JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:419–433
Published online 10 July 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.371

Research

Extended firewall for regression testing: an experience report

Lee White[1], Khaled Jaber[1], Brian Robinson[2]
and Václav Rajlich[3,*,†]

[1]*Department of EECS, Case Western Reserve University, Cleveland, OH, U.S.A.*
[2]*ABB Corporation, Cleveland, OH, U.S.A.*
[3]*Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.*

SUMMARY

Testing firewalls have proven to be a useful approach for regression testing in both functional and object-oriented software. They involve only the modules that are closely related to the changed modules. They lead to substantially reduced regression tests but still are very effective in detecting regression faults. This paper investigates situations when data-flow paths are longer, and the testing of modules and components only one level away from the changed elements may not detect certain regression faults; an *extended firewall* considers these longer data paths. We report empirical studies that show the degree to which an extended firewall detected more faults, and how much more testing was required to achieve this increased detection. Copyright © 2008 John Wiley & Sons, Ltd.

Received 2 May 2007; Revised 25 April 2008; Accepted 23 May 2008

KEY WORDS: regression testing; firewall regression testing; extended firewall testing; object-oriented software testing

1. INTRODUCTION

Regression testing is an indispensable part of software maintenance and evolution, and more recently also of the iterative and agile software development processes. All these processes are

*Correspondence to: Václav Rajlich, Department of Computer Science, Wayne State University, Detroit, MI 48202, U.S.A.
†E-mail: rajlich@wayne.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: CCF-0438970
Contract/grant sponsor: IBM Faculty Award
Contract/grant sponsor: ABB Corporation

based on a repeated software change, and the goal of the regression testing is to find residual problems in the unchanged parts of the software.

A primitive regression testing strategy would re-run all tests that validate functionality of all software components that were not impacted by the change. However, the regression tests need to be repeated frequently, making the re-run of the complete regression test too expensive; therefore, several heuristics were developed as a response. Selective testing means that although an initial test suite has been developed and is available, the programmers select only a subset pertinent to testing the specific changes made in the software [1,2]. Another approach emphasizes prioritization, where the priority is established among the tests and the tests are run within the available time, based on their priority [3].

Selective and prioritization testing assumes that the regression test suite exists and has the desirable quality. However, we have found that often in industrial practice, the initial test suites are inconsistent and poorly constructed. For this situation, we have developed an approach to regression testing called firewall testing that emphasizes an on-demand test creation rather than selection from a given testing suite; of course, existing tests can always be reused, if they satisfy the firewall criterion. The firewall testing considers only components that are directly interacting with the changed components (one level away) and thoroughly tests them. Firewall testing is based on the observation that most residual bugs are caused by a failure of the programmer to propagate the change to some directly interacting program components. The firewall is an imaginary boundary between these components and the rest of the software system.

As a method of regression testing for industrial situations, the *testing firewall* (*TFW*) has proven to be a useful approach for both procedure-oriented and object-oriented software. It has turned out to be particularly useful in the situations where the test suites are incomplete, which is a common situation in industrial software development; the firewall allows the identification of missing tests and provides a simple procedure to develop these missing tests. TFW has been developed for various types of software and documented as successful, leading to substantially reduced regression tests [4–9]. A more complete view of firewalls in object-oriented systems (oo-systems), including polymorphism, is in [7].

In this paper, we study an *extended firewall* (*EFW*) that takes into account the role of data flows between software components. We presented an early version of the study in a short paper by White and Robinson [10], but the present paper provides more details.

Section 2 of this paper explains the concepts of *plain* and *encoded* values and the role of the data flows. Section 3 describes the EFW. Two empirical studies in Section 4 illustrate the extent to which the EFW detected more faults than the TFW, and how much additional testing was required to achieve this increased detection. Section 5 provides a discussion of related work, and Section 6 discusses the conclusions and future work.

2. ROLE OF THE DATA FLOWS

Data flows have long been used for testing in procedural software systems [11–16]. The use of data flows in oo-systems was explored in [17–19] and *inter-class data flows* were defined in [18,19]. Yu and Rajlich [20] investigated *hidden dependencies* in oo-systems, which are data-flow dependencies between two seemingly independent components.

DOI: 10.1002/smr

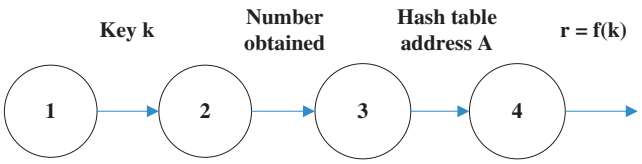


Figure 1. Hash table: an example of an external object.

In oo-systems, an object-oriented component is *inactive* when it is not processing information; in this mode, none of its methods is computing. A component is *active* when one or more methods of it are computing, and the component is processing information. When *activated*, a component makes a transition from inactive to active. This usually happens when a component receives a message, activating that component to perform computations with its constituent methods, and in turn, it may send messages to other components. The messages may contain information in the form of parameters that are transferred to the receiving component. Consider Figure 1 as an example; an arc in Figure 1 corresponds to a message from one component to another, and components are denoted by circles. As messages are sent from component 1 to 2, component 2 becomes activated; subsequent messages activate components 3 and 4, at which time the overall designed functionality is achieved.

A central issue for the analysis of data-flow paths is whether the component deals with the *plain values* of the given inputs, or whether the component must account for how those inputs were generated [20]. As an example of the first case, consider a component that just sorts the given inputs. This sorting component does not depend on the components that generate these values.

Next, consider the situation when a component's inputs are *encoded values*, as illustrated by the hash table in Figure 1. Component 1 generates the key k (which might be just input by the user); component 2 algorithmically transforms k into an integer; the resulting integer is converted to a hash table address A (possibly by a modulo table size operation) by component 3; component 4 accesses address A of the hash table and extracts the result r. The result r of component 4 is not just a function of the input address A but also is a function of key k and the processing that produced address A. Thus, if component 2 or 3 is changed, the result r may change; if k changes, we expect A to be different. We express these observations in the following definition:

Definition 1. Let C be a component with inputs $I_1, \ldots, I_m$ and outputs $O_1, \ldots, O_n$ with postconditions $Q_1, \ldots, Q_n$. If there is $Q \in \{Q_1, \ldots, Q_n\}$ and $I \in \{I_1, \ldots, I_m\}$ such that Q depends not only on value of I but also upon a value generated by a component $C1$ that is connected to I through a data flow through one or more intermediary components $C2, \ldots, Cj$, then I is called the *encoded input*; then C is called the *external component* (E component) *in scope of* $C2$, and $C2$ is the external component in scope of C, making 'external component in scope' a symmetric relation. If there is no such component $C1$, then we call I a *plain value* input. If all inputs of component C are plain values, we call component C an *input/output* (I/O) component.

In the hash table example of Figure 1, the postcondition of component 4 depends on the value k generated by component 1, hence input A is encoded, component 4 is an external component in scope of component 2, and component 2 is an external component in scope of component 4. In the example of sorting component, the postcondition is a sorted set of values and this is dependent only on the values of the input; hence, the input to this component is a plain value input.

DOI: 10.1002/smr

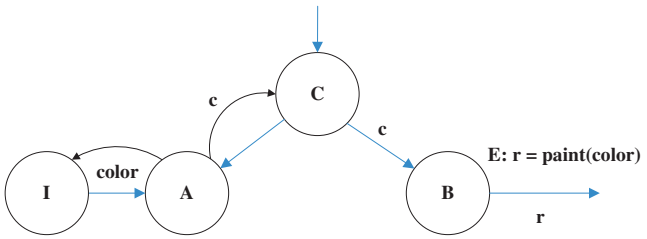


Figure 2. An example of the hidden dependency.

Definition 2. If component *C* is external in scope of component *D* and there is no message send from *C* to *D* or from *D* to *C*, then there is a hidden dependency between *C* and *D*.

In the hash table example of Figure 1, component 2 is external in scope of component 4 while there is no message sent from component 2 to component 4 or *vice versa*, and hence there is a hidden dependency between components 2 and 4.

In the analysis of external components, it is not necessary to know the exact formal definition of their postcondition, but only that a specific dependence on specific values exists. In the remainder of Section 2, we give several illustrations of external components.

It needs to be emphasized that the categories of external and I/O components can only be defined within the context of a given data-flow path through those components. As we consider different data-flow paths, the same component that appears on different paths may be external on some data paths and I/O on other data paths. When a message without data is encountered, then a data path is terminated at that point.

2.1. Effects of return messages

The data-flow path can also contain return messages. Hidden dependencies can consist of both the forward and return messages [20], as shown in Figure 2. In it, class A contains the member function *int A::get()* that receives a choice of a color from the user input *I* and converts it into an integer code *c*, where value '0' means 'red', '1' means 'yellow', and so forth. Class B contains member function *void B::paint(int)* that receives the color code as an argument, decodes it, and paints the screen by the corresponding color. Class C contains the following code fragment that establishes the data flow through it:

```
class C {...
  A a;
  B b; ...
  void foo() { ...
    b.paint(a.get());
    ...
    }
};
```

Then class B is external in scope of A, because its postcondition *r = paint(color)* is dependent on value 'color' that is generated in class *I*, encoded as an integer in class A, and decoded in class B.

DOI: 10.1002/smr

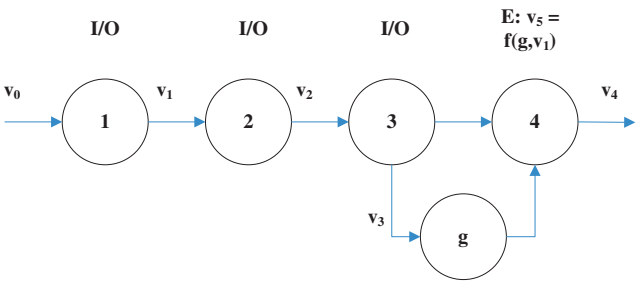


Figure 3. Global variable as part of a data flow.

There is also a hidden dependency between A and B because no message is sent from A to B or from B to A. A code modification in A that changes the encoding of color will prompt a corresponding code modification in B. If the programmer changes only A, the program will compile and run, in spite of the fact that it may have an error in it. All remaining classes of Figure 2 are I/O.

2.2. Effects of global variables

It is also possible that data-flow paths might be using global variables. In Figure 3, the activation path is given as components (1, 2, 3, and 4) and the messages between them; the data path is given as (1, 2, 3, g, 4). The output v_3 of component 3 is then used to modify the global variable g. This same global variable g will then be used as an input to component 4. Note that component 4 is external and dependent on variable v_1. Component 4 still needs to be activated, and this is accomplished with the message from component 3 to component 4, which has no data associated with it.

3. OBJECT-ORIENTED FIREWALLS

The TFW is based on first-level dependencies of modified components [6,7]. An extensive study of changes of components [7] has identified that the most serious effects are in those components that send messages to the changed components, and they need to be extensively tested. TFW also contains components that receive a message from the changed component; the signature and origin of these messages must be checked to ensure that they have not changed. For example, if the method sending the message has changed, or if any parameter in the signature of this message has changed, then this message and the method to which it is sent must be retested. It is important to note that TFW does not examine dependencies further away than one level from a changed component.

The following process determines TFW in oo-systems:

(1) Given two successive versions (or builds) of an oo-system, find the difference of the two versions and identify those classes that have changed.
(2) If any of the changed classes are in an inheritance hierarchy, also consider descendants of the changed class as changed, unless all of the changed parts have been overridden.
(3) For each changed class, identify all the classes that send messages to the changed class or receive messages from the changed class and include them in the TFW.

DOI: 10.1002/smr

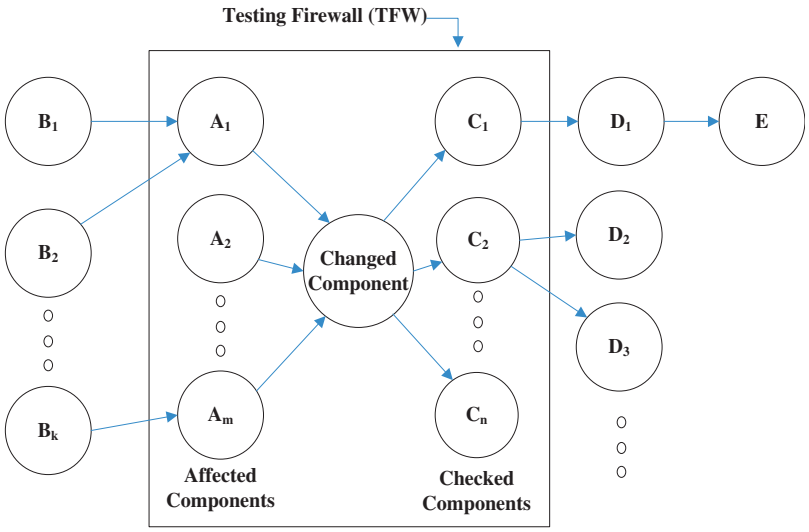


Figure 4. Testing firewall (TFW).

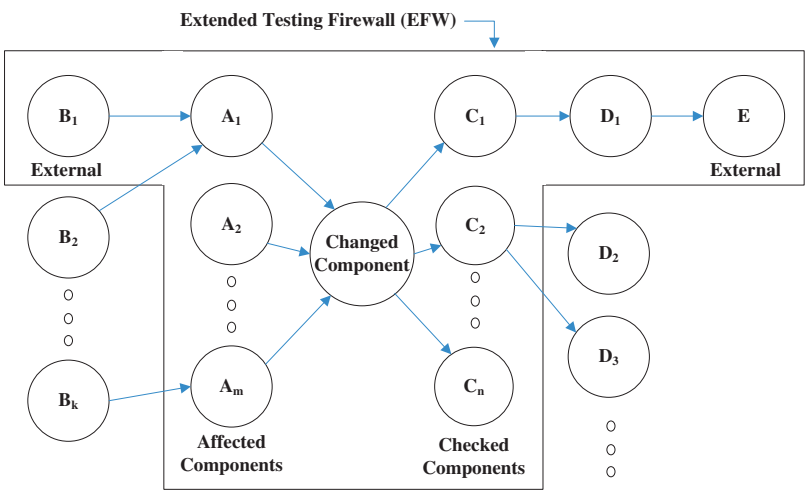


Figure 5. Extended testing firewall (ETW).

This defines the TFW within which all classes must be tested. In our experience, the tester would benefit from the representation of the TFW in a digraph form, similar to Figure 4.

3.1. The extended firewall

EFW extends the TFW by taking into account the data flows and external components as discussed in Section 2; see the illustration in Figure 5. The EFW is obtained by the application of the following

DOI: 10.1002/smr

process:

First apply steps (1)–(3) in order to obtain the TFW.

(4) Identify all data paths from and to a modified class.
(5) Find all external classes in scope of the modified class, such as *E* in Figure 5. Include all such classes in EFW.

Because EFW is a superset of TFW, it will always detect all faults detected by the TFW.

Step (5) requires some additional explanation. If we examine the code of component *E*, and if it contains conditionals that utilize the values of input *I*, then almost certainly *E* is external on the data path that involves input *I*. If, on the other hand, all the conditionals in *E* do not utilize the specific values of input *I*, then almost certainly *E* is an I/O component on the data path that involves input *I*. As an exception to this rule, if *E* compares two of these values to see which is larger, then *E* is also likely to be an I/O component. We applied this heuristics in the case studies explained in the next section and following this heuristics makes the case studies of the next section reproducible. We also used 'light-weight' analysis of the program dependencies and data flows, using the widely available software tools.

4. EMPIRICAL STUDIES OF TFW AND EFW

The objective of the empirical studies described in this section is to compare the tradeoffs in the performance of the TFW and EFW. We particularly seek to understand whether EFW finds additional faults, and what are the costs of this additional effort.

For that purpose, we selected two case studies of very different industrial targets, and different programmers in different companies conducted the studies; the diversity is intentional, for we wanted to see the effects of EWT in very different settings. The study in Section 4.1 utilizes telecommunication software and the study in Section 4.2 uses real-time control software. The telecommunication software is smaller in scale and the study was completed first, while the real-time software is a very large software system. Both systems are implemented in C++.

In both case studies, the programmers constructed TFW and EFW using commonly available software tools in combination with visual inspections of the relevant code. They collected data on the number of the tests, number of the faults discovered by the tests, time required for the analysis, and time required for testing.

4.1. Empirical study of telecommunication software

This study involved an object-oriented telecommunication software system; the base version contains 66 classes with 11K lines of C++ source code that runs on a Unix platform. We have identified three successive versions, constructed after two builds, five builds, and four builds, respectively.

4.1.1. Empirical study description

An industrial expert familiar with this system performed the analysis and testing and was trained in both TFW and EFW processes by one of the authors. The modified classes between successive builds

DOI: 10.1002/smr

were determined manually; then a Unix-based tool *Cscope* was used to determine dependencies that involved the modified classes. The TFW and EFW were then constructed manually based on this dependency information.

4.1.2. Telecommunication empirical study results

Table I shows regression testing information for the three versions of this system and their builds; the builds are indicated by rows in the table. The second column indicates the number of classes changed (or new classes) for each build. The data in the next columns correspond to the regression test data for either the TFW or EFW. The last two columns indicate the faults that were found by firewall testing. *Common faults* are the faults that were detected by both the TFW and the EFW, and the last column represents faults detected only by the EFW.

Note that in build 1.1, we detected one common fault and another distinct EFW fault; in this instance, two objects were modified and we found two distinct faults, one due to each modified object. For all the other builds in Table I, we detected only one fault. At the end of the builds for each version, the total number and type of faults found for that version are indicated. Please note that a total number of eight faults were detected by TFW and additional four faults were detected by EFW only, hence EFW provided an increase in 50% of faults detected.

Some additional information can be provided about the faults reported in Table I. The EFW faults reported in builds 3.1 and 3.4 were new faults not previously known for this software, leading to the conclusion that the EFW detected more subtle faults than the TFW. In build 2.5, EFW reported a fault that was missed previously by unit/system testing.

Table I. TFW and EFW classes, methods tested, and faults detected in a TelComm Study.

Builds	Modified classes	TFW methods	EFW methods	TFW classes	EFW classes	Common faults detected	Faults detected by EFW only
Version 1							
1.1	2	5	13	5	7	1	1
1.2	1 (new)	2	3	2	3	1	
Total	3	7	16	7	10	2	1
Version 2							
2.1	1	3	3	3	3	1	
2.2	1	1	1	1	1	1	
2.3	1	2	3	2	3	1	
2.4	1	3	5	3	4	1	
2.5	1 (new)	1	2	1	2		1
Total	5	10	14	10	13	4	1
Version 3							
3.1	1	1	2	1	2		1
3.2	1	4	4	4	4	1	
3.3	1	1	3	1	2	1	
3.4	1	1	2	1	2		1
Total	4	7	11	7	10	2	2

DOI: 10.1002/smr

Table II. Effort required to test TFW, EFW: number of tests and time.

Version	TFW tests	EFW tests	Extra EFW tests (%)	TFW analysis (h)	EFW analysis (h)	TFW test time (h)	EFW test time (h)	Extra EFW time (%)
1	88	115	31	6	13	15.7	20.2	53
2	90	105	17	7	11	16.0	18.5	28
3	80	110	38	5	10	14.3	19.3	52

An important issue is also the *severity* of these faults. This industrial firm classifies the *severity* of software faults as follows:

Severity	Description
1	Fatal application error
2	Application is severely impaired (no workaround can be found)
3	Some functionality is impaired (but workaround can be found)
4	Minor problem not involving primary functionality

In Table I, all the faults reported were at severity 3, except for build 1.1, where the EFW fault is at severity 2. No faults were at severity 1 or 4.

The other aspect of this EFW study is to determine the extent of the extra effort it took to detect these additional faults. Table II indicates this extra effort. Columns 2–4 deal with the number of required tests. Another aspect of effort is the amount of time required to test the modified build; this involves both the analysis time and the testing time; columns 5–8 provide these data for both the TFW and EFW tests. The last column of Table II indicates the per cent increased time for EFW testing.

For both TFW and EFW testing, the total time for testing was the sum of the time for analysis plus test time. The test time consisted of setup time plus time to execute the tests plus time to verify results. These data are shown in Table II. The last column of Table II shows the per cent additional EFW total time for testing over that for the TFW.

Table II shows that the results of additional EFW tests (in column 4) and additional EFW time for testing (column 9) were quite close for versions 1 and 3, but are much less for version 2, since the number of additional tests and additional time for EFW analysis were both lower for version 2. The average figures here are 28% additional tests and 44% additional EFW time for testing over the three versions.

Tests for TFW were obtained by using some existing tests and creating additional tests, as indicated in Table III. All extra tests just for EFW had to be created from scratch, and their numbers are in the last column of Table III.

4.2. Empirical study of real-time control software

This study involves three consecutive versions of an object-oriented real-time software system at ABB Inc. running in a Windows environment. There is a base version that has over one million

DOI: 10.1002/smr

Table III. Reused versus new tests.

Version	TFW tests	TFW reused tests	TFW new tests	EFW test	EFW new tests
1	88	58	30	115	27
2	90	61	29	105	15
3	80	53	27	110	30

Table IV. TFW and EFW classes, methods tested, and faults detected.

Version	Modified classes	TFW methods	EFW methods	TFW classes	EFW classes	Common faults detected	Faults detected by EFW only
1	29 (12 new)	181	239	86	101	5	1
2	18 (3 new)	145	163	82	94	8	2

lines of code and contains thousands of classes; the two consecutive versions contain changes due to both fault corrections and product enhancements.

4.2.1. Empirical study description

One of the authors of this paper at ABB provided the analysis of both versions. Several members of the testing staff were helping in this effort and they were unaware of the specific faults that had previously been found in this software. They selected tests from previously defined tests, and in the case of EFW, they defined and executed additional tests. Although this industrial firm has begun to use the TFW in their testing process whenever modifications to the software are made, TFW had not been applied to this specific software before. Two software tools were utilized in this process, *Araxis* merge and *D'oxygen*. *Araxis* merge was used to identify the modified C++ code by taking differences between versions. After these modified classes were determined, *D'oxygen* was applied to determine dependencies for each modified class, and then produced graphs indicating these dependencies. From this dependency information, the TFW and EFW were constructed manually.

4.2.2. Real-time empirical study results

Table IV shows the regression testing information for the two versions of this system. Note that six faults were found when these firewalls were utilized for each of the 29 modified (or new) objects for version 1. As for the severity of these faults, we will use the same severity coding as we did for the telecommunication system in Section 4.1.2. For the common faults, three were at level 4, one was at level 3 (workaround possible), and one at level 2 (no workaround possible). The one ETW fault was at level 2.

In version 2, of the eight common faults, six were at level 4, and two at level 2. For the two ETW faults, one was at level 3 and the other fault was at level 2.

DOI: 10.1002/smr

Table V. Effort required to test TFW, EFW: number of tests and time required.

Version	TFW tests	EFW tests	Extra EFW tests (%)	TFW analysis (h)	EFW analysis (h)	TFW test time (h)	EFW test time (h)	Extra EFW time (%)
1	168	219	30	20	25	35	48	37
2	112	141	26	17	20	21	27	29

In summary, TFW detected 13 faults while EFW detected additional three faults, an increase of 23% in fault detection.

Table V has been constructed similar to Table II to allow comparison.

4.3. Discussion of the results of the case studies

In both case studies, the EFW detected more faults than the TFW. The increase in the number of faults found by EFW as opposed to TFW was 50% in the first case study and 23% in the second case study. On the side of the cost, increase in the number of tests required by EFW in Table II was 31, 38, and 17%, and in Table V, additional EFW tests amounted to 30 and 26%; if we consider the figure 17% seems to be an outlier, the increased cost of EFW indicates a considerable consistency.

4.4. Threats to validity

As with other case studies, the results of this case study should be generalized with caution. We reported that the EFW produced more faults than the TFW at the cost of approximately 30% of extra effort, but it must be understood that if applied to other software, no additional faults might be found and different costs may be reported. In addition, it must be understood that EFW does not guarantee to find all regression faults as there may be regression faults lying outside EFW, or the tests may not discover some faults that lie within EFW.

There were separate analysis and test generation processes for the two case studies. While we made a considerable effort to guarantee the consistency between the two case studies, the differences still may have influenced the results.

The collected data depend on the technical skills of the programmers who conducted the case studies and on their knowledge of the system under analysis. The background of our programmers may differ from the background of other testers, who may be more (or less) familiar with the system they are testing. Our programmers might be more experienced with the use of EFW.

The programmers who conducted the case studies were also the co-authors of this paper and they were familiar with the objectives of the case studies; therefore, they were potentially biased toward the positive results.

5. RELATED WORK

Examples of selective regression testing are in References [1–3,21–23]. The implicit assumption of this approach is that the initial test suite is of high quality and thoroughly tests the software.

DOI: 10.1002/smr

Although these selective regression testing methods do not preclude adding further tests, this is not the major thrust of selective regression testing.

The firewall approach was developed for industrial situations where testing is not performed on a systematic basis and tests may be incomplete. In that situation, the selective regression testing will not work well, because a relatively large number of new tests would be needed and the industrial firm would need some guidance in the generation of these new tests. The firewall approach offers a systematic way that efficiently develops missing regression tests. For example, ABB Inc. could not perform effective regression testing several years ago, but since they adopted the firewall approach and added it to their testing arsenal, the TFW test data have been generated for all new software releases. The effectiveness of the TFW approach for ABB is documented in Reference [9]. In the empirical studies done in Section 4.2, the additional tests for EFW were not previously available and had to be generated.

The study of dependencies in maintenance testing has been an important issue for many years. For example, since 1989, Norman Wilde systematically studied the types of dependencies to be addressed by regression testing and provided an automated tool for detecting these dependencies. This research and related work is given in references [24–28]. Another example of research in both regression testing and impact analysis is that of Yau *et al.* [29].

A TFW for procedural software was proposed by Leung and White [6], and extensive evidence for its effectiveness for procedural software is given in Reference [9]. The TFW was applied to object-oriented software, and the firewall was defined for this case by Kung *et al*. [4,5] and by White and Abdullah [7], who provided for additional features of oo-systems, such as polymorphism. In 2003, White *et al*. [8] applied the firewall concept to the regression testing of graphics user interfaces. In 2005, White *et al.* [9] showed how the firewall could be applied to other aspects of real-time software, such as the problem of deadlock detection.

The motivation for the EFW was in part based on the research done on data-flow testing over the years; for example, the data-flow testing research of Bogdan Korel [13–16,30]. Yu and Rajlich explored hidden dependencies in Reference [20]; they are dependencies between two seemingly independent components (classes), where a change in a class A propagates through an unchanged intermediate class C to another class B.

Since the EFW is based on the analysis of the data flows, the issue of precise identification of data flows in oo-systems is relevant to the development of the tools that would support the EFW. Analysis of data flows was discussed in many publications; see, for example, References [17–19].

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the EFW and conducted the case studies where the extra cost for EFW over TFW was about 30% in additional tests, and 30–50% in extra time for analysis and test execution. This extra effort discovered additional 30% of faults in the first case study and 23% of faults in the second case study. Further investigation is needed, both analytical and empirical, to see whether this continues to be the case.

These results could serve as a data point to the programmers and managers, when they decide whether to incorporate EFW into their process and what should be its role. For example, it might turn out that the TFW could be recommended for routine incremental changes of a software system

DOI: 10.1002/smr

to achieve lower testing cost, whereas the EFW could be a good investment for critical situations, such as issuing a new release to customers or dealing with a major design change.

We are currently working on an automated tool for the TFW and EFW. This tool will automate nearly all of the analysis for determining the firewall and be an aide to test development and the reuse of existing tests for the TFW and EFW. The use of such a tool should substantially reduce the analysis overhead reported in this paper.

ACKNOWLEDGEMENTS

The authors would like to thank Brent Nye for his assistance in the analysis and testing of the telecommunication software and also acknowledge several useful conversations with Dapeng Liu.

This study was partially supported by grant CCF-0438970 from National Science Foundation (NSF) and 2005 and 2006 IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF or IBM. We also wish to acknowledge the current financial support of ABB Corporation to develop an automation tool for the TFW process.

REFERENCES

1. Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 1998; **24**(6):401–419.
2. Rothermel G, Harrold MJ, Dedhia J. Regression test selection for C++ software. *Journal of Software Testing, Verification and Reliability* 2000; **10**(2):77–109.
3. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
4. Kung D, Gao J, Hsia P, Wen F, Toyoshima Y, Chen C. Change impact identification in object oriented software maintenance. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1994*). IEEE Computer Society: Los Alamitos CA, 1994; 202–211.
5. Kung DC, Gao J, Hsia P. Class firewall, test order, and regression testing of OO programs. *Journal of Object-Oriented Programming* 1995; **8**(2):51–65.
6. Leung HKN, White L. A study of integration testing and software regression at the integration level. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1990*). IEEE Computer Society: Los Alamitos CA, 1990; 290–301.
7. White L, Abdullah K. A firewall approach for the regression testing of object-oriented software. *Proceedings of the Software Quality Week* (*SQ 1997*). Software Research Inc.: San Francisco CA, 1997; 27.
8. White L, Almezen H, Sastry S. Firewall regression testing of GUI sequences and their interactions. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 2003*). IEEE Computer Society: Los Alamitos CA, 2003; 398–409.
9. White L, Jaber K, Robinson B. Utilization of extended firewall for object-oriented regression testing. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 2005*). IEEE Computer Society: Los Alamitos CA, 2005; 695–698.
10. White L, Robinson B. Industrial real-time regression testing and analysis using firewalls. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 2004*). IEEE Computer Society: Los Alamitos CA, 2004; 18–27.
11. Antoniol G, Fiutem R, Lutteri G, Tonella P, Zanfei S, Merlo E, Irst T. Program understanding and maintenance with the CANTO environment. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM'97*). IEEE Computer Society: Los Alamitos CA, 1997; 72–81.
12. Gupta R, Soffa ML. A framework for partial data flow analysis. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1994*). IEEE Computer Society: Los Alamitos CA, 1994; 4–13.
13. Korel B. PELAS—program error-locating assistant system. *IEEE Transactions on Software Engineering* 1988; **14**(9): 1243–1252.

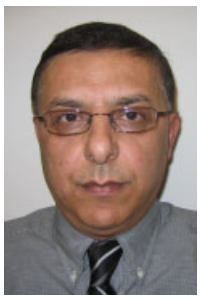
DOI: 10.1002/smr

14. Korel B, Al-Yami AM. Automated regression test generation. *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA 1998*). ACM: New York NY, 1998; 143–152.
15. Korel B, Laski J. A tool for data flow oriented program testing. *Proceedings IEEE Conference on Software Development Tools*, *Techniques*, *and Alternatives* (*Softfair II 1985*). IEEE Computer Society: Los Alamitos CA, 1985; 34–37.
16. Korel B, Tahat LH, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 2002*). IEEE Computer Society: Los Alamitos CA, 2002; 214–223.
17. Harrold MJ, Rothermel G. Performing data flow testing on classes. *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering* (*FSE 1994*). ACM: New York NY, 1994; 154–163.
18. Souter AL, Pollock LL. Contextual def–use associations for object aggregation. *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering* (*PASTE 2001*). ACM: New York NY, 2001; 13–19.
19. Souter AL, Pollock LL, Hisley D. Inter-class def–use analysis with partial class representations. *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering* (*PASTE 1999*). ACM: New York NY, 1999; 47–56.
20. Yu Z, Rajlich V. Hidden dependencies in program comprehension and change propagation. *Proceedings IEEE International Workshop on Program Comprehension* (*IWPC 2001*). IEEE Computer Society: Los Alamitos CA, 2001; 293–299.
21. Agrawal H, Horgan JR, Krauser EW, London SA. Incremental regression testing. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1997*). IEEE Computer Society: Los Alamitos CA, 1993; 348–357.
22. Bible J, Rothermel G, Rosenblum DS. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology* 2001; **10**(2):149–183.
23. Elbaum S, Kallakuri P, Malishevsky A, Rothermel G, Kanduri S. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing*, *Verification and Reliability* 2003; **13**(2):65–83.
24. Lejter M, Meyers S, Reiss SP. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering* 1992; **18**(12):1045–1052.
25. Lieberherr KJ, Xiao C. Object-oriented software evolution. *IEEE Transactions on Software Engineering* 1993; **19**(4): 313–343.
26. Wilde N, Huitt R. Maintenance support for object oriented programs. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1991*). IEEE Computer Society: Los Alamitos CA, 1991; 162–170.
27. Wilde N, Huitt R. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering* 1992; **18**(12):1038–1044.
28. Wilde N, Huitt R, Huitt S. Dependency analysis tools: reusable components for software maintenance. *Proceedings IEEE International Conference on Software Maintenance* (*ICSM 1989*). IEEE Computer Society: Los Alamitos CA, 1989; 126–131.
29. Yau S, Collofello JS, MacGregor T. Ripple effect analysis of software maintenance. *Proceedings IEEE International Computer Software and Applications Conference* (*COMPSAC 1978*). IEEE Computer Society: Los Alamitos CA, 1978; 60–65.
30. Laski JW, Korel B. Data flow oriented program testing strategy. *IEEE Transactions on Software Engineering* 1983; **9**(3):347–354.

AUTHORS' BIOGRAPHIES

Lee White is a Professor Emeritus of EECS at Case Western Reserve University, which he joined in 1988. He received his BSEE from the University of Cincinnati, and his MSc and PhD degrees in EECE from the University of Michigan. He served as an American Editor of the *Journal of Software Testing*, *Verification and Reliability* from 1990 to 2007, and as the Chair of computer science at Ohio State University, University of Alberta and CWRU. His research interests include software testing, most recently in GUI testing and regression testing.

DOI: 10.1002/smr



Khaled Jaber is a PhD student in computer science at Case Western Reserve University. His research is in software testing. He received his MS in computer science from Northeastern Illinois University in 1990. He has an extensive industrial experience in software development and testing. He was a software engineer at Lucent Technologies (now Alcatel-Lucent), Bell Laboratories. Currently, he is a supervisor at Emerson Network Power leading a software automation development and testing team.

Brian Robinson is a Principal Scientist in ABB's Corporate Research group. He received his PhD in computer science from Case Western Reserve University. At ABB, Brian is the technical lead for all software quality and testing research. His current research involves improving software engineering practices by addressing issues that arise in real development projects. He particularly enjoys bridging academic work into industrial practice and is actively collaborating with many Universities around the world.

Václav Rajlich is a full professor and former chair in the Department of Computer Science at Wayne State University. He published extensively in the areas of software evolution, comprehension, and agile development. He received his MS from Czech Technical University and PhD from Case Western Reserve University. Contact him at rajlich@wayne.edu.

DOI: 10.1002/smr