# Automated Session Data Repair for Web Application Regression Testing

**Nadia Alshahwan**
National Company for Cooperative Insurance
Central Regional Office
P.O. Box: 52991 Riyadh - 11573, Saudi Arabia

**Mark Harman**
King's College London
Strand, London
WC2R 2LS, UK

## Abstract

*This paper introduces an approach to web application regression testing, based upon repair of user session data. The approach is entirely automated. It consists of a white box examination of the structure of the changed web application to detect changes and a set of techniques to map these detected changes onto repair actions.*

*The paper reports the results of experiments that explore both the performance and effectiveness of the approach. The effectiveness experiment uses an implementation of the repair algorithm applied to the online bookstore application over a series of 10 releases.*

**Keywords:** Regression Testing, Web Applications.

## 1. Introduction

User session data has been shown to be an effective and valuable resource in web application testing, finding faults not found by other techniques. However, web applications are typically the subject of frequent changes that invalidate some existing session data. This rapid evolution can thus lead to an equally rapid reduction in the value of the session data. Though much attention has been paid to testing issues for web applications there has been very little previous work that utilizes user session data for web application testing and none that uses session data in regression testing of web applications.

Notwithstanding the paucity of previous work on session–based web application testing, the little work that does exist has been very encouraging; revealing that session data may expose faults not found by other white box techniques [6]. Session data represents realistic test cases because they are created directly and naturally by the users of the system. Furthermore, it is usually easy to collect session data due to the server–side execution of much web application functionality and the wide availability of logging mechanisms.

However, when a change is made to a web application (for example, introducing a new page, removing a link or changing a parameter set of a dynamic page) some of the session data becomes potentially obsolete; it will no longer represent a valid session in the new structure of the application. Therefore, any attempt to exploit the value locked in session data will have to overcome the problem posed by changes that render the sessions invalid. However, it is precisely these 'newly invalid' session data that test new functionality denoted by changes to the web application.

This paper introduces an algorithm based on the concept of 'session repair'. Where a session has become newly invalid due to changes in the application, our algorithm will attempt to repair the session. It constructs a new version of the session that closely resembles the original, thereby retaining the realism of the original session.

The repair approach mines the session data for suitable parameter values to use in cases where the parameters of an individual request have become modified. It also attempts to find new paths to connect sections of an existing session for which navigation has become impossible due to structural modifications. Where such paths do not exist, the session is split into two smaller but valid sessions. The first half (subsequence) of such a split is guaranteed to be applicable, though, clearly one would prefer to preserve longer sequences, where possible. Since sessions are a sequence of URLs together with associated parameters, the second half of such a split is also potentially valuable.

Of course, it is possible simply to split a session into two sub-sessions as soon as a problem is encountered, but we prefer, to first attempt repair. This approach is based on the philosophy that we should strive to retain as much of the original length of a session as possible. Longer sessions may be required in order to lead the application into special cases and to establish values for internal state variables.

The repair approach requires a pre-processing white box analysis phase that determines the structure of the modified web application. This new structure is then used to drive the search for newly inapplicable parameter sets and newly invalid sequences of URL requests. One advantage of this approach is that it is entirely automated; the overall algorithm simply takes a (possibly changed) web application and a set of session data (obtained from the original web application)

and produces a repaired set of session data.

In order to assess the effectiveness of the approach, a controlled experiment was performed on a previously well studied web application: the online bookstore [6, 10, 11]. The bookstore was changed in response to a set of 10 user-requests for new functionality. These user–requested changes were used to produce a series of 'releases' of the application, to which an implementation of the repair algorithm was applied. The results show that the approach can produce substantial improvements in regression testing effectiveness by increasing the amount of session data which can be re–used in regression testing. The approach also remains effective in the presence of significant numbers of changes without the re–collection of updated session data. The primary contributions of this paper are:

1. **Algorithm**
   The paper introduces an algorithm for regression testing based on session data repair. This algorithm represents the first session–based approach to web application regression testing.

2. **Effectiveness Study**
   The paper presents the results of a controlled experiment into the effectiveness of the approach when applied to 10 versions of an online bookshop. The results indicate that the approach is effective at recovering the value in session data that would otherwise be lost when the web application changes.

   Without repair, about one third of the sessions would have been unusable in regression testing. The repair approach is not only able to retain these, but in the majority of cases it does so without recourse to session splitting, thereby retaining the value of the original session as a longer sequence of realistic and potentially complex transactions.

3. **Scalability Study**
   The paper also presents results concerning the scalability of the approach, indicating that the performance of the algorithm will allow it to be used even in demanding scenarios such as those where daily regression testing is required.

The rest of the paper is organized as follows: Section 2 presents the algorithm for repair. Sections 3 and 4 describe the controlled experiment and the results obtained from it using a prototype session repair tool that implements the algorithm, whilst Section 5 considers the threats to the validity of the findings. Section 6 presents related work and Section 7 concludes.

## 2. The Repair Algorithm

Our overall approach is based on two phases: an analysis phase that identifies the changes made to the original version of the application and a manipulation phase that repairs the sessions rendered invalid by these changes. The overall algorithm is set out in Figure 1. Steps 1,2 and 3 form the analysis phase of the overall approach, whilst Step 4 performs the manipulation phase that implements the necessary repair actions.

Step 1 is an initialization phase in which three data structures used by the algorithm are constructed from a set of session data. The variable $SL$ is a list of sessions. Each session is a sequence of node–edge pairs, recording the pages (nodes) visited and the edges (links) followed in order to move from one node to the next. The variable $PLT$ is a Parameter Look up Table; a database relation that allows parameters used in a dynamic page to be indexed by their page identifier and vice versa. The variable $PL$ is simply a list of page identifiers used in the web application (essentially a node set).

When all else fails and no suitable parameter can be found in $PLT$, an appropriate value has to be 'invented'. In order to allow values to be invented for parameters that are not enumeration types, a Constant Value Table, $CVT$ is constructed. This table provides a valid entry for unbounded variable types (such as strings) that cannot be given a random enumerated value. Our experience is that such values are seldom used to determine page navigation. Rather, navigation parameters are typically offered to the users as a choice from a set of possibilities, thereby ensuring valid input.

Step 2 constructs the graph structure of the modified web application as a set of nodes (pages) and edges (links). This phase is essentially a white box analysis phase.

If a web `FORM` is encountered for which the input parameters are identical to those in the original web application, then the original parameters are copied into the spider to allow it to continue navigation. If there is a changed set of parameters, our approach is to check to see if there exist other pages in the original web application that have the same parameter identifiers. Since the new application is a 'version' of the old one, it seems reasonable to recoup and reuse such values. In the worst case, where an entirely fresh parameter has been introduced, which has hitherto not been used elsewhere in the old version of the application, we invent a value for this parameter. Where the parameter has an enumerate type value, for example, a menu, check box or radio button, we simply generate a valid random value. Where the parameter type is unbounded (for example a string) we look up a suitable pre–determined value from the Constant Value Table $CVT$.

The re–used and invented parameters play an important role in the subsequent repair phase which occurs in Step

4. They form the new values for changed and freshly introduced parameters that are inserted into the session data. This allows for re-navigation to reconstruct links that have become newly invalid due to web application evolution.

Step 3 is an enabling step for the repair phase that follows. It constructs the set of linearly independent paths from the graph–based data structure constructed in Step 2. The set of linearly independent paths is a set of paths such that each member of the set contains an edge that is not found in any other member of the set. Strictly speaking, the step is not necessary; it is possible to construct the paths required by Step 4 directly from the graph produced in Step 2. However, since repeated paths must be generated in Step 4, it is often more convenient to have sequences of valid paths pre-stored.

Importantly, each edge is contained in precisely one path and all edges are contained in one or other path. This allows the repair phase to construct a valid sequence of steps to move from one page to another in the new version of the web application. The final phase implements the repair actions for three kinds of newly invalid link. The repair algorithm is at the heart of the approach and it is now described in more detail.

In order to perform repair, two kinds of task have to be accomplished:

**1. Individual URL Repair**. An individual URL may no longer use the same parameter. Each such individual URL request must be repaired to give meaningful values to any parameters newly introduced.

2. **Sequence Repair**. Each session may no longer correspond to a valid sequence of URL requests in the new version of the application. Sequences of URLs from the existing session database may contain nodes and edges that no longer exist in the new version of the application. Where possible, these sequences should be repaired to give valid (and hopefully realistic) sessions for the new version of the application. Where it is not possible to build a valid sequence from an existing session, the session may be decomposed into smaller valid sequences of URL requests. This may allow some of its original value to be retained.

Consider a URL in the session data ($old$) and a corresponding URL in the new version of the web application ($new$). Two steps must be performed: First, any parameters in $old$ no longer required by $new$ must be removed. Second, any parameters required by $new$ but no longer present in $old$ must be created to result in a request that is both valid and hopefully, realistic. Here 'realism' is a property that must come from the implicit domain knowledge trapped in the existing session data, since this is the only resource that we can reliably assume will be available for this purpose.

Turning from repair of individual URLs to the repair of entire sequences of URL requests, three classes of sequence repair are considered:-

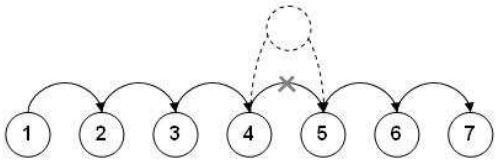


**Figure 2.** **An Illustration of Edge Deletion Repair. If the link from node 4 to node 5 is removed, but the nodes remain, then a new path must be found to navigate from node 4 to node 5.**

**1. Edge Deletion**. A link (edge) between two pages is removed, but the two pages remain. In this situation, a new path must be found from the source page to the target page that is valid in the new version of the application. This situation is illustrated in Figure 2.

**2. Node Deletion**. A page (node) could be deleted from the web application so that any link to this page becomes invalid in the session data. In this situation, it is possible to link the old source page to the next–but–one page in the old session data (so long as the next–but–one page has not also been deleted). This can be achieved by attempting to find a path from the old source page to the old target of the target page. The situation is illustrated in Figure 3. Of course, this process could be repeated for arbitrarily long sequences of deleted pages, but we are likely to receive diminishing returns; a long sequence of deleted pages may indicate a significant shift in functionality. Therefore, we have taken the view in this paper, that we shall stop attempting such repair when two or more pages in sequence are missing.

**3. Default: Neither of the above applies**. It can happen that a sequence of two or more pages are deleted, or that only a single edge or page is deleted, but it is not possible to find a path between the original source and the original target (or target of the target) node. In this 'catch all' situation, we adopt a 'damage limitation' approach, in which the original session is split into two sessions at the point at which the problem occurs.

Each session can then be recursively repaired according to the overall approach. The situation is illustrated in Figure 4. This default approach facilitates some degree of reuse. It loses the value that accrues from continuing testing along an extended sequence of URL requests.

## 3. Experimental Set Up

A prototype regression testing tool was implemented to allow experimentation with the algorithm introduced in Figure 1. The current version of the tool does not implement edge repair (these are currently handled with the default session repair action of session splitting).

Step 1 is achieved using a standard relational database implementation with session data are loaded from an XML

Step 1: **Upload Original Session Data**
Step 1.1: Construct Session List, $SL$
Step 1.2: Construct Parameter Lookup Table, $PLT$
Step 1.3: Load PageList, $PL$
Step 1.4: Set up predetermined Constant Value Table, $CVT$

Step 2: **Spider Modified Web Application**
Perform a traversal of the web application's structure, starting at the home page.
Record each edge (link) in $EdgeList$ and each node (page) in $NodeList$.
Construct a graph data structure, $G$, to represent the connectivity of the nodes in the modified application.
For dynamic node, $n$, (a page of the modified application) visited,
**if** $n$ IsIn $PL$
**then** look up parameter list in $PLT$
**if** parameters unchanged in modified page $n$
**then** use parameters from $PLT$
**else if** enumeration type parameter
**then** use random value in range
**else** use constant value from $CVT$
**else if** parameter used by some other existing page in $PLT$
**then** use parameter value from $PLT$
**else if** enumeration type parameter
**then** use random value in range
**else** use constant value from $CVT$

Step 3: **Identify Independent Paths in Modified Web Application**
**set** $WorkList$ **to** the set of all edges in $EdgeList$
**set** $IndependentPathList$ **to** Empty
**while not** Empty($WorkList$)
**do**
**set** $Path$ **to** the singleton list that consists of an arbitrary edge in $WorkList$, removing this edge from $WorkList$
**while** there exists $succ$ **in** Successors(Last($Path$), $G$) **such that** $succ$ IsIn $WorkList$
**do** Append $succ$ to end of $Path$ **od**
Add $Path$ to $IndependentPathList$
**od**

Step 4: **Repair Session Data**
**set** $WorkList$ **to** the set of sessions in $SL$
**while not** Empty($WorkList$)
**do**
**set** $s$ to arbitrary session in $WorkList$, removing $s$ from $WorkList$
**set** $RepairedSession$ to singleton containing Head($s$)
**while not** IsEmpty(Tail($s$))
**do** **set** $e$ **to** the edge connecting $n$ to Head(Tail($s$))
**if** $e$ IsIn $EdgeList$
**then** */* No action required */*
**else** */* Repair required */*
**if** Head(Tail($s$)) IsIn $NodeList$
**then set** $\pi$ **to** FindPath(Head($s$),Head(Tail($s$)),$G$)
**set** Append $\pi$ to end of $RepairedSession$
**else if** Head(Tail(Tail($s$))) IsIn $NodeList$
**then set** $\pi$ **to** FindPath(Head($s$),Head(Head(Tail($s$))),$G$)
**set** Append $\pi$ to end of $RepairedSession$
**else** */* Split session into two */*
Append singleton containing Head($s$) to end of $RepairedSession$
Add Tail($s$) to $WorkList$ *the remainder of the session is repaired too */*
**exit** */* leave the while loop */*
**od**
Replace $s$ with $RepairedSession$
**od**

**Figure 1.** **The Repair–Based Algorithm. The functions Head, Last, Tail, Append, Empty, IsEmpty and IsIn standard list processing functions. The function Successors, takes an edge, $e$ and and graph, $g$ returns the set of successors of the node upon which $e$ is incident in $g$. FindPath($n_1$,$n_2$,$G$) attempts to find a path from $n_1$ to $n_2$ in $G$. If such a path cannot be found then the function returns an 'abort' value and this causes this attempt at repair to fail, following instead, the default (split session) case.**

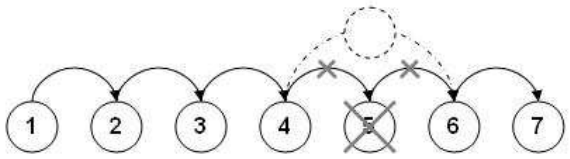


**Figure 3.** **An Illustration of Node Deletion Repair. If a page (node 5 in this case) should be removed, then a new path must be found to navigate from its predecessor to its successor.**

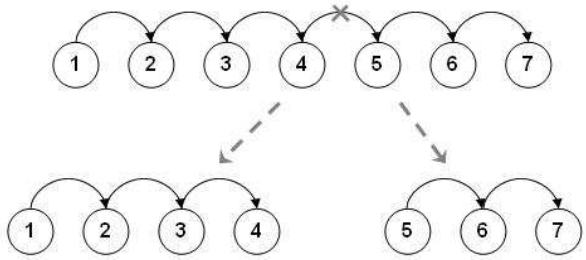


**Figure 4.** **An Illustration of the Default Repair Operation: When no path in the new structure can be found to navigate from node 4 to node 5, then the old session is split into two sub–sequences to which the repair algorithm is recursively applied.**

session log. Step 2 is implemented using a version of the popular web-spidering tool, `JSpider`, tailored to handled forms and dynamic pages in the manner described in Figure 1. Steps 3 and 4 from Figure 1 are implementation components constructed specifically for the regression session repair tool.

The evaluation of the approach is concerned with two aspects of the approach corresponding to two research questions we seek to answer:

1. **RQ1: How effective is the approach?**
   The effectiveness of the approach is measured by considering the percentage of user session data that was reused over a series of changed versions of the application. For comparison (and as a measure of the level of change from version–to–version) the reusable percentage using the repair algorithm is compared to the percentage of session data that would have been reusable were no repair to have been applied.
2. **RQ2: Is the approach sufficiently computationally cheap to be applicable?**
   In order to be applicable, it must be possible to perform the entire repair process in a period of time that is commensurate with the time allocated to all other regression testing activities. The approach may be required in situations where web applications are changing on an almost daily basis, requiring a nightly build-and-test approach to regression testing. In such a scenario, the tool must be able to scale to large web applications with performance measured in minutes.

The theoretical algorithmic performance of the approach is clearly determined by the size of both the web application and the session data base. However, it is theoretically linear in both the number of edges in the web application and the number of sessions on the session data base, and so it *should* be scalable to larger systems. In order to give these theoretical observations some practical interpretation, data will be presented for the run–time performance of our prototype implementation. It should be remembered that this is a prototype and that an industrial strength version might reasonably be expected to achieve an order of magnitude improvement in performance. However, real world regression testing problems may also be an order of magnitude larger than those studied here.

Our main focus for experimentation is RQ1, since we are concerned to investigate the way in which the regression testing effort (using repair) degrades over successive releases of the system, when compared to a regression testing effort which does not use repair. The experiment was conducted in the following manner:

## Web Application Selection

A relatively simple but realistic web application was chosen for the study. The application, an online bookstore, is one used by other researchers working on Web Application Testing [3, 6, 10, 11] and is available from `www.gotocode.com`. The Web application provides the customer with the ability to register, login, search, browse, purchase books and to use a simple 'virtual shopping cart'. The application also has an 'administrative user' feature with different levels of privilege to regular users. The bookstore web application was not altered in any way. Initially, there were 22 books in the book database.

## Session Collection

The site was set up on an Apache Tomcat server, with a session logging tool. A group of 14 participants was invited to use the original version of the web application and the session data they created through this use was stored as the initial session data. The participants were asked to explore the site and to attempt to use all of its features, but they were given no other specific directions on how the site should be used. None of the participants were taught about the structure of the web application, the nature of the experiment nor the details of the algorithms for repair. The participants were given neither reward nor tasks to perform on the web application, so that their use would be as unconstrained as possible. The participants were also uncon-

strained as to when and for how long they used the web application. In total, 40 sessions were logged over a period of 10 days.

### Web Application Evolution

A series of 10 changes were made to the web application to implement new functionalities. The changes made were designed to implement realistic functionality enhancement, not mere 'random mutation'. The changes were cumulative, so that each new version of the web application involved changing the previous version. The aim was to produce a sequence of releases. The changes applied were not necessarily of the same size. Although each change was implemented either by one of the experimentors, creating a potential source of bias, all the changes were implemented in response to change requests from the participants. These requests arose naturally; they were requests for improvement in the functionality of the system. The change requests were not filtered in any way and were implemented purely to achieve the additional or changed functionality requested by the participants.

The types of changes made were as follows:

1. **Structure**: The way pages were linked together was changed.
2. **Forms**: Parameters were added or deleted and names of parameters were changed.
3. **Pages**: Pages were deleted and others were added.
4. **Files**: File names were changed.

Basic data on the types of changes performed are recorded in Table 1, together with a brief description of the nature of each change performed.

### Tool Application

The prototype Regression Testing Tool was applied to each version of the web application using the initial set of sessions collected from the execution of the original version of the application. No update of session data was performed between each release. The aim was that the original session data would become increasingly 'out of date', presenting a monotonically increasing level of challenge to the repair algorithm.

### Data Collection

Results were collected for execution times for the four stages of the algorithm, for the percentage of successfully repaired individual URL requests and for the percentage of all session repair operations that were performed in each of the session repair categories.

## 4. Results

The primary focus of the experiment was upon the effectiveness of the approach, for which the results are presented in Section 4.1.

Whilst an approach to regression testing may be effective at increasing the amount of test cases which may be available following a change, it will not be practical if the performance of any implementation is not commensurate with the time available for regression testing. Our implementation is only a research prototype, so care is required when interpreting performance results. Nonetheless, we separately performed an experiment to collect data on its performance over a series of increasingly large session databases and over a set of increasingly large web applications. These results are presented in Section 4.2. To collect these results, the tool was executed on an Intel Pentium III Mobile CPU, running at 1.19GHz with 256MB RAM.

### 4.1. Effectiveness

Figure 5 shows the percentage of URLs requested that would be discarded were no repair actions to have taken place and the percentage that the algorithm discards because it is unable to find a suitable repair for the individual URL.

As can be seen, without repair, the number of URLs that can be reused, steadily falls as the level of change rises. By contrast, the number of URLs that the repair algorithm is forced to discard is far lower; after the 10 change requests have been implemented, the 'no repair' option is forced to abandon four times as many individual requests as the repair algorithm.

In determining the number of URLs that would be discarded were there to have been no repair, we adopt an approach that gives the most favourable outcome for the 'no repair' option. That is, where parameters are merely removed from a URL, no error occurs, so some form of regression testing with sessions containing this form of old 'broken' URL would be possible. If we also count these 'decreased parameter URLs' as discarded in the 'no repair' approach, then the results for the repair algorithm would appear to be better than those reported.

Figure 6 shows the number of sessions repaired using each of the repair approaches, compared to the percentage that would be discarded were there to be no repair. The results for the different categories of repair action are represented as percentages of the overall number of repair operations applied. Therefore, the sum of their values for each release will always be 100%. These results give an indication of how the proportion of repair operations changes over the evolution of the web application's releases.

Because our philosophy is to retain, where possible, the full length of user sessions, the 'split session' repair operation is the least favoured. The results in Figure 6 are extremely encouraging with respect to the number of occasions on which a session split was required. For example, throughout the change process, twice as many node repair operations are performed as session splits, indicating that

| Version | Number of modified pages | Number of added pages | Number of deleted pages | Number of file name changes | Brief Description of Changes Applied to Produce This Release |
|---|---|---|---|---|---|
| V1 | 0 | 0 | 0 | 0 | Original version. |
| V2 | 4 | 0 | 1 | 0 | Reduce sequence of steps required to login; layout changes. |
| V3 | 2 | 1 | 1 | 0 | Add 'change quantity' option to shopping cart and reduce crowding. |
| V4 | 2 | 0 | 1 | 0 | Add 'continue shopping' button; add rounding up; move book rating to fresh page. |
| V5 | 5 | 0 | 2 | 1 | Link voting to book details; decrease number of search results per page. |
| V6 | 2 | 0 | 0 | 0 | Add more instructions and guidance; add sign-out option to each page. |
| V7 | 8 | 0 | 0 | 0 | Restructure administrators' menus; increase size of book titles. |
| V8 | 3 | 0 | 0 | 0 | Add 'search on author and title' to home page; add currency to price; disallow re-register while logged on. |
| V9 | 7 | 0 | 0 | 0 | Add 'back' button to each administrators' page; move 'browse' to separate page. |
| V10 | 2 | 0 | 1 | 0 | Add 'my account' link to top of each page; separate password change from other fields. |

**Table 1.** **Basic Data on Web Application Evolution. The on–line bookstore application evolved, through a series of changes according to a set of user requests for additional and changed functionality. The number of modified pages are those which retained their identity, but the content was modified.**

the algorithm has the potential to retain a substantial amount of the value of the original sessions in their entirety.

Figure 6 also shows the number of session that would remain valid and, therefore, available for regression testing were there to be no session repair (labelled 'valid sessions'). Using the session repair algorithm, *all* sessions remain available (though some are split), whereas with no session repair, the number available quickly falls. In our sequence of implementations of change requests, there are two points in the application's evolution that cause an 'extinction event' to occur in which a number of sessions become invalid. This causes the number of session available with 'no repair' to fall, first by a quarter and subsequently, after 5 versions by a third, from those available to the original version of the website.

Of course, it should be noted that the repairs are being applied to an increasingly altered web application, but always with reference to the original set of session data, collected from version 1 of the application. In reality it would be possible to retain repaired session data, constructed part way through the evolution of the web application, thereby improving the repair algorithm's effectiveness. We have chosen not to do this in our experimentation, because we wish to see how the effectiveness of the repair operation degrades over a set of increasingly changed versions of the original web application.

Measuring the 'size' of a change to a web application is an inherently imprecise activity destined to be ill-defined.

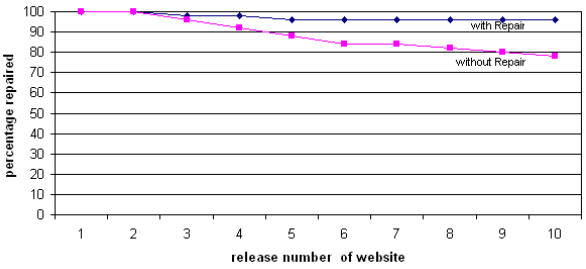


**Figure 5.** **Reuse of URLs in Regression Testing. The percentage of all URLs that would be discarded with and without repair as the web application increasingly changes.**

As a result, we cannot infer any meaning to the slope of the graph of the functions denoting the percentage of each repair activity in Figure 6. However, each release is produced by modifying the previous release to add new functionality. Therefore, we can say that the level of change is monotonically increasing. This allows us to see how the effectiveness of the repair activities change with an increasing level of change.

## 4.2. Performance

The performance of Steps 1 and 4 of the algorithm described in Figure 1 is determined by the size of the session database. Figure 7 shows how the performance (in execu-

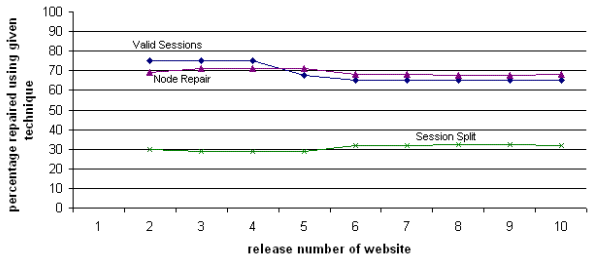


**Figure 6.** **Reuse of Sessions in Regression Testing. The figure shows the percentage (of all repair actions performed) that fall into each of the categories of repair. It also shows the percentage (of all sessions) that would remain valid were no repair actions to be performed. Using the repair algorithm, all sessions are retained (though a proportion of these do become split).**

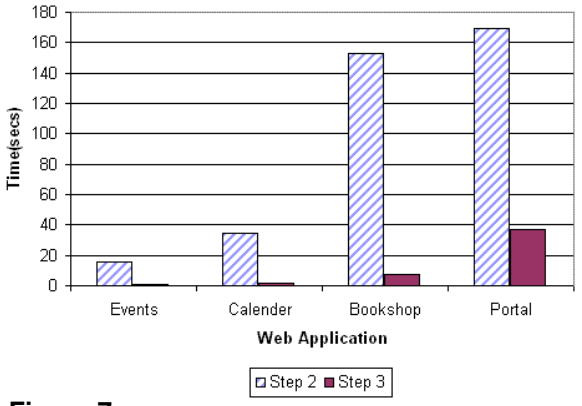


**Figure 7.** **Relationship between Execution Time and Number of Sessions for Steps 1 and 4 of the Algorithm in Figure 1**

tion time) varies with the number of sessions in the session data base.

For session databases containing 10 or fewer sessions, the time to process the session is dominated by the initialization and startup time of the application. As the results for session sizes show, the application is able to process sessions at a reasonable rate, covering all sessions used in the experiment in a matter of seconds on standard computing machinery. This is well within acceptable bounds, even for frequent regression testing scenarios.

The performance of Steps 2 and 3 of the algorithm described in Figure 1 is determined by the size of the web application to which they are applied. In order to asses this performance, steps 2 and 3 were applied to four websites, including the on–line book store used to measure the effectiveness of repair in the previous sections.

Table 2 gives a set of size measurements for each of the four web applications. The column 'files' records the num-

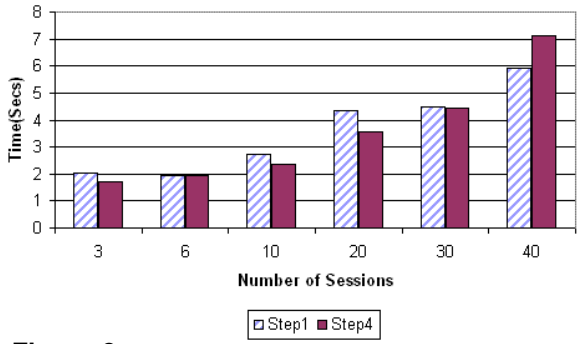


**Figure 8.** **Relationship between Execution Times and Application Size for Steps 2 and 3 of the Algorithm in Figure 1**

| Web Application | Files | URLs | Edges |
|---|---|---|---|
| Events | 14 | 14 | 49 |
| Calendar | 4 | 29 | 253 |
| Bookshop | 52 | 145 | 1210 |
| Portal | 30 | 150 | 3252 |

**Table 2.** **Information About the Applications Used for Evaluation of the Steps 2 and 3 in Figure 8**

ber of files in the application (this number includes image files should they correspond to click-able links). The column 'URLs' records the number of statically distinct URLs recorded by the spidering tool and the 'edges' are the number of links traversed by the tool. From Table 2 we roughly rank the web applications according to overall size. Figure 8 shows how the performance (in execution time) varies with the overall size of the web application under white box analysis. Naturally the spidering process dominates the overall analysis phase, since every edge must be traversed. As can be seen, the tool is able to complete its entire white box analysis phase within three minutes for all of the applications considered. This suggests that the analysis phase of the algorithm and the tool that implements it are likely to have an acceptable performance, even for the most demanding web application regression testing scenarios.

## 5. Threats to Validity

There are two kinds of threats to validity for the experiment presented in this paper: internal and external. The primary and most pressing threat to the validity of the work comes from the application of the approach to only a single web application. Clearly, in order to make more general statements about the behaviour of the algorithm on other web applications, it will be necessary to apply the tech-

niques and algorithms introduced in this paper to more and varied web applications.

In our experiment, we traded a degree of realism for a degree of control, in a carefully planned manner. To promote realism, we allowed independent participants to specify the changes to be made to the web application and we collected session data from unsupervised participants. However, to obtain a degree of control, the changes were all performed on the same web application; one which previously has been widely studied by other authors.

There is of course, a danger that the experimentor may make changes in a manner that biases the results of the application of the repair tool and so we allowed independent web programmers to make some of the changes to see if this produced noticeable differences in results. Furthermore, we ensured that all changes were made only in response to requests from the participants.

The experiment is perhaps a little unrealistic in the sense that we did not re–collect session data after any of the changes had been performed. We did this because we were interested in the worst case performance of the approach over a sequence of changes. In a more realistic scenario, it would be possible to augment our session data base throughout the modification process. However, by basing all attempts at repair on the original session data, we are essentially applying the notion of 'testing to destruction' to our repair algorithm. We wish to see how well it performs with repeated changes and no new session data. In more realistic scenarios, the approach would be more effective, because it would be able periodically to update the available session data, resulting in improved effectiveness.

Also, it could be argued that some of the changes were relatively small, corresponding to simple 'fixes' requested by the users. Changes of this nature would be unlikely to form the basis for a full scale release. It is only at the point of full release that new user session data would accrue. Therefore, the results presented on effectiveness could be regarded as describing the effectiveness of the repair approach over a series of increasingly large changes.

## 6. Related Work

Much previous work on regression testing has focused on problems associated with managing the inherent complexity of the task of re-executing a large suite of test data on a changed application [2, 4, 5, 12]. However, the concern of such work has been with classical implementations of systems, not with web applications. Where work on regression testing `has` considered web applications, the focus has been on white box techniques and design issues [7, 15]; no previous work on regression testing web applications has used session data.

There has also been a larger previous body of work on more general aspects of web application testing, apart from regression testing issues [11, 13]. Even in this more general body of work, there is a surprising lack of previous work that considers the possibility of exploiting session data. Indeed the first authors to suggest the use of session data in web application testing were Rothermel et al. in 2003 [6, 3], which showed that the use of session data can reveal faults not found by other techniques such as white box techniques.

The first authors to suggest the use of session data in web application testing are Tonella and Ricca [14], who present an approach to the statistical testing of web applications. Like Rothermel et al., Tonella and Ricca are not concerned with regression testing. Rather, they suggest session data as one possible source of probabilities for a Markov model in order to determine likely paths traversed to guide test effort.

The results of Rothermel et al. point to the important value denoted by the session data collected from the normal execution of a web–based system. However, their work was concerned with comparing the effectiveness of session–based testing with white box testing for fault finding. It was not concerned with regression testing. The present paper is the first paper to present techniques for regression testing of web applications based on the use and re–use of session data.

The work by Rothermel et al. also combines existing session data to from new session data and so it shares with our work an 'active' rather than 'passive' approach to the session data. However, in the work of Rothermel et al., the goal is to generate new valid sequences for the existing web application, rather than to repair existing sessions.

The closest work to ours on (non–regression based) testing of web applications is that of Ricca and Tonella [10, 11], who present automatic support techniques for analysis and testing of web applications. Two tools where developed by Ricca and Tonella for this purpose: ReWeb and TestWeb. ReWeb downloads a web application and creates a UML graph of the relationships between its different parts representing web pages as nodes and links as edges between the nodes. ReWeb also has features to indicate which parts of a web application have been altered and uses a spidering approach, similar to that used in our work and in other web crawling work [8, 9]. However, the ReWeb system does not currently perform any form of regression testing and is not based upon the use of session data.

TestWeb uses the graph produced by ReWeb to generate test suites. The whole process is semi-automatic, with user interventions required at several points. Most noticeably, the user has to provide the set of required parameters and their values for every form encountered by each of the tools. In our work we aim to fully automate the process of regression testing. Because we have session data available to us, we are able to identify possible candidate parameter values rather than relying upon human interven-

tion. Our approach to pre–determining a system of values for cases where no parameter value can be found is similar to the approach used in VeriWeb [1], which introduces a similar 'smart-profiles' concept.

Rothermel et al. [3, 6] also used the 'Online Bookstore' web application to evaluate their work from which they obtained 85 user sessions from 75 participants. The participants were students, who were given specific tasks to perform with the promise of a reward. In our study the participants were 14 members of the general public who were neither giving specific task nor reward and from whom we obtained 40 sessions. Rothermel et al. removed some of the administrator functionality that was deemed irrelevant to their study. In our work, the web application was used unaltered, since we did not wish to make assumptions about the kinds of session data would be relevant to the study.

## 7. Conclusion

This paper has introduced an algorithm for repair of web session data collected from users of the original version of a web application. Repair allows old session data to be reused when testing changes made to the web application, even when these changes would otherwise render the original session invalid. The paper represents the first application of repair algorithms to the problem of maintaining the value of session data as a site evolves.

The paper also reports on the results of controlled experiments into the effectiveness and scalability of the repair–based approach. The results show that the approach can be used to recover much of the value locked in otherwise unusable legacy web sessions, even over multiple evolutionary changes in which no new session data is created.

## References


[1] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *Proceedings of 11th International World Wide Web Conference*, Honolulu, Hawaii, USA, 2002. Appears in ACM SIGSOFT Software Engineering Notes 29(5):1-10, 2004.

[2] D. W. Binkley. The application of program slicing to regression testing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):583–594, 1998.

[3] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, Mar. 2005.

[4] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197. IEEE Computer Society Press, Apr. 1998.

[5] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Los Alamitos, California, USA, 1992. IEEE Computer Society Press.

[6] S. Karre, S. Elbaum, and G. Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Portland, Oregon, USA, May 2003.

[7] T. Margaria, O. Niese, and B. Steffen. A practical approach for the regression testing of IP-based applications. In *IP Applications and Services 2003: A Comprehensive Report*, pages 195–208. International Engineering Consortium (IEC), 2002.

[8] R. C. Miller and K. Bharat. Sphinx: a framework for creating personal, site-specific web crawlers. In *Proceedings of the seventh international conference on World Wide Web 7*, pages 373–388, Brisbane, Australia, 1998. Springer–Verlag.

[9] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *In Proceedings of the 27th International Conference on Very Large Data Bases*, pages 129–138, New York, NY, USA, 2001.

[10] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[11] F. Ricca and P. Tonella. Building a tool for the analysis and testing of web applications: Problems and solutions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 373–388. Springer–Verlag, 2001.

[12] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Transactions on Software Engineering*, 27(10):929–948, Oct. 2001.

[13] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection of web applications. In *Proceedings of International Conference on Automated Software Engineering (ASE 2005)*, pages 253–262, Long Beach, California, USA, 2005.

[14] P. Tonella and F. Ricca. Statistical testing of web applications. *Journal of Software Maintenance*, 16(1-2):103–127, 2004.

[15] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *COMPSAC*, pages 652–656, 2003.