On the Effectiveness of Test Extraction without Overhead

Andreas Leitner[1], Alexander Pretschner[2]*, Stefan Mori[1], Bertrand Meyer[1], Manuel Oriol[3]
[1]Department of Computer Science, ETH Zürich, Switzerland
[2]Fraunhofer IESE and TU Kaiserslautern, Germany
[3]University of York, United Kingdom

Abstract

Developers write and execute ad-hoc tests as they implement software. While these tests reflect important insights of the developers (e.g., which parts of the software need testing and what inputs should be used), they are usually not persistent and are easily forgotten. They cannot always be re-executed automatically, for example to debug or to test for regressions. Several methods that make such test cases persistent and automatically executable have been proposed. They rely on capturing state and/or events at run-time and thus induce significant overhead or require specialized hardware. In previous work we proposed a method that, in the event of a failure, extracts test cases solely from the state at the time of the failure (and not from before the failure). We call this method "failure-state extraction." Capturing the state only at the moment of failure reduces the run-time overhead to zero, but comes at a cost: state extracted in this way cannot always be used to reproduce the failure. This paper provides an experimental evaluation of failure-state extraction. The results show that the method is highly effective: in the experiment, 90% of all failures were reproducible using failure-state extraction and thus could be extracted without run-time overhead.

1 Introduction

The advent of unit testing frameworks such as xUnit has largely facilitated—and encouraged—the definition of unit and integration tests. One central feature of such frameworks is that tests become first class citizens and, akin to the code itself, managed entities. Among other things, this approach makes regression testing a lot easier, even though we start to see the problem that too many managed tests impose special maintainability challenges in themselves.

Arguably not all developers use such management and execution frameworks though; and even if they do, they may cast not all tests in them (the question of if they ought to do this is outside the scope of this paper). Instead, in particular during early stages of development, there is a chance that they interleave testing code with code that implements the business logics, just to see if it works. Similarly, in interactive applications, there is the possibility that there are no dedicated driver components, but that the respective tests are performed manually. In other words, developers may, in addition to writing managed test cases, also "play around" with the code. These unmanaged program executions tend to evolve along with the code, and often rely on external state (databases, configuration files) or user interactions that are not made explicit and hence hinder reproduction of the respective test cases.

Several methods that make such tests persistent have been proposed. These methods work by repeatedly recording program state (i.e., stack and heap) and/or events (e.g., interactive user input) [1, 9, 10, 5]. This repeated recording during program execution induces run-time overhead.

In previous work [6] we proposed a method that does not induce any run-time overhead: it captures program state only when a failure occurs and not before. The goal is to reproduce the failure by invoking the routines that are on the stack at the time of the failure. Since only the failure state is captured, the routines are invoked from this state, rather than from the states from which they were originally invoked. The problem with this approach is that state possibly changes in-between routine invocation and the moment of failure—invoking routines with the failure state may not reproduce the failure.

This paper investigates the effectiveness of our approach through experimental evaluation: despite its potential problem our technique reproduces 90% of all failures; capturing state at invocation time increases reproducibility by 4 percent points.

Problem We empirically study if zero-overhead test case extraction is effective. We analyze if test cases that are extracted from a failure state—rather than from a state that occurred *before* a failure state—can reproduce the failure.

*This work was supported by the FhG Internal Programs under Grant No. Attract 692166.

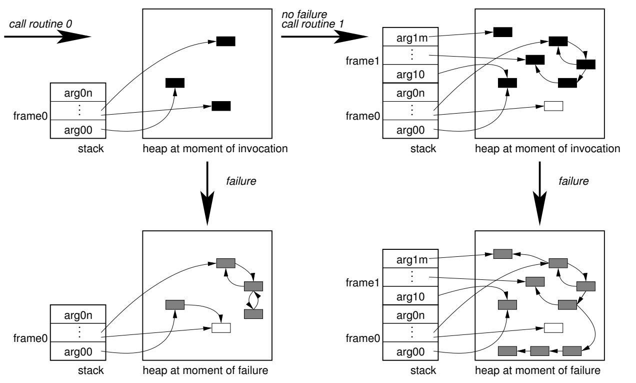

Figure 1. Evolution of stack and heap. Top row: moment of routine invocation (also content of shadow structures). Bottom row: moment of failure in routine 0 (left) or 1 (right).

Solution We asked 19 groups of 2-4 students each to implement one of two projects in an Eiffel IDE that supports automatic test case extraction. An analysis of the log files of the IDE provided evidence as to how many failures are reproducible. The results have been sketched above.

Contribution While we had described our test case extraction tool prior to our experiment [6], we did not know how effective it was. We are not aware of any studies that investigate the quality of zero-overhead test case extraction. §4 presents some related studies that do not discuss zero-overhead test case extraction but do consider an efficient and effective technique called second-chance. Our experiments show that zero-overhead extraction is highly effective and that a hybrid approach that combines it with Second Chance is a very promising approach to test case extraction.

Overview §2 provides the necessary background for test case extraction. §3 describes our experiment, presents the results, analyzes them, and highlights the threats to validity. §4 puts our work in context, and §5 concludes.

2 Background

This section describes the two test case extraction variants compared in this paper: invocation-state extraction [1], which invokes routines from the state that they were originally invoked from; and failure-state extraction [6], which invokes routines from the state when the failure occurred.

2.1 Invocation-State Extraction

The runtime system of a computer system maintains a stack and a heap. The stack consists of frames that are pushed whenever a routine is invoked. When a routine returns control to the calling routine, the topmost frame is removed. Frames include the address of the code, arguments (including the object on which the routine is to be executed), values or pointers to values of local variables, and a return address. The heap consists of all dynamically created data structures. Heap elements can be referenced by stack elements, and we will consider only those parts of the heap that are indeed referenced by a stack element. As an example, the upper left part of Figure 1 shows the stack at the time the main routine (routine 0) is invoked. In the example, this routine takes n arguments, or $n + 1$ arguments if we include the object on which routine 0 is executed.

The idea of run-time test extraction is to maintain a copy of the stack, a so-called shadow, by taking a snapshot of the respective frame (not the entire stack) whenever a routine is invoked. When a routine completes execution, the frame is popped, and its shadow deleted. Maintaining the shadows is the major reason for the overhead of test case extraction: upon every routine invocation, we essentially have to mimic what the highly optimized runtime system does. In addition, whenever a routine is invoked, we have to maintain a copy of those elements of the heap that are referenced by the corresponding stack frame. This can be achieved by deep or shallow copying. Several variants that differ in how much state they copy have been proposed by Artzi et al. [1].

Note that implementations of this approach take a snapshot of a stack frame when a routine is invoked, as opposed to when the failure occurs. We hence refer to this kind of test case extraction as **invocation-state extraction**.

In Figure 1, the upper row shows on its left the heap at the moment of invoking routine 0, and thus the corresponding shadow. On the right, we see the state of the heap at the moment of invoking routine 1. The black heap elements are those that are referenced by the shadow stack frame for routine 1. Note that on the right, there is one element that is referenced both by an argument of routine 0 and an argument of routine 1, and it might well be that this element was altered in-between the invocations of routines 0 and 1. This does not pose any problem, however, because the shadow that corresponds to routine 0 is a snapshot of this element at the moment of invoking routine 0—exactly the state that is depicted on the left.

If a routine does not terminate normally (e.g, by throwing an exception) all existing shadows (i.e., as many as there are frames on the stack) together with their copies of the relevant heap elements are serialized. One, some or all shadow frames are serialized: all relevant objects in one shadow frame (i.e., routine arguments) are turned into a piece of code that can directly be used to reconstruct these objects. A test deserializes the objects and invokes the routine to which the frame corresponds. One might think that test extraction could be done by simply serializing any element of the shadow stack. This is in general not the case, however.

Interactive applications depend on (non-deterministic) external events that are sent to the program by the operating system, external processes, or an interactive user. These events pose a problem for state-based test case extraction (all test extraction methods studied in this paper are state-based), because state-based test cases are unable to represent and hence re-trigger these events. The test extractor increases the chances to reproduce a failure by creating a test out of each shadow frame. There is a chance that the upper elements of the stack correspond to routine invocations that did not interact with external events, and are hence more likely to lead to tests that reproduce the failure. Generally speaking, upper-level frames will lead to tests that correspond to unit tests, whereas lower-level frames will lead to tests that correspond to integration or system-level tests.

To reduce the run-time overhead of invocation-state extraction, Artzi et al. proposed an optimization called Second Chance [1]. It improves the performance of invocation-state extraction, but depends on a failure to occur at least twice. Initially, no shadow stack frames are saved at all. When a failure occurs for the first time, the extractor marks the routines on the stack at this moment as observed. From this point onward, whenever the program invokes an observed routine, the extractor saves a shadow stack frame for it. When the failure occurs a second time, just as with normal invocation-state extraction, the extractor creates test cases using the shadow stack. Second Chance extractors hence record considerably fewer routine invocations. Artzi et al. [1] showed that this significantly reduces the overhead of invocation-state extraction. It does require a failure to occur twice though.

The performance overhead of invocation-state extraction is considerable. With full monitoring and deep copying enabled, Artzi et al. [1] found it to induce 12,000% to >638,000% overhead over normal execution with deep copying. In that same study, monitoring only a subset of the routines and relaxing the deep-copy requirement reduces the overhead to roughly 15-65%. Second Chance reduces the overhead to 0-2%.

2.2 Failure-State Extraction

The approach we proposed earlier [6] does not maintain a shadow at all, thus reducing run-time overhead to zero. In this approach, the extractor waits for an exception to occur. When this happens, it creates test cases from the current state of the stack and heap only. Since this kind of extractor does not need to maintain a shadow stack, it does not slow down the execution of the program under test. We refer to this approach as **failure-state extraction**.

Failure-state extraction requires some control over the runtime system: a) it needs to be notified when an exception is thrown and b) it must access stack and heap when notified. We propose to use test extraction during development, when developers run their programs via a debugger. Debuggers provide everything failure-state extraction requires. They stop in the event of an exception and display the state of both stack and heap. To implement a failure-state extractor one can change the debugger so that it notifies the extractor when an exception happens and then have the extractor serialize stack and heap using the facilities of the debugger to display stack and heap.

Zero overhead comes at a potential cost, however. One problem is that routines that are (transitively) invoked by a routine r can alter the heap elements that correspond to the arguments of that routine r. Reconsider Figure 1, top right, that shows the heap at the time of invoking routine 1. We have seen that there is one object referenced by both the first and the second frame. Assuming that this object has changed in-between the invocations of routine 0 and 1, and assuming that the failure occurs directly after starting to execute routine 1, the serialization of the bottom frame would yield an argument that does not correspond to the original object that was provided as argument to routine 0.

Moreover, even if we consider only one frame at a time, the heap elements pointed to by stack frames *at the time of failure* are likely to differ from the heap elements pointed to *at the time of invoking the routine*. The problem is, once again, that routine executions can alter elements on the heap, and these heap elements may be referenced by the stack. Consider the bottom left of Figure 1 which depicts the moment of failure during execution of routine 0. The grey elements have been created or modified since this routine's invocation. A failure-state test case extractor realizes that routine 0 has crashed and serializes the current stack frame, including a deep or shallow copy of its reference to the grey elements on the heap (the white element is of course also copied). Executing the generated test on the grey rather than the original objects with grey rather than the original objects may or may not lead to the failure that triggered test case extraction. Similarly, as more frames are pushed on the stack and more routines are executed, many heap objects may, at the time of failure, differ from the objects at the time of routine invocation. Figure 1, bottom right, shows this for routine 1 with m arguments. The generated tests from stack and heap at the moment of failure may or may not be able to reproduce the failure—some tests will not lead to a failure at all, some will lead to a different failure, and some will require external input.

The failure-state extractor invokes a routine using the failure state, and not the state from the routines original invocation. The failure state is only an approximation of the invocation state. How often the difference between these two states is relevant for test case extraction is the subject of this paper.

2.3 Exceptions Modifying Control Flow

Test extractors produce tests when a failure in the form of an exception occurs. However, programs also use exceptions to explicitly redirect control flow *if there is no failure*. Since tests that reproduce this kind of exceptions do not reveal a failure, a test case extractor should not create tests for it. A test case extractor can disambiguate the two kinds of exceptions based on the fact that an exception which signals a failure is not handled (and thus crashes the application), while an exception which simply redirects control flow is handled.

This approach does not work if programs handle failure-signaling exceptions to fail gracefully. The extractor should extract tests for such exceptions, but does not. To extract tests for handled failure-signaling exceptions, that we assume to be infrequent, the developer needs to mark the places in the code where the programs handles such exceptions (e.g., via a routine call, which the test extractor recognizes and intercepts at run time). The programs studied in §3 did not use exceptions to redirect control flow, hence our test extractor does not need to be able to disambiguate.

2.4 Reproducibility

This paper empirically studies how well failure-state extraction reproduces failures. Recall that our setting is based on a programmer who "plays around" with his code by writing ad-hoc tests with ad-hoc external data sources and ad-hoc user interactions. At the time of a failure, we want to generate a managed (e.g., xUnit-style) test case that reproduces the failure and that can be used for regression testing. As explained, the extractor generates one test case from each frame on the stack, but not necessarily every test case will actually reproduce the failure. We consider two perspectives. We say a *test case* is *reproducing* if it can reproduce the failure that led to its generation. Conversely, we say a *failure* is *reproducible* if there is at least one generated test case that can reproduce this failure. Arguably, for a developer it is important that a test extractor reproduces as many failures as possible. It seems less relevant whether every extracted test case is reproducing.

In order to assess the reproducibility of a failure, we first need to define the notion of failure. Failures are observable differences between expected and actual behaviors. In our context, failures come in the form of exceptions. Exceptions that signal runtime problems can either be contract violations (differences with the explicitly stated intended behavior) or other uncaught runtime problems that, because they usually make the system halt, are likely not intended either. Exceptions of this latter kind include null pointer exceptions, divisions by zero, etc.

For reasons that will become apparent later, we want to be able to distinguish between failures. A *unique failure* is determined by the type of exception together with the location in the program code where it was thrown. Assume that a programmer triggers three failures while "playing around" with the code. If two of these failures occur at the same line of code and generate identical exceptions but the third is different, there are two unique failures. In terms of test case extraction, ideally all tests extracted from the stack frames would lead to the same unique failure.

For reasons of expediency, we will only consider the reproducibility of unique failures in our experiment. This of course leads to the question if the abstraction is not too coarse. After all, we are interested in test cases for debugging purposes, not the failures themselves. As we will see in §3.5, it is adequate for our purposes.

3 Experiment

We set out to answer the question of how many failures can be reproduced by test case extracted from a failure state. If a large proportion can be shown to be reproducible, this would be evidence that the run-time overhead of maintaining a shadow stack is not necessary.

3.1 Experimental Setup

We conducted a study with 59 6th semester students from ETH Zurich who took the *Software Engineering* class in 2008. The students were divided into 19 groups of 2-4 students. 10 groups had to complete an assignment involving a geneaology tree, and 9 different groups had to complete an assignment involving V-Cards.[1]

1. The goal of the genealogy assignment was to implement a genealogy database. The database was to be searched for properties such as closest common ancestor. The students had to write a library, and they were provided with an application that used this library.

2. The goal of the other assignment was to implement the v-card standard for electronic addresses. Students were required to write both a library and an application that uses this library.

On average, solutions to the genealogy assignment consisted of 3284 lines of code and 11 classes while solutions to the v-card assignment consisted on average of 3995 lines of code and 34 classes. Table 1 conveys relevant statistics. Students had one month to complete their assignment.

Both applications made use of external events as discussed in §2.1 through user interfaces and file access. Both assignments involved a text-based user interface where the

[1]Assignments available from `http://se.ethz.ch/people/leitner/as.pdf`.

Table 1. Statistics for project submissions

	Genealogy		V-Card	
	LOC	# Classes	LOC	# Classes
min	2302	5	2998	32
avg	3284	11	3995	34
med	3183	8	3872	33
max	4886	26	5301	37
stdev	822	7	858.6	2

Table 2. Statistics for original (top) data and its considered subset (bottom)

original data					
criterion	min	avg	med	max	stdev
per individual failure (total 714)					
# extracted tests	0	2.3	2	12	2.3
failure stack size	2	8.1	6	25	3.8
per unique failure (total 319)					
# individual failures	1	2.2	1	17	2.3
extracted tests	0	5.2	4	40	5.6
considered subset					
criterion	min	avg	med	max	stdev
per individual failure (total 411)					
# extracted tests	0	2.5	3	12	2.2
failure stack size	2	8.4	6	18	3.0
per unique failure (total 189)					
# individual failures	1	2.2	1	17	2.3
extracted tests	0	5.4	4	40	5.8

user interactively inputs text and both assignments contained features that accessed the file system: the v-card assignment included functionality to read and write v-cards to the file system, and the genealogy assignment included the feature to load commands from a batch file.

We gave the students an integrated development environment, CDD EiffelStudio, which is equipped with a failure-state extractor, and we included a logging mechanism for information relevant to the experiment, including failures, extracted test cases, and information on whether or not tests were reproducing. Students were required and agreed to commit these logs together with their program to a version control system.

Overall, 714 individual failures were logged. This corresponds to 319 unique failures as defined in §2.4. 1664 tests were extracted, corresponding to an average of 2.3 tests per individual failure. The failures aggregated as one unique failure caused on average the extraction of 5.2 test cases. Relevant statistics are shown in Table 2, top. The stack size is not the same as the number of extracted test cases because the test case extractor ignores four classes of stack frames.

Table 3. Reproduction results for original data (depth-limited extractor)

	original data	post-proc. subset
ind. failures	714	411
unique failures	319	189
– contract violations	209 (66%)	125 (66%)
– other exceptions	110 (34%)	64 (34%)
extracted tests	1664	1024
reproducing tests	390 (23%)	271 (27%)
repr. unique failures	139 (44%)	92 (49%)
– contract violations	85 (61%)	58 (63%)
– other exceptions	54 (39%)	34 (37%)

1. It ignores frames of calls from Eiffel to C, because the test extractor cannot serialize the memory which these C functions use.
2. It ignores frames of agents—roughly, the Eiffel counterpart of delegates in C#, because our implementation cannot handle them.
3. It ignores frames of routines from the base libraries, since we did not expect developers to be interested in such test cases.
4. It did not add redundant test cases. If (due to a previous extraction) an identical test already existed, no new test was generated.

An initial analysis revealed that 139 (or 44%) of the unique failures were reproducible and that 390 (or 23%) of the tests were reproducing. The middle column of Table 3 summarizes these numbers; the rows relating to contracts as well as the rightmost column will be discussed below.

These rather low numbers made us analyze the results. We realized that we had been too optimistic in terms of the necessary depth of serializing heap structures when the failure occurred. We had worked with a maximum depth of five consecutive links, and this turned out to be too low. In the genealogy assignment, for instance, students were happy to implement genealogies with more than four generations. Our analysis indicated that we could expect much better results with more deeply serialized structures.

3.2 Failure-state Extraction without Depth Limit

Consequently, we decided to manually regenerate the failures and have the IDE re-extract the test cases, this time with a sufficient depth of serialized heap structures.[2] Note

[2]We distinguish between a "regenerated" failure, for which we manually re-ran the program in order to apply the unlimited depth test extractor and a "reproduced" failure, which extracted test cases were able to reproduce. Regenerated failures can but need not necessarily be reproducible.

Table 4. Statistics for manually regenerated failures (unlimited depth)

criterion	min	avg	med	max	stdev
per individual failure (total 209)					
# extracted tests	0	2.5	3	11	1.8
failure stack size	2	6.8	6	17	2.2
per unique failure (total 97)					
# individual failures	0	2.1	2	13	1.8
extracted tests	0	5.5	5	17	3.9

that like the limited-depth extractor, this unlimited extractor does not introduce any overhead during regular program execution, but that it does of course introduce more overhead at extraction time. Since the manual regeneration turned out to be extremely labor-intensive, we could not post process all failures. We restricted ourselves to 7 out of the 19 groups and picked those groups that exhibited the most unique failures. 4 out of these 7 groups had implemented the genealogy assignment and 3 had implemented the v-card assignment. These groups had triggered 411 out of the entirety of 714 individual failures. Correspondingly, they had triggered 189 out of all 319 unique failures.

There is some evidence that the 7 groups are representative of all 19 (Table 3, rightmost column). In terms of reproducibility, using the depth-limited test case extractor, 49% of all unique failures from the 7 groups are reproducible. This is comparable to the 44% of reproducible failures from all groups. Conversely, the percentage of reproducing tests, 27%, seems sufficiently close to the original 23%. Moreover, the statistics for stack sizes, number of extracted tests, and the number of individual failures that correspond to one unique failure are very close to the statistics of the original data set, as shown in Table 2, bottom.

Recall from §2.4 that our primary interest is in determining how many failures can be reproduced rather than in how many test cases reproduce the failure from which they were extracted. This allowed us to reduce the number of failures we had to manually regenerate: it is safe to assume that all failures reproducible with the depth-limited extractor (92 out of 189) are also reproducible with the unlimited extractor. As a consequence, we only needed to manually regenerate those failures that were not reproducible with the depth-limited extractor—because of our assumption we already knew that they were reproducible with the unlimited extractor. This left only 97 unique failures that we had to manually regenerate. These correspond to 210 individual failures. We proceeded as follows.

1. For every unique failure, we determined its first occurrence in the log files.
2. We retrieved the version of the program that was committed to the version control system as close as possible to the date and time of the occurrence of this failure.
3. We ran the program and provided input that we thought would reproduce the failure. This task was rendered feasible with the information from the stack trace of the failure that we were trying to regenerate.
4. When we were able to manually regenerate a failure, we ensured that the failure was similar to the one we were after by applying it to the depth-limited failure-state extractor. We considered the failures equal only if the depth-limited failure-state extractor indeed failed to extract reproducing test cases for the regenerated failure. Otherwise, we searched for different inputs until we could regenerate the exception.

Table 5. Regeneration results for post-processed data

	manually regenerated	not regen.	both
ind. failures	209	201	410
unique failures	97	92	189
extracted tests	530	602	1132
reproducing tests	148	271	419
% repr. tests	28	45	37
repr. unique failures	78	92	170
% repr. unique f.	80	100	90

Each previously non-reproducible failure was regenerated (at least) twice: once applying the limited extractor, and once applying the unlimited extractor. The regeneration of non-reproducible failures resulted in a total of 209 individual failures and 530 new test cases out of which 148, or 28%, reproduced the failure they were extracted from, with an average 2.5 tests extracted per individual failure and an average 5.5 tests available per unique failure. Table 4 shows relevant statistics for the manually regenerated failures and the corresponding tests. Note that it does not include the failures we did not have to regenerate because there was no problem with the depth limit when copying heap structures (see Table 5 below). Also note that the minimum number of failures per unique failure is zero. This captures the fact that we did not succeed in regenerating all unique failures, which was the case for 12 of them. We counted these unique failures as non-reproducible, thus introducing a slight negative bias against failure-state extraction.

We have argued that for developers it is important whether a failure is reproduced, not by how many tests it is reproduced. We found that out of the 97 failures that we had to regenerate (because of the limited extractor), 78, or 80%, were reproducible with the failure-state extractor, i.e.,

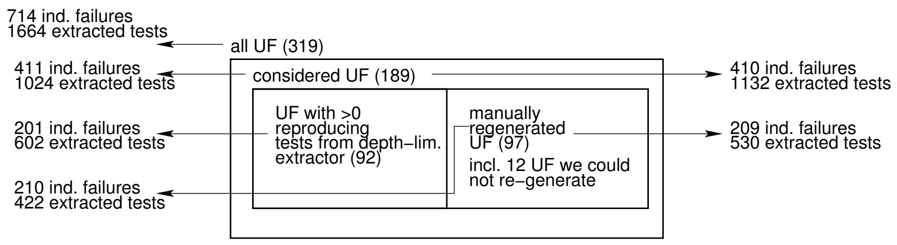

Figure 2. Distribution of unique failures (UF). Left: original, non-post-processed values (depth-limited extraction). Right: values after post-processing (unlimited depth).

there was at least one extracted test case that could reproduce it. The second column of Table 5 shows relevant data (it is pure coincidence that we manually regenerated 209 individual failures, a number that is close to the 210 individual failures triggered with the depth-limited extractor).

Figure 2 shows how the unique failures relate to each other. Numbers on the left relate to the original data obtained with the limited-depth test case extractor. Numbers to the right relate to the subset that we post-processed with the depth-limited extractor.

Note that the above considers only those failures that were irreproducible with the depth-limited failure-state extractor. Since we assume that any failure reproduced by the limited failure-state extractor is also reproduced by the unlimited failure-state extractor, we can combine the numbers to learn how the unlimited failure-state extractor performs on all failures of the selected 7 groups: 92 failures were reproduced by the limited extractor and 78 by the unlimited extractor, leaving 19 non-reproduced by either (Table 5, third column). **Hence, in our experiment that is restricted to 7 out of the 19 groups of student programmers, the unlimited failure-state extractor reproduced 170 out of 189 unique failures, or 90%.**

3.3 Invocation-state Extraction

To find out how much more effective invocation-state test extraction is when compared to the far more efficient failure-state extraction, we iterated on the approach described in § 3.2. As discussed above, we manually regenerated the unique failures for which the limited depth extractor did not extract a reproducing test (making sure the exception is similar in nature to the original one) and then used an invocation-state extractor to produce new test cases. For fair comparison, we considered the same 7 out of 19 groups of student programmers. We assumed (1) that any failure reproducible by the unlimited failure-state extractor was also reproducible by the invocation-state extractor and, once more, (2) that any failure reproducible by the limited failure-state extractor was also reproducible by the unlimited failure-state extractor. As a consequence, we used the invocation-state extractor to reproduce only those 19 unique failures that were not reproducible with the unlimited failure-state extractor. The right column of Table 6 shows the results. **Applying the invocation-state extractor to these remaining 19 unique failures rendered another 7 reproducible, or 3.7 percent points.** Overall, test case extraction could hence reproduce 177 unique failures, or 94%. Incidentally, the 12 missing unique failures are exactly those that we could not reproduce manually, and for which the invocation-state extractor could equally not be applied. For these 12 unique failures, we do not know if invocation-state extraction, failure-state extraction, both, or none would be able to generate reproducing tests.

Table 6. Regeneration results by type of exception (post-processed)

	failure-state extraction	invocation-state extraction
unique failures	189	189
– contract violations	125 (66%)	125 (66%)
– other exceptions	64 (34%)	64 (34%)
repr. unique failures	170 (90%)	177 (94%)
– contract violations	112 (90%)	118 (94%)
– other exceptions	58 (91%)	59 (92%)

3.4 Contract vs. other Runtime Failures

The failures considered in this paper stem from programs annotated with contracts. The following discusses whether the results can be generalized to programs that do not contain such specification annotations. The failures studied can be divided into two groups. *Contract violations* correspond to failures that stem from contract violations (i.e. precondition violations, assert violations). Such failures only occur in programs with assertion annotations. *Other violations* stem from other types of runtime-violations (i.e. null pointer dereferences). Such failures occur also in programs without assertion annotations.

Table 6 shows the reproducibility of contract violations and violations due to normal exception separately. The 189 unique failures processed with the unlimited failure-state extractor can be divided into 125 contract violations and 64 other violations. The failure-state extractor reproduced 90% of the contract violations and 91% of the other violations (see Table 3 for respective statistics of the original data set). The invocation-state extractor reproduced 94% of all contract violations and also 92% of all other violations. For both test case extractors the difference between contract failures and other failures is less than 3%. **In our experiment, there is no difference between failures that are a result of contract violations and other exceptions.**

3.5 Unique Failures

Unique failures (§2.4) group exceptions that (1) indicate the same runtime problem (e.g., precondition violation violation or division-by-zero) and (2) have been triggered by the same line of code. The idea is that unique failures group failures with similar properties. Two failures stemming from different faults are likely to have different properties. When grouped by kind and location, it is technically possible that two failures that stem from distinct faults are grouped together. For example, assume a routine a which takes an argument, that must not be null. Two distinct routines may each invoke a with null as argument. There are two faults, one in each calling routine, but the two failures are put into the same group. The fact that the source code from our experiment was annotated with contracts in the form of preconditions, postconditions and invariants lowers the chances of such a scenario. Routines that do not accept, say, null arguments often state this in their precondition. We considered precondition violations to be triggered from the calling routine. Hence the two failures from the above example would end up in two distinct groups.

While contracts help they do not remove the chance that two failures with dissimilar properties are put into the same group. We studied the relevance of this potential issue by considering the following criteria. Unique exceptions are equivalence classes. In our context, two members of the same equivalence class should lead to qualitatively similar test cases. To capture this fuzzy notion of qualitative similarity, we considered the following criteria. We consider two tests to be qualitatively similar if (1) the failure stacks of the corresponding individual exceptions have the same height; and (2) two frames of the failure stacks of the corresponding individual exceptions at the same vertical position correspond to the same routine—routine arguments are not considered; and (3) the tests generated from two frames at the same vertical position in the failure stack of the corresponding individual exceptions either both pass or fail.

These criteria capture the intuition of similar tests. Criteria (1) and (2) approximate that the control flow of the program execution until the failures are similar, and criterion (3) very roughly approximates that the data flow problems in the two executions are similar.

In our experiments, a large majority—277 out of 319, or 87%—of the unique exceptions satisfies these three criteria (the first and second criterion together are satisfied by 89%). **In other words, there is some evidence that in our experiment, the kind and location of a raised exception are good determinants for equivalence classes because these two criteria relate to the nature of the generated test cases rather than the exceptions themselves.**

3.6 Interpretation and Consequences

Out of the 189 considered unique failures, we could not manually reproduce 12. The unlimited failure-state extractor generated 170 reproducing tests. Invocation-state extraction led to another 7 reproducing tests. Our results do not depend on whether or not contract violations or other exceptions occur. Due to the problem with limited depth of copying heap structures, we had to group individual failures into classes, so-called unique failures, and we have argued that the simple definition of these classes can be justified. Failure-state test case extraction is both effective and efficient.

We conjecture but do not know for sure if the 12 unique failures that we could not regenerate—and for which we do not know if any of the extractors would be able to generate reproducing tests—were not reproducible because of external events.

Modulo the threats to validity discussed below our experiments clearly indicate that the huge run-time overhead of invocation-state extraction is not necessary. A mere additional 4 percent points of reproducible failures will, in most situations, not be worth several orders of magnitude of slow-down.

A hybrid approach, based on a combination of failure-state extraction and Second Chance (§2.1) combines the benefits of both approach, however. If a failure occurs, the failure-state extractor generates test cases (without imposing run-time overhead). The IDE automatically executes the new test cases. If the failure is reproduced by at least one test case nothing else happens. If the failure is not reproducible, the routines that were on the stack at the time of the failure are marked as observed. The IDE turns on the invocation-state extractor, but restricts it to routines marked as observed. Whenever a routine marked as observed is entered, its relevant state is captured. For other routines nothing happens. If the failure that was not reproducible with the failure-state extractor occurs a second time, the extractor uses the previously saved invocation states to generate test cases. The resulting test cases are identical to the ones generated by a pure invocation-state extractor, but because of Second Chance the extractor requires much less overhead. As soon as a failure is reproduced via a test case, the routines previously marked as observed are unmarked. The developer may also unmark routines manually.

3.7 Threats to Validity

The most obvious threat to the validity of a generalization of our results lies in the restricted class of programs, both in terms of their application domain and the language that we used. State-based test extraction is (as discussed in §2.1) restricted by external events. We do not know if our results generalize to programs with more frequently occurring external events. Another apparent threat is the size of the programs used in the study. The two programs were medium sized, and while they are not trivial, they are not comparable to large multi-million line industrial projects. The number and kind of failures occurring during development of a small project may not be representative for large projects.

Two threats stem from our manual regeneration of failures. First, we limited manual regeneration to 7 out of 19 groups. The failures of these groups might not be representative of the failures of all 19. We assume that they are, because (as shown in Table 3) the properties of the unprocessed data of the 7 groups is similar to the properties of all 19 groups. Second, the failures we regenerated might be different from the original ones. To address this concern, we thoroughly compared the failures we regenerated to the original ones, as described in Section 3.2.

The conclusions drawn in this paper assume that for a developer it is sufficient when a failure is reproduced even by only a single test case. Not all test cases are equally useful to the developer, however. Failures reproduced only by tests useless to the developer should not be counted as reproduced. For example, a test case might reproduce the failure, but not execute the code containing the fault. A test case could invoke a routine with a null argument, but this routine is not supposed to handle null arguments. Since the present study considers contracted programs, it is not affected by this problem as much as if it would consider programs without contracts. In the studied programs, routines not accepting null arguments often state this in their precondition, which is checked before the routine is executed. To avoid extracting useless tests our extractor did not create test cases for routines that had their precondition violated. In previous work [6], we describe how contracts influence extraction in detail.

4 Related Work

Much research has gone into the direction of using capture and replay, which tries to replay program runs in general. Most capture and replay techniques are based on checkpointing the state of the system at certain intervals and recording a log of non-deterministic events in-between the intervals [3, 4, 7, 11]. These approaches try to replay arbitrary executions. Test case extraction, as presented in this paper, focuses on the failing executions only.

Clause and Orso recently presented a system, which in addition to the checkpointing and log-recording, minimizes the log [2]. Test case extraction implicitly minimized the number of instructions to replay: each extracted test only executes one routine. Current extractors do not minimize the state stored in test cases. We believe that approaches like *Delta Debugging* [12] will yield promising results.

A novel capture and replay approach that logs the interaction of software components at user-defined borders has been proposed independently by Orso et al. [9, 8] (Selective Capture and Replay), Ernst [10] (Test Factoring), and Elbaum [5] (Test Carving). The technique works as follows: a program is divided into two parts. During capture the interactions (routine calls etc.) that cross the boundary are logged. During replay one part of the program is replaced by mock objects which are hard-coded to behave according to the previous recording. The advantage of this method lies in the freely definable border between the two parts. Most programs have parts that interact little and parts that interact frequently. By wisely choosing the border the amount to record can be reduced significantly. We believe that this work is orthogonal to ours. If sources of non-determinism are found to be a problem for a given application, they can be mocked out using Selective Capture and Replay, the rest of the state can be extracted as proposed by our method.

ReCrash [1] which is based on our work on test case extraction, observes production runs of programs and extracts test cases in the case of a failure. Similar to our approach, they extract one test case per routine on the stack. The authors of ReCrash experiment with a number of variations of the algorithm. The variations differ in how much invocation-state is captured and how many routines are observed. The more often and the more state is captured at routine invocation time, the more reliable, but also the slower a variant becomes. They measured the slow down on two popular open source applications (SVNKit and Eclipse). Full failure-state extraction induces 12,000 to >638,000% overhead. A more efficient variant that makes a shallow copy of memory and depends on static analysis to detected used fields reduces the overhead to 13% to 60%. They also introduce a mechanism called *Second Chance*—an extraction mechanism that requires a failure to occur at least two times. In such cases the mechanism is both efficient and effective. In contrast to our approach all of their methods induce some run-time overhead, but they require less control over the stack at the time of the failure.

5 Conclusions and Future Work

Our work is based on the observation that developers test their programs by "playing around" with it during development. These implicit test cases are not managed and can not be used for regression testing. Since they may rely on external data sources (user interfaces, configuration files, data bases), reproducibility is a serious issue.

The problem of turning implicit into managed tests has been tackled at various levels. External input can be intercepted and serialized (capture-and-replay), and information on a program's execution state can be recorded and transformed into test cases. The work described in this paper is based on state-based extraction. There are several ways of implementing this idea. One is to maintain a shadow copy of those frames that are pushed on the stack of the runtime system, together with a copy of the heap data structures that are referenced by the stack. When a failure occurs, these shadow frames are turned into test cases. One problem with this approach is the huge run-time overhead. This run-time overhead can be reduced to almost zero by waiting for a second occurrence of the failure. After the first failure, only relevant routines are monitored. However, this obviously requires a failure to occur twice. In earlier work, we proposed failure-state extraction where methods are invoked from the state captured at the time of the failure and not with their original invocation state. This extraction induces zero runtime overhead, but the state it invokes methods from is not necessarily able to reproduce the failure it aims to reproduce. This paper sets out to assess the effectiveness of our approach.

The student experiment described in this paper showed that zero-overhead failure-state extraction is capable of reproducing 90% of the (unique) failures that occurred. Invocation-state extraction leads to another 4 percent points of reproducible failures at the cost of a significant run-time overhead. Modulo the threats to validity that we carefully discussed, in particular the influence of more frequently occurring external events, our results suggest that the run-time overhead of invocation-state extraction cannot be justified. Note that the application domain of invocation-state and failure-state extraction may be slightly different however: since failure-state extraction requires access to the program stack at the time of a failure (e.g. via a debugger), it may be more suited in a development context whereas invocation-state extraction may be more useful when the software is deployed. Moreover, our results suggest that the combination of failure-state extraction with Second Chance appears both efficient and effective, and we have hence implemented it in our IDE.

At least two important questions remain to be studied. How useful are extracted test cases for debugging a fault? How useful are extracted test cases for regression testing? Since the stack frames at the moment of routine invocation correspond to argument values while failure-state frames correspond to transformations of these values, we would conjecture that invocation-state test extraction leads to tests that are easier to understand.

References

[1] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, Paphos, Cyprus, July 9–11, 2008.

[2] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[3] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71, New York, NY, USA, 2006. ACM.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[5] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264, New York, NY, USA, 2006. ACM.

[6] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test-cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, September 2007.

[7] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[8] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *Proceedings of the Fourth International ICSE Workshop on Dynamic Analysis (WODA 2006)*, pages 3–10, May 2006.

[9] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, may 2005.

[10] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123, New York, NY, USA, 2005. ACM.

[11] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, 2003.

[12] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, Feb. 2002.