Reduce First, Debug Later

Alexander Elyasov, Wishnu Prasetya, Jurriaan Hage, Andreas Nikas
Department of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
{A.Elyasov, S.W.B.Prasetya, J.Hage}@uu.nl, a.nikas@students.uu.nl

ABSTRACT

The delta debugging minimization algorithm (*ddmin*) provides an efficient procedure for the simplification of failing test-cases. Despite its contribution towards the automation of debugging, *ddmin* still requires a significant number of iterations to complete. The delta debugging (DD) search space can be narrowed down by providing the test-case circumstances that are most likely relevant to the occurred failure. This paper proposes a novel approach to the problem of failure simplification consisting of two consecutive phases: 1) failure reduction by rewriting (performed offline), and 2) DD invocation (performed online). In the best case scenario, the reduction phase may already deliver a simplified failure, otherwise, it potentially supplies DD with extra information about where to look for the failure. The proposed solution has been prototyped as a web application debugging tool, which was evaluated on a shopping cart web application — *Flex Store*. The evaluation shows an improvement of the DD execution time if the offline reduction over-approximates the failure.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, diagnostics, testing tools, tracing*

General Terms

Verification, Experimentation

Keywords

Automated debugging, delta debugging, test-case simplification, rewriting

1. INTRODUCTION

The actual debugging process starts once the application failure has been witnessed. The sequence of program statements, system events or user steps that caused the application to crash is a crucial artifact for debugging, because it presumably provides a persistent procedure for reproducing the crash. With such a failing test-case in hand, the debugging process can be initialized and to some extent automated [16]. And, if the usefulness of the knowledge about the hypothetical error location in the program casts some doubts on the practicality of automated debugging [12], then the need of the failing trace is beyond question.

In automated debugging, failures are often revealed by means of randomly generated test-cases such as unit-tests [3]. In the case of testing in the field, log files, if available, play the role of failing test-cases when the application has crashed. The both ways for tracking down the failures often suffer from the problem of lengthy and cumbersome test-cases. Therefore, simplification of the failure revealing test-cases is certainly an essential step towards the debugging.

Minimizing Delta Debugging (*ddmin*) has been introduced by Zeller [16] as an effective and efficient procedure for simplification of the failing test-case by performing at most a quadratic number of tests. The practical applications of delta debugging (DD) include simplification of failing GNU C programs, and HTML pages as rendered by Mozilla. Another algorithm introduced by Zeller [16] is *dd* — a generalization of *ddmin* for failure isolation, which has the worst-case complexity similar to *ddmin*. A more recent study performed by Yu [15] has shown that the execution time of *dd* [16] for isolation failures in C programs can vary from 8 hours to 5 minutes, and so improvement is desirable.

In this paper, we investigate the applicability of minimizing delta debugging to the field of web applications, where the failure is usually caused by a sequence of user events triggered by means of interaction with the application. Whenever we refer to delta debugging, the minimization algorithm (*ddmin*) is assumed, unless otherwise specified. Thus far, we have not seen any attempt to apply *ddmin* for simplification of the failing sequence of user events.

To address the foreseen issue with the *ddmin* execution time, we propose a modified version of *ddmin*, which is extended with the reduction of the failing sequence prior to running DD. It is suggested to conduct minimization in two phases. During the first phase the failing sequence is subjected to reduction by means of rewriting with the set of rewrite-rules inferred from previously collected successful executions. The reduced sequence might, however, lose the ability to replicate the original failure or not even be executable at all. These problems can be alleviated by applying DD procedure to the complement of the reduced sequence in the original failing one. In the second phase, each initial DD run should contain the reduced sequence as the fixed

AST'14, May 31 – June 1, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2858-6/14/05...$15.00
http://dx.doi.org/10.1145/2593501.2593510

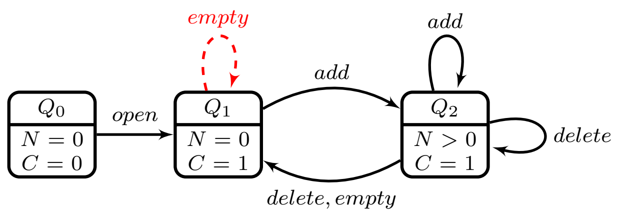

Figure 1: Shopping Cart Application Model

L=[event]	N	C
`init`	0	0
`open`	0	1
`add(1)`	1	1
`empty`	0	1
`empty`	0	1
`add(2)`	1	1
`add(3)`	2	1
`delete(3)`	1	1
`delete(2)`	0	1
`add(3)`	1	1
`add(2)`	2	1

Rule Patterns:
$[X(a)] \equiv \epsilon$
$[X(a); Y(a)] \equiv \epsilon$
$[X(a); Y(b)] \equiv [Y(b)]$
$[X(a); Y(b)] \equiv [Y(b); X(a)]$

Figure 2: Log File and Rewrite Rule Patterns

part and vary the complement until the failure is exhibited or the complement is exhausted.

The paper makes the following contributions:

- It proposes a novel approach of combining DD with offline reduction that can reduce the overall cost of DD minimization.
- The standard *ddmin* algorithm is adjusted to incorporate the reduction into its main flow.
- Both *ddmin*, and the new minimization algorithm, *rddmin* (*ddmin* with the prior reduction), are implemented in a web application automatic debugger, BAD, which currently supports Flash applications.
- The debugger is used to carry out a preliminary evaluation *ddmin* and *rddmin*.

The paper is structured as follows. Section 2 illustrates the approach on a model shopping cart application, and introduces the modified version of DD (*rddmin*). The web application debugger is presented and evaluated in Section 3. Related and future work is discussed in Section 4. The paper is concluded by Section 5.

2. HYBRID DEBUGGING APPROACH

2.1 Illustrative Example

To walk the reader through the details of the suggested debugging approach, we consider a simplified version of the Flex Store application, the discussion of which is postponed until Section 3.3. The hypothetical example used here is the application that only consists of the shopping cart, and is able to respond to the following user events:

- `open` the product catalog
- `add` item to the shopping cart
- `delete` item from the shopping cart
- `empty` content of the shopping cart.

The FSA in Figure 1 models the semantics of the Shopping Cart application. The state Q_0 is the initial state of the application. Internally the state of application is fully defined by two variables $\{N, C\}$, where N is the number of items in the shopping cart and C encodes whether the catalog is opened or not. The given application exhibits an error that will be observed as soon as the `empty` event is invoked from the state Q_1. The event sequence below provides an example of such a failing execution:

$$\begin{aligned}&\texttt{open} \to \texttt{add(1)} \to \texttt{empty} \to \texttt{add(2)} \to \texttt{add(3)} \to \\ &\quad \to \texttt{delete(2)} \to \texttt{delete(3)} \to \texttt{empty}.\end{aligned} \tag{1}$$

It consists of eight events and can serve as a test-case reproducing the failure. Obviously, not all events in this sequence are essential for the failure to occur. The following event sequence is the minimal subsequence of (1) that still produces the same failure:

$$\texttt{open} \to \texttt{empty}. \tag{2}$$

2.2 Reduction

In [5] we have proposed a two stage approach for reduction of the failing event sequence, that consists of 1) mining rewriting rules from the set of collected logs, and 2) applying these rules to the failing sequence with the purpose of sequence length reduction. The rewriting rules have the form $\tau_1 \equiv \tau_2$, which represents the following property: for all application states where both the event sequences τ_1 and τ_2 are executable, the final states after the execution of τ_1 and τ_2 from that state are equal as well.

2.2.1 *Executable Reduction*

The log file L displayed on the left side of Figure 2 presents a successful execution of the Shopping Cart Application before it contained any bugs. Each log entry records the invoked event and the application state, which is represented, in our case, by two variables N and C, after the event has occurred. The approach proposed in [5] introduces algebraic rule patterns, such as those in Figure 2, that are used for mining the rewrite rules from logs. Applying inference to the log L, we can obtain the following concrete rewrite rules:

$$[\texttt{add(i)}; \texttt{empty}] \equiv [\texttt{empty}] \tag{r1}$$
$$[\texttt{add(i)}; \texttt{delete(i)}] \equiv \epsilon \tag{r2}$$
$$[\texttt{empty}; \texttt{empty}] \equiv [\texttt{empty}] \tag{r3}$$
$$[\texttt{add(i)}; \texttt{add(j)}] \equiv [\texttt{add(j)}; \texttt{add(i)}] \tag{r4}$$

The second rule, for example, says that adding an item and removing it from the shopping cart does not have any effect on the application state at all. Applying the rules (r1)–(r4) to the event sequence (1) we can carry out the following reduction process:

$$\begin{aligned}(1) &\xRightarrow{(r1)} \texttt{open} \to \texttt{empty} \to \texttt{add(2)} \to \texttt{add(3)} \to \\ &\quad \to \texttt{delete(2)} \to \texttt{delete(3)} \to \texttt{empty} \\ &\xRightarrow{(r4)} \texttt{open} \to \texttt{empty} \to \texttt{add(3)} \to \texttt{add(2)} \to \\ &\quad \to \texttt{delete(2)} \to \texttt{delete(3)} \to \texttt{empty} \\ &\xRightarrow{(r2)} \texttt{open} \to \texttt{empty} \to \texttt{add(3)} \to \texttt{delete(3)} \to \\ &\quad \to \texttt{empty} \\ &\xRightarrow{(r2)} \texttt{open} \to \texttt{empty} \to \texttt{empty} \\ &\xRightarrow{(r3)} \texttt{open} \to \texttt{empty}\end{aligned}$$

At the end, this process produces exactly the same event sequence as (2). And we can even make sure that it still exhibits the failure by replaying each event in this small trace and observing the result.

2.2.2 *Non-executable Reduction*

Since the rules are *dynamically* inferred from the collected logs, some rules may be false candidates. Moreover, some rules may only be valid per given state abstraction. For example, imagine that the variable C has been excluded from the application state, because we decided, for some reason, that the state of the catalog is irrelevant for our purposes. This gives us a new rewrite-rule in addition to those known already, namely:

$$[\texttt{open}] \equiv \epsilon \qquad \text{(r5)}$$

It is valid for the given state abstraction but does actually not work. This rule allows us to reduce the sequence (2) further by removing the first event:

$$(2) \xRightarrow{(r5)} \texttt{empty} \qquad (2^*)$$

If we now tried to replay the sequence (2*), we would not even be able to successfully execute it and consequently would not reproduce the original failure. On the other hand, the sequence (2*) still contains some meaningful part of the shortest reproducible test-case (2). Can we repair this *re-produceability issue*? Our answer is: "**Yes, we can!**". How? By exploiting delta debugging.

2.3 Debugging

Delta debugging [16] has been known for years as a productive approach for failure simplification. Alternatively to the reduction approach discussed in Section 2.2, we could try to minimize the failing sequence (1) by means of DD. The whole process of stepwise simplification is described in Table 3(a). In short, DD tries to replay various subsequences of the original failing sequence, splitting it initially into two parts and gradually increasing the granularity until a smaller subsequence reproducing the failure is found. This process stops when the subsequence can not be reduced further without losing the ability to produce the failure. In this case, it is called *1-minimal* because each event in this sequence matters for the failure. Despite the complete automation of the debugging process and the speed of the search convergence, DD is an expensive algorithm to use in practice [15]. For example, the reduction in Table 3(a) requires 14 iterations. The overhead of one reduction step consists not only in the effort to re-execute the new subsequence, but it also requires the initial state of the system to be reconstructed. Depending on the concrete application domain it can be recompilation, data-base restore, or, as it was the case for our example, web-page reload.

By applying the reduction procedure in Section 2.2.2, we have managed to get a significant simplification, but unfortunately the resulting trace was not executable. We can employ DD to complement the reduced sequence (2*) with some extra events to make the whole sequence executable again. In order to do so, DD is applied to the difference between the original sequence and the reduced one. But the reduced sequence itself should always be included as a persistent part of the replayed sequence. The proposed stepwise simplification process in application to the failing sequence (1) is presented in Table 3(b). The highlighted column indicates the event(s) that should be the permanent part of the sequence under test, whereas DD is applied to the rest. At step #2 we generate an event sequence that is both executable and failure revealing. From now on, we can already apply a conventional DD to simplify that sequence further. As a result, taking into account the note under Table 3 about the number of actual iterations, we can conclude that the improved version of DD saved us four unnecessary iterations of the standard DD.

Algorithm 1: Minimizing Delta Debugging Algorithm Complemented with Reduction: $rddmin(\mathcal{E}, \mathcal{R})$

```
Input   : $\mathcal{E}$ is a failing events sequence,
          $\mathcal{R}$ is a reduction of $\mathcal{E}$
Require: $\mathcal{R} \subseteq \mathcal{E}$, Test($\mathcal{E}$) = ✘
Output  : minimal failure revealing subsequence of $\mathcal{E}$
if $|\mathcal{R}| = 0$ then
    return $ddmin(\mathcal{E})$
else if $Test(\mathcal{R})$ = ✘ then
    return $ddmin(\mathcal{R})$
else
    return $rddmin'(\mathcal{E}, \mathcal{R}, 2)$

Procedure $rddmin'(\mathcal{E}, \mathcal{R}, n)$
    $\mathcal{C} \longleftarrow \mathcal{E} \setminus \mathcal{R}$
    if $n < |\mathcal{C}|$ then
        $\{c\}_{i=0}^{n} \longleftarrow$ Partition($\mathcal{C}$, n)
        for $i \leftarrow 1$ to $n$ do
            if $Test(c_i \cup \mathcal{R})$ = ✘ then
                return $ddmin(c_i \cup \mathcal{R})$
        for $i \leftarrow 1$ to $n$ do
            if $Test((\mathcal{C} \setminus c_i) \cup \mathcal{R})$ = ✘ then
                return $ddmin((\mathcal{C} \setminus c_i) \cup \mathcal{R})$
        return $rddmin'(\mathcal{E}, \mathcal{R}, min(|\mathcal{C}|, 2n))$
    else
        return $ddmin(\mathcal{E})$

Procedure $ddmin(\mathcal{E})$        // as described in [16]
```

2.4 Formal Algorithm

This section presents a modified version of the Minimizing Delta Debugging algorithm (*ddmin*) from [16] extended here with the prior reduction of the failing sequence. If the set of events contributing to the failure can be approximately localized, this may speed up *ddmin* by letting it focus only

#	n	Δ	Events								R	M	T
0	$n = 1$	Δ_1	`o`	`a(1)`	`e`	`a(2)`	`a(3)`	`d(2)`	`d(3)`	`e`	✗		
1	$n = 2$	Δ_1	-	-	-	-	`a(3)`	`d(2)`	`d(3)`	`e`	?		1
2	$n = 2$	∇_1	`o`	`a(1)`	`e`	`a(2)`	-	-	-	-	✓		4
3	$n = 4$	Δ_1	-	-	`e`	`a(2)`	`a(3)`	`d(2)`	`d(3)`	`e`	?		1
4	$n = 4$	Δ_2	`o`	`a(1)`	-	-	`a(3)`	`d(2)`	`d(3)`	`e`	?		4
5	$n = 4$	Δ_3	`o`	`a(1)`	`e`	`a(2)`	-	-	`d(3)`	`e`	?		5
6	$n = 4$	Δ_4	`o`	`a(1)`	`e`	`a(2)`	`a(3)`	`d(2)`	-	-	✓		6
7	$n = 4$	∇_1	`o`	`a(1)`	-	-	-	-	-	-	✓	2	
8	$n = 4$	∇_2	-	-	`e`	`a(2)`	-	-	-	-	?	3	
9	$n = 4$	∇_3	-	-	-	-	`a(3)`	`d(2)`	-	-	?	1	
10	$n = 4$	∇_4	-	-	-	-	-	-	`d(3)`	`e`	?		1
11	$n = 8$	Δ_1	-	`a(1)`	`e`	`a(2)`	`a(3)`	`d(2)`	`d(3)`	`e`	?		1
12	$n = 8$	Δ_2	`o`	-	`e`	*	*	*	*	*	✗		2
13	$n = 2$	Δ_1	`o`	-	-	-	-	-	-	-	✓	2	
14	$n = 2$	∇_1	-	-	`e`	-	-	-	-	-	✓	3	

(a) Standard delta debugging procedure

#	n	Δ	Events								R	M	T
0	$n = 1$	Δ_1	`o`	`a(1)`	`e`	`a(2)`	`a(3)`	`d(2)`	`d(3)`	`e`	✗		
1	$n = 2$	Δ_1	-	-	-	`a(2)`	`a(3)`	`d(2)`	`d(3)`	`e`	?		1
2	$n = 2$	∇_1	`o`	`a(1)`	`e`	-	-	-	-	`e`	✗		3
3	$n = 2$	Δ_1	-	-	`e`	-	-	-	-	`e`	?		1
4	$n = 2$	∇_1	`o`	`a(1)`	-	-	-	-	-	-	✓	2	
5	$n = 4$	Δ_1	-	`a(1)`	`e`	-	-	-	-	`e`	?		1
6	$n = 4$	Δ_2	`o`	-	`e`	-	-	-	-	*	✗		2
7	$n = 2$	Δ_1	`o`	-	-	-	-	-	-	-	✓	2	
8	$n = 2$	∇_1	-	-	`e`	-	-	-	-	-	?	3	

(b) Improved delta debugging procedure

*** For the sake of compactness, each evens has been abbreviated in the table by its first letter. There are three possible outcomes: success (✓), failure (✗) and unresolved (?). The last column M refers to the previous step of the DD where a given sequence has been already checked (prefix of a success or has unresolved as a prefix). Memorization decreases the number of required iterations from 14 to 9 for Table 3(a) and from 8 to 5 for Table 3(b). The last column T stores the number of successfully executed events, including the first unresolved one.

Figure 3: Comparison of two delta debugging strategies

on the part of the failing sequence that has higher chance to produce the failure. To localize the failing events, we rely on the reduction procedure presented in Section 2.2. But, in fact, our modified DD algorithm can also be used with any other oracle able to point to the failure relevant events.

Let us consider the workflow of Algorithm 1, denoted here as *rddmin*. As arguments it takes the initial failing sequence $\mathcal{E}$ and the subsequence $\mathcal{R}$ of $\mathcal{E}$ which is expected to be an approximation of the possible shortest one $\mathcal{E}^*$. The procedure on line 20 refers to the standard *ddmin* algorithm as it is described in [16]. Our algorithm calls *ddmin* once the failure is reproduced for the first time or no other options are left except applying it to $\mathcal{E}$.

The algorithm starts by checking whether the original sequence is reduced to the empty one, in that case we have to apply *ddmin* to the former (line 2). Practically, it means that our set of rules is probably too liberal and should be re-examined. If the reduction $\mathcal{R}$ is not empty, it should be tested for the failure (line 3). In case the failure is immediately observed, we proceed by applying *ddmin* to $\mathcal{R}$ (line 4). Otherwise, the algorithm tries to extend the reduction with the rest of the events from the initial sequence $\mathcal{E}$ until the failure is produced or the entire sequence is consumed (line 6). Looking for an extension by applying *rddmin′* (line 7), we, first, construct the complement to $\mathcal{R}$ in $\mathcal{E}$ (line 8); next, partition the complement (line 10), into n disjoint pieces starting from $n = 2$ and until n is equal to the complement length, doubling n each time (line 17); and, finally, test each subsequence in the partition, as well as its complement, in union with the reduced sequence $\mathcal{R}$ (lines 12, 15). These tests do not guarantee to reproduce the failure, therefore, the search has to be closed with the call to *ddmin* (line 19) on the original failing sequence $\mathcal{E}$.

The worst-case complexity of *rddmin* is obviously higher than that of the standard *ddmin* algorithm. It adds $\mathcal{O}(4|\mathcal{E} \setminus \mathcal{R}|)$ on top of the *ddmin* worst-case complexity, which is $\mathcal{O}(|\mathcal{E}|^2 + 3|\mathcal{E}|)$, but the order stays the same.

Our algorithm *rddmin* resembles the Augmented Delta Debugging (*ADD*) algorithm from [14]. The main difference is that *ADD* is built on top of the DD algorithm for failure *isolation* (called *dd* in [16]), whereas we address the failure simplification problem (called *ddmin* in [16]). This choice is motivated by the need to reproduce the failure instead of isolating the failure inducing changes, that Zeller's *dd* algorithm does.

3. EVALUATION

As we pointed in Section 2.3, the number of necessary iterations to simplify the failure is what makes DD expensive to use in practice, especially if each iteration requires a separate spadework. In case of web applications, the extra iteration workload is produced when the application is requested by the client for the first time: the server connection has to be set up, auxiliary data, such as graphics and libraries, should be transferred etc. One representative class of web applications are Flash applications. They often take a considerable amount of time to load all complex visual content, but once this process is finished, the response time resumes back to normal. Because of this feature of Flash, we have chosen the *Flex Store*[1] application to be used as a case-study for evaluation of DD complemented with reduction.

3.1 Automatic Debugger

To perform the evaluation of DD complemented with reduction, we have implemented both standard *ddmin* and our own algorithm as a prototype of web application debugger called *BAD* (**B**lack-box **A**utomatic **D**ebugger). The debugger is a JavaScript library that should be embedded into the HTML source of the target application under test. It launches together with the application and provides the user a simple GUI interface for uploading the failing sequence and applying delta debugging to this sequence. The instrumented flash application [10] generates logs in response to each GUI event. The debugger is using FITTEST Flash logger [2] to replay the recorded events as part of the debugging process. The user can choose which instance of DD to use: the standard DD or DD complemented with reduction. During the debugging session, the result of each testing attempt is memorized, and later on it is used to decide whether the

[1]`http://www.adobe.com/devnet/flex/samples/flex_store_v2.html`

new sequence should be tested or the test outcome is known already. That is, if *abc* was checked and the outcome was *pass*, each new testing sequence that matches an *abc* prefix should also pass. The similar property holds for suffixes of the *unresolved* sequences. Currently, the debugger only has a backend for the Flash event system, but due to its modularity can also be extended to support HTML DOM events. The BAD source code together with the Flex Store case-study and the evaluation data are available online [1].

3.2 Case-study Set up

The Flex Store application is developed by Adobe to illustrate the capabilities of Flex SDK for building of modern e-commerce applications. The Flex Store use-cases include: applying various filters to the list of items in the catalog, comparing and purchasing items, and order processing. Ten users were requested to interact with the application without any supervision for a fixed time frame. As result, we obtained ten log files listed in Table 1. We have chosen the log number three to be used for evaluation of delta debugging. This log consists of 44 events, which is still smaller than the average length of the collected logs. But at the same time, as we will see later, the DD reduction time starts to exceed one hour even for logs of such a moderate length. The complete set of logs was used for inference of the algebraic rewrite-rules [5]. We set the confidence level to 90%, which is equivalent to demanding a rewrite-rule candidate to be witnessed four or more times before accepting it (given confidence level has shown to be sufficient [5] to guarantee low rate of false positives). That gave us in total 99 rewrite-rules. The rules together form an event-reduction system, and can be exploited for log reduction. In accordance with the reduction strategy [5], the chosen log (#3) was reduced from 44 events to just 8 events by applying the rewrite-rules inferred on the previous step. Let us refer to the original log by $\mathcal{E}$ and its reduction by $\mathcal{R}$. Let $\mathcal{E}^*$ be the minimal subsequence of $\mathcal{E}$ imposing the failure.

Table 1: User logs

logs:	1	2	3*	4	5	6	7	8	9	10
size:	5	5	44	79	106	109	153	180	230	240
								total:	**1151 events**	

By performing the evaluation in this paper, we intend to address the following research questions:

RQ1: Does the use of *rddmin* give a benefit over *ddmin*?

RQ2: Under which conditions is this benefit possible to achieve?

3.3 Evaluation Results

There are five possible alternatives of how $\mathcal{R}$ and $\mathcal{E}^*$ can be aligned relatively to each other. Three of them are outlined in Table 2, and two others, discussed first, are listed below:

case 4: $\mathcal{E}^* = \mathcal{R}$

case 5: $\mathcal{E}^* \cap \mathcal{R} = \emptyset$

The forth case implies that `Test`$(\mathcal{R})$ = ✘, which means that the minimized failing sequence is discovered straight away after the reduction phase. In the fifth case, we cannot expect any guaranteed advantage of our failure minimization algorithm over *ddmin*. This situation should be avoided by defining a state abstraction that is more appropriate to the failure. For instance, if it is known that some variables are likely to be error-prone, they should be included in the state abstraction and logged.

Let us proceed by discussing the alternatives presented in Table 2. For the log #3 from Section 3.2, we introduced 15 different errors. Each error consists of a predefined sequence of events that should be executed in the order of their occurrence in the initial sequence for the failure to be exposed. For instance, if *abcd* is the log, we can define the error to be imposed by *bd*. It means that the sequence *abd*, which itself is a subsequence of *abcd* and has *bd* as a subsequence, also produces the same failure. Following this principle, we chose 15 different subsequences of the log #3 to be responsible for the failure. With that respect, we covered three variants of how $\mathcal{E}^*$ and $\mathcal{R}$ can be related. Each time the error was imposed by the event sequence $\mathcal{E}^*$ with the following property:

$$Lev = |\mathcal{R} \setminus \mathcal{E}^*| + |\mathcal{E}^* \setminus \mathcal{R}| \leq 7$$

This property says that the reduced sequence deviates from the DD minimum by at most seven events, i.e. the *Levenshtein distance* between $\mathcal{R}$ and $\mathcal{E}^*$ is not greater than seven.

First, each failure was minimized with the standard algorithm (*ddmin*), and then the minimization procedure was conducted with *rddmin*. The summary of results is presented in Table 2. We measured the following parameters (listed in same order as they occur in Table 2): the elapsed reduction time, the total number of executed events, the number of required tests, and the number of actually executed testes (some tests do not need to be executed because their outcome was previously memorized), and finally, the relation between the *ddmin* and *rddmin* execution time, as well as the relation between the number of actual tests for both algorithms. The execution time of one event was set up to one second, whereas each new DD test took four seconds to refresh the application in order to recover the application initial state.

In the first case ($\mathcal{E}^* \subset \mathcal{R}$), the reduced sequence subsumes the minimal failing one. The *rddmin* has shown an improvement over *ddmin* in all seven experiments, with the best time ratio for $Lev = 5$, which gave us 6.4 times faster reduction. In the second case ($\mathcal{R} \subset \mathcal{E}^*$), the reduced sequence is subsumed by the failing one. For the values of $Lev = 1$ and $Lev = 2$, *rddmin* has gain the speed up by 2 and 1.4 times respectively, but with the growth of *Lev*, *ddmin* has managed to outperform *rddmin* by less than two times. Interestingly, the ratio of tests number (last column) is close to one for the values of $Lev = 4, 3$. This implies that *rddmin* probably tried to test longer subsequences of $\mathcal{E}$ than *ddmin*. In the third case ($\mathcal{E}^* \cap \mathcal{R} \neq \emptyset$), the reduced and failing subsequences partly overlap. In all four experiments, the execution time of two algorithms was almost equal (except for $Lev = 4$, when *ddmin* has beaten *rddmin*). We again see the pattern observed in the first case, namely, that the success of *rddmin* higher if the reduction has fewer distance from the failure. Over all 15 experimental runs, *rddmin* has shown better execution time in 12 runs, and has been beaten by *ddmin* in 3 others. In our evaluation, the optimal distance between the reduced sequence and the minimal failing sequence for *rddmin* to outperform *ddmin* turned out to be one or two events.

Table 2: Evaluation of *ddmin* and *rddmin*

case i:	Lev	$\|\mathcal{R} \setminus \mathcal{E}^*\|$	$\|\mathcal{E}^* \setminus \mathcal{R}\|$	DD	time (sec.)	#events	#tests	#actual.tests	Δ_{time}	$\Delta_{act.tests}$
1: $\mathcal{E}^* \subset \mathcal{R}$	7	7	0	*ddmin*	158	85	11	6	1.1	1.2
				rddmin	138	59	8	5		
	6	6	0	*ddmin*	451	275	63	22	3.8	2.4
				rddmin	118	72	23	9		
	5	5	0	*ddmin*	843	593	114	51	6.4	4.6
				rddmin	131	83	31	11		
	4	4	0	*ddmin*	1703	874	160	70	4.97	3.5
				rddmin	342	131	43	20		
	3	3	0	*ddmin*	981	492	122	49	2.3	2.04
				rddmin	419	162	54	24		
	2	2	0	*ddmin*	1377	882	168	72	5.8	4
				rddmin	239	129	31	18		
	1	1	0	*ddmin*	1336	774	148	54	5.7	3
				rddmin	235	134	37	18		
2: $\mathcal{R} \subset \mathcal{E}^*$	4	0	4	*ddmin*	3515	2697	360	159	0.59	0.96
				rddmin	5944	3823	400	166		
	3	0	3	*ddmin*	3189	1918	286	129	0.65	1.1
				rddmin	4911	2669	279	118		
	2	0	2	*ddmin*	2167	969	171	65	1.4	0.85
				rddmin	1581	943	166	76		
	1	0	1	*ddmin*	2019	1112	190	85	2.01	1.8
				rddmin	1003	540	117	47		
3: $\mathcal{E}^* \cap \mathcal{R} \neq \emptyset$	6	2	4	*ddmin*	2290	1229	204	76	1.2	0.77
				rddmin	1895	1366	256	99		
	4	1	4	*ddmin*	2145	1261	203	66	0.38	0.52
				rddmin	5597	3040	328	126		
	4	1	3	*ddmin*	2058	1053	189	69	1.09	0.79
				rddmin	1894	1026	200	87		
	3	2	1	*ddmin*	1988	1254	228	95	1.12	1.27
				rddmin	1769	866	189	78		

In conclusion, trying to answer the research questions suggested at the end of Section 3.2, we have seen some examples of failures which were faster to minimize with *rddmin* than with the plain *ddmin*. The evaluation has also shown that the likelihood to gain some performance over *ddmin* gets higher if the reduced sequence closely approximate the failure, and rather over- than under- approximates it, as the first case of our evaluation has illustrated.

3.3.1 Threats to Validity

The first threat to external validity is related to the fact that we considered only one case-study. This does not allow us to confidently generalize our conclusions about the effectiveness of *rddmin* for other applications.

Lacking a driver for random generation of Flash applications, we had to initiate testing sessions with real users. The resulted logs were used for two purposes: 1) infer rewrite rules, and 2) simplify log files as containing failures. This process creates the internal threats to validity of failure simplification results. The inferred rewrite rules define how significantly we can reduce the failure, whereas the initial length of the failure revealing log influences the number of steps required by delta debugging to carry simplification.

Another validity threat comes from the artificial nature of the failures used for simplification. We have not found any information about real faults in *Flex Store*, so we had to define them ourselves. We decided to inject faults not by modifying the application source code, but, instead, we associated the faults with some set of application events triggered in certain order at run-time.

Having a fixed system of rewrite rules, just one log file for simplification, and only 15 different faults injected in this log file might not be representative enough to statistically confirm the answers on the research questions given in Section 3.3. The execution time of delta debugging soon starts to exceed one hour which makes it expensive to include more faults in the experiment. We could have taken a smaller log file for the experiment to decrease the DD execution time, but this could possibly make negligible the benefit of applying *rddmin* due to the short execution time of *ddmin* on small logs. We believe more experiments are needed in this direction to gain higher confidence.

4. RELATED AND FUTURE WORK

4.1 Related Work

As it has been already noticed in the original paper on DD [16], the number of test steps required for minimization can be decreased by maximizing the probability of failure occurrence for a given sequence in question. If it is known, that one subsequence is more likely to cause the failure than the others, then that sequence should be tested first. But the paper does not suggest the strategy of prioritization. Given a test-case that passes on program version and fails on the other, Ya et al. [14] suggest applying delta debugging to concentrate on those program statements that are at least covered by one of the test-case runs (passing or failing).

These statements form a *dubious set*, which is similar to the simplified failing trace resulted from the offline reduction in our approach. The *rddmin* algorithm is a variation of Ya's augmented delta debugging algorithm, adjusted to address the problem of failure simplification rather than isolation.

Hierarchical delta debugging (HDD) [11] decreases the number of testing attempts by considering only structured tree-like inputs such as XML. It applies DD level by level starting from the root. The paper mentions, that hierarchy can be also imposed on GUI events as a dependency relation between high-level events (*purchase item*) and corresponding sequences of low-level events (*mouse-click*).

When multiple failures exist in one program, probably overlapping with each other, Iterative Delta Debugging can be applied [4] to gradually isolate the changes and failures over multiple versions of the same program.

Regehr et al. [13] have suggested to generalize test-case minimization problem as a generic *search problem* which can be tuned with various *fitness functions*. They have introduced three new reducers and applied their framework for the reduction of C compiler bugs.

Unit-tests are another subjects for automatic minimization, especially if they are randomly generated. Delta debugging can also be employed to address this problem [7]. In this case, the reduction can particularly benefit from slicing preceding DD [8].

Within a given test budget — the time given for testing — the test suite can be optimized to increase the effectiveness of testing [3]. Groce et all [6] have adapted delta debugging for test suite simplification with respect to code coverage.

4.2 Future Work

In our approach, the success of failure minimization heavily depends on what comes out of the reduction phase. The more events relevant to the failure are eliminated, the harder it is to reconstruct the original failure. In the worst case, we can be left with the original failing execution, which makes our approach unproductive. This issue should be addressed by studying the relation between the observed failure and the application state sampled in the logs. A different state abstraction provides a different set of rewrite-rules. If the sequence after reduction does not produce the failure, the set of rules could be altered and the whole reduction process could be repeated until the failure appears again.

On the other hand, our approach can be combined with other variations of delta debugging such as IDD [4] and HDD [11].

There is a room for improvement in the web debugger (BAD) itself. First, the debugger can be extended to support the HTML DOM event system. Second, the current implementation of the debugger requires to alter the source code of the application under test to hook up the debugger to the application. In general, we would like to see the debugger as a standalone application, for example, the browser extension such as Firebug [9] or as a web service.

5. CONCLUSION

The automatic test-case minimization by means of delta debugging often suffers from large number execution steps required to carry out the simplification process. Not all exercised subsequences of the original test-case contain only failure relevant circumstances, and apparently there are some tests that will result to non-executable test-cases anyway. Assuming that the test-case is represented by a sequence of GUI events, the failing test-case can be first minimized by applying offline reduction, and only then subjected to minimization with delta debugging. In this paper we presented the variation of DD minimization algorithm that is complemented with a priori test-case reduction phase. The two algorithms were implemented in the web application debugger, BAD, that was applied for failure simplification on the web shop application, Flex Store.

6. ACKNOWLEDGMENTS

Thanks to anonymous reviewers for their valuable comments. This work is supported by the EU project FITTEST (No. 257574).

7. REFERENCES

[1] Balck-box automatic debugger. `https//github.com/aelyasov/BAD`.

[2] Fittest flash loger. `https://code.google.com/p/fittest/wiki/FlashLogger`.

[3] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. ASE '08, pages 19–28, 2008.

[4] C. Artho. Iterative delta debugging. *Int. J. Softw. Tools Technol. Transf.*, 13(3):223–246, June 2011.

[5] A. Elyasov, I. Prasetya, and J. Hage. Guided algebraic specification mining for failure simplification. In *Testing Software and Systems*, LNCS 8254. 2013.

[6] A. Groce, M. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. ICST '14, 2014.

[7] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. ISSRE '05, pages 267–276, 2005.

[8] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. ASE '07, pages 417–420. ACM, 2007.

[9] R. Lerner. At the forge: Firebug. *Linux J.*, 2007(157):8–, May 2007.

[10] A. Middelkoop, A. Elyasov, and W. Prasetya. Functional instrumentation of actionscript programs with asil. In *IFL '12*, LNCS 7257.

[11] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. ICSE '06, pages 142–151, 2006.

[12] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? ISSTA '11, pages 199–209. ACM, 2011.

[13] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. PLDI '12, pages 335–346. ACM, 2012.

[14] K. Yu, M. Lin, J. Chen, and X. Zhang. Practical isolation of failure-inducing changes for debugging regression faults. ASE 2012, pages 20–29, 2012.

[15] K. Yu, M. Lin, J. Chen, and X. Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *Journal of Systems and Software*, 85(10):2305 – 2317, 2012.

[16] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.