Minimization of Randomized Unit Test Cases

Yong Lei and James H. Andrews
Department of Computer Science
University of Western Ontario
London, Ontario, CANADA N6A 5B7
Email: {leiyong,andrews} (at) csd.uwo.ca

***Abstract*— We describe a framework for randomized unit testing, and give empirical evidence that generating unit test cases randomly and then minimizing the failing test cases results in significant benefits. Randomized generation of unit test cases (sequences of method calls) has been shown to allow high coverage and to be highly effective. However, failing test cases, if found, are often very long sequences of method calls. We show that Zeller and Hildebrandt's test case minimization algorithm significantly reduces the length of these sequences. We study the resulting benefits qualitatively and quantitatively, via a case study on found open-source data structures and an experiment on lab-built data structures.**

I. Introduction

Software testing consists of three main activities: selecting test inputs, running the inputs on the software under test, and evaluating the correctness of the outputs. The first and third of these activities can be labour-intensive and error-prone. If testing results in the discovery of failing test cases (test inputs that result in incorrect outputs), then the failing test cases must be used as input to the process of debugging. The quality of those failing test cases affects how useful they are to debugging.

A. Randomized Testing and Test Case Minimization

The authors have recently been exploring the technique of randomized testing with test oracles. In this technique, a randomized test generator selects test inputs, and a test oracle (a program that evaluates the output of another program) checks the outputs. The randomized test generator allows the automatic production of a high volume of varied test inputs, and the test oracle allows the outputs to be checked automatically. The goal is not to build a fixed test suite, but rather to keep drawing test cases randomly from a large test case space until either the software under test fails or a stopping condition is reached. This technique has been known for years in specific domains, like compiler testing [1] and functional program testing [2], and our previous research has quantified some benefits of it [3], [4]. An outstanding problem, however, has been the quality of the failing test cases. In some instances the failing test cases are hundreds of lines of random input, which can be difficult for a human debugger to use in debugging.

In this paper, we apply Zeller and Hildebrandt's Delta Debugging minimization algorithm [5] to the problem of cutting down the size of the failing test cases resulting from randomized testing. It is clear from the outset that test case minimization will have some effect, but there are four main questions. First, *how much* effect will it have? For instance, will it cut down on the size of the test inputs enough to have a significant impact on the feasibility of randomized testing? Second, *how useful*, subjectively speaking, are the minimized test case inputs? Third, *how effic ient*, in terms of running time, is the minimization process in practice? Fourth, can we use test case minimization to *improve the process* of randomized testing, for instance by re-running randomized testing and minimization in order to get smaller and smaller failing test cases?

B. Unit Testing

In order to tighten our research focus, we have been concentrating on *unit testing*, the testing of classes, source files, or closely-related groups of methods. Unit testing is acknowledged to be an important step in the testing process, which speeds debugging and facilitates later integration and system testing [6]. It is also emerging as an essential component in rapid development techniques, through easy-to-use tools like JUnit [7].

We interpret a test case input to a unit under test (UUT) to be a sequence of calls to the methods in the unit, together with the arguments that are given to the method calls and the method calls that are needed to construct those arguments. To check whether the software has performed correctly on a test case, we check such things as the return values of the methods and the post-state values of the parameters that may have been changed by the method calls.

To perform unit testing on a UUT, we need a *driver*, a main program that exists only for the purpose of testing the UUT. In order to more conveniently control the inputs and outputs of the driver, we use drivers that take as input a text file consisting of a sequence of commands, one command per line, and that produce as output a text file reporting on the results of those commands. Each command in the input file may specify a method call, a step in building a parameter for a method call, or a step in extracting information from the results of a method call. The output file is given to a test oracle for evaluation.

It is the text input files, interpreted as test cases, that are the target of the randomized generation. A randomized test case generator generates random sequences of method call, parameter building, and information extraction commands. A test case that forces a failure may therefore contain many

Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)
1071-9458/05 $20.00 © 2005 **IEEE**

unnecessary commands, such as commands to call methods that are not the cause of the failure.

C. Research Contribution

The first contribution of this paper is to describe a general framework for randomized unit testing that is a generalization of both conventional unit test drivers and of our own previous research on randomized data structure testing.

The other contributions have to do with test case minimization of failing test cases found by randomized drivers. We found that Zeller and Hildebrandt's Delta Debugging minimization algorithm ($ddmin$) cut down the size of these randomly generated failing test cases by an average of 71% to 93%, depending on the UUT; the sources of variation from one UUT to another included the number of methods in the UUT and the complexity of the implementation. We found that the minimization process had only a moderate cost in CPU time. We also found that repeating the randomized generation and minimization processes resulted in a range of test cases that clarified the failing behaviour. However, we found that repeated random generation and minimization did not usually result in significantly smaller test cases than a single round of generation and minimization.

D. Structure of Paper

In this paper, we first describe a general framework for randomized unit testing (Section II), showing how conventional unit test drivers and our previous work on data structure testing are instances of the framework. We then describe a case study on minimization of randomized unit test cases, performed on three units from two data structure packages found on the Sourceforge free software repository (Section III). We then describe an experiment on mutants of a set of data structure units that we have used in previous research, an experiment aimed at quantifying and visualizing the amount of benefit obtained by the test case minimization process (Section IV). We discuss previous research in this area and relate it to the research reported here (Section V). Finally, we give conclusions and suggestions for future research.

II. Randomized Unit Testing

In this section, we describe the randomized unit testing technique that we study in the paper, a generalization both of conventional unit testing and of our previous work, which concentrated on randomized testing of data structures.

We use, as an example of a UUT, an object class `Course` containing a method `register`. We assume that `register` takes one parameter of class `Student`, and allows the student to register in the course if the size limit for the course has not been reached and if the student has completed the prerequisites for the course.

A. Controlflow and Dataflow Views of Drivers

In a flowchart view of the operation of a unit test driver (Figure 1a), the driver performs a sequence of test cases, checks the results, and judges the success or failure of each test case. For example, test cases for the `Course` class may include one that tries to register a student having the prerequisites, one that tries to register a student not having the prerequisites, and one that tries to register students when the size limit has been reached. The driver could be programmed from scratch or built using a framework such as JUnit [7], in which each test case would be represented by one test method.

For the purpose of generalizing a test driver to a randomized test driver, it is more useful to view the driver not in terms of its control flow, but rather in terms of its data flow, because the randomized operations will randomly change the values of the data used. In this dataflow-oriented view (Figure 1b), the driver executes a series of *test fragments*, each of which might either perform a call to a UUT method, set up parameters for future method calls, or extract information from the results of past method calls.

The driver will typically have one or more local variables, such as instances of a class under test or variables to be used as parameters or results of UUT methods. Each local variable whose value is set by one test fragment and whose value is then used by a future test fragment is here called a *persistent variable* (PV in Figure 1b). The driver can therefore be seen as a program that sets initial values for the persistent variables, and then executes test fragments, each of which may change the value of one or more of the persistent variables. For example, a `Course` driver may have one `Course` persistent variable and one `Student` persistent variable; the test fragments may initialize or reinitialize the `Student` to a new object instance, may add more information to the `Student` about past courses taken, and/or may call the `register` method, giving the `Student` as a parameter.

B. Randomized Test Drivers

The kind of randomized test drivers that we consider are generalizations of the dataflow view of unit test drivers. The randomized driver initializes the persistent variables, and then randomly selects and executes a test fragment. It keeps randomly selecting and executing test fragments until some stopping condition is met, such as when a certain number of test fragments has been executed. Each of these test fragments may evaluate preconditions and skip further processing of its target method calls if the preconditions are not met – for instance, if the persistent variables are not yet in a form that can be passed as a parameter. Each fragment may itself contain a random element, such as in the selection of scalar arguments for method calls. Each fragment may evaluate results, or just output information about the results for later evaluation, as with a test oracle.

For example, a randomized driver for `Course` might contain a test fragment that reinitializes the `Student`, another that randomly selects a "past course taken" from a fixed list of courses and adds it to the `Student`, and another that calls the `register` method. A random sequence of test fragment executions is very likely to eventually make both valid and invalid calls to `register`, and may execute test sequences

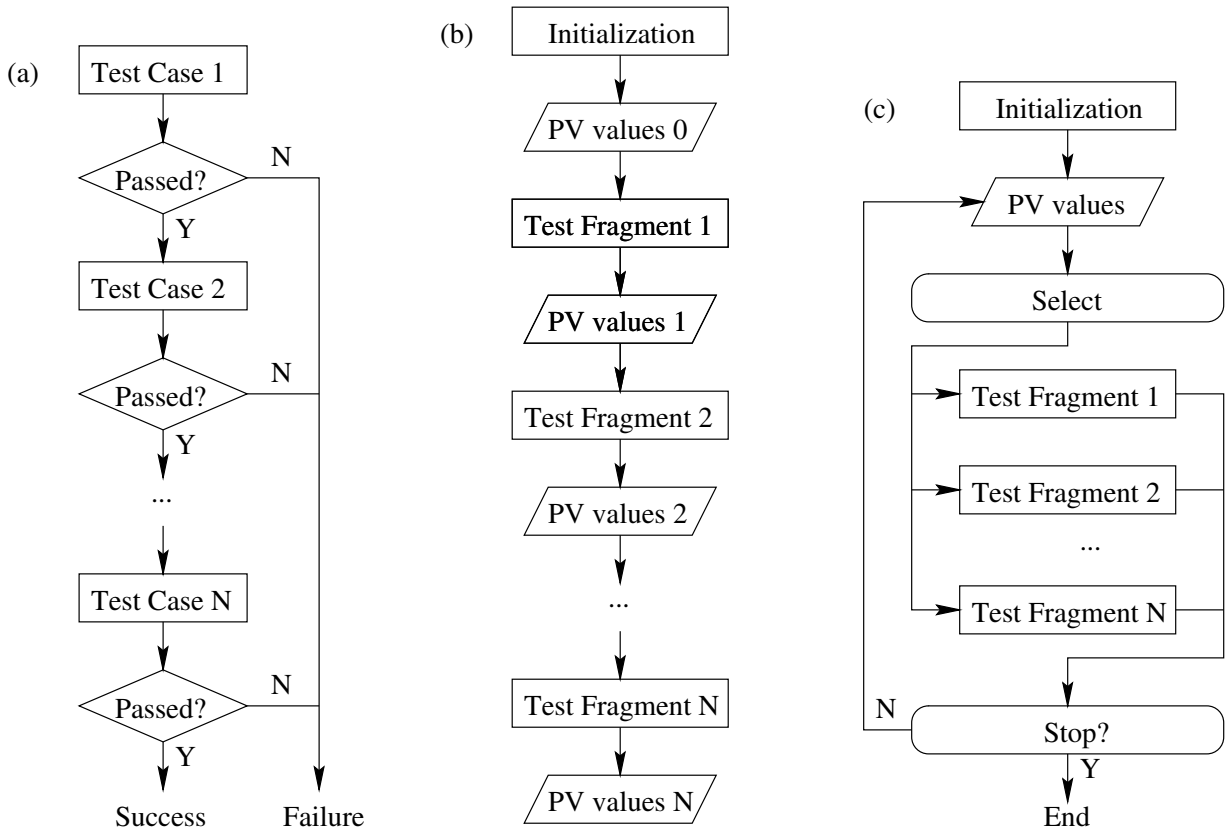

Fig. 1. (a) Controlflow-oriented view of a unit driver. (b) Dataflow-oriented view of a unit driver. (c) A randomized unit driver based on the dataflow view.

not accounted for by the writer of a non-randomized test driver, such as calling `register` twice with the same `Student`.

Such randomized drivers are a strict generalization of the dataflow view of conventional non-randomized drivers, such as JUnit drivers. Since conventional drivers are commonly written for units (for instance, JUnit is widely used in industry), it is reasonable to expect test engineers to be able to write randomized test drivers from scratch. However, there is clearly potential for automating parts of the process of writing the randomized test drivers (see Related Work).

In order to facilitate the generation of test inputs and the analysis of test outputs, we have in our research studied drivers that take text files as input and produce text files as output. The input files contain sequences of commands; each command is a request to execute a particular test fragment, possibly with selected values. The burden of random selection is thus shifted to a separate, randomized test case generator program. For instance, say that the `register` method will work only if the `Student` has already taken the course `210`. The randomized test case generator may produce a command file consisting of the commands

```
reinitStudent
addCourse 110
register
addCourse 210
register
```

The driver, on receiving this file as input, may produce the output

```
reinitStudent
addCourse 110 succ
register fail
addCourse 210 succ
register succ
```

indicating that both `addCourse` method calls succeeded, but that only the second `register` call succeeded. The format of both the inputs and the outputs is decided by the programmer, but should reflect the actions performed by the driver.

Randomized unit testing may perform many useless sequences of actions. For instance, another generated test case for `Course` might initialize the `Student`, add a course to it, and then reinitialize the `Student` before doing anything useful with it. However, because randomized testing is automatic, we can perform an arbitrary number of very long sequences of test fragments, that are likely to eventually include all useful sequences. Previous research indicates that random drivers can achieve high code coverage without a significant performance penalty when compared to conventional drivers [3].

C. Test Oracles

Conventional unit test drivers execute sequences of method calls that the programmer knows are supposed to achieve a

given result. However, randomized drivers execute random sequences of method calls; it is possible to check the results of these sequences manually, but doing so requires tediously inspecting the parameters and return values of each method call. A test oracle, or program to automatically check the test results, is a virtual necessity [8].

Much past research (e.g., [9], [10], [11]) has centred on generating oracles from formal specifications. In our research, we follow the closely-related approach of writing test oracles in a programming language named LFAL designed for that purpose [12]. LFAL is based on logic and state machines, and a unit test oracle written in it is equivalent to a formal specification of the input/output behaviour of the unit.

Previous studies have shown that randomized unit test drivers coupled with LFAL oracles can be highly effective at forcing failures of the UUT [3], and that the overall technique can be applied to real-world units at a reasonable cost [4]. A study with human users [13] has shown that LFAL is judged to be reasonably useful and reasonably easy to use.

D. Test Case Minimization

Although randomized unit testing can accurately find failing test cases, the test inputs that cause the failure can contain both commands relevant to the failure and random commands that are irrelevant, in the sense of not contributing to forcing the failure. Separating the relevant parts from the irrelevant parts is a difficult task for the human debugger; for example, Andrews [4] estimated that half the time spent in randomized testing of some units consisted of tracking down and fixing bugs.

It is therefore natural to ask how much benefit could be obtained by automatically reducing the size of the failing test cases. The state of the art in this area is Zeller and Hildebrandt's Delta Debugging minimization algorithm, $ddmin$ [5]. We briefly summarize this algorithm here.

First some definitions. A *failing test case* is a test case that causes the SUT to perform incorrectly. Since we assume software with text inputs, and since the unit of input to our test driver programs is a one-line command, the *length* of a test case is the number of lines in its input file. A *global minimized* failing test case is one whose length is less than or equal to the length of all other failing test cases. A *local minimized* failing test case is one such that deleting any one line in the input file does not result in a failing test case. The $ddmin$ algorithm does not attempt to find global minimized failing test cases; instead it finds local minimized failing test cases based on a given failing test case.

$ddmin$ is given as input a failing test case and a granularity level, initially set at 2. The algorithm proceeds by progressively reducing the size of the failing test case, by repeatedly trying the three following strategies.

- Reduce to subset. Given the granularity level n, we split the original test case text file into n pieces, all of them approximately equal in length. We test those n test cases separately. If one of them causes the SUT to fail, then we take that split test case as the new working test case.
- Reduce to complement. We again split the original test case into n pieces, but now we form the complements TC_1 to TC_n of the test case. Each complement TC_i is the original test case with piece i deleted. We test the n complements separately, and if one causes the SUT to fail, we take it as the new working test case.
- Increase granularity. We multiply the granularity level n by 2. If this results in an n which is larger than the number of lines of input, we set n to be equal to the number of lines of input.

The above three strategies are tried repeatedly until n is the number of lines of input and neither of the first two strategies has resulted in a reduction in the test case size. The result is clearly a local minimized test case, although it might not be a global minimized test case.

Zeller and Hildebrandt showed in [5] that the time complexity of the algorithm is $O(n^2)$, where n is the size of the original failing test case. They also found that the algorithm worked well for failing test inputs of a number of complex pieces of software, reducing the size of the test inputs substantially. It was therefore a natural choice for use in our research on unit test case minimization. See [5] for more details and analysis of the algorithm.

III. Case Study

We applied failing test case generation and minimization to a group of open-source data structure units. The units were chosen because they were real-world units with known faults that arose during the course of development of the units. We found that the minimization process substantially reduced the size of the failing test cases and had other benefits for the debugging process.

A. Units Under Test

The units under test were drawn from the GCS and LIBDS data structure packages, found on the Sourceforge open-source software repository [14]. These were the packages that were used in a previous real-world case study [4]. It is very difficult to find real-world units together with clearly-identified faults, for the purposes of research in testing; in fact, we are not aware of any others in C in the literature.

In the previous study, a randomized driver and an LFAL test oracle was written for each of 12 units. The drivers generally had one persistent variable each (the data structure in question), and each test fragment typically called one of the methods of the UUT, possibly after selecting and constructing some non-persistent arguments to the method call.

As an example of a driver, one of the test fragments in the driver for the `htable` (hash table) unit of GCS selected a random key from a range of integers, constructed an element to be added based on that key and a random piece of data, added the element to the "hash table" persistent variable by a call to the appropriate method, and reported on the success or failure of the add request based on the return values. Another test fragment randomly selected a key to delete, attempted to

Unit	Package	SLOC	Methods	Faults
`avltree`	LIBDS	611	26	2
`htable`	GCS	102	8	2
`parray`	LIBDS	114	16	2
Total		827	50	6

TABLE I

PROPERTIES OF UNITS OF CASE STUDY. SLOC IS SOURCE LINES OF CODE, I.E. CODE WITHOUT COMMENTS, WHITESPACE OR DECLARATIONS. METHODS IS THE NUMBER OF FUNCTIONS INTENDED TO BE CALLED FROM A CLIENT UNIT.

UUT	Fault	$AvgOrig$	$AvgMin$	$AvgRatio$	CPU
`avltree`	1	200.29	9.70	0.10	6.4
`avltree`	2	104.36	4.93	0.07	1.7
`htable`	1	51.69	5.12	0.11	0.92
`htable`	2	27.74	3.00	0.15	0.61
`parray`	1	38.53	3.00	0.12	0.76
`parray`	2	57.09	4.42	0.11	0.58
Average		79.95	5.03	0.11	1.8

TABLE II

RESULTS OF CASE STUDY. NUMBERS SHOWN ARE AVERAGES OVER 100 RUNS. THE LAST ROW IS AN AVERAGE OVER ALL FAULTS.

delete the key, and again reported on success or failure based on what was returned.

Each unit was subjected to a total of 60,000 test fragment invocations, and the results were checked by the test oracles. The problems found, coverage achieved and effort expended were documented. Despite the fact that the units came with test suites and were reasonably mature, a total of 18 problems were found across the 12 units. Some of these problems were clearly failures, and others were reasonable behaviour that was only inconsistent with the requirements as stated, indicating that it was the stated requirements rather than the software that contained the fault.

The randomized drivers achieved 90.35% line coverage over the units, and the rest of the code was verified or judged infeasible by inspection. The total time taken to write the drivers and oracles, perform the testing and fix the faults in the units was 180 hours, or approximately 40 minutes per entry point. For more detailed information on the study, see [4].

B. Faults Under Study

Ten failures were found that were clear violations of reasonable requirements. Six of these failures corresponded to faults that were fixed in the source code without requiring parallel changes in the test driver or test oracle, and so were good candidates for re-introduction for the purposes of the case study. These six faults happened to be distributed two apiece among three units: LIBDS `avltree` (an AVL tree), GCS `htable` (a hash table), and LIBDS `parray` (an extensible array). Table I contains statistics about these units[1].

As an example, in one of the `htable` faults, when a hash table element was deleted or updated, every element that appeared in its hash bucket before it was erroneously also deleted. The original developer had extensively tested the implementation, but missed the precise sequence that would force the failure: adding two elements so that they appeared in the same hash bucket, deleting or updating the *second* element, and then searching for or otherwise trying to access the *fr st* element. The random testing process found the failure, although the average length of the failing test cases generated was about 52 lines.

[1]Note that we are specifically concerned here with *unit* testing. Thus, although the sizes of the units are small compared to a typical whole program, they are of reasonable size when considered *as units*.

C. Procedure

For each of the faults, we re-introduced the fault into the source code and performed randomized unit testing on the faulty unit. The unit test drivers of the original study integrated the random selection with the actual method calls, so we first had to separate each into a randomized test input generation program and a deterministic test driver, as described in Section II-B.

For each faulty unit, we generated 100 failing test cases. We generated each failing test case by repeatedly running the driver, gradually increasing the number of test fragment invocations requested, until a failure was observed. We then minimized each of the failing test cases. For each faulty unit, we calculated the following statistics:

- $AvgOrig$, the average length of the original failing test case (where the length of a test case is defined as the line number of the first line in the input that causes the incorrect output).
- $AvgMin$, the average size of the minimized test case.
- $AvgRatio$, the average over all test cases of the ratio (minimized size)/(original size). This is not necessarily the same as $AvgOrig/AvgMin$ for the fault, since $AvgRatio$ measures the average reduction in size for a given test case.
- CPU, the average CPU time in seconds needed for the entire process of finding and minimizing a failing test case. `htable` and `avltree` were run on a 29s Mhz UltraSPARC-II; `parray`, which exhibits one of its faults only under Linux, was run on a 600 MHz Pentium III.

D. Results

Table II shows the statistics for the faulty units of the case study. The results show that test case minimization decreased the size of the failing test cases on average by 89%, to an average of 11% of the original failing test case size. This is a significant reduction, and significantly increases the value of the failing test cases to the debugging process, allowing human debuggers to inspect much smaller failing test cases in their attempts to trace faults.

The benefit was greater for the faults in the `avltree` unit than with the other units. It is not clear whether this is because `avltree` had more methods and thus the driver had more opportunities to generate irrelevant method calls, or

Implementation	LL	BST	AVL tree	B-tree	(total)
NLOC	61	93	158	369	681
Number of conditional stmts	9	16	39	44	108
Mutants generated	92	99	381	653	1225
Mutants compiled	85	94	347	576	1102
Mutants non-equivalent	75	82	288	500	945

TABLE III
PROPERTIES OF CODE STUDIED.

whether it is somehow related to the greater code complexity of `avltree`.

In addition to the benefit obtained by finding minimized test cases, the diversity of test cases obtainable by the process proved to be valuable. Repeating the random generation and minimization process yielded a set of different test cases that would help a human debugger to understand better the precise conditions under which failures occur.

For instance, in the example of the hash table fault described above, many different minimized failing test cases were generated, but all of them shared the features of adding two keys, removing or updating one of them, and then performing some operation on the other key. The failure always consisted of the final operation behaving as if the other key was no longer in the hash table. Some of the failing test cases consisted only of these four calls, for instance the following input file:

```
htable_put 7 8156
htable_set 23 31095
htable_del 23
htable_upd 7 9933
```

Closer inspection of the keys and the hashing algorithm would have led a human debugger to conclude that the key pairs always mapped to the same hash bucket, information that would have quickly led to pinpointing the fault (in a function called by both the remove and update methods).

IV. EXPERIMENT

In addition to the case study, we also performed an experiment to quantify and visualize the benefit obtained from randomized test case generation and minimization. The subject units in the study all had the same interface, eliminating the confounding factors of number of functions and number of parameters. The faulty versions of the units were generated using mutation operators; this gave us a large and diverse collection of faults, including some that were very difficult to find. We then performed various statistical and visualization analyses on the data.

A. Units Under Test

The UUTs of the experiment were a set of four implementations (linked list (LL), binary search tree (BST), AVL tree, and B-tree) of a Dictionary ADT, developed previously in our research group based on code found on the Internet. These units had two main advantages. First, they all had the same API (an "initialize" method, an "add" method, a "delete" method and a "find" method) and the same input/output specification, but varied widely in code size and implementation complexity. This allowed us to control for such factors as the number of methods in the unit, the number of parameters to each method, and variations in specification. It also allowed us to use the same LFAL oracle and the same randomized driver for each unit, eliminating any source of variation in how efficient the oracle was or how effective the randomized driver was. The driver had one persistent variable – the data structure itself – and one test fragment for each of the add, delete and find methods.

Second, we had generated *mutants* for all the units. A mutant of a piece of source code is a variant of the source code obtained by changing some part of one line of code, for instance by changing a relational operator to another relational operator, changing an integer constant to 0, or deleting a statement entirely [15]. Many mutants can be generated from a piece of source code, increasing the statistical significance of the results obtained and providing a set of faulty variants with a wide range of behaviour. Recent research suggests that the behaviour of mutants that are not equivalent to the original software can closely approximate the behaviour of faulty versions of software observed in actual practice [16].

Table III gives an overview of the properties of these units, in number of lines of code, number of conditional statements as a rough indicator of code complexity, number of mutants generated and compiled, and number of mutants found to be faulty, i.e. not equivalent to the original UUT.

B. Procedure

For each of the mutants of each of the units, we ran the randomized driver in order to find a failing test case. If none was found, then the mutant was judged to be equivalent to the original and discarded. If a failing test case was found, we minimized the test case and then performed failing test case generation and minimization nine more times, for a total of ten failing and minimized test cases. We then calculated the measures $AvgOrig$, $AvgMin$ and $AvgRatio$ as in the case study (see Section III-C).

We also took the opportunity to see whether running the test case generation and minimization process ten times and taking the *smallest* minimized test case out of the ten resulted in a significant advantage. We therefore also recorded the measure $MinMin$ for each mutant, the smallest minimized test case size out of the ten generation and minimization runs.

In order to measure the efficiency of the test case minimization, we also recorded $AvgTime$, the CPU time averaged over the ten minimization runs for each mutant. All runs were made on a Sun Ultra 5 Solaris machine (purchased 1998) with a 300Mhz CPU and 128 MB memory.

Finally, we computed the averages of all five measures *across all mutants* of each of the four implementations.

C. Results

Table IV gives an overview of the measures $AvgOrig$, $AvgMin$, $AvgRatio$, and $AvgTime$ obtained from the ex-

	$AvgOrig$ (lines)	$AvgMin$ (lines)	Avg $Ratio$	$AvgTime$ (seconds)
LL	9.61	1.91	0.29	2.52
BST	16.04	2.58	0.28	3.55
AVL tree	143.35	12.05	0.15	60.38
B-tree	49.72	7.52	0.22	13.40
Average	54.68	9.38	0.24	19.96

TABLE IV
OVERVIEW OF EXPERIMENT RESULTS.

	$AvgMin$ (lines)	$MinMin$ (lines)	Ratio
LL	1.91	1.88	0.98
BST	2.58	2.54	0.98
AVL Tree	12.05	8.66	0.72
B Tree	7.52	6.88	0.91
Average	9.38	4.99	0.90

TABLE V
THE EFFECT OF REPEATING MINIMIZATION.

periment. The last line of the table shows the average across the four UUTs. We now discuss separately the reduction in size of the test cases, the efficiency of the procedure, and the effect of repeatedly generating and minimizing test cases.

a) Reduction of failing test case size: Across the four UUTs, the test case minimization process resulted in a reduction of the failing test cases to an average of 24% of the original size. The benefit was greatest for the AVL tree UUT, perhaps because it had the largest average size of failing test case; its failing test cases were reduced to an average of only 15% of the original size.

Since all four UUTs had identical APIs and specifications, this indicates that at least some of the benefit is connected to the complexity of the unit, and not to the number of callable methods or number of parameters to the methods. This conclusion is also supported by the fact that the most complex unit in terms of code size and number of conditional statements (the B-tree) also had a low $AvgRatio$. However, the case study reported in the last section suggests that the number of methods in the unit also has some effect on $AvgRatio$, since those units had more methods and received even greater benefit from the minimization process.

b) Visualization of reduction: In order to visualize the relationship between original test case size and minimized test case size, we produced a scatter plot with those axes for each UUT. Figure 2 is a typical such graph, for the AVL tree UUT. It shows a roughly linear relationship between the two, except that the largest original test cases tend to get minimized to a sublinear level, indicating a greater benefit for larger original test cases.

In order to better visualize this effect, we then ordered the mutants for each UUT by $AvgOrig$, the average size over ten runs of the failing test cases originally found. We then plotted a graph in which the x-axis was the rank of a mutant in this ordering, and the y-axis was test case length. We plotted two measures on these axes: $AvgOrig$ and $AvgMin$, the average length over the ten runs of the minimized test cases.

Figure 3 shows a typical such graph, for the AVL tree UUT. The thin upper line is the $AvgOrig$ line, and the thick lower line is the $AvgMin$ line. Note that the upper line is necessarily nondecreasing because of how we have structured the graph. The lower line is not monotonic but does show a slight overall increase. The graphs for the other UUTs are similar.

Note the final "spire" at the right of the upper line in the graph. This represents 12 mutants having 400-800 lines on average in their failing test cases. We may view these as mutants with extremely difficult-to-find bugs. The test case minimization process found locally minimized test cases with an average of 35 lines or fewer for each of these mutants, indicating that the minimization process yielded more benefit for the hardest bugs.

c) Efficiency: Table IV indicates that the average amount of CPU time needed was 20 seconds, which is a relatively long time even on a 7-year-old machine. The results were skewed by the behaviour of the AVL tree UUT, which had average times of about 60 seconds. They were also skewed by the fact that some of the mutants exhibited their faulty behaviour by going into an infinite loop; although we controlled this to some extent, still each run took at least 1 CPU second.

In order to better visualize the relationship between CPU time and original test case size, for each of the UUTs we made a scatter plot of $AvgOrig$ vs. $AvgTime$, in which each point represents one mutant. Figure 4 shows the CPU time scatter plot for the AVL tree implementation; the other graphs are not significantly different.

Note that the 12 extremely difficult mutants (those with $AvgOrig$ of 400 or more) are also among the mutants that take the most CPU time to process. Since we obtain more benefit from the minimization process with these difficult mutants, there is more justification for the increased CPU time.

When we exclude the 12 hardest mutants, the rest of the scatter plot forms a rough parabola. This is consistent with Zeller and Hildebrandt's observation [5] that the analytic complexity of the $ddmin$ algorithm is quadratic in the size of the original test case. To lower the average processing time further, we would probably need to perform optimizations to the minimization algorithm.

d) Repeated minimization: We also measured the effect that could be obtained by repeating the minimization and then selecting the smallest minimized failing test case. The measure $MinMin$ represented the length of the smallest of the ten minimized test cases found for each mutant.

Table V shows the effect of the repeated minimization, along with the average ratio between the length of the smallest of the ten minimized test cases and the average length of the minimized test cases. The average reduction in test case length over the four implementations is 10%, but this is only because the AVL tree implementation sees a reduction of 28%. Most of the units obtained less than a 10% benefit from the repeated minimization.

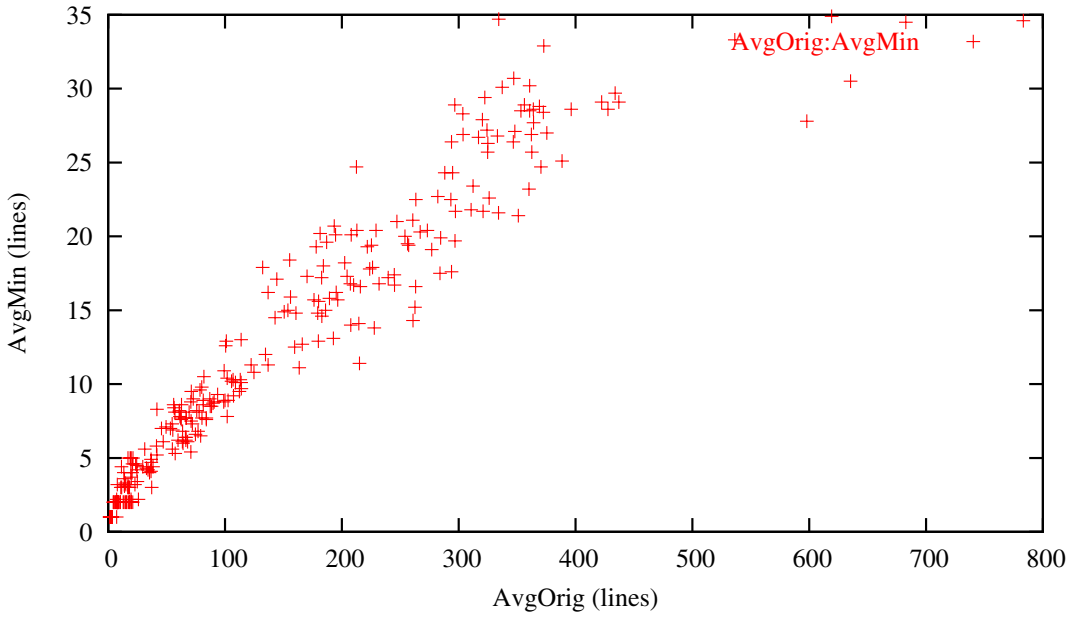

Fig. 2. AVL Tree: scatter plot, $AvgOrig$ vs. $AvgMin$.

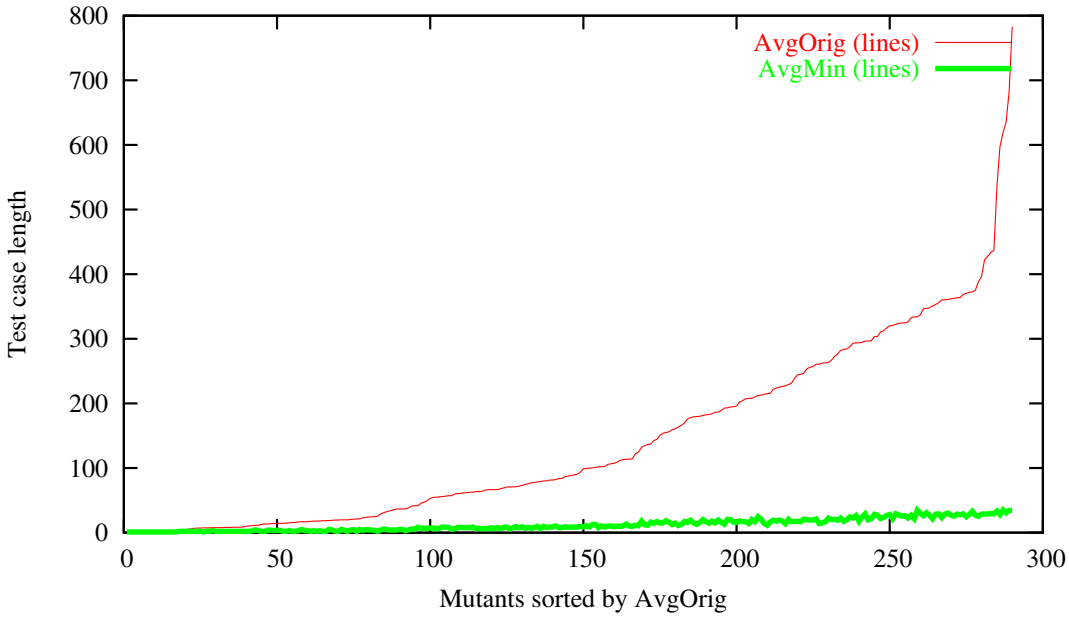

Fig. 3. AVL Tree: Sorted line graph of $AvgOrig$ and $AvgMin$.

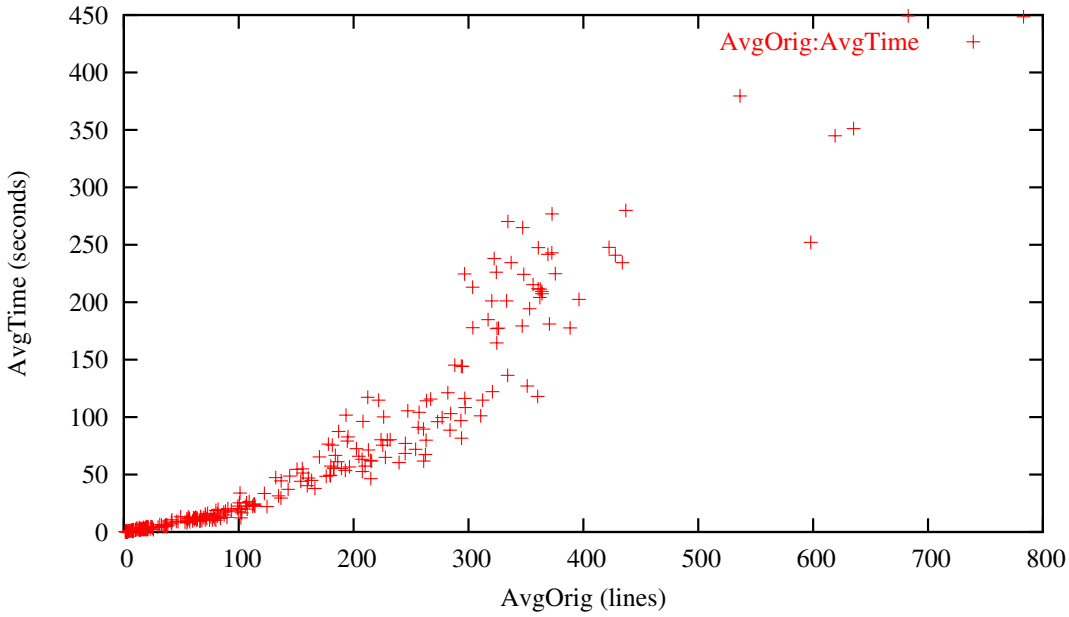

Fig. 4. AVL Tree: scatter plot, $AvgOrig$ vs. $AvgTime$.

Considering the cost of the minimization process, we can give no definite recommendation about whether repeated generation and minimization is worthwhile for any given UUT. Test engineers can experiment informally with re-running the generation and minimization process to see if they obtain a benefit that they judge to be worthwhile.

D. Threats to Validity

Threats to internal validity of the experiment include the possibility of faulty experiment software. We controlled for this threat by formally reviewing all scripts and programs. Threats to construct validity include the possibility that the mutants generated are not good representatives of the space of possible faults. We controlled for this threat by using mutation operators that have been shown to be sufficient [15] and shown to simulate the behaviour of real faults well [16].

The main threats to validity are to external validity. In order to control for number of methods and specification complexity, the experiment studied four units with the same API and specification. The results may not therefore generalize across all units, although the case study suggests that even more benefit can be obtained when the number of methods is larger. We have not yet performed randomized unit testing and test case minimization when the number of persistent variables is greater than one, or on units written in languages other than C. We plan to address this by performing experiments on a broad range of units written in Java.

V. Related Work

The phrase "random testing" is often taken in earlier literature to mean either the use of a small number of arbitrary, unplanned test cases [17] or strict random selection of test cases unguided by coverage criteria [18], [19]. Our work is more in the tradition of McKeeman's work on compiler testing [1] and Claessen and Hughes' QuickCheck for Haskell programs [2]; in that work, the tester uses coverage measures to intelligently guide the selection of ranges for random parameters, weights for function calls, and so on. We also assume that the randomized driver will be modified until high coverage is achieved [4].

Previous approaches to unit testing that use formal specifications often attempt to generate drivers, test cases and/or oracles; examples include ASTOOT [20] and TACCLE [21]. These approaches tend not to take account of code coverage, limiting their practical effectiveness. A further difficulty arises when parameters to the methods of the UUT are themselves complex. Leow et al. [22] describe a system based on an AI planning algorithm for generating code that builds complex parameters; they provide some small case studies in [23]. Tkachuk et al. [24] describe a system for building drivers to encapsulate units for modular software model checking; these drivers have many of the features of unit testing drivers. We believe that these tools for generating drivers and method parameters could be integrated with our work in the future. For now, we rely on the programmer's ability to write test fragments that build method parameters, something the programmer would have to do for a conventional driver already, for instance when using JUnit [7].

The research that is closest in spirit to our own is recent research on the Tobias and Jartege tools [25], [26]. Tobias is given a schema of test cases and generates all instances of the schema, whereas Jartege is able to randomly generate test driver programs, each of which runs one test case. These systems embody good ideas that we hope to use in our own future work on building a practical toolset for randomized unit

testing. Godefroid et al. [27] also employ a closely-related approach in which randomized path exploration is combined with static analysis of programs with assertions.

The standard reference for test case minimization is that of Zeller and Hildebrandt [5]. To our knowledge, the research reported here is the first to combine randomized test case generation and general-purpose minimization, although McKeeman [1] did apply domain-specific heuristics to reduce the size of the failing test cases generated to test compilers.

VI. Conclusions and Future Work

Randomized unit testing is a testing practice that can achieve high coverage and is effective at forcing failures, but tends to generate long failing test cases. In this paper, we have shown that Zeller and Hildebrandt's $ddmin$ algorithm can significantly reduce the length of these failing test cases, making them more valuable for the debugging process. This in turn increases the practical applicability of randomized unit testing.

The CPU time needed for test case minimization is not trivial, but is reasonable given the benefit obtained. We did not find that repeating the generation and minimization process usually resulted in significantly smaller failing test cases, but we did find that it resulted in a range of diverse failing test cases that clarified the nature of the failure.

We make no claims here about the applicability of randomized testing to system-level testing, concentrating instead on unit testing. We also recognize that the process of writing an oracle for a unit is non-trivial, although we are encouraged by recent research by us on the reasonable cost of oracles [4] and by others on the closely-related problem of randomized testing using assertions instead of oracles [27].

In the future, we wish to develop a JUnit-style or Jartege-style framework that could perform the randomized testing and test case minimization semi-automatically. This framework may allow for a semi-automatic generation of the randomized driver as well.

Acknowledgments

Thanks to Akbar Siami Namin for reviewing an earlier draft and making many detailed suggestions. Thanks also to Andreas Zeller for valuable discussions. This work is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[2] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, Montreal, Canada, September 2000, pp. 268–279.

[3] J. H. Andrews and Y. Zhang, "General test result checking with log file analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 7, pp. 634–648, July 2003.

[4] J. H. Andrews, "A case study of coverage-checked random data structure testing," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, September 2004, pp. 316–319.

[5] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, February 2002.

[6] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed. New York: Van Nostrand Reinhold, 1993.

[7] "JUnit web site," 2005, `www.junit.org`.

[8] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, November 1982.

[9] D. B. Brown, R. F. Roggio, J. H. Cross II, and C. L. McCreary, "An automated oracle for software testing," *IEEE Transactions on Reliability*, vol. 41, no. 2, June 1992.

[10] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based test oracles for reactive systems," in *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 1992, pp. 105–118.

[11] D. K. Peters and D. L. Parnas, "Using test oracles generated from program documentation," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 161–173, 1998.

[12] J. H. Andrews and Y. Zhang, "Broad-spectrum studies of log file analysis," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 105–114.

[13] G. Huang and J. H. Andrews, "Learning and initial use of a software testing technology: An exploratory study," in *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE 2004)*, Edinburgh, Scotland, May 2004, pp. 77–86.

[14] "Sourceforge web site," 2005, `sourceforge.net`.

[15] A. J. Offutt and R. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000, pp. 45–55.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, May 2005, to appear.

[17] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[18] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438–444, July 1984.

[19] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, December 2001.

[20] R.-K. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, pp. 101–130, April 1994.

[21] Chen, Tse, and Chen, "TACCLE: A methodology for object-oriented software testing at the class and cluster levels," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 56–109, 2001.

[22] W. K. Leow, S. C. Khoo, and Y. Sun, "Automated generation of test programs from closed specifications of classes and test cases," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, Edinburgh, UK, May 2004, pp. 96–105.

[23] W. K. Leow, S. C. Khoo, T. H. Loh, and V. Suhendra, "Heuristic search with reachability tests for automated generation of test programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, September 2004, pp. 282–285.

[24] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu, "Automated environment generation for software model checking," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, October 2003, pp. 116–127.

[25] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron, "Filtering TOBIAS combinatorial test suites," in *Proceedings of ETAPS/FASE'04 – Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 2984. Barcelona, Spain: Springer-Verlag, 2004, pp. 281–294.

[26] C. Oriat, "Jartege: A tool for random generation of unit tests for Java classes," Institut d'Informatique et de Mathématiques Appliquées de Grenoble (IMAG), Grenoble, France, Tech. Rep. RR 1069, June 2004.

[27] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, June 2005, pp. 213–223.