Efficient Unit Test Case Minimization

Andreas Leitner
Chair of Software Engineering
ETH Zurich, Switzerland
andreas.leitner@inf.ethz.ch

Manuel Oriol
Chair of Software Engineering
ETH Zurich, Switzerland
manuel.oriol@inf.ethz.ch

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-sb.de

Ilinca Ciupa
Chair of Software Engineering
ETH Zurich, Switzerland
ilinca.ciupa@inf.ethz.ch

Bertrand Meyer
Chair of Software Engineering
ETH Zurich, Switzerland
bertrand.meyer@inf.ethz.ch

ABSTRACT

Randomized unit test cases can be very effective in detecting defects. In practice, however, failing test cases often comprise long sequences of method calls that are tiresome to reproduce and debug. We present a combination of static slicing and delta debugging that automatically minimizes the sequence of failure-inducing method calls. In a case study on the EiffelBase library, the strategy minimizes failing unit test cases on average by 96%.

This approach improves on the state of the art by being far more efficient: in contrast to the approach of Lei and Andrews, who use delta debugging alone, our case study found slicing to be 50× faster, while providing comparable results. The combination of slicing and delta debugging gives the best results and is 11× faster.

Categories and Subject Descriptors: D.2.5 [Software Engineering] Testing and Debugging–Testing tools (e.g., data generators, coverage testing)

General Terms: Experimentation, Verification

1. INTRODUCTION

Although automated test generation is more attractive than ever, randomly generated unit test cases can become very large in size of inputs and the number of method invocations. While large unit test cases can be useful for triggering specific defects, they make the subsequent diagnosis hard, and take resources to execute.

In this paper, we introduce a novel test case minimization method based on *static program slicing*. The key idea of using slicing for minimization is straightforward: we start from the oracle in the failing test case—that is, the failing assertion or another exception. By following back data dependencies, we establish the subset of the code—the *slice*—that possibly could have influenced the oracle outcome; any instructions that are not in the slice can therefore be removed. In contrast to related work (Section 6), this approach is far more efficient.

2. CREATING TEST CASES

To generate test cases, we use the AutoTest [7] tool. AutoTest automatically tests Eiffel classes. by invoking their methods at random. The Eiffel contracts (pre- and postconditions and class invariants), constitute natural test oracles and can be exploited to narrow down the test generation space [7].

As an example of a test generated by AutoTest, consider Listing 1, testing the interplay of various classes in the *EiffelBase library*—the library used by almost all Eiffel applications, covering fundamental data structures and algorithms.

```
    ...
67  v_61. forget_right
68  create {PRIMES} v_62
69  v_63 := v_62.lower_prime ({INTEGER_32} 2)
70  create {STRING_8} v_64.make_from_c(itp_default_pointer)
    ...
146 create {ARRAY2 [ANY]} v_134.make ({INTEGER_32} 7, {
      INTEGER_32} 6)
147 v_134.enter (v_45, v_131)
148 create {RANDOM} v_135.set_seed (v_63)
149 v_136 := v_135. real_item
```

Listing 1: A generated random test case. Line 149 fails with a nested precondition violation.

As shown in Listing 1, an AutoTest-generated test case only needs four kinds of instruction—object creation, method (routine) invocation, method invocation with assignment, and assignment. This language is as complete as required by the test synthesis strategy and as simple as possible. *Control structures,* for instance, are not needed as tests are generated on the fly. Likewise, *compos-*

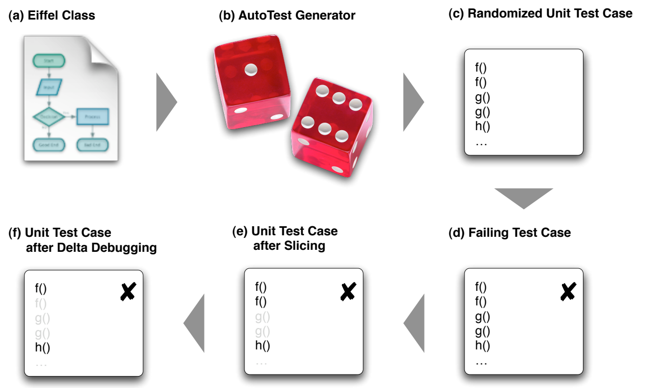

Figure 1: Unit test generation and minimization. An Eiffel class file (a) is fed into the AutoTest tool (b), which generates a randomized unit test case (c) as a sequence of method calls. Failing unit test cases (d) are automatically minimized—first by static analysis (e), and then by delta debugging (f).

ite expressions can be transformed into a sequence of instructions using only atomic expressions.

The test case in Listing 1 has successfully detected a defect in the EiffelBase library. The invocation in Line 149: *v_136* := *v_135*. *real_item* results in a precondition violation (of the *RANDOM*.*i_th* method), raising an exception.

How did this violation come to be? In general, a violated precondition indicates a defect in the caller or further upstream in the execution. In Listing 1, though, "upstream" means 148 lines of code, not counting all the code that is executed in method calls. Such a large test case makes diagnosis hard and brings a recurrent performance penalty when executed repeatedly.

A means to *simplify failing test cases* to the bare minimum of instructions required to reproduce the failure is therefore needed. This helps the programmer understand the failure conditions and narrow down the defect—and it makes test execution more efficient. A simple test case will also be easier to explain and communicate. Finally, test case simplification facilitates the identification of duplicate failures (multiple failing test cases that relate to the same defect), as the simplified test cases all contain the same sequence of instructions.

3. MINIMIZING TEST CASES

A common diagnosis strategy for debugging works as follows: starting with the failing instruction, proceed backwards ("upstream") to identify those instructions that might have influenced the failure. This can be done dynamically (e.g. in a debugger) or statically (by analyzing the code). In the case of Listing 1, this backward reasoning could be as follows:

1. Line 149 fails when invoking *v_135*.*real_item*.
2. Object *v_135* was created in Line 148 as a *RANDOM* object, setting the seed from *v_63*.
3. Object *v_63*, in turn, was created from *v_62* in Line 69.
4. Object *v_62* was created from a constant as a *PRIMES* object in Line 68.

Not only does this backward reasoning help us understand the chain of events that led to the failure, but even more importantly: *no other objects were involved in the computation.* In other words, the above backward sequence of events could easily be reversed to a forward subset of instructions, as shown in Listing 2.

```
68  create {PRIMES} v_62
69  v_63 := v_62.lower_prime ({INTEGER_32} 2)
148 create {RANDOM} v_135.set_seed (v_63)
149 v_136 := v_135.real_item
```

Listing 2: Example test case, minimized

Now the reason for the failure becomes clear: creating an instance of class *RANDOM* using the *set_seed* constructor to initialize its seed, we cannot access the random value with *real_item*. What happens here is that the *set_seed* constructor fails to initialize the random generator, which results in the failure. The other *RANDOM* constructor, far more frequently used, works fine.

The general technique of reasoning backwards and identifying those statements that could have influenced a statement at hand is known as *slicing,* and the set of influencing statements is known as a *slice* [9]. In our example, the statements in Listing 2 constitute the *backward slice* of Line 149. Put in other words, Line 149 *depends* on the statements in the backward slice, because they all potentially influence the input to Line 149. Furthermore, the slice is a *static slice,* as it is obtained from program code alone—in contrast to a *dynamic slice* [1, 5], which would be extracted from a program run, tracking all concrete variable accesses.

Test scope	Avg. no. original inst.	Avg. no. min. inst. (slicing)	Avg. reduction factor (%)
EiffelBase	35.29	1.62	95.39
Data struct.	100.24	2.90	97.10

Table 1: Reduction in number of lines of code of test cases by slicing. Slicing reduces the size of test cases by over 90%.

These ideas are the basis of our approach to minimize test cases: a straightforward static analysis that follows data dependencies backwards from the failing method call, and returns a slice within the generated test case. This slice then serves as a minimized test case as well.

This approach is summarized in Figure 2. The core of the algorithm is the definition of the *ssmin* function, which gets a sequence of instructions to be minimized.

The slicing approach to simplify test cases is surprisingly simple but highly effective. In our test setting, executing the full test case (Listing 1) took 2344 ms. Executing the minimized test case takes only 105 ms—that is, the test case is now 22 times faster. Due to the verifying test run at the end, the whole run for *ssmin* took 105 ms as well. The slicing step itself (*slice*) took less than 1 ms, which means virtually no cost.

4. CASE STUDY: EIFFELBASE LIBRARY

In order to evaluate the efficiency and effectiveness of the *ssmin* algorithm we have generated over 1300 failing test cases for the EiffelBase library with AutoTest.

Table 1 summarizes the results of the case study in terms of test case size. The number of instructions per test case is averaged out over all processed test cases.

Slicing produced test cases reduced by 95% (EiffelBase) and 97% (data structures). This clearly indicates that slicing is a very effective minimization technique despite its simplicity.

The overall overhead of minimization compared to random testing (as summarized in Table 2) is small: the first testing session lasted for 900 seconds producing 1316 failing test cases and the second session lasted for 300 seconds producing 168 failing test cases. Minimizing the failing test cases was much faster with 94 seconds for the first session and 19 seconds for the second.

Test scope	Number of failing TCs	Total testing time (s)	Total minimization time (s)
EiffelBase	1316	900	94
Data struct.	168	300	19

Table 2: Performance of testing compared to minimization. Minimization is quicker than testing itself.

Table 4 (column *ssmin*) shows how much time is spent slicing and how much time executing the resulting slice. Values are again averaged out over all processed test cases. Most of the minimization time is spent verifying the slice (via execution). The slicing step in particular, takes negligible time; this seems to confirm the advantage of a shallow but fast slicing. The sum of slicing and execution time is slightly smaller than the total minimization time, because the algorithm also performs some extra benchmarking.

Let tc be a test case, represented as a sequence of instructions $tc = [i_1, \ldots, i_n]$. Our *static slicing minimization algorithm* $ssmin(tc)$ returns a *minimized* sequence $tc' = ssmin(tc) = [i'_1, \ldots, i'_m]$ with $m <= n$ and $i'_1, \ldots, i'_m \in tc$.
$ssmin$ is defined from the function $slice$. $slice$ returns all instructions in the original test case on which the last instruction i_n depends upon. In other words, $slice$ returns only those instructions that can actually affect the execution of the last instruction, the one that makes the test fail:

$$slice\big([i_1, \ldots, i_n]\big) \triangleq \big[i \in [i_1, \ldots, i_n] \mid i_n \rightarrow^\star i\big]$$

In this definition, $[a \in A \mid b]$ is the subsequence of A including only those elements for which b holds, and $i_n \rightarrow^\star i$ is the transitive reflexive closure over the *dependency* relationship. An instruction i_2 is *dependent* on an instruction i_1, written $i_2 \rightarrow i_1$, if there is some variable v that is written by i_1, read by i_2, and not written in-between:

$$i_2 \rightarrow i_1 \triangleq \exists\, v \in \big(\, WRITE(i_1) \wedge READ(i_2)\big) \setminus WRITTEN\_BETWEEN(i_1, i_2)$$

(Note that we ignore control dependencies, as the generated test cases do not contain any control instructions.)

The set of variables read and written depend on the individual type of instruction:

$$READ(i) \triangleq \begin{cases} \{source(i)\} & \text{if } i \text{ is an assignment} \\ \{arguments(i)\} & \text{if } i \text{ is an object creation} \\ \{target(i)\} \cup arguments(i) & \text{if } i \text{ is a method invocation} \\ \{target(i)\} \cup arguments(i) & \text{if } i \text{ is a method invocation with assignment} \end{cases}$$

$$WRITE(i) \triangleq \begin{cases} \{receiver(i)\} & \text{if } i \text{ is an assignment} \\ \{target(i)\} & \text{if } i \text{ is an object creation} \\ \{target(i)\} & \text{if } i \text{ is a method invocation} \\ \{receiver(i), target(i)\} & \text{if } i \text{ is a method invocation with assignment} \end{cases}$$

In these definitions, $source$(i) is the right-hand side of instruction i, $receiver$(i) is the left-hand side of an assignment, $target$(i) the target variable of a method call, and $arguments$(i) is the set of actual variables (constants are ignored).

Finally, the set of variables that may be written by some other instruction executed between i_1 and i_2 is defined as

$$WRITTEN\_BETWEEN(i_1, i_2) \triangleq \{v \mid i \in i_1 \leftrightarrow i_2 \wedge v \in WRITE(i)\}$$

Here, $i_1 \leftrightarrow i_2$ denotes the sequence of instructions that can be executed between i_1 and i_2, but excluding i_1 and i_2.

Finally, we define $ssmin$ on top of $slice$. Since $slice$ is unsound, we check whether the minimized test case returned by $slice$ produces the same failure as the original:

$$ssmin(tc) \triangleq \begin{cases} slice(tc) & \text{if } failure(tc) = failure\big(slice(tc)\big) \\ tc & \text{otherwise} \end{cases}$$

where *failure* returns the contract that failed the test.

Figure 2: Static slicing minimization in a nutshell. The $ssmin$ function takes a test case as a sequence of instructions and returns a minimized test case, keeping only those instructions in the test case that might affect the failing instruction.

5. DELTA DEBUGGING

Our work is not the first one that minimizes randomized unit tests. In 2005, Lei and Andrews [6] presented an approach based on *delta debugging* [10], a general and automated solution to simplify failure-inducing inputs. The delta debugging minimization algorithm (abbreviated as $ddmin$) takes a set of factors that might influence a test outcome, and repeats the test with subsets of these factors. By keeping only those factors that are relevant for the test outcome, it systematically reduces the set of factors until a minimized set is obtained containing only relevant factors.

We have implemented $ddmin$ as described in the original paper [10]. The minimized result of $ddmin$ (Table 3, column $ddmin$) is only slightly smaller than the result of $ssmin$ discussed in Section 3. But $ddmin$ takes vastly more time. In EiffelBase (Table 4), the average minimization with $ddmin$ takes 3570 ms, which is 50 times slower than the minimization with $ssmin$ (71 ms). The time of $ssmin$ comes essentially from the single test execution.

Why is $ddmin$ able to minimize further than $ssmin$? The reason is that $ssmin$ relies on *dependencies,* whereas $ddmin$ actually experimentally verifies *causality.* In the sequence y := 1; z := 0; x := $y * z$, for instance, $ssmin$ determines that *x* is dependent on *y*, and therefore keeps y := 1 in the minimized test case. $ddmin$, however, finds that removing y := 1 makes no difference for the value of *x* (nor to the test outcome, for that matter), and therefore removes the instruction as not being a cause of failure. In general, such findings are hard to produce without actual experiments.

It turns out, though, that the strengths of both approaches can be combined, by first applying $ssmin$ and then $ddmin$. As shown in Tables 3 and 4, the combination

$$sdmin(tc) \triangleq ddmin\big(ssmin\,(tc)\big)$$

is just as effective as $ddmin$ alone, yet conserves the efficiency gains of $ssmin$.

Test scope	Average original LOC	Average minimized LOC			Average reduction factor (%)		
		$ssmin$	$ddmin$	$sdmin$	$ssmin$	$ddmin$	$sdmin$
EiffelBase	35.29	1.62	1.42	1.42	95.39	95.97	95.97
Data struct.	100.24	2.90	2.05	2.05	97.10	97.95	97.95

Table 3: Reduction in number of lines of code of test cases by all 3 strategies

Test scope	Minimization average total time (ms)			Average #executions			Average execution time (ms)		
	$ssmin$	$ddmin$	$sdmin$	$ssmin$	$ddmin$	$sdmin$	$ssmin$	$ddmin$	$sdmin$
EiffelBase	71.40	3570.25	316.68	1	14.01	2.10	71.25	3551.42	315.16
Data struct.	110.42	11554.51	738.88	1	29.35	6.05	110.92	11510.41	741.75

Table 4: Minimization times for all 3 strategies

6. RELATED WORK

Delta debugging [10] is a general and automated solution to simplify failure-inducing inputs. If, as in our case, the input is strongly bound by a known semantic of the computation, other domain-specific techniques can be much more efficient. Our experiments indicate that slicing is nearly as effective as delta debugging, and in cases where it fails often reduces the number of executions needed by delta debugging and hence the overall minimization time.

Program slicing was first introduced in the testing area for improving regression testing [2, 4]. It executes a test case designed for previous versions of the code only if the new version of the code changes some of the code executed by the test case. In an empirical study Zhang et al. [11] compare several dynamic slicing algorithms for detecting faulty statements. They conclude that data slices are the most efficient version.

Combining delta debugging and program slicing for testing programs was already proposed by Gupta et al. [3] with a different purpose from what is described here. In that approach, the test cases are minimized with delta-debugging and then the input is taken as the input for dynamic slices of the failing program. In the present work, slicing and delta debugging are both applied to a program only. In this case, delta debugging and slicing are then competing solutions rather than complementary techniques.

While many approaches concentrate on the generation of failure-producing test cases, most do not consider test case minimization. We are only aware of two approaches that minimize test cases, both using delta debugging. The work of Lei and Andrews [6] minimizes randomized unit tests, just as we do, whereas the work of Orso et al. [8] minimizes captured program executions. As discussed in Section 5, our approach is far more efficient than delta debugging alone; we use delta debugging as an optional step to further minimize sliced test cases and as a fall-back mechanism for when program slicing fails. By doing so, it is possible to speed up the minimization process by a factor of 11 comparing to a pure delta debugging minimization technique—which makes it applicable to large industrial code bases, such as the EiffelBase library.

7. CONCLUSION

Failing random unit test cases should be minimized—to locate defects faster, to make execution more efficient, and to ease communication between developers. Our proposed minimization approach based on static slicing is highly efficient, practical, and easy to implement. In our evaluation, running static minimization came at virtually no cost, and the speed gain of the resulting minimized test case pays off with the first execution. To gain another 1% in minimization, one can combine the approach with an additional delta debugging step. This combination yields the same results as delta debugging (the state of the art) alone, but is far more efficient.

On a larger scale, our approach also demonstrates what can be achieved by combining different techniques in quality assurance—in our case, random test generation, static program analysis, and experimental assessment. We believe that the future of program validation lies in such pragmatic combinations, and this is also the focus of our future work.

To encourage further research in these and other directions, everything needed to replicate our experiments is publicly available. A package including all data, intermediate results, the source of the tester and minimizer, as well as the compilers and tools used to compile the source is available at

`http://se.ethz.ch/people/leitner/ase_min/`

Acknowledgements. We thank the Hasler foundation for financial support through project Managing Complexity (project #2146).

8. REFERENCES

[1] AGRAWAL, H., AND HORGAN, J. R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)* (White Plains, New York, June 1990), vol. 25(6) of *ACM SIGPLAN Notices*, pp. 246–256.

[2] AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. A. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance* (Washington, Sept. 1993), D. Card, Ed., IEEE Computer Society Press, pp. 348–357.

[3] GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. Locating faulty code using failure-inducing chops. In *ASE* (2005), D. F. Redmiles, T. Ellman, and A. Zisman, Eds., ACM, pp. 263–272.

[4] GUPTA, R., HARROLD, M. J., AND SOFFA, M. L. Program slicing-based regression testing techniques. *Softw. Test, Verif. Reliab 6*, 2 (1996), 83–111.

[5] KOREL, B., AND LASKI, J. Dynamic slicing of computer programs. *The Journal of Systems and Software 13*, 3 (Nov. 1990), 187–195.

[6] LEI, Y., AND ANDREWS, J. H. Minimization of randomized unit test cases. In *ISSRE* (2005), IEEE Computer Society, pp. 267–276.

[7] MEYER, B., CIUPA, I., LEITNER, A., AND LIU, L. L. Automatic testing of object-oriented software. In *Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)* (2007), J. van Leeuwen, Ed., Lecture Notes in Computer Science, Springer-Verlag.

[8] ORSO, A., JOSHI, S., BURGER, M., AND ZELLER, A. Isolating relevant component interactions with JINSI. In *Proceedings of the Fourth International ICSE Workshop on Dynamic Analysis (WODA 2006)* (May 2006), pp. 3–10.

[9] WEISER, M. Programmers use slices when debugging. *Commun. ACM 25*, 7 (1982), 446–452.

[10] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering SE-28*, 2 (Feb. 2002), 183–200.

[11] ZHANG, X., HE, H., GUPTA, N., AND GUPTA, R. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging* (New York, NY, USA, 2005), ACM Press, pp. 33–42.