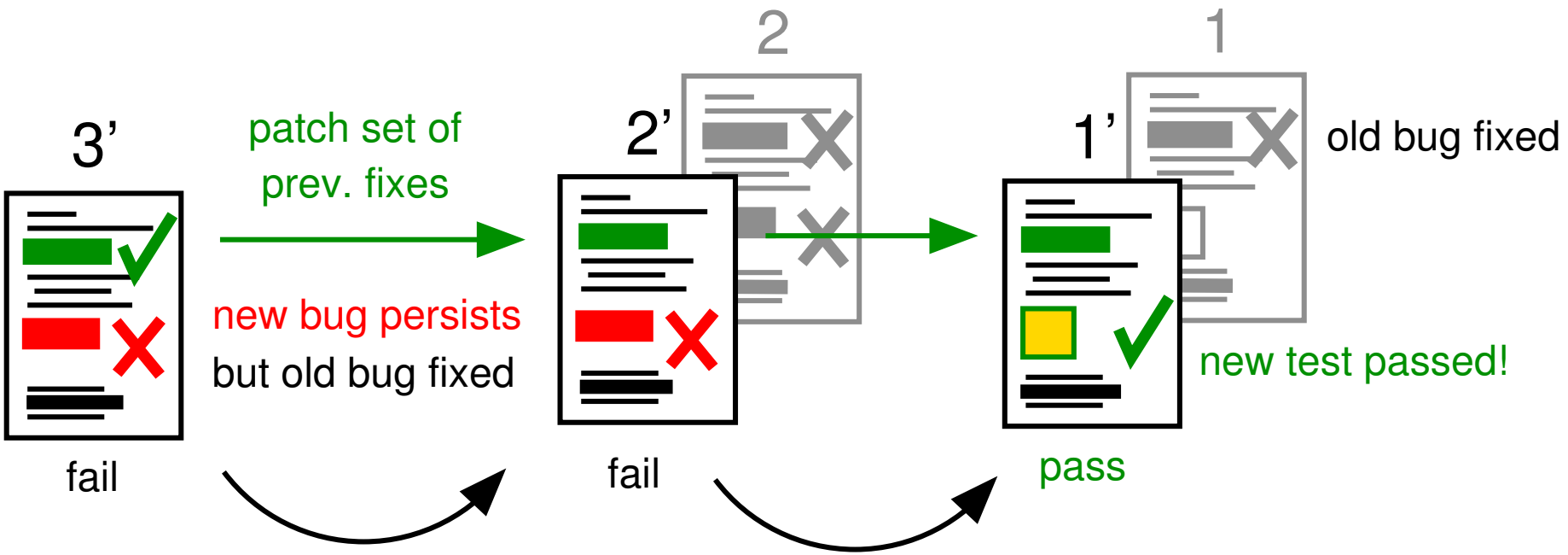
Try to find an older “good” version.

What if older version cannot run test?

- Use delta debugging to back-port test.
- Try to find which changes to apply.
- DD generates patch.

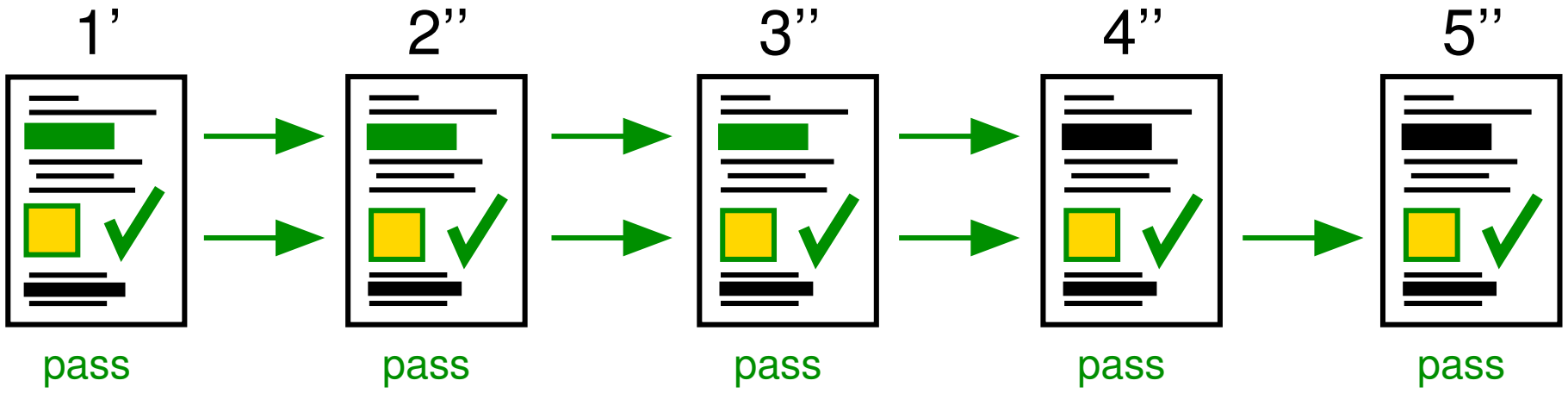


Iterated Delta Debugging



- Current change set (patch) is back-ported to older versions.
- If new problem encountered, DD is used again.
- Process is repeated until test passes!

Back to the present



- We don’t care about how to fix last year’s software!
- Use same idea to forward-port patch to current version.
- DD minimizes patch whenever necessary.

Problem 1: What is correct?

- Correct version not known (but test result can tell).
- Incorrect (but not totally flawed) version not well-defined.
- Easy to exclude obviously broken runs.
- Small flaws are difficult to avoid!
  - Removal of conditional or loop statement.
  - Removal of important function calls.
- Avoid elimination of correct functionality.

Use metrics (memory usage, coverage) to avoid such pruning.

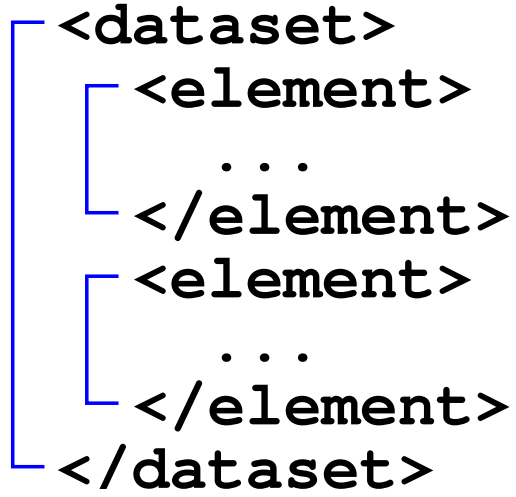
Problem 2: Wasted test attempts when using naive DD on program source code

- Lines of code are not independent!
- Block structure of program code.

```
+ #if 0                          - static void
+ static int                     - foo(...)
+ foo(void *data)                - {
+ {                              -    int x;
+     int x;                     -    x = ...
+     x = ...                    -    if (x) {
+ }                              -    }
+ #endif                         - }
```

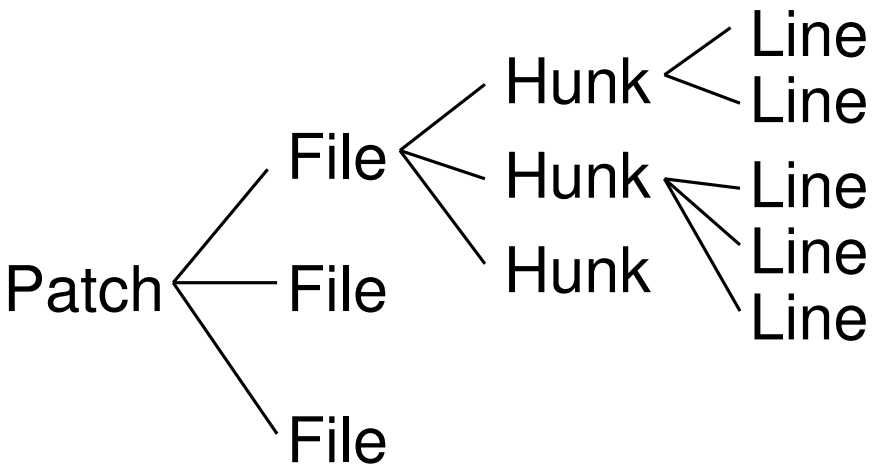
Addressing hierarchical structures: HDD [Miserghi 2006]

- Hierarchical Delta Debugging (for XML).
- Bisection search follows tree structure.
- Substantially better performance than DD, with better results.



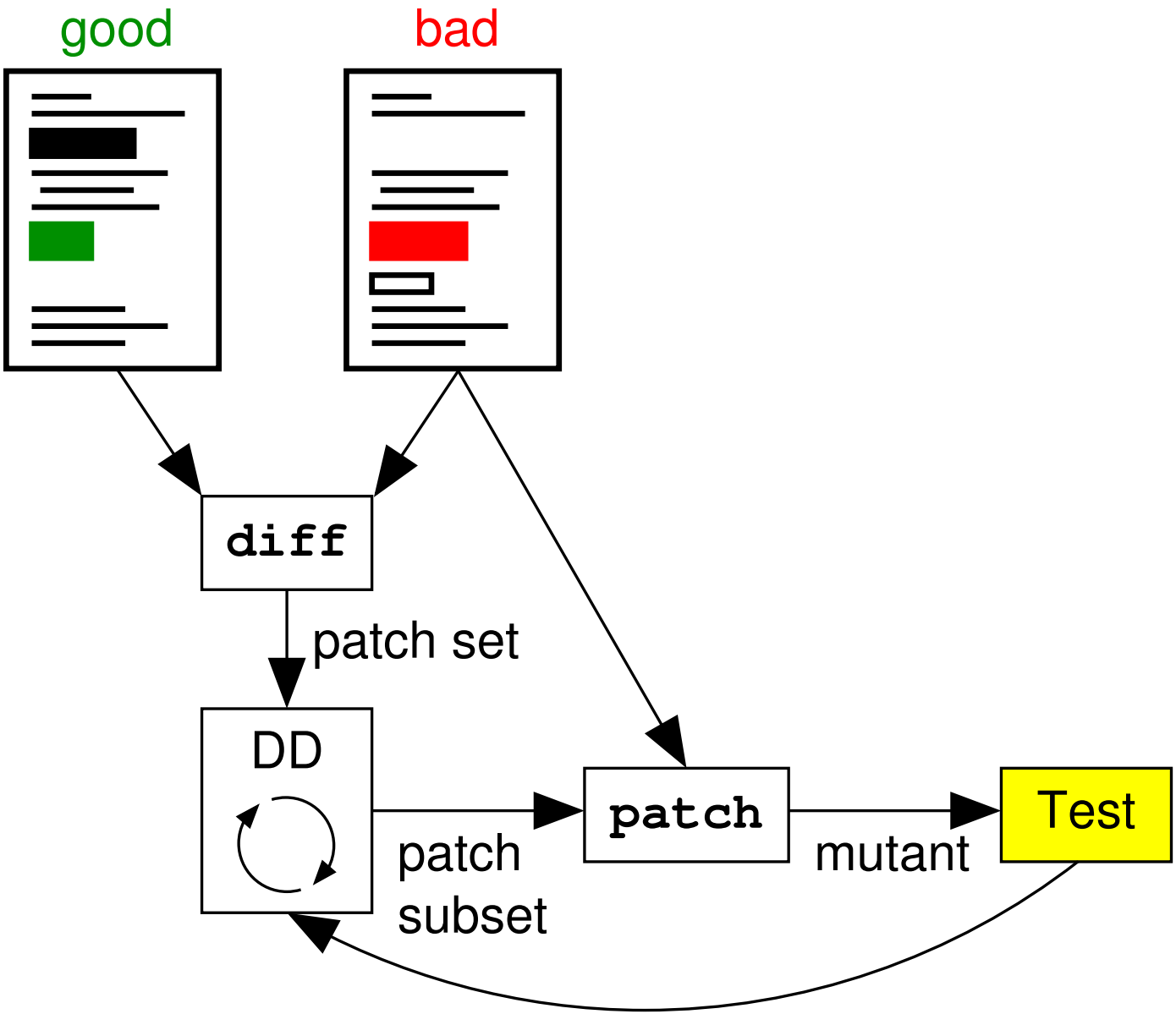
Patch file structure has similar hierarchy.

HDD for patches

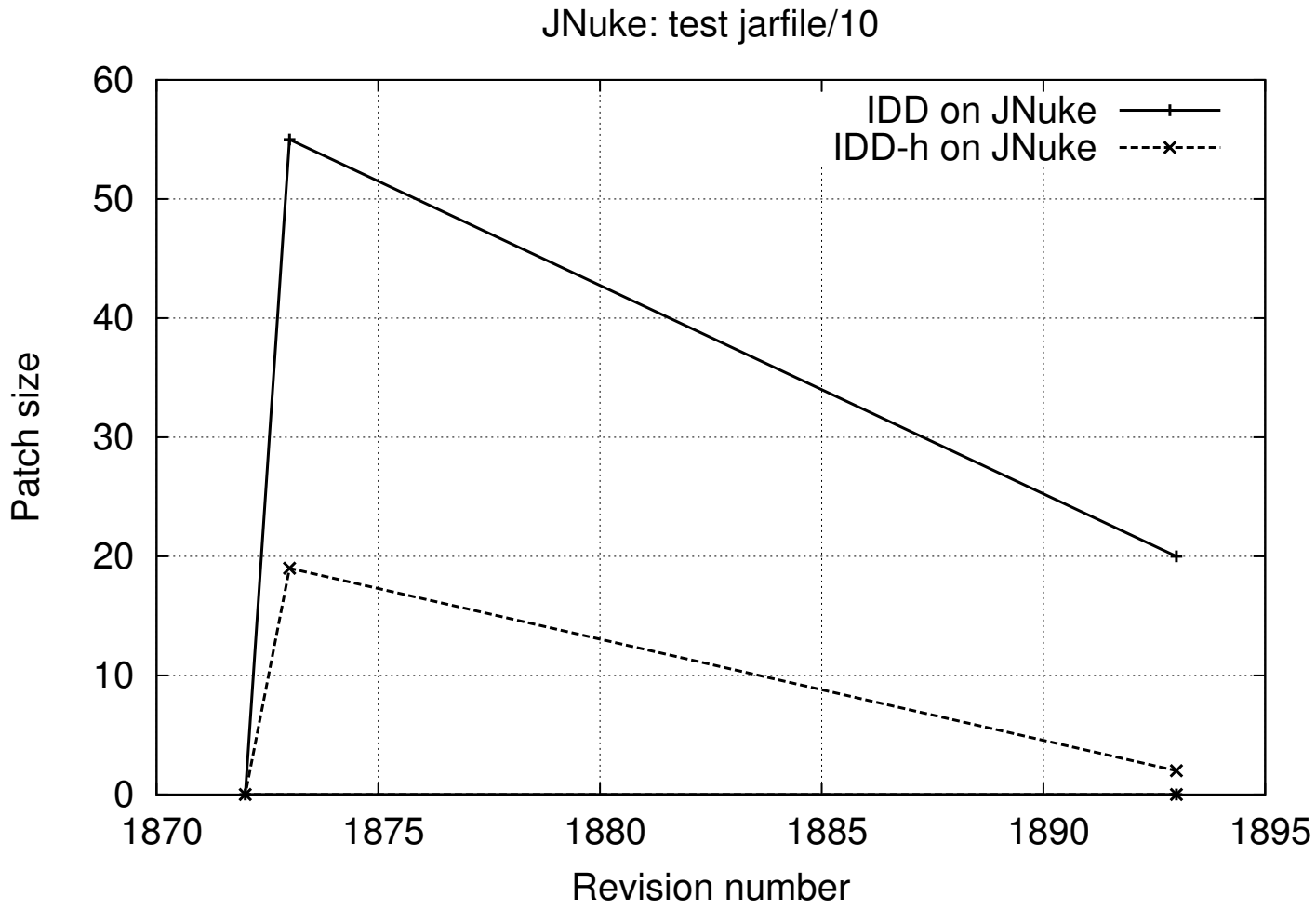


- Use „natural” boundaries of patch elements (files, hunks).
- Often, hunk corresponds to entire code block/function.
- In these cases, HDD is much better.

Implementation

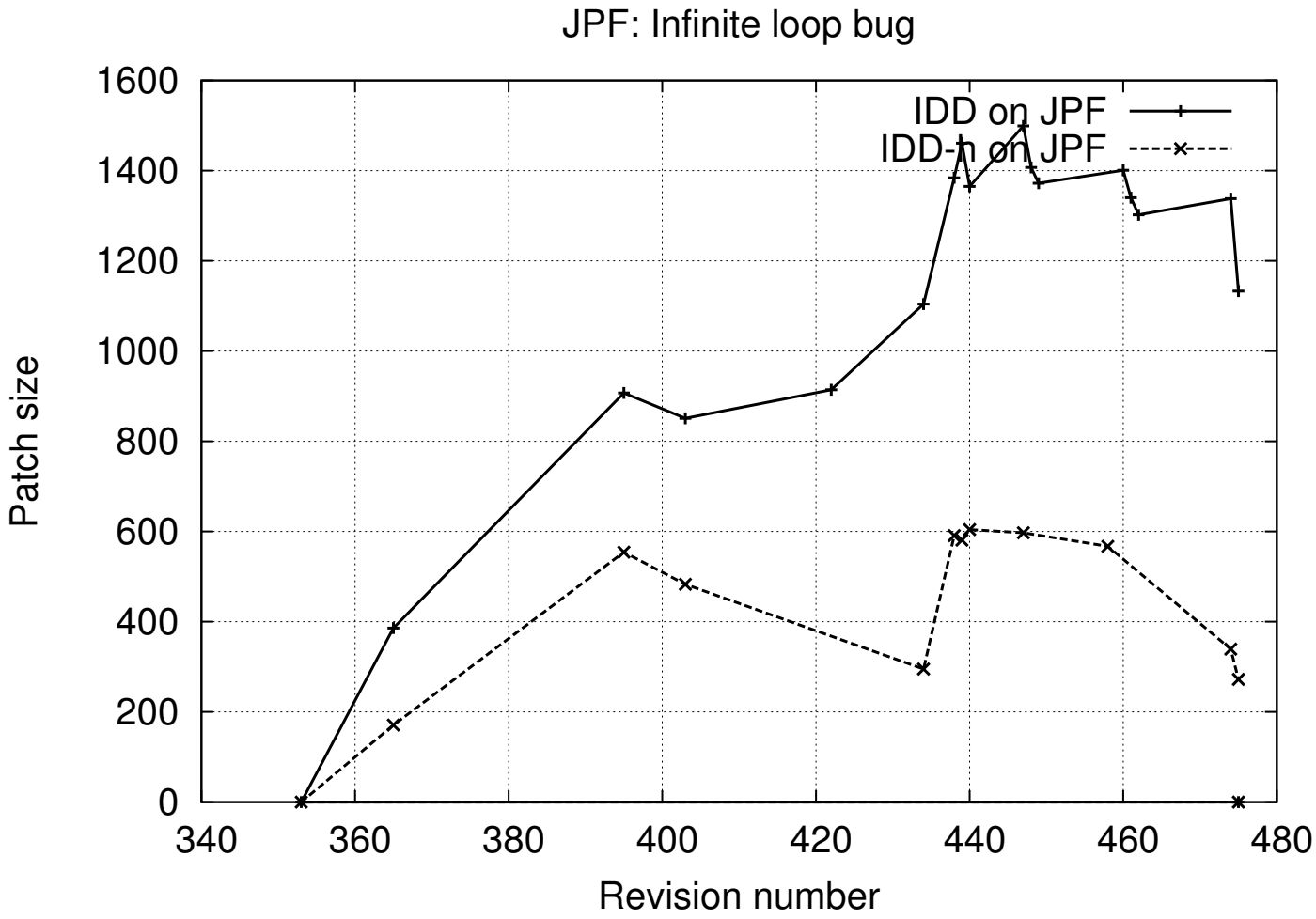


Experiment: JNuke: Jar file parser



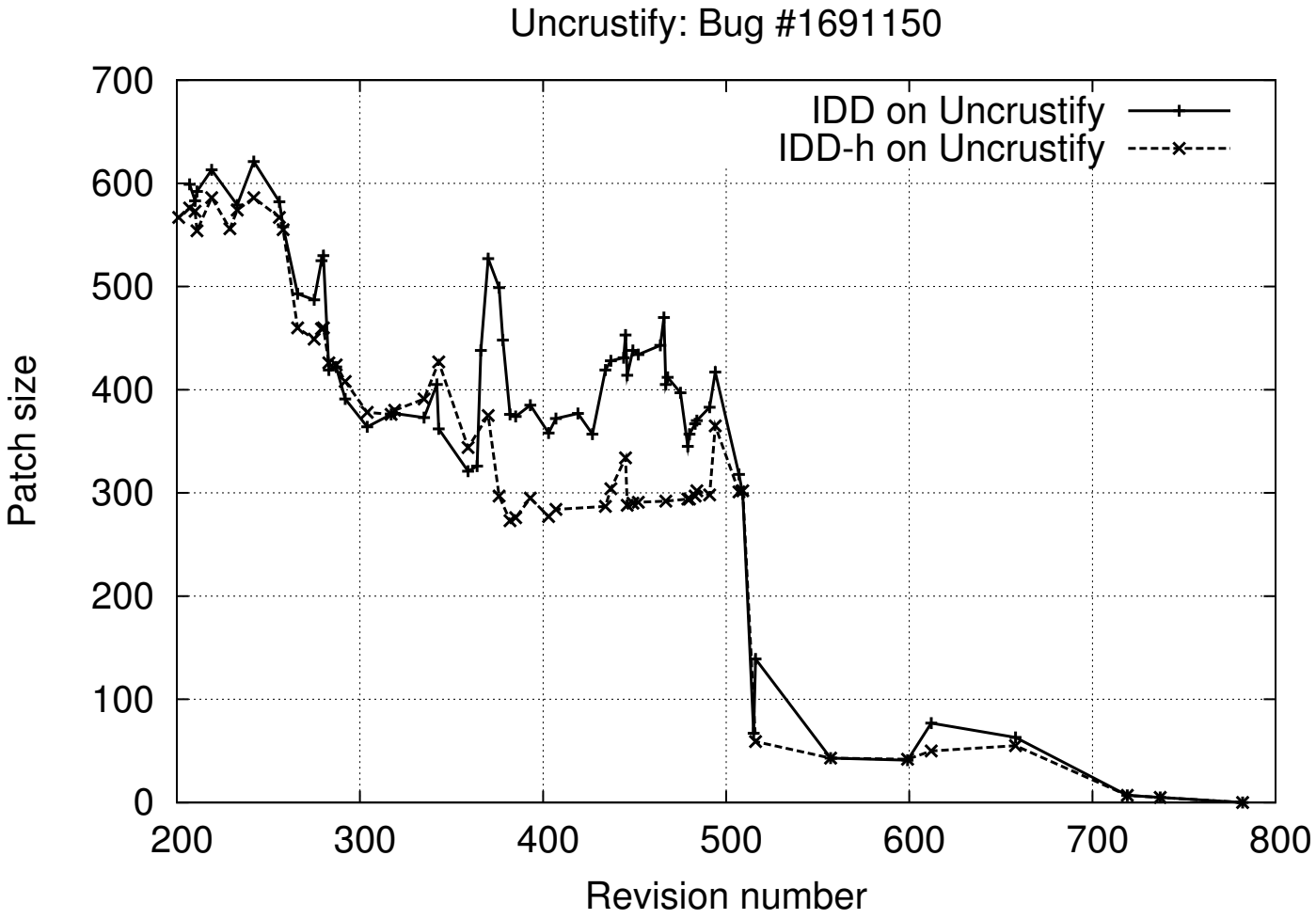
- Java VM for verification, written in C.
  - Code written under Linux fails under Mac OS 10.4.
  - Success: HDD found two-line explanation for error.

Experiment: Java PathFinder: Model checker



- Java model checker, written in Java.
  - Complex bug (5 minutes run time, > 1 GB data), more than a year old.
  - Good version found, but patch too large, breaks other features.

Experiment: Uncrustify: Source code formatter



- Program source reformatting tool, written in C++.
- Test successfully back-ported, but no good version found.

HDD itself is not good enough for programs

- Advantageous in some cases; going in the right direction.

Open issues:

1. Changes are usually not line-based!
   - DD tool needs other platform than just **`diff`**/**`patch`**.
2. Program code is not a tree structure.
   - Bisection search does not address such dependencies.

Conclusion

- Replace human effort with automated debugging.
- Delta Debugging: minimize change between „good” and „bad”.
- Iterative Delta Debugging: find „good” version if not known!
- Caveats:
  - Current `diff`/`patch` tools are too coarse (line-based).
  - Tools tailored to program source code needed.

Iterative “mining” of old revisions: a promising approach.

Observations

- „Unstable” patches: context information from `diff` is line-based.
- Lots of „cruft” → fragile patches → more cruft, etc.
- Tool chain not optimized:
  - Freshly obtains source each time (10 – 20 seconds wasted).
  - Fresh build each time (almost one minute wasted).
  - Many syntactically invalid versions generated!

  „Waste”: 90 – 98 %.
- In reality, hard-to-find bugs are usually committed together with refactorings and other changes!
- Worst-case scenario for DD; but only such „bad” commits lead to hard-to-fix bugs where DD could help.