Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Iterative Delta Debugging

Cyrille Artho

Research Center for Information Security (RCIS), AIST, Tokyo, Japan

The date of receipt and acceptance will be inserted by the editor

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 a series of defects step by step, until the originally unresolved defect is isolated. The method is automated and managed to localize a bug in some real-life examples.

Key words: Fault localization, automated debugging, automated repair, mining, delta debugging

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [29,31]. Automated testing enables 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 to facilitate fault-finding in complex software [25,41]. Delta debugging (DD) is one such technique [28,43]. It takes two sets of inputs, one which yields a correct result and another one which causes a failure. DD minimizes their difference while preserving the successful test outcome. 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 input (the program to be analyzed), and a run-time step, in which the outcome of that mutation is verified by testing.

The minimized change set obtained by DD constitutes an explanation of a test failure. The scenario considered here is where a test fails for the current version. If an older correct version exists, DD can be used to 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 [43].

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 other defects introduced earlier that prevent them from doing so. By successively back-porting fixes to these earlier defects, one may eventually obtain a version that is capable of executing the test in question correctly [4].

IDD yields a set of changes, going back to previous revisions. The final change, applied to the oldest version after removal of earlier defects, yields a revised old version that passes the given test. At that point, the same algorithm that is used to back-port fixes to older versions can also serve to port the found bug fix forward to the latest revision (or a version of choice). Our approach is fully automated. We have applied the algorithm to several large, complex real-world examples in different programming languages and types of repositories. Even though it was not known a priori whether a working revision could be found, we have found working revisions in some cases.

This paper is organized as follows: Section 2 introduces the idea behind IDD. DD and IDD are described in Sections 3 and 4, respectively. The implementation and experiments carried out are described in Sections 5 and 6. Related work is discussed in Section 7. Section 8 concludes, and Section 9 outlines future work.

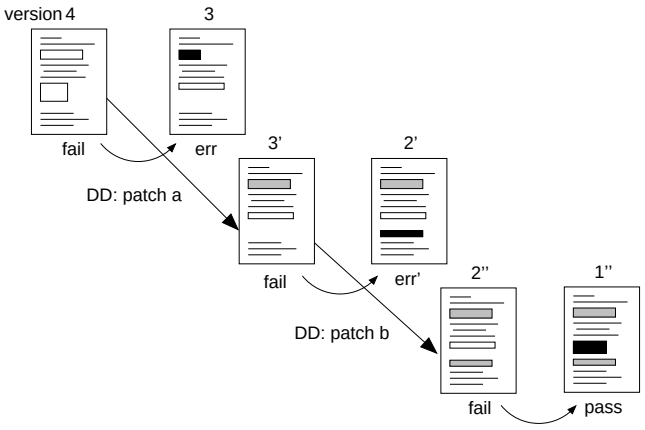

Fig. 1. Iterative delta debugging.

```
prev_version = current_version = latest_version();
patch = {}
original_result = current_result = test(current_version);
while (current_result ≠ pass) {
  current_version = predecessor(current_version);
  current_result = test(current_version ⊕ patch);
  if (current_result ≠ original_result) {
    patch = DD(current_version, prev_version);
  }
  prev_version = current_version;
}
return patch;
```

Fig. 2. IDD algorithm in pseudo code.

2 Intuition Behind IDD

Developers have used change sets for debugging before. When a regression test fails on a given version, one way to diagnose the problem is to compare the current version to an older version, which is known to pass the test. If a good old version is not known a priori, then it might be found by mining older versions from the history of revisions. Iterative Delta Debugging (IDD) automates this process:

1. A test fails on the current version. It is supposed that there might be an older version that passes the test.
2. By successively going back to older versions, one tries to obtain a version that passes the test.
3. One tries to distill a minimal difference between the "good" and the "bad" version, which constitutes the change that introduced the defect.

Step 2 tries to isolate two successive revisions, one that passes a test, and another one that fails. Step 3 attempts to minimize the difference between these two revisions. In that step, we assume that delta debugging (DD) is used to minimize a given change set while preserving the given test outcome [43].

If a version passing the test (a "good" version) is known a priori, then the search for the latest defective version can be optimized by using binary search instead of linear search. This idea has been implemented in the source code management tool `git`, which is used to maintain the Linux kernel sources [37].

The process becomes more complex in the general case, when the "good" version is not known, and a test might not be executable on older versions. Assume there exists a test that fails on the 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.*

A test case may not be applicable to older revisions due to missing features or other defects that prevent the test from executing properly. In this case, IDD utilizes DD to apply the necessary changes from a newer version to an older version, to allow a test to run. Figure 1 shows how IDD builds on DD. IDD starts from a version that fails (version 4 in Figure 1). Unlike in the original scenario for DD, a version that passes is not known a priori. IDD successively goes back to previous versions and assesses whether the same failure persists. If the test outcome differs, DD is 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 (a) skips a version where the outcome does not change, (b) finds a correct version, or (c) eliminates a older defect (*err*) that covered a newer one.

In the example in Figure 1, version 3 produces case (c): It does not pass the test, and even fails to reproduce the behavior of version 4. DD is then applied to versions 3 and 4. The resulting minimal change, patch a, can then be applied as a patch to version 3. Patching fixes the earlier defect in version 3 and produces a new version $3'$. Version $3'$ fails again in the same way as version 4 did. In Figure 1, the change set (patch) that is back-ported is shown by a thick arrow pointing to the correct, changed code.

The iterative process of IDD continues by applying this patch to older versions (such as version 2) before running the test. In the example, when applied to version 2, IDD produces version $2'$, on which the test is run. That version behaves differently from $3'$, so DD is applied again to find the minimal change required to fix the program. This produces a new patch b (the change between $3'$ and 2). This patch usually contains changes of the previous patch a. The resulting new version is therefore called $2''$. After version 2 is repaired, IDD continues. The patched version 1, $1''$, passes, and IDD terminates successfully in this example.

IDD will eventually find a version that passes the test, or run out of older versions (see Figure 2). IDD is fully automatic as long as the test process does not require human intervention. The process starts from a version where the test in question successfully compiles, and produces a known, well-defined output. (Even though that output is erroneous, it is important to use it as a measure against changes.) IDD successively tries older revisions, and uses DD as a subroutine upon failure. DD produces a minimal change set from a newer (working but not fully correct) revision. This change set is back-ported to the older version, reproducing the previously observed test outcome.

IDD is best applicable to a source code repository containing many small, versioned (or tagged) changes. The sizes of changes committed mainly depends on the development policy. Some development practices, such as Extreme Programming [7], advocate frequent, incremental changes. Each change is tested immediately, and committed to the development version of the source code. This strategy results in small change sets and facilitates automated analysis. Conversely, large changes, such as changes resulting from merging two versions, typically contain many unrelated changes and affect large parts of the code base.

3 Delta Debugging on Source Code

Delta Debugging (DD), as introduced initially, analyzes a flat change set, typically consisting of line-based or character-based changes [43]. If changes are independent, DD yields a one-minimal change set, meaning that the property of interest is no longer preserved if any one element is removed. DD typically requires a linear number of steps (in the size of the input), but may not produce an optimal output if changes are interrelated. Furthermore, the performance of DD is not optimal if a data is hierarchical. In that case, a search that divides the state space according to the hierarchical structure is much more efficient [28]. This section explains the application of both algorithms to program code.

3.1 Delta debugging

Delta debugging uses bisection to minimize the difference between two versions of program input while preserving the behavior that the first input generates. It can also be applied to the program source code itself. When applied to program source code, DD typically operates on a description of the difference between two revisions. The difference set generated by the Unix tool `diff` provides this change set readily. The usage of `diff` also has the advantage that variations of the change set can automatically be applied by `patch`, followed by compilation and execution of the program.

Delta debugging evaluates changes in contiguous subsets only, and ignores many possible subsets of a change set. Since the size of a power set is exponential in the size of the original set, an exhaustive evaluation of all possible change sets is intractable. In the following discussion, a "successful" or passing test is a test whose outcome corresponds to the desired value, such as the same kind of test failure observed in a previous revision.

Figure 3 illustrates DD on a simple example. The change set includes eight elements, the presence or absence of each is illustrates by a "1" or "0", respectively. Let the first element by the leftmost bit in the change set, and the eighth element be the rightmost bit. Initially, the empty change set is tried. The test, when executed unchanged, fails. DD then attempts to isolate the reason of the failure. The assumption is that if the test passes with only the first half of the change set being active, then the reason for the failure must lie in the first half of the change set. Conversely, if the test fails with the second half of the change set disabled, then at least a part of the second half of the change set is necessary for the test to succeed. This is what happened in the example in Figure 3: The test still fails after the initial bisection of the state space. Therefore, at least one element of the last four elements in the change sets is necessary to achieve the desired outcome.

Step	Change set	Test verdict
1	`00000000`	`fail`
2	`11110000`	`fail`
3	`11111100`	`pass`
4	`11111000`	`fail`
5	`11110100`	`pass`
6	`00000100`	`pass`

Fig. 3. Delta debugging illustrated.

In the next DD step (3), the subset of the last four elements is again bisected, and the last two elements are ignored. As the test passes, they can be safely dropped from the change set. Now, the algorithm backtracks and analyzes the subset containing elements 5 and 6. A direct implementation of backtracking would again analyze the change set of step 2. As that one is known to fail, it can be skipped. The change set in step 4 therefore contains elements 1–5; the test fails, confirming that element 6 is necessary. Step 5 confirms that element 5 can be dropped; further backtracking analyzes the first half of the change set, which does not contribute to the test outcome in this example. The final set contains only one change, and is one-minimal w. r. t. set inclusion.

3.2 Delta debugging applied to program executions

The original DD algorithm, called *ddmin* [43], operates on individual characters or lines of a failure-inducing input to a program. Besides success or failure, it also considers *indeterminate* outcomes. An outcome is undetermined if a program crashes, loops forever, or produces a different kind of faulty outcome that cannot be directly classified.

The *ddmin* algorithm analyzes indeterminate outcomes further, sometimes yielding better results than the DD algorithm described above in this paper. In its original version [43], the number of test invocations may be quadratic in the size of the input, in the worst case. Like many other implementations of DD, our algorithm considers any kind of indeterminate outcome as a failure. This reduces the worst-case performance to $2n - 1$ invocations in the size of the input. We found that this simplification works relatively well on program code, as the resulting algorithm still produces a one-minimal result, and avoids the quadratic worst-case scenario. However, it shares the weakness with *ddmin* of not analyzing all combinations of pairs, triples, and larger change sets. This is often a problem for program source code, where different parts (such as the declaration and use of a variable) are interdependent; see Section 6 for a detailed discussion of experimental results.

Function addition

```
+ #if 0
+ static int
+ foo (void *data) {
+   int x;
+   x = 42;
+ }
+ #endif
```

Function removal

```
- static void
- bar(int y) {
-   int x;
-   x = y;
-   if (y)
-     bar(--x);
- }
```

Fig. 4. Different use cases for delta debugging.

It is simple to derive the complexity of our algorithm: For an input of size n, a test may either succeed (preserve the desired outcome) or fail. If a test is successful, the state space search terminates. A failed test will lead to a bisection of the state space, and require two recursive invocations, on inputs of size $\frac{n}{2}$ each. In the worst case, with all tests failing, recursion proceeds $\log n$ times until, at the end of recursion, changes of size 1 are analyzed. This results in n invocations on deltas of size 1, $\frac{n}{2}$ invocations on deltas of size 2, etc., through all $\log n$ recursions. The total number of invocations is equal to

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + \frac{n}{n} = 2n - 1.$$

3.3 Analysis of program code

DD assumes that each element of a change set is independent of all others. As the format of the Unix `diff` tool is line-based, DD works best on data where one line is independent of any other line, such as a flat configuration file. Obviously, program source code does not conform to this property. For local code changes, though, DD provides a good approximation of the minimal change set. DD fails in cases where the hierarchical structure of program source code conflicts with line-by-line analysis of changes. In typical programming languages, a class scope surrounds declarations of functions and methods, which contain declarations of local variables and code. Higher-level syntactic entities cannot be removed without removing everything within their scope, or else the resulting program is no longer syntactically valid. Due to its line-based nature, when trying to eliminate the addition of a new method, DD tends to eliminate statements and declarations but not necessarily the entire method scope. When dealing with a change consisting of the removal of an entire function, that change cannot be broken down into smaller parts.

Figure 4 shows some of these problems. The two examples are shown in "unified diff" format, as produced by `diff -u`, with file names and line numbers omitted. On the left hand side, a C function is added to the code, on the right hand side, a function is removed. DD analyzes the given changes in conjunction with many other changes, spanning hundreds of lines. Bisection of the state space may happen at any location, and may not coincide with function or method boundaries.

Incomplete functions will not compile, so a bisection of the state space produces invalid code whenever the syntactic structure of the target language is violated. Because of this, DD can only manage to remove unused functions if bisection hits the boundaries of its declaration. In Figure 4, for DD to succeed in removing the changes shown, bisection needs to match the syntactic hierarchy. For the function addition on the left side, DD should hit the function boundaries and the scope of the preprocessor `#if`/`#endif` construct. In other cases, DD only manages to remove the code inside the function, and, if it works from bottom to top, the declaration of local variable `x`. On the right hand side, where a function is removed, the situation is even worse. A reduction of the change set corresponds to the elimination of a function removal from the change. In other words, if one starts with such a change as given, a reduction of the change set adds back code that was previously removed. As statements cannot be compiled without an enclosing function declaration, bisection needs to hit the function boundary exactly, or none of the changes can be removed. This example shows why DD is of limited usefulness if large parts of the code, especially function declarations and interfaces, change.

3.4 Hierarchical delta debugging

The problem of having to conform to a syntactic structure also occurs when applying DD to hierarchical data, such as XML. Previous work has implemented hierarchical delta debugging to produce correctly nested changes of XML documents [28]. In that work, it was shown that a hierarchical approach avoids generation of changes where opening and closing tags are mismatched. Because of this, the hierarchical approach tends to be faster and yield smaller change sets than the original DD algorithm [28].

When this idea is applied to program source code, the hierarchy of changes on a local scope corresponds to the hierarchy of syntactic elements. On a global scope, dependencies between callers and callees also have to be considered [45]. This information requires knowledge of the programming language, and is difficult to generalize across programming languages.

Luckily, the changes produced by the `diff` utility are hierarchical: Change sets consist of changes to individual files, which are in turn broken up into so-called "hunks", which contain a number of line-based changes. Figure 5 illustrates this hierarchy using the "unified diff" format. In real code repositories, some changes are entirely local in the scope of a single file or a block of code where it occurs. Extending DD to include the hierarchy of the generated patches can therefore improve precision of DD in these cases, as shown in the experiments in Section 6.

In this work, the patch file hierarchy provides boundaries for the state space bisection. Instead of bisecting the state space in the middle, based on the number of lines involved, bisection proceeds hierarchically, across files, hunks, and lines. First, sets of changes across files as a whole are analyzed. If a change for an entire file cannot be ignored, a more fine-grained search proceeds on hunk level, then on line level.

Taking advantage of the patch file hierarchy improves the recognition of local code changes, but it does not take care

Patch	Hierarchy
`Index: file1`	File
`@@ -42,2 +42,3 @@`	Hunk
`  context`	
`+ addition`	Line
`  context`	
`@@ -84,4 +85,3 @@`	Hunk
`  context`	
`- removal`	Line
`+ addition`	Line
`- removal`	Line
`  context`	

Fig. 5. The hierarchy of a change set produced by `diff -u`.

First call to DD	Second call to DD
`+   foo (NULL);`	
...	
`+ static void`	`+ static void`
`+ foo (void *data) {`	`+ foo (void *data) {`
`+   int x;`	
`+   x = 42;`	
`+ }`	`+ }`

Fig. 6. Recursive dependencies.

of interdependent changes. Addition or removal of a function may be taken care of by multiple iterations of DD: Once all calls and references to a function are removed, then the function itself may be removed as well. Unfortunately, even hierarchical DD cannot deal with certain changes affecting multiple files. For example, in cases where the signature of a function changes, its definition and all instances of its usage have to be changed simultaneously. A bisection-based algorithm such as DD cannot isolate such changes and produces overly large change sets.

3.5 Recursive dependencies

Delta debugging may be called multiple times on a given change set, if recursive dependencies are present. For instance, a change set may include the addition of a new function and a call to that function. The function itself cannot be removed without removing the call as well. If DD analyzes the function first, it may remove the function body but not the function itself. If DD subsequently removes the function call, then another invocation of DD on the result of the first one may also remove the function itself.

Figure 6 shows this example. Assuming DD works from bottom to top, it is able to remove the function body of `foo` during the first time, and the invocation of that function. Another execution of DD is needed to remove the function itself.

When using DD iteratively, subsequent invocations of DD on different revisions eventually remove recursive dependencies that were present in earlier steps. While a fix-point recursion at every step would produce a smaller change set, we found that code elimination in subsequent iterations works well, so we do not call DD multiple times on one revision. An exception to this is the final revision, where a fix-point iteration is performed. This is a design choice, the results of which are shown in Section 6.

4 Iterative Delta Debugging

IDD starts with a test failure in a new revision. The goal is to find an older revision that passes the given test. Whenever the outcome of a test changes in a different way than succeeding, e. g., by executing an infinite loop, delta debugging is used to create a minimal patch that preserves the former outcome. Iteration proceeds until no older revisions are available, a revision where the test passes is found, or a timeout is reached.

4.1 Iteration strategy

A given test case (or a set of test cases) is compiled, executed, and evaluated by a shell script. This evaluation is successively applied to older revisions, until a working version of the code is found. In most cases, the outcome of a particular test does not change from one revision to the next one. Instead, older revisions contain changes affecting other parts of the program. Delta debugging is only necessary when the behavior of the test case of interest changes. Such changes include compilation errors or other changes preventing test execution in the first place.

Iteration proceeds revision by revision. If a correct version was known a priori, binary search such as implemented in `git` could be used [37]. As by far the most time is spent in the DD part of the algorithm, improvements in the iteration across revisions would not improve performance significantly.

Previous work implemented this sequential iteration strategy but involved manual patch creation [6]. The usage of DD for patch creation automates our method. Unlike in previous work [6], patches are not accumulated, but replaced with a new patch each time delta debugging (DD) is invoked.

IDD starts with an empty patch set, and uses patches generated by DD whenever the test case in question behaves differently than previously observed. Each step of IDD, as shown in Figure 2, is guaranteed to terminate. Given enough time, IDD is therefore guaranteed to either find a “good” version, or reach the initial revision of a project, where no more change sets exist.

IDD assumes that each revision contains a set of related changes. In each iteration, when test behavior is affected, IDD either terminates (regarding the outcome as a success), or undoes the changes. If multiple changes affecting a test are present in a given revision, or if syntactic changes prevent a patch from being applied, IDD may regard the outcome as a failure. In such cases, IDD may miss a good revision because several changes combined in one revision produce a different outcome than individual changes. This problem in inherent in large change sets: In general, automated methods cannot deduce if a large change set can be logically decomposed into independent smaller changes.

4.2 Forward porting

So far, the given algorithm attempts to locate a correct version of the program by going back to older revisions. Patches (minimized by DD) are applied whenever a test cannot be executed in an older version. If the algorithm is successful, it will eventually find a revision where the given test passes. The final change set will contain a bug fix for the given test, as well as all the "infrastructure" needed to execute it.

Unfortunately, that final change set applies to an old revision, rather than the current one. If the bug fix applies to a version that is much older than when the test was implemented, the portion of the change set containing intermediate changes may be large, and the change set might not applicable to the latest revision. Of course, the primary concern is usually to fix the latest revision of the software rather than an old one. Therefore, after having found a successful revision, IDD is again applied in reverse, going from the correct (old) version to the current (new) one. DD is again invoked as a subroutine whenever a patch cannot be applied. In this way, the forward iteration generates a patch for the current version.

Forward porting always succeeds, although in the worst case, a patch set containing the entire change set between the correct version and the newest version may be generated. Furthermore, it may not always be practical to wait until the patch is fully forward ported when it becomes too large. In our experiments, we therefore also considered the size of the initial patch set fixing the defect (on an old version) versus the size of that patch applied to the latest version.

This final patch constitutes an explanation of the test failure on the current version. In general, the patch cannot be used to automatically repair the program. While the patch causes a given test to succeed, it may adversely affect other functionality in the program. Despite this, having a small candidate set of changes to consider when debugging narrows down the search for the defect tremendously.

5 Implementation

This section discusses the implementation, including a more fine-grained view of test compilation and execution, which is at the heart of each individual step in IDD.

5.1 Compilation and test setup

Within each iteration of IDD, and each step of DD, IDD uses a set of scripts to automate program compilation and testing. In detail, this involves the following steps:

1. Updating the source code to a new revision. This step may fail due to unavailability of the source repository (e. g. due to a network outage) or because an older version is unavailable. In the case of network outage, this step is repeated until successful.
2. Patching the source code. Patching is a likely point of failure, as any major changes in the source code, including formatting changes, make it impossible to apply a patch to a version other than the one the patch was generated for.
3. Configuration, preceded by the deletion of all compiled files. Configuration may require re-generating a build file. Deletion of all compiled files is necessary, as the patching process may create files that do not compile. This may result in the object file of the older version (which successfully compiled) being used in a current test run, and falsify test results.
4. Compilation. Also referred to as the build process, this step generates the executable program and may fail because the given version cannot be compiled before or after the application of a patch. If a given version does not compile even when unpatched, then it can be skipped.[1]
5. Test execution. Upon successful compilation, given test cases are executed. Test execution failure may be detected by a given test harness, e. g. if a known correct output is not generated by the test case. However, it also frequently happens that tests fail "outside" the given test harness. There are two ways for this to happen:
   (a) Catastrophic failure: Programming errors, such as incorrect memory accesses, may lead to failures that cannot be caught by the test harness. Therefore, the test harness of the application has to be complemented by an "outer" test harness that is capable of detecting such failures.
   (b) Incomplete test output verification: A given revision may not contain the full code to verify the test output. In that case, such a revision may erroneously report a failed test as successful. The outer test harness has to double-check that the lack of a reported failure actually implies success.

Any failure in the steps above (except for network problems in step 1) is treated as a critical failure that requires the current version to be fixed. The only exception is if a test fails in exactly the same way as a previous revision, in which case it is assumed that the test outcome has not changed. Otherwise, any change in the result of test execution requires invocation of delta debugging.

5.2 Test evaluation accuracy

It should be noted that IDD cannot guarantee that a correct revision (which passes a given test case) is detected as such. If DD removes code that does not contribute to a test failure, but is vital for a test to pass, then IDD cannot recognize a future successful test run as such anymore, as the functionality for the test to pass has been removed by DD. Figure 7 illustrates the problem on an example that could be run in C or in Java.

Let us assume that the test fails in the current revision, i. e., `runTest` returns 0. This fact is preserved when line 4 that calls `runTest` is removed. Unfortunately, this removes

[1] It sometimes occurs that a repository contains a version that cannot be compiled, for instance, because a developer committed only a subset of all necessary changes.

```
void test1() {
  int result;
  result = 0;
  result = runTest();
  assert(result != 0);
}
```

Fig. 7. Example showing a potential code removal that would prevent a test from passing.

the entire test functionality from the program. A different revision where `runTest` returns 1 would not be recognized as being correct. Even worse, if the code is compiled as a C program, the initialization of `result` and the `return` statement may be removed as well. This causes the return value of the test function to be undefined. During execution, the value probably corresponds to the contents of an element of a previous stack frame that occupied the same memory, but such behavior is platform-dependent. The test harness has to be augmented in order to prevent such deviations.

In order to maintain the exact behavior of failing tests, static or dynamic slicing [21,41] could be used. Slicing would ensure that DD does not remove any lines that contribute to the value of the test result. However, slicing tools are not portable across programming languages, and do not scale well to large programs. We have therefore chosen a less stringent approach. In addition to monitoring the output and log files as closely as possible (using an external test harness), memory consumption and time usage are also checked for deviations from the expected previous value. Furthermore, usage of uninitialized data in C programs is prevented by inspecting compiler warnings and the final change set generated by DD. This process is currently not fully automated, but could be automated by static analysis tools, and by using memory checking tools such as valgrind [30]. For Java, the compiler and run-time environment already enforce that memory cannot be corrupted.

5.3 Implementation architecture

The iterative step of IDD, which applies a given patch across several revisions, is implemented as a shell script using the compilation and test setup described above as a subroutine. The script iterates through existing revisions until a change in behavior is detected. After that, it stops, and control is transferred to the DD program.

When used on a subversion code repository, IDD can take advantage of the fact that all versions are globally and consecutively numbered. Stepping through older revisions is therefore trivial. When using CVS, though, global revision numbers are not available. They are recovered by pre-processing the code repository. During this step, a revision counter is assigned to each revision that is more than five minutes apart from the next one. Revisions being less than five minutes apart are regarded as a single version that was committed using multiple CVS invocations.

DD is implemented as a Java program that takes a patch set derived by the Unix `diff` tool as input (see Figure 8).

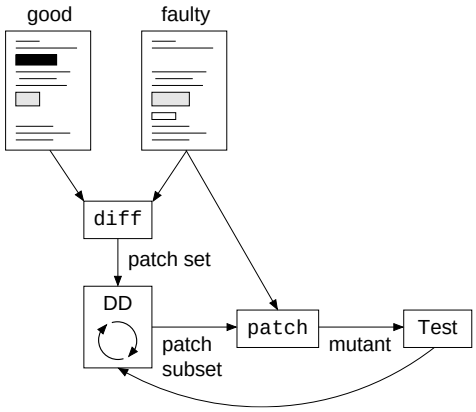

Fig. 8. Implementation architecture of DD.

It parses the patch file and produces an internal representation of the state space of all possible patch sets. It then iterates over the state space of all possible change sets. Each change set is produced as a modified patch, which is applied to the faulty version using the `patch` tool. The resulting version reflects a subset of all changes between the good and the faulty version. Compilation and testing are then used to decide whether the mutated version still produces the desired outcome. DD continues its state space search based on this outcome, until the state space is exhausted.

For the hierarchical implementation of DD on patches generated from program source code, three levels exist: files, hunks, and lines. In our implementation, this hierarchical dependency is reflected by three classes extending an abstract base class. The base class implements the actual hierarchical search algorithm. The subclasses contain code that correctly composes a representation of the current component state (including subcomponents) as text in the patch format, a parser for the respective component of a patch file, and in the case of the lowest hierarchy level, direct calls to the bisection state space search of DD. This implementation allowed us to switch from a flat representation of patch files (a large set of lines) to a hierarchical one.

When bisecting the state space given by a change set, bisection starts within the top-level hierarchy and recursively proceeds until the leaf nodes are reached. If these leaf nodes contain multiple (nested) components, state space bisection may proceed at the lowest level. The presence of nested components has to be tested recursively as well, as it may be the case that, for example, a single file contains a single hunk consisting of multiple lines. In that case, only one component is present on the level immediately below the current one, but at levels further down in the hierarchy, multiple elements may still exist.

5.4 Caching previous test executions

The usage of the Unix `diff` and `patch` tools eliminated the need for a custom representation of the difference between two program versions. This made the DD algorithm very elegant, except for the cache structure, which was difficult to

implement for hierarchical DD. Caching adds a subtle interaction between recursion hierarchies of completed, partially completed, and incomplete parts of the search space. To ensure that the state space is bisected correctly even in the case of a nested hierarchy, about 400 unit tests were used. These tests covered many corner cases, but two additional incorrect corner cases were found by inspecting the output when an initial version of the tool was run.

The state space search starts with an empty change set and then subsequently enlarges that change set until a one-minimal change set with the desired test behavior is found. Before a test is actually invoked, though, a cache containing all previously visited states, along with their test outcome, is consulted. This avoids revisiting certain states when backtracking: Whenever a test outcome in the initially investigated half of the bisected state space is successful, the recursive algorithm backtracks and analyzes the other half of its substates. The state immediately after backtracking corresponds to the state before the previous bisection step. In pathological cases, where every second element of a change set can be eliminated, one invocation for each change set of size 2 can be avoided through caching. This eliminates $\frac{n}{2}$ tests, n being the size of the input (initial change set). Caching comes at the cost of having to keep a copy of a bit string representing each DD state. This requires $O(n)$ storage space per state, for up to $2n-1$ states. Implemented naïvely, cached DD therefore provides at most a linear gain in time for a quadratic cost of storage. In practice, the memory overhead for caching was never noticeable. While at a given point in time, some cached states may not be reachable anymore and could be discarded, we never needed to reclaim used memory and have not implemented this.

6 Experiments

This section describes how the experiments were chosen, which test cases were analyzed, and conclusions drawn from the results.

6.1 Selection of experiments

A major challenge for this work was to find suitable experiments. It is desirable to validate a tool on actual faults reported by users or developers, rather than artificial (seeded) faults created for the purpose of a case study. While seeded single faults share some characteristics with real ones [2], such a relationship is not confirmed for more complex faults spanning multiple locations. Furthermore, this project, which mines the history of all revisions for potential repairs of software, was inspired by a real application problem. Therefore, we decided not to use any fault seeding, to avoid a bias towards small, localized faults. Small changes resulting in single faults or small sets of faulty components can be analyzed relatively well by existing approaches, but such faults can also be found more easily by a human. Success is less likely on long-standing faults where automation may direct a developer towards the root of the problem.

In software quality assurance, a *regression* denotes a defect that is absent in an older version but introduced in a newer one. In other words, a regression fault occurs when a change in the software results in a feature not functioning correctly anymore. The approach presented here works best on regression faults, and scenarios where human defect analysis cannot determine the problem cause. Therefore, sufficiently large projects were chosen that warrant automated debugging, and are mature enough to contain many old versions that can be mined for changes. Except for the JNuke project, where we had direct access to a local copy of the source repository, we used projects hosted on the popular development platform `sourceforge.net`. From projects hosted on that web page, the following technical constraints had to be fulfilled to make the use of our tool feasible:

- The full history of all revisions had to be available for anonymous access. This was only the case for about 20% of all projects on a given platform. In some cases, the repository was available as a CVS repository, but it was not documented which module to check out; this problem does not occur with SVN, where all branches can be checked out at once.
- Tests had to be fully automated and repeatable, including any configuration files needed to run the test. This criterion ruled out any applications using a graphical user interface or processes running in the background without generating any directly visible output.
- The application had to be sufficiently self-contained to allow automated compilation throughout its life cycle. This was a major problem in particular with large Java or C++ applications. A long list of dependencies on external libraries or tools ruled out about half of all projects on a given platform. For large projects, dozens of external libraries are common, and automated downloads of these often fail at a given time because one of the servers is not available. In other cases, incompatibilities between the platform used for experiments, and the development system the project was created on, prevented analysis.

Projects fulfilling the criteria above were mined for regression defects by consulting their bug tracker database. The bug tracker contains defect reports submitted by end users, and has no standardized way of marking a defect as a regression bug. Reports were therefore studied manually in search of likely regression defects. This classification itself was done according to the subjective experience of the submitter and not always correct, as experiments showed. Reading the bug reports actually proceeded very quickly, as the majority of bug reports contains no code or example input, and thus are not suitable for automated testing.

After initial case studies [4], the scope of the experiments was enlarged by choosing the 50 most popular projects from `sourceforge.net` in the programming languages Java, Ruby, Python, and Haskell. Given the constraints above, a handful of successful candidates emerged. In particular, for

Ruby, Python, and Haskell, no candidates that fulfilled all the criteria above could be found. The main reason for this is that in these communities, other servers usually host the source code; `sourceforge.net` is used to represent the project on that web page, but not to host its code or bug repository. For Java and C++, the odds of getting access to a working SVN repository were much better, which is why Java in particular has the largest share of projects used for experiments. For each eligible project, the 50 most recent open bug reports were scanned for candidate test cases that could be reproduced automatically on our machine. Overall, several hundred bug reports of about 20 projects were evaluated, out of which 13 case studies were chosen. These case studies were marked as potential regressions in the bug database, or had defects related to features of which the implementation had undergone major changes, making them a possible regression. Only a few different projects remained in the final selection because the need for sample input and output data effectively narrowed the selection of projects down to tools that are used non-interactively with a well-defined input and output.

Table 1 lists the case studies taken for the experiments. The first three columns show the project name and a brief description of the purpose of the application, together with the kind of defect that was analyzed. The next three columns show the implementation language, the size of each project (in the version that analysis started at), and the type of revision control system used. The penultimate column shows how long it took for a human to report the defect, from the time when it was introduced into the system. This could of course only be analyzed post mortem, after IDD had found a good version. For defects that could not be confirmed as regressions, and where no good version could be found, "*" is indicated. For the bytecode instrumentation problem in JNuke, a bug that was originally fixed after a code review, but not tested against, was back-ported for this case study. In that project, 212 days had passed between the initial bug fix and the test that confirms the fault. Finally, the time it took for a human developer to repair the defect is shown in the final column; as can be seen, some problems took up to two years to be repaired, so it can be said that these defects are hard to analyze. For JNuke, the project had not been under development for some time when it was ported to another platform, where the defects showed up; so the time to fix would not provide a meaningful metric here.

6.2 Uncrustify

Uncrustify is a source code formatting tool for various programming languages including C, C++, and Java [16]. In addition to formatting code, changing whitespace, Uncrustify is also capable of changing the code itself. For instance, one-line statements following an `if` statement do not have to be surrounded by curly braces. Uncrustify can be used to add these optional braces, facilitating code maintenance.

The bugs registered in the sourceforge bug tracker under numbers 1691150, 1723794, and 1739348 were used as case studies. These bugs were not reported as regression bugs, so it was not certain that mining old revisions could yield a bug fix. However, given the rapid addition of new features to Uncrustify during that time, these bugs seemed to be good candidates.

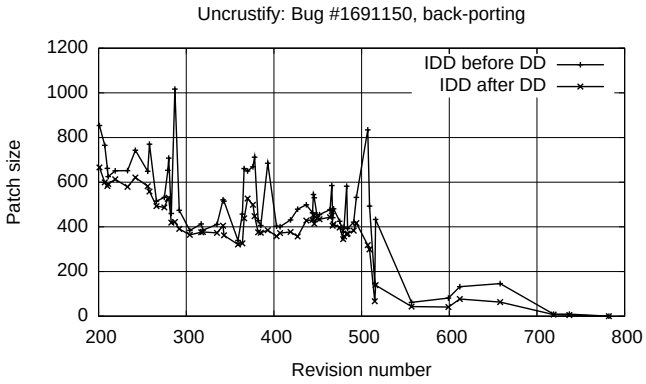

Fig. 9. Result of using IDD on Uncrustify.

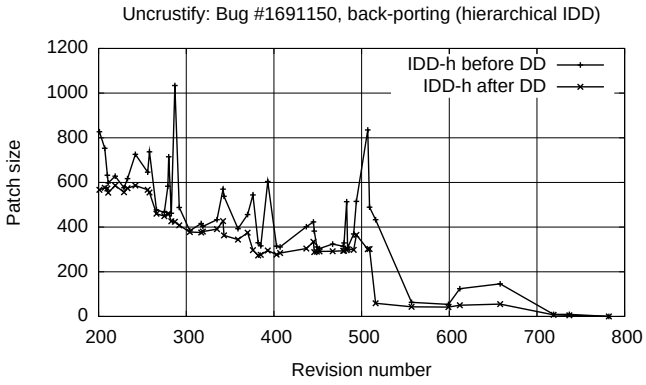

Fig. 10. Result of using hierarchical IDD on Uncrustify.

6.2.1 Placement of comments

This bug describes how C++–style one-line comments (starting with `//`) are sometimes moved to the wrong code block when they are converted to C-style comments (enclosed by `/*` and `*/`), if, at the same time, optional curly braces are also added for `if` and `while` statements. While the bug report described three other issues, we focused on comment placement with our test.

Figures 9 and 10 show the result of running IDD on that case. The sizes of the change sets at each step before (above) and after reduction by DD (below) are shown. The difference between the two points for the same revision number (horizontal axis) corresponds to the reduction of the change set by DD. Note that a large reduction, resulting in a small "output" patch, tends to lead to a smaller "input" patch at the next DD iteration. Where the given patch could be applied directly, no attempt was made to reduce it further, so each point in the curve is equivalent to an invocation of DD during back-porting. The sudden growth of the change set at revi-

Table 1. Overview of case studies.

Project name	Description	Defect	Impl. lang.	Code size [KLOC]	Repository system	Time to detect	Time to fix
Uncrustify	Source code formatter	Comment placement	C++	20	Subversion	*	125 d
		C++ functions				*	4 d
		Conditionals				*	1 d
Java PathFinder	Java model checker	Infinite loop	Java	70	Subversion	60 d	690 d
		Cyclic deadlock				*	282 d
		Lock counter				*	733 d
JNuke	Java VM/run-time analyzer	Jar file parser	C	130	CVS	22 d	*
		Class loader				542 d	*
		Bytecode instrumentation				0 d (212 d)	
itext	PDF manipulation library	Unicode problem	Java	180	Subversion	*	$>$ 49 d
hsqldb	Database	Compatibility	Java	250	Subversion	*	122 d
		Deadlock				193 d	10 d
		NullPointerException				24 d	0 d

sion 509 corresponds to a refactoring where the large number of command line options was specified differently in source code. DD could not eliminate this refactoring. As older versions supported fewer and fewer options, the expected growth in the patch set was sometimes compensated by the shrinkage of the part of the patch that concerned command-line options.

IDD could not find a version passing the given test. Unfortunately, revisions 200 and older could not be checked out with the given subversion (`svn`) client. This problem was independent of our IDD implementation; even a fresh checkout of that old version failed. After this experiment had been carried out [4], the defect was eventually fixed by the developer, four months after the bug had originally been reported.

6.2.2 Other case studies

In addition to the experiment previously performed [4], two other case studies were taken. In these cases, the bugs were not long-standing issues, but it was possible to evaluate them with the same test harness that was used in the earlier case study, giving two more cases to test our approach on.

Incorrect handling of C++ functions with no parameters Bug 1723794 addresses a cosmetic problem where functions in C++ are not formatted consistently. The defect affects functions with an empty list of parameters. The semantics of the compiled code are not affected, but the bug was still repaired quickly, within four days.

Brackets around boolean conditionals Bug 1739348 relates to an erroneous syntactic change: Boolean expressions in brackets were enclosed in additional brackets. For instance, expression `!(a || b)` became `(!(a) || (b))`. This change affects the semantics of the expression and is of course highly undesirable in a code reformatting tool. This bug is much more critical than the earlier ones, so it is not surprising that it was addressed within one day.

For both C++ functions and boolean conditionals, IDD was able to evaluate the given test without having to back-port any code. Again, revisions 200 or older could not be checked out. Therefore, these two newer defects are not regressions, as no older version of the code could be found that passed these tests.

6.3 Java PathFinder

Java PathFinder (JPF) is a Java model checker that analyzes concurrent Java programs for assertion violations, deadlocks, and other failures [39].

6.3.1 Infinite loop

In a new version of the tool, a major architectural change caused JPF to run into infinite loops for certain programs. Because the execution of a correct version takes a few minutes, the program has to be run with a timeout of ten minutes to ensure that the version is indeed faulty and not just a bit slower

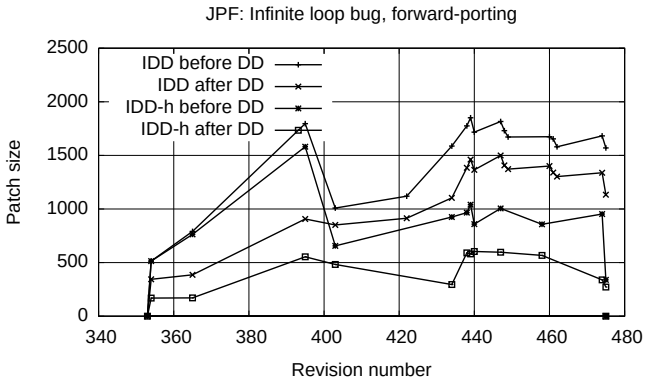

Fig. 11. Result of using IDD on Java PathFinder.

than the correct one. The long test execution time makes a manual search for the problem very tedious, and inspired this work.

A much older version of JPF, 3.0a, which is not maintained in the same repository, produces a correct result. 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. Perhaps the bug was introduced after the architectural changes that took place prior to the migration to a public source repository. In that case, IDD could find it.

We have applied IDD using revision 475 of version 4 as a starting point. IDD iterated through older revisions up to version 353, which passed the test (see Figure 11; both versions of IDD are shown in one plot here). In this case, no changes had to be back-ported to that revision. Therefore, the visible part of the graph shows the size of the patch fixing the defect. Delta debugging was unable to identify a very small change set to fix the newer, defective versions. During forward-porting, the initial patch of 386 and 171 lines, respectively, grew successively. Sometimes, a later iteration of DD took advantage of previously resolved dependencies and shrank the patch again. However, at the end, an unwieldy patch of more than 1,000 lines remained for standard IDD, and a patch of 272 lines for hierarchical IDD.

This is still too large for the patch being meaningful to a human programmer. The generated change set still includes refactorings amidst functional changes, and while it fixes the given bug, it breaks other features of JPF. Because of this, we did not follow up with this bug fix further to actually repair JPF. After this initial case study [4], the problem was eventually fixed, more than two years after it had been introduced.

6.3.2 Additional case studies

Failure to detect cyclic deadlock Bug 1727279 describes a failure of JPF to detect a cyclic deadlock between two locks. In later versions, internal failures in the maintenance of the lock state of the virtual machine cause errors. The fault was eventually repaired by a change that addressed at least eight unrelated issues, according to the log file of the source code repository. Because of this, it is unclear whether the defect repair was just subsumed by other changes or specifically intended, as this particular issue was not mentioned in the log, and the bug tracker still reports this problem as being open.

Incorrect internal lock counter A different defect, reported as bug 1523912, relates to an incorrect lock counter in the virtual machine. The defect evolved over time into a case where JPF failed to report an error for a faulty program. Eventually, the defect was repaired when a large change set from the development branch was merged.

Both additional case studies turned out not to be regressions, as no working earlier version could be found. Therefore, IDD was not able to synthesize a fix.

6.4 JNuke

JNuke is a tool platform written in C to analyze Java programs [5]. It includes a Java virtual machine, static and dynamic analysis components, and a bytecode instrumentation tool [3]. The class loader of the virtual machine includes a jar file parser as a component; that parser was analyzed in a first experiment [4]. This publication includes two large IDD runs that have been completed recently, on two unrelated components: The code pretty printer of the class loader, used for testing, and the bytecode instrumenter.

6.4.1 Jar file parser

In the first experiment on JNuke, IDD was applied to find the reason of a memory access problem in the jar file reader under Mac OS X (10.4) that was not found under Linux. This is a typical case in which IDD can be applied: Code that is often tested on one platform (in this case, Linux) but rarely on another one (Mac OS). Such tests may pass on the main development and test system, but fail on a different platform. As the system is periodically tested on the other platforms, some known good versions exist, but regression defects may go undetected for some time. IDD is used to identify the most recent good version.

As Figure 12 shows, IDD was very successful here. The test could be run without any adaptations on older versions, until it passed at revision 1872. The graph shows the size of the patch fixing the problem. Hierarchical IDD even found a minimal patch of just two lines, identifying the fault precisely. This example is a case where IDD would have found the test quickly enough to be useful for replacing human effort in debugging.

6.4.2 Class loader

In this experiment, a buffer overrun in the class loader produced a failure on Mac OS X (10.4) that was not detected under Linux. This case differs from previous cases in that

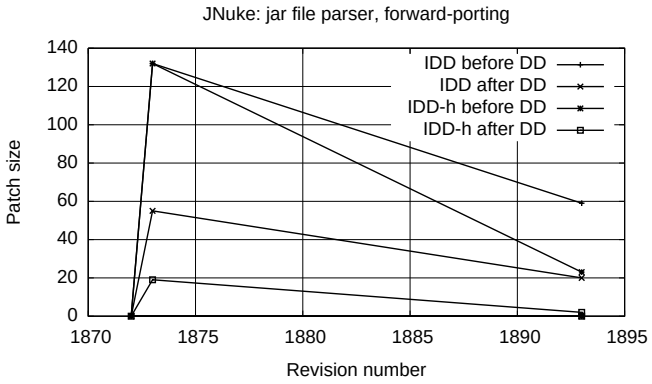

Fig. 12. Result of using IDD on JNuke: jar file parser.

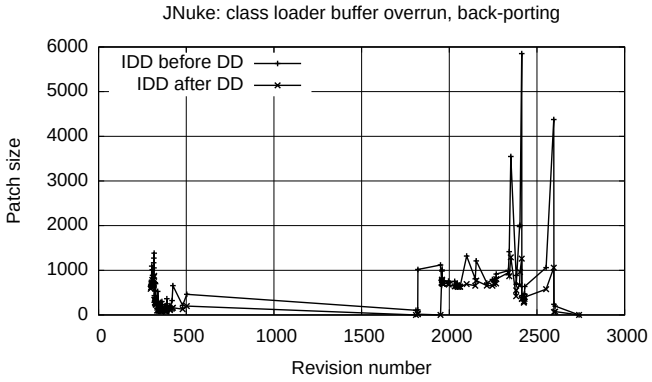

Fig. 13. Result of using IDD on JNuke: buffer overrun in class loader.

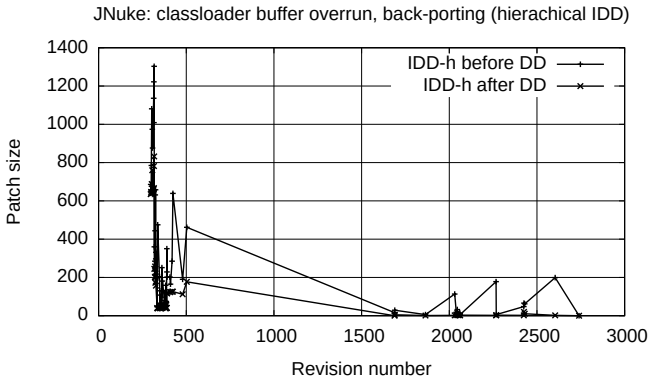

Fig. 14. Result of using hierarchical IDD on JNuke: class loader.

the failure could be located using traditional debugging techniques and memory analysis tools [30]. Because the defect was not discovered for a very long time, it was still considered to be an interesting candidate for an experiment.

When considering the plots in Figures 13 and 14, two "phases" of the experiment can be picked out: In an initial phase (revisions 2604–1950), hierarchical DD produces a very small patch, whereas the line-based version produces patches with sizes up to 1000 lines and more. In a second phase (revision 503 and lower), both approaches fare about the same. The experiment was stopped at revision 295, because introduction of the formatting tool `indent` affected the entire code base at this point, generating a patch of about 25,000 lines, which was too large for DD to handle.

The fact that the code base had undergone several years of evolution manifested itself in a compilation problem on slightly older revisions. Revision 2740, where the defect was encountered, could be run and tested normally with a newer version of gcc than the one that was used at development time. In revisions 2604 and lower, a code artifact that is no longer tolerated by newer versions of gcc prevented successful compilation. In this case, the DD part of our algorithm therefore did not target functionality, but syntax. The change responsible for the compilation error consists of only two lines. While hierarchical DD could efficiently isolate the file and change in question, the non-hierarchical version did not hit reasonably small change sets that were also syntactically valid. As a consequence of this, hundreds of lines of unnecessary changes were kept and moved back throughout revisions 2604–1950. At that point, application of the patch resulted in another invocation of DD. Because the "offending" code in question was no longer part of the repository at that point, the patch to compile on a newer gcc had become entirely obsolete. The first DD invocation that attempted to use an empty change set therefore succeeded, eliminating the entire change set. The same effect was encountered on the hierarchical run at a different revision, and again later for a similar syntactic problem. As the performance of DD mostly depends on the locality of a change set, the nature of the problem (compilation or test failure) is immaterial. Hierarchical DD was therefore much better for this initial problem.

When going back to earlier revisions (503 and lower), though, actual functional changes had to be back-ported in order to keep the necessary infrastructure to execute the unit test in question. In that part of the experiment, both variants of DD fared about the same. The hierarchical variant was slightly better at first, but the "flat" DD generated a slightly smaller patch in the last few revisions.

This may seem counter-intuitive at first. However, one should be reminded that the given hierarchical implementation uses an approximation of the change hierarchy, and that changes may be interspersed throughout files. If a file affected by a patch is unchanged, a hierarchical implementation may reach a local fix point (for that file) fairly quickly. This is because once a file and a change set are stable, a hierarchical bisection will always analyze the same subsets of changes. In contrast to that, the state space bisection of non-hierarchical DD sometimes includes parts of several files. State space bisection starts across files, working its way down to smaller intervals. These intervals hit slightly different sets of lines within a file in each invocation, because the size of the overall patch set changes from one revision to another, even if the part affecting one file is constant. This fact sometimes allows the non-hierarchical variant to reduce the change set more than the given hierarchical implementation can.

This behavior does not indicate that the hierarchical algorithm per se is not ideal. However, its implementation has to

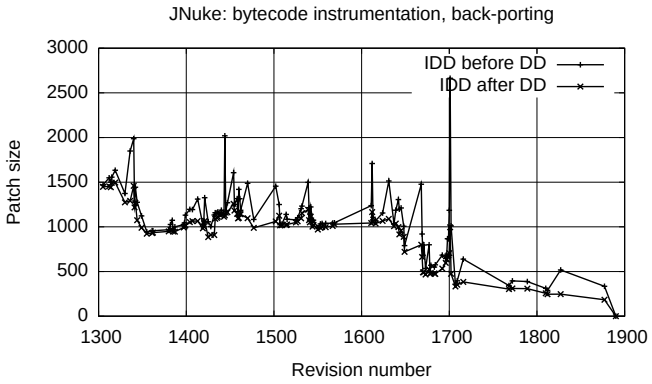

Fig. 15. Result of backward iteration on JNuke: bytecode instrumentation.

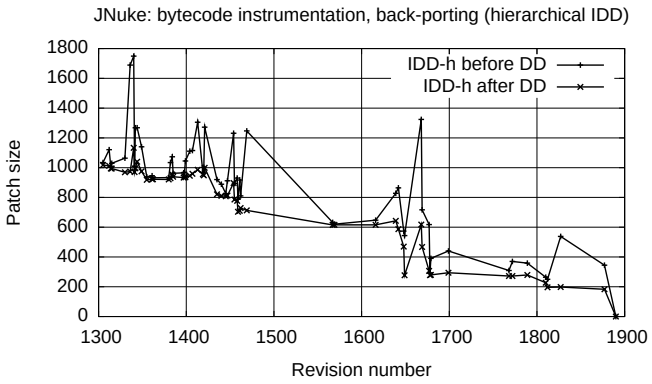

Fig. 16. Hierarchical IDD, backward it. on JNuke: bytecode instrumentation.

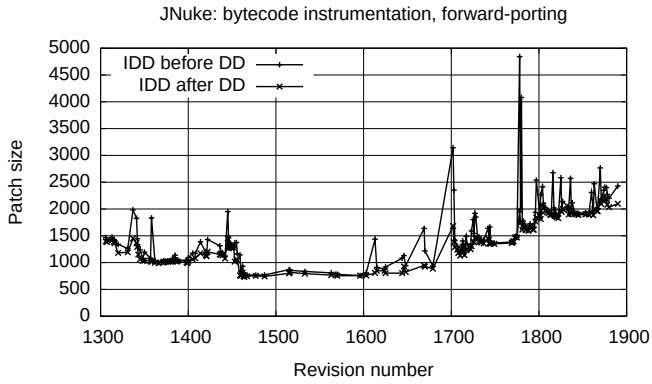

Fig. 17. Result of forward iteration on JNuke: bytecode instrumentation.

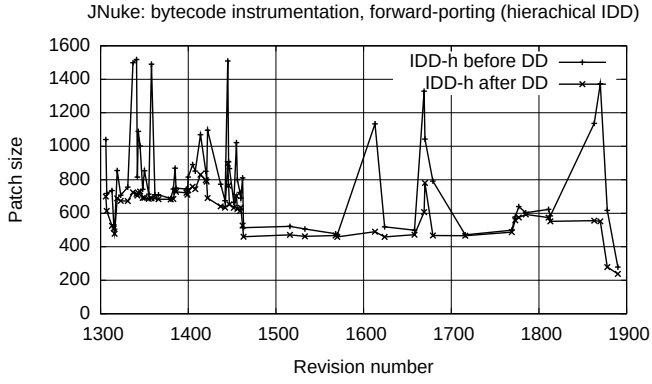

Fig. 18. Hierarchical IDD, forward it. on JNuke: bytecode instrumentation.

fully reflect dependencies of changes on a semantic level, in a given programming language, in order to be most effective.

6.4.3 Bytecode instrumentation

The last experiment targets a defect that was found through code inspection, but not verified by a unit test until much later in the project. If the defect had not been found by inspection, it may have persisted in the system for over 600 revisions, occurring over the time span of about six months. After that time period, it is likely that the localization of the defect would have taken a significant human effort.

For this experiment, the defect that was introduced in revision 1306 and repaired immediately after, was applied to revision 1890, which was the first one containing a unit test that confirms the problem.

The results are presented as two plots here, to make it easier to read the information. Figures 15 and 16 show the back-porting phase of IDD. Various refactorings and other code changes required adaptation of older revisions to the newer test case. Both versions of DD produced a large change set, resulting in more and more frequent patch conflicts, which in turn required more calls to DD. The graph shows how this problem was exacerbated over time, with data points that correspond to DD calls clustering on the left, on older revisions. The patch set to be pack-ported reached more than 1,000 lines at the time when the good revision (1305) was hit.

Forward porting (see Figures 17 and 18) allowed for a successive reduction of the patch size. One reason for the immediate drop in patch size in the hierarchical version of DD is that certain guards against possibly unwanted changes can be dropped for forward-porting. For instance, memory corruption in C programs may occur after a patch is reduced. In the backward porting phase, this has to be avoided because memory corruption may prevent a test case from ever succeeding. When forward porting a patch, certain memory problems, such as the use of initialized variables, are permissible if they are a consequence of incompletely eliminated spurious changes [4].

6.5 PDF library (itext)

Project “itext” is a PDF library written in Java, which can generate PDFs and also display documents in that format. While most bugs reported online either had not automated way of being reproduced, or concerned the graphical user interface of the tool, one bug relating to an incorrect handling of UTF-8 encodings was used as a case study (bug 2792227). It is one of a few cases where a defect was taken by a programmer and encoded as an automated unit test, intended to

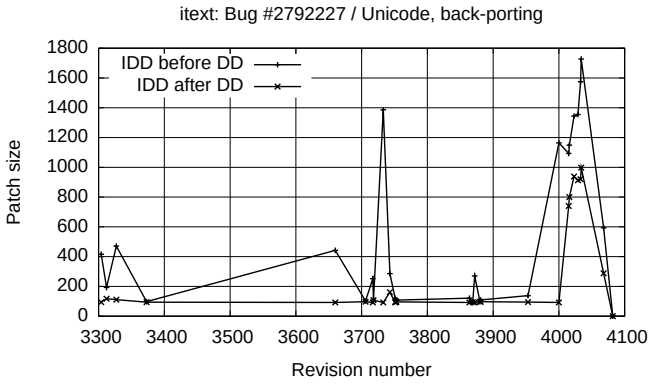

Fig. 19. Result of using IDD on itext: PDF library

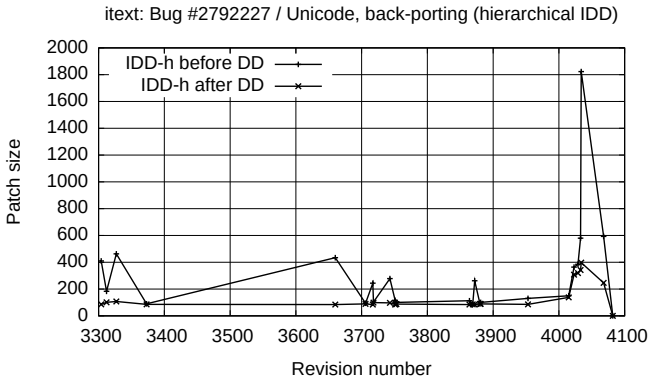

Fig. 20. Result of using hierarchical IDD on itext: PDF library

both automate testing and serve as an example for future unit tests.

IDD was not able to find a working version. Analysis was hampered by the fact that the test driver was set up to abort upon the first test failure. This required IDD to repair a number of other tests when going back to older revisions. As Figures 19 and 20 show, relatively large patches were generated between revisions 4079 and 4000. At older revisions, the problematic unit tests were no longer present in the repository, and only a small patch containing the test code of the new test had to be back-ported. This allowed IDD to progress efficiently and analyze over 800 older revisions, down to revision 3174. After that, the subversion (`svn`) client was not able to proceed, because a directory had been added and deleted again over the course of time, causing an interference between patch generation (through `svn diff`) and a subsequent use of that patch. At the time of writing, the bug has not been repaired yet, after over one month of being encoded as an automated test.

6.6 Hyper-SQL database (hsqldb)

Hsqldb is a popular light-weight database written from scratch in Java. This makes it a suitable candidate for IDD, as there are no dependencies on third-party libraries. Furthermore, a non-persistent in-memory database mode is supported, allowing for easily automated testing. Three issues from the bug database were chosen. Two were encoded as SQL scripts that execute the test, and one was implemented as a small application.

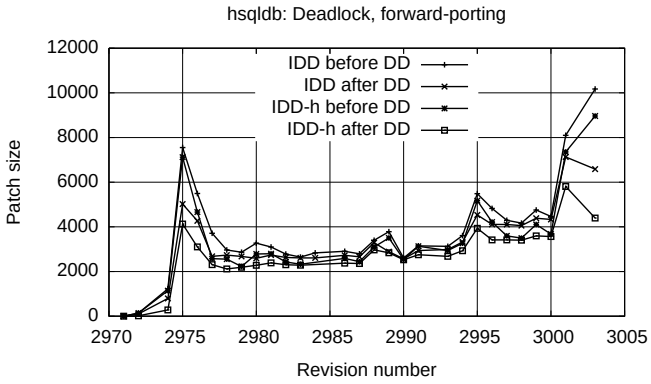

Fig. 21. Result of using IDD on hsqldb: Read lock not released

6.6.1 Backward compatibility problem

The first bug, number 2795205, refers to a change that breaks backward compatibility. As various applications already rely on the old (non-standard) behavior, the developers changed the query parser to accommodate for this. IDD was unable to find a working older revision, but the given test application could be applied to all existing older versions of hsqldb without having to back-port any functionality.

6.6.2 Read lock not released

The second case (bug 2887855) refers to a case where a read lock is not released, causing subsequent queries on the same table to block indefinitely. The problem was a symptom of a design change in the query engine, confirmed by the developer to be “currently normal”. With work on the engine continuing, the bug was addressed as part of a subsequent larger change.

This bug was known to be a regression from the previous stable release, and IDD was able to find the exact change in question. The change set found was relatively small, and allows a human to understand the problem. Unfortunately, it was not suitable to forward-port that change set to a newer version. Frequent refactorings interfered with the patch mechanism, causing very large patch sets to be forward-ported where a manual intervention could have ignored certain changes such as class renamings easily (see Figure 21). The problem affects both the non-hierarchical and the hierarchical delta debugging approaches. The bug fix was forward-ported across 32 versions in this case, after which it became clear that the resulting patch would be very large, and complete forward-porting would take a large amount of time.

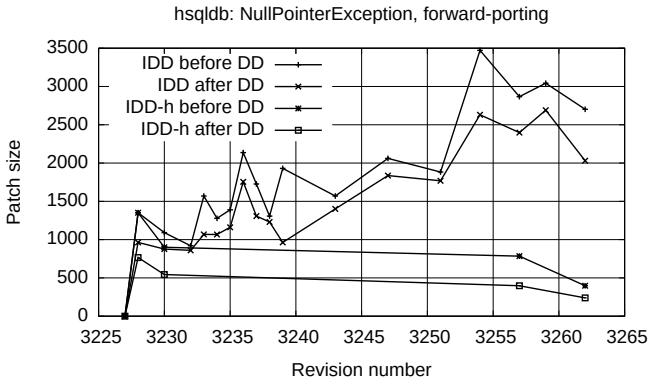

Fig. 22. Result of using IDD on hsqldb: NullPointerException

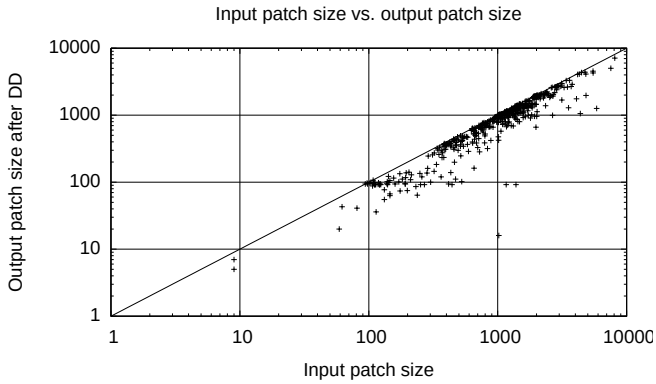

Fig. 23. Effectiveness of DD

6.6.3 NullPointerException

The last case, bug 2887964, was easily diagnosed, but still included as a case study because it was also a known regression. As expected, a good version could be found. For the hierarchical approach, the resulting synthesized bug fix could be forward ported efficiently, resulting in a relatively small patch. Only a small part of the synthesized bug fix actually coincides with the change by the developer. The non-hierarchical approach ended up being much slower and less precise than the hierarchical one, as shown in Figure 22.

6.7 Reasons for iteration and DD failures

It is interesting to look at the reasons why the backward or forward iteration could not proceed, and similarly, at what kinds of problems were encountered during delta debugging.

Table 2 shows the reason why a given non-empty patch could not be applied to a given revision. Four different categories of problems are listed:

1. Patch: The `patch` tool reported a conflict, so the patch could not be applied. When using the `patch` tool outside the context of this project, a human has to resolve a patch conflict. As described in Section 5, a patch failure can be regarded as an overall failure in the full patch-build-test cycle, so delta debugging can also be used to treat such problems.
2. Build: The given version could not be compiled successfully.
3. Test error: The test failed, or, in backward iteration, produced an outcome different from the existing failure. Such a different erroneous outcome may be a different output or a premature termination.
4. Timeout: The patched version exceeded its time limit, usually because the patch generated an infinite loop.

Table 2 shows that patch failures make up at least 64 % of all cases why the iterative part of IDD fails. While patch problems can be caused by semantic changes in the program, a lot of patch failures encountered in this work were due to minor syntactic changes. As the patch format depends on precise context information in the affected part of the code, and in the immediate vicinity of a change set, it is rather fragile. It often happens that changes unrelated to the intended semantic change prevent automatic patching. A simple renaming of a variable often prevents a patch from being applied successfully. This calls for a next-generation `patch` tool, as outlined in Section 9.

Finally, on the level of delta debugging itself, reasons for a failure of individual DD steps also vary. Table 3 gives an overview of why a subset of a change set produces an invalid program.[2] A generated program mutation may not compile successfully, cause an assertion failure, a timeout, use corrupted memory, or cause another, unclassified failure.[3] Note that memory corruption sometimes caused a C or C++ program to crash in a certain way that it could not be classified as such by our evaluation scripts, so some instances of memory corruption are listed as “other”.

An interesting observation is that build failures are much more common for a Java program. Up to 97 % of all generated mutations for Java programs were rejected by the compiler, while for C and C++, this number was often lower than 90 %, and as low as 43 % for the JNuke/jar file parser case.[4] In general, the hierarchical approach produces fewer build failures, as some syntactically invalid versions are never generated. The fact that build failures make up about 90 % of failed DD iterations overall shows that a more detailed syntactic analysis prior to the bisection search has the potential of improving the performance by an order of magnitude.

The average patch size reduction, shown in the final column, varies between projects, but is always better for hierarchical DD. As each successful test requires fewer successive DD steps, it is clear that the average patch size reduction is

[2] Note that the numbers do not add up to the final counts in table 4, as successful runs are not included here.

[3] Memory corruption may manifest itself in a segmentation fault for C or C++ programs, or an array bounds exception in a Java program.

[4] The reason for more frequent Java compilation errors is that the Java compiler performs method-local data flow analysis and requires that return values and values of each local variable always be defined. This generates compilation error for programs for which the equivalent code would be accepted by a C or C++ compiler. For our work, this “fail fast” behavior was beneficial for the overall accuracy and performance, as no Java programs using uninitialized memory were created.

Table 2. Reasons for failed backward and forward iterations.

Project name		Failure type			
		patch	build	test error	timeout
Uncrustify	IDD	51	7	5	
	IDD-h	43	6	4	
Uncrustify (other cases)	IDD and IDD-h	no need to back-port changes			
Java PathFinder (infinite loop)	IDD	14	1		1
	IDD-h	10			1
Java PathFinder (other cases)	IDD and IDD-h	no need to back-port changes			
JNuke (jar file parser)	IDD	1			
	IDD-h	1			
JNuke (class loader)	IDD	80	19	6	
	IDD-h	49	16		1
JNuke (bytecode instrumentation)	IDD	233	49	11	
	IDD-h	80	34	8	1
itext (Unicode problem)	IDD	15	7	6	
	IDD-h	12	7	6	
hsqldb (backward compatibility)	IDD and IDD-h	no need to back-port changes			
hsqldb (deadlock)	IDD	24	4		
	IDD-h	20	7		
hsqldb (NullPointerException)	IDD	15	1		
	IDD-h	2			

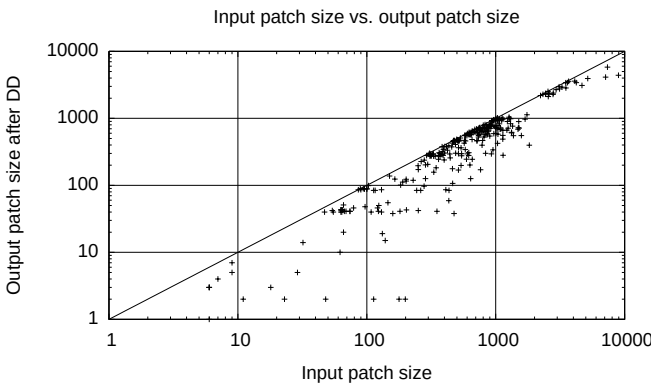

Fig. 24. Effectiveness of hierarchical DD

inversely correlated to the number of times a build/test run had failed in DD (see Table 3). However, the effectiveness of DD is not correlated to the size of the input patch set. Small input patches may be easily reducible due to the simplicity of a change set, or hard to reduce further because irrelevant changes have already been removed at a previous iteration. The same holds for large patches. Figures 23 and 24 show two scatter plots of input patch sizes (before the invocation of DD at each step) and output patch sizes (after DD), for normal and hierarchical DD. A point placed further away from the diagonal corresponds to a more effective patch size reduction. As can be seen, both small and large change sets can sometimes be significantly reduced, and sometimes hardly reduced at all. It should also be noted that usually, a large part of the input patch corresponds to the output of a previous DD

Table 3. Reasons for failed build/test runs in DD, and effectiveness of DD.

Project name		Failure type					Patch size reduction
		build	assertion	timeout	memory	other	(average)
Uncrustify	IDD	43781		222	451	4666	20 %
	IDD-h	31868		81	472	4091	23 %
Java PathFinder (infinite loop)	IDD	53516			271	964	23 %
	IDD-h	12660			74	796	50 %
JNuke (jar file parser)	IDD	192			17	55	60 %
	IDD-h	35			10	35	86 %
JNuke (class loader)	IDD	88994	513	1		68	39 %
	IDD-h	23004	192	72	2	99	44 %
JNuke (bytecode instrumentation)	IDD	578064	12577	1755	7660	24744	12 %
	IDD-h	119209	5269	1149	3562	13934	20 %
itext (Unicode problem)	IDD	10881		2		89	49 %
	IDD-h	2934				101	53 %
hsqldb (deadlock)	IDD	191287			130	163	14 %
	IDD-h	154988			69	165	19 %
hsqldb (NullPointerException)	IDD	106971		18	1	537	19 %
	IDD-h	5533				61	43 %

invocation. That part of the change set can usually not be reduced significantly anymore.

6.8 Evaluation

Table 4 summarizes the outcome of all experiments. For both the linear and the hierarchical DD algorithm, the table shows the size of the patch required to back-port the test, the initial patch repairing an old version, and the final patch for the current version of the application. After that, the size of a bug fix by a human is shown, where applicable. Columns with no data indicate that no repair action could be synthesized. Finally, the number of calls to the DD procedure, and the number of build/test runs in DD, gives an impression of the overall complexity and run-time behavior. DD invocations do not include the final fix-point iteration, which adds 2–11 test cycles and usually has a minor impact on the overall number of iterations.[5]

The fix-point iterations typically reduce the patch size by about 10%, as shown by Figure 25. When hierarchical DD was used on the Jar file parser patch for JNuke, the patch was reduced from 8 to 2 lines, appearing as a very large relative change in Figure 25. From these results, one may decide to forgo the fix-point iteration, if the given patch is already providing a good explanation or fix of a defect.

From Table 4, it can be seen that in 5 out of 13 versions, a "good" version passing the test could be found. Forward-porting always succeeds, but in most cases produces a patch that is much larger than one written by a human. In all cases but one (JNuke/jarfile), the synthesized patch would have repaired the test in question, but caused problems with other parts of the program. The overall success rate of about one tenth is consistent with other automated repair techniques on large programs [13] and show that automated repair actions have a potential for future enhancements.

[5] We made the design choice of only having a fix-point iteration for patch size reduction at the end of the entire IDD process. The reason for this was that during the iterative phase, patch failures often occurred for parts of the code that were not affected in the previous DD invocation. In these cases, the next iteration would also aid the convergence of patch sets generated in previous DD runs, where recursive dependencies prevented a removal of a change in a single step.

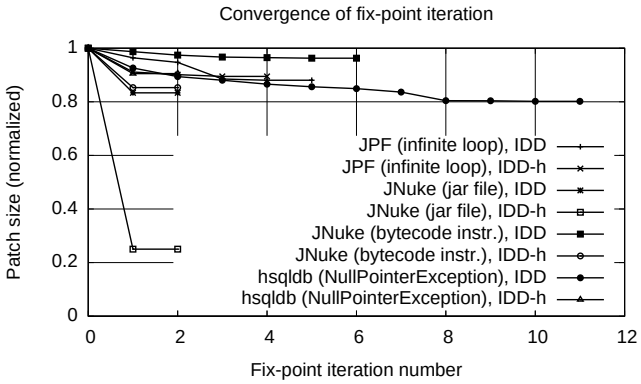

Fig. 25. Relative patch size reduction during fix point iteration

Unfortunately, it is difficult to classify the quality of the generated patch: The lack of an overlap between the generated patch and the actual bug fix (such as in the first JPF and the last two hsqldb case studies) does not mean that the automated fix is not useful. The automated patch attempts to repair the program and maintain the status quo. In many cases, the human-induced changes include adaptations to other requirements, so a meaningful comparison is not possible.

As can be seen, the hierarchical version of IDD always produces significantly smaller patches, although the difference varies. Furthermore, the hierarchical algorithm requires between 1.3 and 18 times fewer build and test runs, a result that is consistent with similar hierarchical approaches [45].

In the given experiments, the number of test cycles within DD would have been prohibitively large in practice in all cases except for the jar file parser of JNuke. The complexity of our DD implementation is linear in the size of failed build/test runs, suggesting a fairly good scalability of DD itself. However, when patches are generated repeatedly, as in IDD, the overall complexity is not linear or quadratic: Larger patches tend to be more vulnerable to patch conflicts in later revisions. Therefore, once a patch has accumulated a number of features such as refactorings, name changes, etc., the `patch` command tends to fail more often. As a result of this, the DD subroutine is used more often, which tends to lead to even larger patches. At some point, the problem exacerbates itself to a level where IDD takes too much time to finish. A better change set elimination algorithm improves the situation, which shows in the better performance of the hierarchical DD implementation. Furthermore, the time to compile a project is also significant. In our experiments, each build/test cycle took up to 30 seconds. Longer IDD runs can take several days, or even months.[6] In practice, the likelihood of a success is highest for quickly detected defects, so DD would typically be run for a few hours or days before it would be aborted.

[6] Intelligent caching of compiled artifacts could reduce this time by at least a factor of 10. However, we did not find a system that supported all the programming languages and revision control systems in question.

7 Related Work

Several solutions have been proposed to isolate defects, and to automatically repair programs. Automated techniques range from program analysis, and statistical methods, to methods analyzing the run-time behavior of the program.

7.1 Program analysis

Static analysis techniques, such as *slicing,* can reduce the amount of code that has to be analyzed [25,41,44]. Program slicing uses a predicate and computes the subset of a program that determines the outcome of that predicate. Like DD, slicing attempts to compute a minimal set that has to be analyzed. The major difference is that the full program, and not just a change set, is analyzed with slicing.

Program slicing has been conceived as a static technique, but can also be used at run-time. A simple yet accurate and effective way of slicing is the usage of coverage information, to show which code is executed by which unit tests [20]. Test coverage and failure information can also be gathered in a distributed way to reduce the overhead on each execution [22]. The coverage of a single test shows if a statement has been executed or not, and provides binary information. Such information is precise but may result in a candidate set that is rather large.

7.2 Statistical techniques

More recently, several quantitative, statistical techniques have been proposed. While they do not work with 100 % precision, they can typically narrow down the set of candidate locations further than qualitative approaches.

When considering the program structure, a set of program edits can be ranked based on structural heuristics [32]. Conversely, when it comes to test coverage, many approaches have been proposed that consider a number of passing and failing tests on a given program. They are known as *spectrum-based* techniques and compare the coverage of tests with different outcomes [2,10,19,34]. The key difference to our approach is that spectrum-based techniques require several passing and failing tests to narrow down the cause of an error. They are thus more geared towards an early selection of likely fault locations, such as a ranking of modules or classes, and less towards pin-pointing a fault in a particular test case. Attempts were made to use a more elaborate model of test executions to improve the precision of spectrum-based location. Such so-called "model-based" diagnosis approaches suffer from scalability problems, though [27,42].

Until recently, spectrum-based localization techniques were applicable to single faults only, but recent improvements made them applicable to multiple faults [1]. Coverage-based techniques consider multiple tests against one version of the program; we consider one test against multiple variations of the program. While coverage-based techniques are not applicable to a single test, one may consider them as a heuristic to

Table 4. Summary of experiments.

Project name	Size of back-ported test		Size of initial patch		Size of final patch			Number of calls to DD		Number of build/test runs	
	IDD	IDD-h	IDD	IDD-h	IDD	IDD-h	Human	IDD	IDD-h	IDD	IDD-h
Uncrustify (comments)	667	567	–	–	–	–	50	63	53	52483	39170
Uncrustify (C++ functions)	0	0	–	–	–	–	37	0	0	0	0
Uncrustify (conditionals)	0	0	–	–	–	–	76	0	0	0	0
JPF (infinite loop)	0	0	344	169	1133	272	255	16	11	57140	15013
JPF (cyclic deadlock)	0	0	–	–	–	–	1344	0	0	0	0
JPF (lock counter)	0	0	–	–	–	–	555	0	0	0	0
JNuke (jar file parser)	0	0	55	19	20	2	2	1	1	335	117
JNuke (class loader)	586	635	–	–	–	–	2	106	67	96870	26141
JNuke (bytecode instr.)	1448	1017	1395	700	2098	238	4	293	124	647490	148619
itext (Unicode problem)	94	86	–	–	–	–	–	27	24	12678	3597
hsqldb (compatibility)	0	0	–	–	–	–	36	0	0	0	0
hsqldb (deadlock)	0	0	102	15	>6000	>4000	411	28	27	200579	164472
hsqldb (NullPointerException)	0	0	964	766	2030	239	107	17	4	110469	6012

prioritize other types of fault localization. Multiple tests may be generated from an initial seed by "fuzzing" the input, randomizing it to alter the test outcome [35,36].

7.3 Behavior-based defect analysis

In software, invariants and internal consistency checks are often specified as rules, encoded as assertions. In the absence of such specifications, rules may be generated by observing the *behavior* of program executions [15]. Rule sets generated in this way can then be used to generate test cases for verification [12], or be verified using existing tests [14]. In this way, inconsistencies in the implementation can be demonstrated by test executions. Recent work has combined the observation of tests against such rules, and used coverage information on these rules to generate possible repair actions for rule violations [13]. Other work considers repair actions directly on data structures in memory, to repair broken invariants and then generate program code that implements such a change [24].

Unlike our approach, which takes existing code as candidates for possible repair actions, techniques that synthesizes new code are also applicable for defects that are not regressions. The weakness of current *synthesis* approaches is that the types of repair actions that can be generated is still limited. Another way to overcome the limitation of strictly history-based approaches is to allow for mutations in repair actions, using genetic programming [40]. The combination of generating changes based on rules, with mining previous revisions for potential fixes, constitutes future work.

7.4 Delta Debugging

Delta Debugging, as introduced by Zeller [43], reduces an input or change set by applying the state space bisection technique described in Section 3 of this paper. Delta debugging in its original version considers an input (or change set) to be unstructured. It can therefore be applied to any kind of non-binary data, including program source code. One project applied delta debugging to a representation of thread context switches of the execution of a concurrent program [11]. The search strategy of DD remained unchanged, though. This also applies to work that combines DD with program slicing [18].

7.5 Domain-specific implementations of hierarchical DD

Many kinds of data, such as XML data or program source code, are not structured into disconnected, independent lines. The original DD algorithm is not necessarily very precise and efficient in such cases. Hierarchical Delta Debugging [28]

breaks down change sets into hierarchical groups, thus improving the effectiveness of the algorithm.

Algorithms for hierarchical debugging for data other than XML have been developed in the meantime. A domain-specific DD algorithm exploits the structure of the input data to reduce the number of invalid mutations generated, and usually leads to much better convergence by taking advantage of the hierarchical structure of the input.

In general, HDD tools operate on a model that is more abstract than the underlying data. The model allows for direct modification, but typically has to be converted back into the original format when evaluating the result of a modification. The experiments in this paper have shown that the quality of the underlying fault localization algorithm is of major importance for IDD. Therefore, existing hierarchical DD algorithms are surveyed in more depth here.

7.5.1 Original HDD algorithm

The original HDD algorithm for XML [28] operates on an abstract syntax tree (AST). Concrete tool implementations and experiments have been carried out for C programs and XML data files. The C version builds on a parser for C that has been extended to allow a modification of the resulting AST. The XML version directly exploits the strict hierarchical structure of XML, in the form of subtrees, nodes, attributes, and finally characters. This work is similar to the approach used in this paper, in that the nesting of data is exploited, in order to enhance the search. Non-hierarchical consistency constraints, such as the fact that a variable used (in statically typed program code) must be defined, were not exploited.

7.5.2 SMT formulae

The satisfiability of logical formulae is an everyday problem when analyzing models of digital circuits [8]. Formulae are typically encoded as a hierarchical structure, and tools operating on them accept only well-formed input. The header of the input includes redundancy, such as the number of elements to follow. A direct implementation of DD is not applicable in such a case, because a direct mutation of data leaves the header and the payload inconsistent. A specialized implementation of delta debugging for Satisfiability Modulo Theories (SMT) solvers has been proposed, which adapts the modification of data such that the generated data is still valid [38,9]. In this case, consistency conditions include the graph structure, the type of nodes, and simplification rules that eliminate redundancy.

7.5.3 Aspect-oriented Java programs

Dependencies in program source code, and changes therein, are complex. In object-oriented systems, changes can be structured into *atomic changes* [33]. Change sets follow the structure of object-oriented code, from classes, methods, and fields, down to the body of a method. The granularity stops at method level and thus is fairly coarse. The Celadon tool extends the set of atomic changes to include changes in aspect-oriented software, and uses such a representation of changes for delta debugging [44].

Celadon introduces ranking as a part of *three-phase* delta debugging [45]. Phase 1 constructs an internal representation of changes. Phase 2, which does not exist in other tools, ranks changes according to code coverage information; such a ranking is applicable when multiple tests are analyzed. Our work focuses on analyzing one test, so a ranking is not applicable here. In Celadon, coverage information is used to determine which changes affect how many unit tests [45]. Finally, phase 3 corresponds to the actual invocation of DD. The improvement in phase 2 in Celadon is orthogonal to our contribution of applying DD iteratively to a history of changes, and could therefore be combined in future work.

7.5.4 Differences to our tool

Existing work that addresses DD on program source code [28, 45] is tied to a particular programming language. Changes on a higher level can be represented and structured accurately. Our work is language-independent and uses the hierarchy of patch files for a finer-grained analysis. As a drawback, our work suffers from the fact that patch structure may not reflect the hierarchy of underlying program changes accurately. A versatile, fine-grained implementation is subject of future work.

Table 5 shows an overview of the HDD algorithms considered here, in the order in which they have been discussed in this section.

8 Conclusion

Delta debugging automates the task of identifying minimal change sets. However, it requires a correct version, which may not be known when a new defect is discovered. If an old version that passes a given test exists, then a systematic evaluation of older versions may discover it. Sometimes, it is necessary to apply changes of newer versions to older ones in order to allow a newer test to execute on an old version. Iterative delta debugging automates 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. After that, the resulting patch may be forward-ported to the current version, using the same algorithm. In this way, automatic synthesis of potential software repairs is sometimes possible. Our methodology works successfully on some large examples, but depends on the precision of the delta debugging implementation used, and requires much time for large change sets.

9 Future Work

Iterative Delta Debugging works in conjunction with delta debugging, but requires a high-performance, high-precision implementation of DD in order to be successful.

Table 5. Overview of hierarchical delta debugging implementations.

Algorithm	Domain	Data format	Model format
HDD	C source code or XML data	C source code or XML data	Abstract syntax tree
HDD for SMT solvers	SMT formulae	SMT solver input	Tree with consistency constraints
Celadon	Aspect-oriented Java programs	Source code	Atomic changes
Patch-based HDD	Text files	Patch	Patch file hierarchy

The current tool chain has been written as a prototype, lacking several possible optimizations that could be implemented. For instance, DD generates and compiles variations of one revision by checking out a fresh copy of the necessary files and compiling them from scratch in each iteration. Whenever the build configuration does not change, most of these steps can be cached. Recent revision control systems like `git` [37] already cache the entire repository very efficiently. Their design also allows for off-line commits, when no central server is available, and encourages frequent small changes. The resulting small change sets should make IDD more effective.

Incremental compilation would also speed up the process, but requires accurate build information. In our experiments, the changes in the code were accompanied by changes (and bug fixes) in the build configuration itself. Furthermore, some files are compiled and executed prior to the actual build process, to test for the behavior of the given platform. Due to this, we chose a conservative implementation that always recompiles the entire application from scratch. A more precise analysis could not only compile a program incrementally, but also deduce if a given unit test is affected by a particular change. As an augmentation to delta debugging, code analysis such as coverage [20] or slicing [25] could a priori eliminate the need to verify variations where too much code is removed. Code coverage and dynamic slicing on a known "good" version can help to identify which statements are crucial for a test to pass. Such statements should not be removed in the DD process. For C programs, static analysis against memory corruption could further automate the suppression of programs that do not generate consistent results.

The direction of applying hierarchical DD is definitely promising, and delivers faster and better results than standard DD. However, the hierarchy of the patch file structure does not exactly mirror the hierarchy of software source code. Tools that analyze the syntax and call graph of programs [45], or tools that represent software source code in XML format [17, 26] could be used to extract a hierarchical representation of programming constructs. If XML data is used, then the question of how to generate an efficient and expedient difference representation is still open. Tools having their own format exist [23], and may be used in further case studies. Eventually, such work may lead to a successor of the Unix `patch` tool, which would be useful in a much wider context than debugging alone.

References

1. R. Abreu, P. Zoeteweij, and A. van Gemund. Spectrum-based multiple fault localization. In *Proc. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, pages 88–102, Auckland, New Zealand, 2009.
2. S. Ali, J. Andrews, T. Dhandapani, and W. Wang. Evaluating the accuracy of fault localization techniques. In *Proc. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, pages 76–87, Auckland, New Zealand, 2009.
3. C. Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.
4. C. Artho. Iterative delta debugging. In *Proc. 4th Haifa Verification Conference (HVC 2008)*, Haifa, Israel, 2008.
5. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
6. C. Artho, E. Shibayama, and S. Honiden. Iterative delta debugging. In *19th IFIP Int. Conf. on Testing of Communicating Systems (TESTCOM 2007)*, Tallinn, Estonia, 2007. Poster/Tools session.
7. K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., 2000.
8. A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.
9. R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *Proc. 7th Int. Workshop on Satisfiability Modulo Theories (SMT' 09)*, Montreal, Canada, 2009.
10. T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proc. 31nd Int. Conf. on Software Engineering (ICSE 2009)*, pages 34–44, Vancouver, Canada, 2009. ACM SIGSOFT and IEEE, IEEE.
11. J. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proc. ACM/SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 210–220, Roma, Italy, 2002. ACM.
12. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on*

Software Engineering and Methodology (TOSEM), 17(2):1–37, 2008.
13. V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proc. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, pages 550–554, Auckland, New Zealand, 2009.
14. D. Engler, D. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.
15. M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
16. B. Gardner. Uncrustify code beautifier, 2009. `http://uncrustify.sourceforge.net/`.
17. K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understanding tools for embedded software in C. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 336–345, Washington, USA, 2004. IEEE Computer Society.
18. N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proc. 20th Int. Conf. on Automated Software Engineering (ASE 2005)*, pages 263–272, Long Beach, USA, 2005. ACM.
19. J. Jones and M. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proc. 20th Int. Conf. on Automated Software Engineering (ASE 2005)*, pages 273–282, Long Beach, USA, 2005. ACM.
20. J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. 24th Int. Conf. on Software Engineering (ICSE 2002)*, pages 467–477, Orlando, USA, 2002. ACM.
21. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
22. B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.
23. T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *Proc. 2006 ACM symposium on Document engineering (DocEng 2006)*, pages 75–84, New York, USA, 2006. ACM.
24. M. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *Proc. 24th Int. Conf. on Automated Software Engineering (ASE 2009)*, pages 620–624, Auckland, New Zealand, 2009.
25. R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proc. 12th Int. Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 63–72, Newport Beach, USA, 2004. ACM.
26. K. Maruyama and S. Yamamoto. A tool platform using an XML representation of source code information. *IEICE – Trans. Inf. Syst.*, E89-D(7):2214–2222, 2006.
27. W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Proc. 23rd Int. Conf. on Automated Software Engineering (ASE 2008)*, pages 128–137, L'Aquila, Italy, 2008. IEEE Computer Society.
28. G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proc. 28th Int. Conf. on Software Engineering (ICSE 2006)*, pages 142–151, Shanghai, China, 2006. ACM Press.
29. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
30. N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. 3rd Int. Workshop on Run-time Verification (RV 2003)*, volume 89 of *ENTCS*, pages 22–43, Boulder, USA, 2003. Elsevier.
31. D. Peled. *Software Reliability Methods*. Springer, 2001.
32. X. Ren and B. Ryder. Heuristic ranking of Java program edits for fault localization. In *Proc. ACM/SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 239–249, London, UK, 2007. ACM.
33. X. Ren, B. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for Java programs. In *Proc. 27th Int. Conf. on Software Engineering (ICSE 2005)*, pages 664–665, St. Louis, USA, 2005. ACM.
34. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th Int. Conf. on Automated Software Engineering (ASE 2003)*, page 30, Montreal, Canada, 2003. IEEE Computer Society.
35. M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
36. A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, USA, 2008.
37. L. Torvalds and C. Hamano. git-bisect(1) manual page, 2009. `http://kernel.org/pub/software/scm/git/docs/git-bisect.html`.
38. A. Vida. Random test case generation and delta debugging for bit-vector logic with arrays. Master's thesis, Johannes Kepler University, Linz, 2008.
39. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
40. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. 31nd Int. Conf. on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009. IEEE Computer Society.
41. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
42. F. Wotawa, M. Stumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In *Proc. 15th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2002)*, pages 746–757, London, UK, 2002. Springer-Verlag.
43. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering*, 28(2):183–200, 2002.
44. S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Celadon: a change impact analysis tool for aspect-oriented programs. In *ICSE Companion 2008: Companion of the 30th Int. Conf. on Software Engineering*, pages 913–914, Leipzig, Germany, 2008. ACM.
45. S. Zhang, Y. Lin, Z. Gu, and J. Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *Proc. 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2008)*, pages 77–83, Atlanta, Georgia, 2008. ACM.