Iterative Delta Debugging

Cyrille Artho[1], Etsuya Shibayama[1], and Shinichi Honiden[2]

[1] Research Center for Information Security (RCIS), AIST, Tokyo, Japan
[2] National Institute of Informatics, Tokyo, Japan

Abstract. Automated debugging attempts to locate the reason for a failure. Delta debugging minimizes the difference between two inputs, where one input is processed correctly while the other input causes a failure, using a series of test runs to determine the outcome of applied changes. Delta debugging is applicable to inputs or to the program itself, as long as a correct version of the program exists. However, complex errors are often masked by other program defects, making it impossible to obtain a correct version of the program through delta debugging in such cases. Iterative delta debugging extends delta debugging and removes series of defects step by step, until the final unresolved defect alone is isolated.

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [4,5]. Automated test runs allow efficient verification of software as it evolves. However, in the case of failure, defect localization (debugging) is still a largely manual task.

In the last few years, several automated debugging techniques have been developed in order to facilitate fault-finding in complex software [2,7]. Delta debugging (DD) is one such technique [3,8]. It takes two sets of inputs, one which yields a correct result and another one which causes a failure. DD then modifies both inputs with the goal of minimizing their difference while preserving the test outcome (one input succeeds, another one fails). DD can be applied to the program input or the source code of the program itself, using two different revisions of the same program. The latter variant treats the source code as an input to DD. It therefore includes a compile-time step, which produces mutations of the program to be analyzed, and a run-time step, where the outcome of that mutation is verified by testing.

DD obtains an explanation for a test failure. The scenario considered here is where a test fails for the current version. If an older correct version exists, DD can distill the essential change that makes the new version fail on a given test, and thus reduce a large change set to a minimal one [8]. DD is applicable when there exists a known version that passes the test. For newly discovered defects, this may not be the case. For such cases, we propose iterative delta debugging (IDD). The idea is based on the premise that there exists an old version that passes the test in question, but older versions of the program may have earlier defects that prevent them from doing so. By successively back-porting fixes to these earlier defects, one eventually obtains a version that is capable of executing the test in question correctly.

IDD yields a set of changes, going back to previous revisions. The final change, applied to the oldest version after removal of earlier defects, constitutes an explanation for the newest defect.

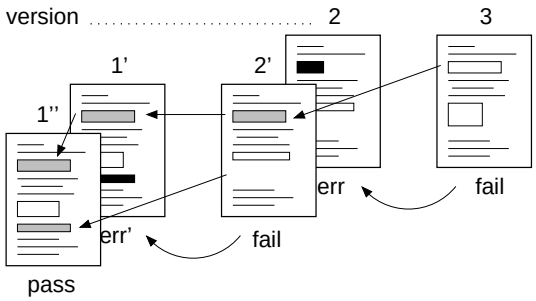

Fig. 1. Iterative delta debugging.

```
original_version = current_version = latest_version();
patchset = {}
original_result = current_result = test(original_version);
while (current_result ≠ pass) {
   current_version = predecessor(current_version);
   current_result = test(current_version ⊕ patchset);
   if (current_result ≠ original_result) {
      delta = DD(current_version, original_version);
      patchset = patchset ∪ delta;
      original_version = current_version;
   }
}
return delta;
```

Fig. 2. IDD algorithm in pseudo code.

2 Iterative Delta Debugging

Assume there exists a test that fails on a current version. We will call this outcome *fail*. A correct result is denoted by *pass*. Besides these two outcomes, there may also be different incorrect outcomes of the test, denoted by *err*. A set of changes between two versions of a program is referred to as a *patch*. Delta Debugging (DD) identifies a minimal change set that preserves the behavior of a test case between two versions [8]. Usually, this change set is used to explain a failure.

Figure 1 shows how IDD builds on DD. IDD is starts from a new version, which fails (version 3 in Figure 1). Unlike in the original scenario for DD, a version that passes does not exist a priori. IDD successively goes back to previous versions and assesses whether the same failure persists. If the test outcome differs, DD is then applied to the last failing version and the older version. This identifies the source code change that made the old version behave differently. One IDD iteration thus either produces a correct version, or eliminates a older defect that covered an newer one.

In many cases, IDD will not find a successful version in one iteration. Instead, an older version fails in a different way (*err*). Assume version 2 is such a version. DD is then applied between versions 2 and 3. The resulting minimal change can then be applied as a patch to version 2, fixing the earlier defect in version 2, and producing a

new version $2'$. The resulting version fails again in the same way as version 3 did. In Figure 1, the change that is back-ported is shown by an arrow pointing to the correct, changed code, in grey. Assume that one change does not completely repair the program. In that case, the iterative process is repeated. However, before the test is run on an even older version (such as version 1), the fix is back-ported again, producing version $1'$, on which the test is run. If that version behaves differently from $2'$, DD is applied again to find the minimal change required to fix the program. This produces a version with another patch (the change between $2'$ and $1'$) applied on top of the original one (the change between 3 and 2), resulting in a new version, $1''$. If IDD terminates, it will eventually find a version that passes the test, or run out of older versions (see Figure 2).

IDD can be partially automated by using DD to identify the patch to back-port. However, full automation is not possible. A given patch set is not always applicable to earlier revisions without changes, due to changes in the source code layout and structure. When the difference between two revisions grows too large, the patch tool fails. Therefore, large changes prevent full automation of the selection and back-porting of patches. However, application of DD still constitutes partial automation for extremely difficult problems where fully manual approaches are prohibitively difficult to apply.

IDD is best applicable to a source code repository containing many small, versioned (tagged) changes. For instance, subversion automatically assigns a unique tag to each change set, making such repositories an ideal candidate for IDD.

3 Experiments

We have applied IDD to the source repository of Java PathFinder (JPF), a Java model checker [6], using revisions 1 – 208 of version 4. In version 4, a complex input is not processed correctly. The defect is manifest in an incorrect result after several minutes of processing: The analysis report of JPF states that a faulty input (a Java program containing a known defect) is correct. Because the point of failure is extremely difficult to locate, this problem has persisted for several months and not been resolved yet.

A much older version, 3.0a, which is not maintained in the same repository, produces a correct result, though. However, using that version for identifying a change set would not be useful, because the entire architecture of JPF has been redesigned since version 3.0a was released. Hence, IDD was applied to different revisions of the source repository containing all revisions of version 4. The goal was to find a revision in the new repository that could pass the test.

Running a test takes several minutes in recent revisions, and five hours when going back to older revisions. IDD automates test execution, allowing these lengthy test runs to run unattended. Our IDD implementation also ignores failed builds (due to incomplete revisions in the repository), and has successfully found several defects, which could all be resolved by back-porting patches. Unfortunately, after applying ten nested patches throughout almost 200 revisions, a correct revision still could not obtained in that case study. The case study, however, lead to a systematic approach of back-porting changes, as presented in this paper.[3]

[3] At the time of writing, the defect still persists. However, in revisions that were made after our experiment, the fault is now masked by a `NullPointerException`, which almost immediately

4 Future work

When using IDD, challenges do not only occur when having to back-port patches throughout many revisions, but also in selecting the right patch. Sometimes, several faults are introduced when going back to an older revision, or there exist several ways to fix a fault. Therefore, the algorithm presented here may even have to "fork" into multiple revision trees in order to find a correct version. We have not attempted this in our case study, as this multiplies the amount of work to be done. Instead, we have relied on our own experience with a similar model checker [1] when selecting the patch of choice. IDD involving trees of revisions requires a more sophisticated tool than our prototype.

5 Conclusion

Delta debugging can identify minimal change sets. However, it requires a correct version, which may not be known when a new defect is discovered. Still, it is possible that an old revision would have processed a particular input correctly, if subsequent bug fixes of newer revisions were applied to it. Iterative delta debugging systematizes this process and successively carries changes from newer versions back to older ones, until either a correct version is found or the process is aborted. Future work includes more case studies, and attempting to investigate trees of modifications on older revisions.

References

1. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int'l Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
2. R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proc. 12th Int'l Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 63–72, Newport Beach, USA, 2004. ACM.
3. G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proc. 28th Int'l Conf. on Software Engineering (ICSE 2006)*, pages 142–151, Shanghai, China, 2006. ACM Press.
4. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
5. D. Peled. *Software Reliability Methods*. Springer, 2001.
6. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
7. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
8. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering*, 28(2):183–200, 2002.

leads to a failure. Even though our case study was not successful, this experience underlines the difficulty of manual debugging. The fact that a new failure now masks the fault we were looking for, gives further support for the viability of our idea.