Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact

Hyunsook Do, Sebastian Elbaum, Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE
{dohy, elbaum, grother}@cse.unl.edu

July 22, 2005

Abstract

Where the creation, understanding, and assessment of software testing and regression testing techniques are concerned, controlled experimentation is an indispensable research methodology. Obtaining the infrastructure necessary to support such experimentation, however, is difficult and expensive. As a result, progress in experimentation with testing techniques has been slow, and empirical data on the costs and effectiveness of techniques remains relatively scarce. To help address this problem, we have been designing and constructing infrastructure to support controlled experimentation with testing and regression testing techniques. This paper reports on the challenges faced by researchers experimenting with testing techniques, including those that inform the design of our infrastructure. The paper then describes the infrastructure that we are creating in response to these challenges, and that we are now making available to other researchers, and discusses the impact that this infrastructure has had and can be expected to have.

Keywords: software testing, regression testing, controlled experimentation, experiment infrastructure.

1 Introduction

Testing is an important engineering activity responsible for a significant portion of the costs of developing and maintaining software [4, 28]. It is important for researchers and practitioners to understand the tradeoffs and factors that influence testing techniques. Some understanding can be obtained by using analytical frameworks, subsumption relationships, or axioms [37, 40, 46]. In general, however, testing techniques are heuristics and their performance varies with different scenarios; thus, they must be studied empirically.

The initial, development testing of a software system is important; however, software that succeeds evolves, and over time, more effort is spent re-validating a software system's subsequent releases than is spent performing initial, development testing. This re-validation activity is known as *regression testing*, and includes tasks such as re-executing existing tests [34], selecting subsets of test suites [9, 41], prioritizing test cases to facilitate earlier detection of faults [16, 42, 49], augmenting test suites to cover system enhancements [6, 39], and maintaining test suites [20, 21, 30]. These tasks, too, involve many cost-benefits tradeoffs and depend on many factors, and must be studied empirically.

Many testing and regression testing techniques involve activities performed by engineers, and ultimately we need to study the use of such techniques by those engineers. Much can be learned about testing techniques,

however, through studies that focus directly on those techniques themselves. For example, we can measure and compare the fault-revealing capabilities of test suites created by various testing methodologies [18, 22], the cost of executing the test suites created by different methodologies [5], or the influence of choices in test suite design on testing cost-effectiveness [38]. Such studies provide important information on tradeoffs among techniques, and they can also help us understand the hypotheses that should be tested, and the controls that are needed, in subsequent studies of humans, which are likely to be more expensive.

Empirical studies of testing techniques, like studies of engineers who perform testing, involve many challenges and cost-benefits tradeoffs, and this has constrained progress in this area. In general, two classes of empirical studies of software testing can be considered: controlled experiments and case studies. Controlled experiments focus on rigorous control of variables in an attempt to preserve internal validity and support conclusions about causality, but the limitations that result from exerting control can limit the ability to generalize results [44]. Case studies sacrifice control, and thus, internal validity, but can include a richer context [51]. Each of these classes of studies can provide insights into software testing techniques,[1] and together they are complementary; in this article, however, our focus is controlled experimentation.

Controlled experimentation with testing techniques depends on numerous software-related artifacts, including software systems, test suites, and fault data; for regression testing experimentation, multiple versions of software systems are also required. Obtaining such artifacts and organizing them in a manner that supports controlled experimentation is a difficult task. These difficulties are illustrated by the survey of recent articles reporting experimental results on testing techniques presented in Section 2 of this article. Section 3 further discusses these difficulties in terms of the challenges faced by researchers wishing to perform controlled experimentation, which include the needs to generalize results, ensure replicability, aggregate findings, isolate factors, and amortize the costs of experimentation.

To help address these challenges, we have been designing and constructing infrastructure to support controlled experimentation with software testing and regression testing techniques.[2] Section 4 of this article presents this infrastructure, describing its organization and primary components, our plans for making it available and augmenting it, some examples of the infrastructure being used, and potential threats to validity. Section 5 concludes by reporting on the impact this infrastructure has had, and can be expected to have, on further controlled experimentation.

2 A Survey of Studies of Testing

To provide an initial view on the state of the art in empirical studies of software testing, we surveyed recent research papers following approaches used by Tichy et al. [43] and Zelkowitz et al. [52]. We selected

[1] A recent study by Juristo et al. [24], which analyzes empirical studies of testing techniques over the past 25 years, underscores the importance of such studies.

[2] This work shares similarities with activities promoted by the International Software Engineering Research Network (ISERN). ISERN, too, seeks to promote experimentation, in part through the sharing of resources; however, ISERN has not to date focused on controlled experimentation with software testing, or produced infrastructure appropriate to that focus.

Table 1: Research papers involving testing and empirical studies in six major venues, 1994-2003.

Year	TSE			TOSEM			ISSTA[3]			ICSE			FSE			ICSM		
	P	T	EST	P	T	EST	P	T	EST	P	T	EST	P	T	EST	P	T	EST
2003	74	8	7	7	0	0	-	-	-	75	7	3	33	4	4	39	4	3
2002	74	8	4	14	2	0	23	11	4	57	4	4	17	0	0	61	9	8
2001	55	6	5	11	4	3	-	-	-	61	5	3	28	3	3	68	4	3
2000	62	5	2	14	0	0	21	10	2	67	5	2	17	2	2	24	0	0
1999	46	1	0	12	1	0	-	-	-	58	4	2	29	1	1	36	4	2
1998	73	4	3	12	1	1	16	9	1	39	2	2	22	2	1	32	3	3
1997	50	5	4	12	1	1	-	-	-	52	4	2	27	3	1	34	2	1
1996	59	8	2	13	5	2	29	13	1	53	5	3	17	2	1	34	1	1
1995	70	4	1	10	0	0	-	-	-	31	3	1	15	1	1	30	4	1
1994	68	7	1	12	3	1	17	10	1	30	5	2	17	2	0	38	3	1
Total	631	56	29	117	17	8	106	53	9	523	44	24	222	20	14	396	34	23

two journals and four conferences recognized as pre-eminent in software engineering research and known for including papers on testing and regression testing: IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), the ACM/IEEE International Conference on Software Engineering (ICSE), the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), and the IEEE International Conference on Software Maintenance (ICSM). We considered all issues and proceedings from these venues over the period 1994 to 2003.

Table 1 summarizes the results of our survey with respect to the numbers of full research papers appearing in each venue per year. Of the 1,995 full research papers contained in the issues and proceedings of the six venues we considered, we identified 224 papers on topics involving software testing issues such as testing techniques, test case generation, testing strategies, and test adequacy criteria. We examined these papers, and determined that 107 reported results of empirical studies. In this determination, we included all papers that either described their results as "empirical" or clearly evaluated their proposed techniques or methods through empirical studies.

The table contains three columns of data per venue: P (the total number of papers published in that year), T (the number of papers about software testing published in that year), and EST (the number of papers about software testing that contained some type of empirical study). As the table shows, 11.2% (224) of the papers in the venues considered concern software testing, a relatively large percentage attesting to the importance of the topic. (This includes papers from ISSTA, which would be expected to have a large testing focus, but even excluding ISSTA, 9% of the papers in the other venues, that focus on software engineering generally, concern testing.) Of the testing-related papers, however, only 47.7% (107) report empirical studies.

These data might be considered to be biased because the authors of this article have been responsible for several of the papers considered in this survey. To assess whether such bias had influenced our results, we

[3]ISSTA proceedings appear bi-annually.

also considered the same data with those papers eliminated from consideration. In this case, however, the foregoing percentages become 10.4% (papers concerning software testing), 8.2% (papers concerning software testing excluding those at ISSTA), 43.4% (empirical studies among testing-related papers), respectively, and continue to support our conclusions (Table 5 in Appendix A presents full results for this view).

We next analyzed the 107 papers on testing that reported empirical studies, considering the following categories, which represent factors important to controlled experimentation on testing and regression testing:

- The type of empirical study performed.
- The number of programs used as sources of data.
- The number of versions used as sources of data.
- Whether test suites were utilized.
- Whether fault data was utilized.
- Whether the study involved artifacts provided by or made available to other researchers.

Determining the type of empirical study performed required a degree of subjective judgement, due to vague descriptions by authors and the absence of clear quantitative measures for differentiating study types. However, previous work [2, 27, 52] provides guidelines for classifying types of studies, and we used these to initially determine whether studies should be classified as controlled experiments, case studies or examples. An additional source for distinguishing controlled experiments from other types of studies was Wohlin et al.'s criteria [47], and focused primarily on whether the study involved manipulating factors to answer research questions.

Many studies were relatively easy to classify: if a study utilizes a single program or version and it has no control factor then it is clearly a case study or an example; if a study utilizes multiple programs and versions and it has multiple control factors then it is a controlled experiment. However, some studies were difficult to classify due to a lack of description of experiment design and objects.

The categories listed above (the number of programs and the number of versions empirical studies used, whether they used tests and faults, and whether they were involved in artifact sharing) are essential elements to consider in the process of classification of the type of an empirical study, but further consideration of the experiment design was also required. For example, if a study used all types of artifacts listed in Table 2 but its experiment design did not manipulate any factors, then we classified it as a case study or an example. On the other hand, if a study involved a single program but its experiment design manipulated and controlled factors such as versions or tests, we classified it as an experiment.

Having determined which papers presented controlled experiments, we next considered the remaining papers again, to classify them as case studies or examples. If the studies utilized only single trivial programs (i.e. less than a couple of hundred lines of code), then we classified them as examples.

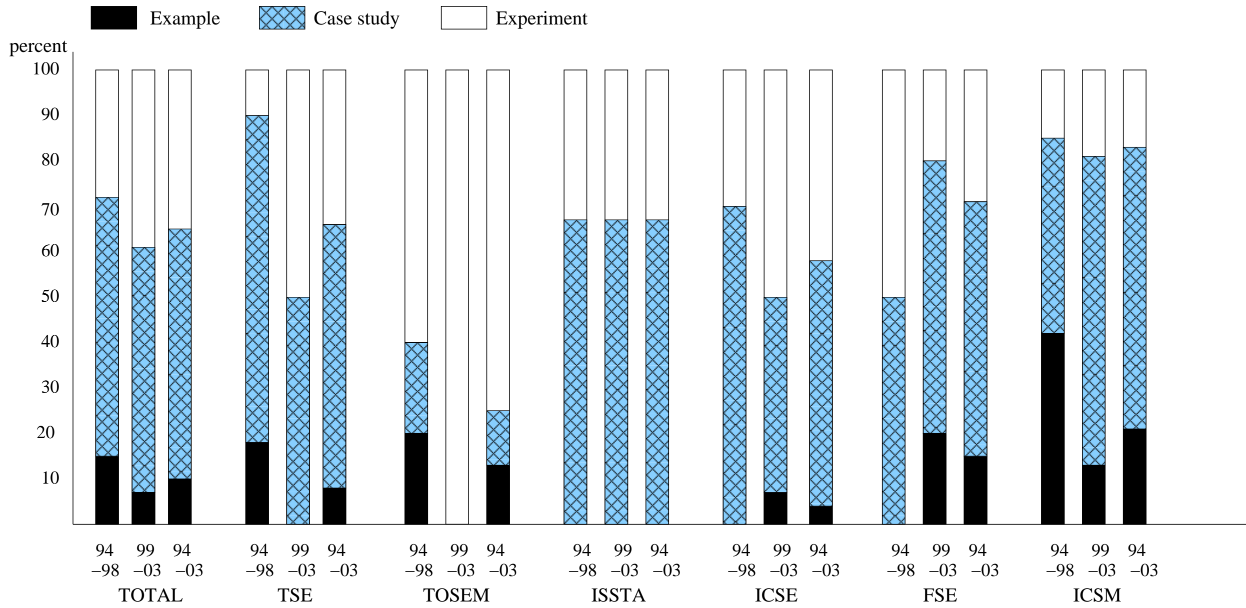

Figure 1: The percentage of empirical studies: example, case study and experiment.

Table 2: Further classification of published empirical studies.

Publication	Empirical Papers	Example	Case Study	Controlled Experiment	Multiple Programs	Multiple Versions	Tests	Faults	Shared Artifacts
TSE (1999-2003)	18	0	9	9	15	6	16	7	5
TSE (1994-1998)	11	1	9	1	6	2	8	2	1
TSE (1994-2003)	29	1	18	10	21	8	24	9	6
TOSEM (1999-2003)	3	0	0	3	3	3	3	3	2
TOSEM (1994-1998)	5	1	1	3	4	2	5	3	1
TOSEM (1994-2003)	8	1	1	6	7	5	8	6	3
ISSTA (1999-2003)	6	0	4	2	5	2	6	1	1
ISSTA (1994-1998)	3	0	2	1	3	1	3	1	0
ISSTA (1994-2003)	9	0	6	3	8	3	9	2	1
ICSE (1999-2003)	14	1	6	7	9	6	14	8	6
ICSE (1994-1998)	10	0	7	3	6	7	10	5	2
ICSE (1994-2003)	24	1	13	10	15	13	24	13	8
FSE (1999-2003)	10	2	6	2	5	2	8	1	0
FSE (1994-1998)	4	0	2	2	3	2	4	2	0
FSE (1994-2003)	14	2	8	4	8	4	12	3	0
ICSM (1999-2003)	16	2	11	3	10	11	10	3	5
ICSM (1994-1998)	7	3	3	1	3	3	2	2	1
ICSM (1994-2003)	23	5	14	4	13	14	12	5	6
Total (1999-2003)	67	5	36	26	47	30	57	23	19
Total (1994-1998)	40	5	24	11	25	17	32	15	5
Total (1994-2003)	107	10	60	37	72	47	89	38	24

Figure 1 summarizes the results of our analysis. The figure reports the data for each venue in terms of three time periods: 1994-1998, 1999-2003, and 1994-2003. Across all venues over the ten year period, 34.5% of the studies presented were controlled experiments and 56% were case studies. Separation of this data into time periods suggests that trends are changing: 27.5% of the studies in the first five years (1994-1998) were controlled experiments, compared to 38.8% in the second five years (1999-2003). This trend occurs across all

venues other than ISSTA and FSE, and it is particularly strong for TSE (9% vs. 50%). The results of this analysis, too, remain stable when papers involving the authors of this article are excluded: in that case, over the ten year period, 23% of the studies were controlled experiments and 65% were case studies; 17.6% of the studies in the first five-year period were controlled experiments compared to 27% in the second five-year period; and the increasing trend is visible across all venues other than ISSTA and FSE, remaining strongest for TSE (0% to 40%) (see Table 6 in Appendix A for full results for this case).

Table 2 reports these same numbers together with additional data listed in the following columns: multiple programs, multiple versions, tests, faults, and shared artifacts. As the table shows, 32.7% of the studies utilize data from only one program (although this is not necessarily problematic for case studies). Also, only 44% of the studies utilize multiple versions and only 35.5% utilize fault data. Finally, the table shows that of the 107 studies, only 22.4% involved artifact sharing. This table exhibits an increasing trend in sharing from 12.5% in the early time period to 28% in the later period. Excluding papers by the authors of this article, the first three percentages become 36% (studies utilizing one program), 37.2% (studies utilizing multiple versions), and 23.2% (studies utilizing fault data), respectively, with 9.3% of the studies involving sharing, and the increasing trend being from 3% to 13.4%, continuing to support our conclusions (see Table 6 in Appendix A).

Further investigation of this data is revealing. Of the 24 papers in which artifacts were shared among researchers, 21 use one or both of a set of programs known as the "Siemens programs", or a somewhat larger program known as "space". (Four of these 21 papers also use one or two other large programs, but these programs have not to date been made available to other researchers as shared artifacts.) The Siemens programs, originally introduced to the research community by Hutchins et al. [22] and subsequently augmented, reorganized, and made available as shareable infrastructure by an author of this article, consist of seven C programs each of no more than 1000 lines of code, 132 seeded faults for those programs, and several sets of test suites satisfying various test adequacy criteria. Space, appearing initially in papers by other researchers [45, 48] and also processed and made available as shareable infrastructure by an author of this article, is a single application of nearly 10,000 lines of code, provided with various test suites and 35 actual faults. In the cases in which multiple "versions" of software systems are used in studies involving these programs, these versions differ only in terms of faults, rather than in terms of a set of changes of which some have caused faults; ignoring these cases, only four exist in which actual, realistic multiple versions of programs are utilized. This is just a starting point for infrastructure sharing; as we describe later in Section 3, the use of the Siemens programs and space poses several threats to validity.

3 Challenges for Experimentation

Researchers attempting to conduct controlled experiments examining the application of testing techniques to artifacts face several challenges. The survey of the literature just summarized provides evidence of the

effects of these challenges: in particular in the small number of controlled experiments, the small percentage of studies utilizing multiple programs, versions, and faults data, and the limited artifact sharing evident. The survey also suggests, however, that researchers are becoming increasingly willing to conduct controlled experiments, and are increasing the extent to which they utilize shared artifacts.

These tendencies are related: utilizing shared artifacts is likely to facilitate controlled experimentation. The Siemens programs and space, in spite of their limitations, have facilitated a number of controlled experiments that might not otherwise have been possible. This argues for the utility of making additional infrastructure available to other researchers, as is our goal.

Before proceeding further, however, it is worthwhile to identify the challenges faced by researchers performing experimentation on testing techniques in the presence of limited infrastructure. Identifying such challenges provides insights into the limited progress in this area that goes beyond the availability of artifacts. Furthermore, identifying these challenges helps us define the infrastructure requirements for such experiments, and helps us shape the design of an experiment infrastructure.

Challenge 1: Supporting replicability across experiments.

A scientific finding is not trusted unless it can be independently replicated. When performing a replication, researchers duplicate the experimental design of an experiment on a different sample to increase the confidence in the findings [47] or on an extended hypothesis to evaluate additional variables [3]. Supporting replicability for controlled experiments requires establishment of control on experimental factors and context; this is increasingly difficult to achieve as the units of analysis and context become more complex. When performing controlled experimentation with software testing techniques, several replicability challenges exist.

First, artifacts utilized by researchers are rarely homogeneous. For example, programs may belong to different domains and have different complexities and sizes, versions may exhibit different rates of evolution, processes employed to create programs and versions may vary, and faults available for the study of fault detection may vary in type and magnitude.

Second, artifacts are provided in widely varying levels of detail. For example, programs freely available through the open source initiative are often missing formal documentation or rigorous test suites. On the other hand, confidentiality agreements often constrain the industry data that can be utilized in published experiments, especially data related to faults and failures.

Third, experiment design and process details are often not standardized or reported in sufficient detail. For example, different types of oracles may be used to evaluate technique effectiveness, different, non-comparable tools may be used to capture coverage data, and when fault seeding is employed it may not be clear who performed the activity and what process they followed.

Challenge 2: Supporting aggregation of findings.

Individual experiments may produce interesting findings, but can claim only limited validity under different contexts. In contrast, a family of experiments following a similar operational framework can enable the aggregation of findings, leading to generalization of results and further theory development.

Opportunities for aggregation are highly correlated with the replicability of an experiment (Challenge 1); that is, a highly replicable experiment is likely to provide detail sufficient to determine whether results across experiments can be aggregated. (This reveals just one instance in which the relationship between challenges is not orthogonal, and in which providing support to address one challenge may impact others.)

Still, even high levels of replicability cannot guarantee correct aggregation of findings unless there is a systematic capture of experimental context [36]. Such systematic capture typically does not occur in the domain of testing experimentation. For example, versions utilized in experiments to evaluate regression testing techniques may represent minor internal versions or major external releases; these two scenarios clearly involve very distinct levels of validation. Although capturing complete context is often infeasible, the challenge is to provide enough support so that the evidence obtained across experiments can be leveraged.

Challenge 3: Reducing the cost of controlled experiments.

Controlled experimentation is expensive, and there are several strategies available for reducing this expense. For example, experiment design and sampling processes can reduce the number of participants required for a study of engineer behavior, thereby reducing data collection costs. Even with such reductions, obtaining and preparing participants for experimentation is costly, and that cost varies with the domain of study, the hypotheses being evaluated, and the applicability of multiple and repeated treatments on the same participants.

Controlled experimentation in which testing techniques are applied to artifacts does not require human participants, it requires objects such as programs, versions, tests, and faults. This is advantageous because artifacts are more likely to be reusable across experiments, and multiple treatments can be validly applied across all artifacts at no cost to validity. Still, artifact reuse is often jeopardized due to several factors.

First, artifact organization is not standardized. For example, different programs may be presented in different directory structures, with different build processes, fault information, and naming conventions.

Second, artifacts are incomplete. For example, open source systems seldom provide comprehensive test suites, and industrial systems are often "sanitized" to remove information on faults and their corrections.

Third, artifacts require manual handling. For example, build processes may require software engineers to configure various files, and test suites may require a tester to control execution and audit results.

Challenge 4: Obtaining sample representativeness.

Sampling is the process of selecting a subset of a population with the intent of making statements about the entire population. The degree of representativeness of the sample is important because it directly impacts the applicability of the conclusions to the rest of the population. However, representativeness needs to be balanced with considerations for the homogeneity of the sampled artifacts in order to facilitate replication as well. Within the software testing domain, we have found two common problems for sample representativeness.

First, sample size is limited. Since preparing an artifact is expensive, experiments often use small numbers of programs, versions, and faults. Further, researchers trying to reduce costs (Challenge 3) do not prepare artifacts for repeated experimentation (e.g., test suite execution is not automated). Lack of preparation for reuse limits the growth of the sample size even when the same researchers perform similar studies.

Second, samples are biased. Even when a large number of programs are collected they usually belong to a set of similar programs. For example, as described in Section 2, many researchers have employed the Siemens programs in controlled experiments with testing. This set of objects includes seven programs with faults, versions, processing scripts, and automated test suites. The Siemens programs, however, each involve fewer than 1000 lines of code. Other sources of sample bias include the types of faults seeded or considered, processes used for test suite creation, and code changes considered.

Challenge 5: Isolating the effects of individual factors.

Understanding causality relationships between factors lies at the core of experimentation. Blocking and manipulating the effects of a factor increases the power of an experiment to explain causality. Within the testing domain, we have identified two major problems for controlling and isolating individual effects.

First, artifacts may not offer the same opportunities for manipulation. For example, programs with multiple faults offer opportunities for analyzing faults individually or in groups, which can affect the performance of testing techniques as it introduces masking effects. Another example involves whether or not automated and partitionable test suites are available; these may offer opportunities for isolating test case size as a factor.

Second, artifacts may make it difficult to decouple factors. For example, it is often not clear what program changes in a given version occurred in response to a fault, an enhancement, or both. Furthermore, it is not clear at what point the fault was introduced in the first place. As a result, the assessment of testing techniques designed to increase the detection of regression faults may be biased.

4 Infrastructure

We have described what we believe are the primary challenges faced by researchers wishing to perform controlled experimentation with testing techniques, and that have limited progress in this area. Some of these challenges involve issues for experiment design, and guidelines such as those provided by Kitchenham

et al. [26] address those issues. Other challenges relate to the process of conducting families of experiments with which to incrementally build knowledge, and lessons such as those presented by Basili et al. [3] could be valuable in addressing these. All of these challenges, however, can be traced at least partly (and some primarily) to issues involving infrastructure.

To address these challenges, we have been designing and constructing infrastructure to support controlled experimentation with software testing and regression testing techniques. Our infrastructure includes, first and foremost, artifacts (programs, versions, test cases, faults, and scripts) that enable researchers to perform controlled experimentation and replications.

We are staging our artifact collection and construction efforts along two dimensions, breadth and depth, where breadth involves adding new object systems to the infrastructure, and depth involves adding new attributes to the systems currently contained in the infrastructure.

Where breadth is concerned, we focused initially on systems prepared in C; however, given the increasing interest on the part of the testing community in object-oriented systems and Java, we have recently shifted our attention to Java systems.

Where depth is concerned, we have been incrementally expanding our infrastructure to accommodate additional attributes. Ultimately, we expect to continually extend these attributes based on the input from other researchers who utilize our infrastructure. Our current set of attributes, however, can be broadly categorized as follows:

- System attributes: source code, and (as available) user documentation. We also store all necessary system build materials, together with information on configurations supported, and compilers and operating systems on which we have built and executed the system.
- Version attributes: system releases created in the course of system enhancement and maintenance. We store versions of source code, versions of documentation, and all associated system attributes for those versions.
- Test attributes: pools of input data, and test suites of various types. We store all harnesses, stubs, drivers, test classes, oracles, and materials necessary to execute test cases.
- Fault attributes: fault data about the system and versions, and information on fault-revealing inputs.
- Execution attributes: operational profiles of the system's execution, expected runtimes for tests or analyses, results of analysis runs, and data on test coverage obtained.

Beyond these artifacts, our infrastructure also includes documentation on the processes used to select, organize, and further set up artifacts, and supporting tools that help with these processes. Together with our plans for sharing and extending the infrastructure, these objects, documents, tools, and processes help address the challenges described in the preceding section as summarized in Table 3.

Table 3: Challenges and Infrastructure.

Challenges	Infrastructure attributes				
	Artifact			Docs,	Share,
	Selection	Organization	Setup	Tools	Extend
Support Replicability	X	X	X	X	X
Support Aggregation		X	X	X	X
Reduce Cost	X	X	X	X	X
Representativeness	X				X
Isolate Effects		X	X		

The following subsections provide further details on each of these aspects of our infrastructure. Further subsections then describe examples of the use of the infrastructure that have occurred to date, and threats to validity to consider with respect to the use of the infrastructure.

4.1 Object Selection, Organization, and Setup

Our infrastructure provides guidelines for object selection, organization, and setup processes. The selection and setup guidelines assist in the construction of a sample of complete artifacts. The organization guidelines provide a consistent context for all artifacts, facilitating the development of generic experiment tools, and reducing the experimentation overhead for researchers.

4.1.1 Object selection

Object selection guidelines direct persons assembling infrastructure in the task of selecting suitable objects, and are provided through a set of on-line instructions that include artifact selection requirements. Thus far, we have specified two levels of required qualities for objects: 1st-tier required-qualities (minimum lines of code required, source freely available, five or more versions available) and 2nd-tier required-qualities (runs on platforms we utilize, can be built from source, allows automation of test input application and output validation). When assembling objects, we first identify objects that meet first-tier requirements, which can be determined relatively easily, and then we prioritize these, and for each, investigate second-tier requirements.

Part of the object selection task involves ensuring that programs and their versions can be built and executed automatically. Because experimentation requires the ability to repeatedly execute and validate large numbers of test cases, automatic execution and validation must be possible for candidate programs. Thus, our infrastructure currently excludes programs that require graphical input/output that cannot easily be automatically executed or validated. At present we also require programs that execute, or through edits can be made to execute, deterministically; this too is a requirement for automated validation, and implies that programs involving concurrency and heavy thread use might not be directly suitable.

Our infrastructure now consists of 17 C and seven Java programs, as shown in Table 4. The first eight

Table 4: Objects in our Infrastructure.

Objects	Size	No. of Versions	No. of Tests	No. of Faults	Release Status
tcas	173	1	1608	41	released
schedule2	374	1	2710	10	released
schedule	412	1	2650	9	released
replace	564	1	5542	32	released
tot_info	565	1	1052	23	released
print_tokens2	570	1	4115	10	released
print_tokens	726	1	4130	7	released
space	9564	1	13585	35	released
gzip	6582	6	217	15	released
sed	11148	5	1293	40	released
flex	15297	6	567	81	released
grep	15633	6	809	75	released
make	27879	5	1043	17	released
bash	48171	10	1168	69	near release
emp-server	64396	10	1985	90	near release
pine	156037	4	288	24	near release
vim	224751	9	975	7	near release
nanoxml	7646 (24)	6	217	33	released
siena	6035 (26)	8	567	3	released
galileo	15200 (79)	16	1537	0	near release
ant	179827 (627)	9	150	21	released
xml-security	41404 (143)	4	14	6	released
jmeter	79516 (389)	6	28	9	released
jtopas	19341 (50)	4	11	5	released

programs listed are the Siemens programs and space, which constituted our first set of experiment objects; the remaining programs include nine larger C programs and seven Java programs, selected via the foregoing process. The other columns are as follows:

- The "Size" column presents the total number of lines of code, including comments, present in each program, and illustrates our attempts to incorporate progressively larger programs. For Java programs, the additional parenthetical number denotes the number of classes in the program.

- The "No. of Versions" column lists how many versions each program has. The Siemens programs and space are available only in single versions (with multiple faults), a serious limitation, although the availability of multiple faults has been leveraged, in experiments, to create various alternative versions containing one or more faults. Our more recently collected objects, however, are available in multiple, sequential releases (corresponding to actual field releases of the systems).

- The "No. of Tests" column lists the number of test cases available for the program (for multi-version programs, this is the number available for the final version). Each program has one or more types of test

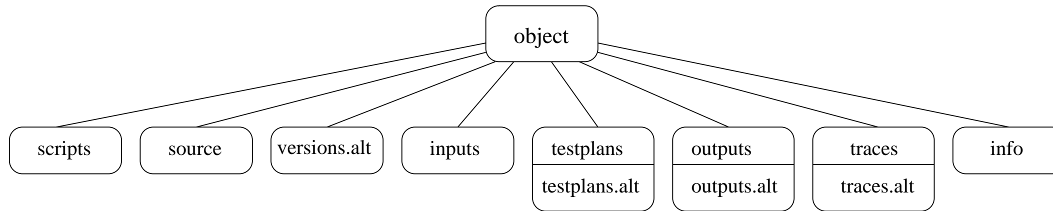

Figure 2: Object directory structure (top level).

cases and one or more types of test suites (described below). Two of the Java programs (nanoxml and siena) are also provided with test drivers that invoke classes under test. The last four Java programs (ant, xml-security, jmeter, and jtopas) have JUnit test suites. The test suites were supplied with each Java program from the open source software hosts.

- The "No. of Faults" column indicates the total number of faults available for each of the programs; for multi-version programs we list the sum of faults available across all versions.

- The "Release Status" column indicates the current release status of each object as one of "released", or "near release". The Siemens programs and space, as detailed above, have been provided to and used by many other researchers, so we categorize them as released. Bash, emp-server, vim, pine, and galileo are undergoing final formatting and testing and thus are listed as "near release". The rest of the programs listed are now available in our infrastructure repository, and we also categorize them as released.

Our object selection process helps provide consistency in the preparation of artifacts, supporting replicability. The same process also reduces costs by discarding earlier the artifacts that are not likely to meet the experimental requirements. Last, the selection mechanism lets us adjust our sampling process to facilitate the collection of a representative set of artifacts.

4.1.2 Object organization

We organize objects and associated artifacts into a directory structure that supports experimentation. Each object we create has its own "object" directory, as shown in Figure 2. An object directory is organized into specific subdirectories (which in turn may contain subdirectories), as follows.

- The scripts directory is the "staging platform" from which experiments are run; it may also contain saved scripts that perform object-related tasks.

- The source directory is a working directory in which, during experiments, the program version being worked with is temporarily placed.

- The versions.alt directory contains various variants of the source for building program versions. Variants are alternative copies of the object, created when the initial copy will not function for some particular purpose, but when a somewhat modified alternative will. For example:

  – The basic variant, which every object has, is contained in a subdirectory of versions.alt called versions.orig, which contains subdirectories v0, v1, . . ., vk, where v0 is the earliest version, and the other vj contain the next sequential versions in turn.

  – A second variant provides faults; for this purpose a directory that may exist is versions.seeded, which also contains subdirectories v1, v2, . . ., vk. In this directory each vk area contains the .c file and .h files (or .java files) needed to build the version with some number of seeded faults that are inserted and a file FaultSeeds.h which contains all declarations needed to define all the faults.

  – A third general class of variants is created to handle cases in which some prototype analysis tool is not robust enough to process a particular syntactic construct; in such cases a variant of a program may be created in which that construct is replaced by a semantically equivalent alternative.

  In the case of Java programs, if a program consists of applications and components, then the versions directory itself is subdivided into multiple subdirectories for these applications and components. Each subdirectory contains variants such as versions.orig and versions.seeded subdirectories.

- The inputs directory contains files containing inputs, or directories of inputs used by various test cases.

- The testplans.alt directory contains subdirectories v0, v1, . . ., vk, each of which contains testing information for a system version; this information typically includes a "universe" file containing a pool of test cases, and various test suites drawn from that pool. We organize the vj subdirectories into four types of files and subdirectories:

  – *General files.* These are .universe, .tsl, and .frame files. A *universe file* (.universe extension) is a file listing test cases. An automated test-script generation program transforms universe files into various types of scripts that can be used to automatically execute the test cases or gather traces for them. TSL and frame files facilitate the use of black-box test cases designed using the category-partition method (described further in Section 4.1.3). TSL files (.tsl extension) are named vk.tsl (k=0,1,2...) for different versions. The sets of test frames generated from .tsl files (.frame extension) are named vk.frame, in reference to their corresponding TSL (vk.tsl) files.

  – *Link files.* These are files that link to previous versions of general files. These links allow the inheriting of testing materials from a prior version, preventing multiple copies.

  – *Testsuites subdirectories.* Some objects have directories containing various test suites built from the universe files. For example, the Siemens programs each have test suites containing randomly

selected test cases from the universe file, test suites that are statement-coverage adequate, and test suites that are both statement-coverage adequate and minimal in size.

  - *The testscripts subdirectory.* If the test cases require startup or exit activities prior to or after execution, scripts encoding these are stored in a testscripts subdirectory.

- The traces.alt directory contains subdirectories v0, v1, ..., vk, each holding trace information for a version of the system, in the form of individual test traces or summaries of coverage information.

- The outputs.alt directory permanently stores the outputs of test runs, which is especially useful when experimenting with regression testing where outputs are compared against previous outputs.

- The testplans, outputs, and traces directories serve as "staging platforms" during specific experiments. Data from a specific "testplans.alt" subdirectory is placed into the testplans directory prior to experimentation; data from outputs and traces directories is placed into subdirectories in their corresponding ".alt" directories following experimentation.

- The info directory contains additional information about the program, especially information gathered by analysis tools and worth saving for experiments, such as fault-matrix information (which describe the faults that various test cases reveal).

- Java objects may contain a testdrivers directory that contains test drivers that invoke the application or its components.

Our object organization supports consistent experimentation conditions and environments, allowing us to write generic tools for experimentation that know where to find things, and that function across all of our objects (or a significant subset, such as those written in Java). This in turn helps reduce the costs of executing and replicating controlled experiments, and aggregating results across experiments. The use of this structure can potentially limit external validity by restricting the types of objects that can be accommodated, and the transformation of objects to fit the infrastructure can create some internal validity threats. However, the continued use of this structure and the generic tools it supports ultimately reduces a large class of potential threats to internal validity arising from errors in automation, by facilitating cross-checks on tools, and leveraging previous tool validation efforts. The structure also accommodates objects with various types and classes of artifacts, such as multiple versions, fault types, and test suites, enabling us to control for and isolate individual effects in conducting experimentation.

4.1.3 Object setup

Test suites

Systems we have selected for our repository have only occasionally arrived equipped with anything more than rudimentary test suites. When suites are provided, we incorporate them into our infrastructure because they

are useful for case studies. For controlled experiments, however, we prefer to have test suites created by uniform processes. Such test suites can also be created in ways that render them partitionable, facilitating studies that isolate factors such as test case size, as mentioned in Section 3 (Challenge 5).

To construct test suites that represent those that might be constructed in practice for particular programs, we have relied primarily on two general processes, following the approach used by Hutchins et al. [22] in their initial construction of the Siemens programs.

The first process involves specification-based testing using the category-partition method, based on a test specification language, TSL, described in [35]. A TSL specification is written for an initial version of an object, based on its documentation, by persons who have become familiar with that documentation and the functionality of the object. Subsequent versions of the object inherit this specification, or most of it, and may need additional test cases to exercise new functionality, which can be encoded in an additional specification added to that version, or in a refined TSL specification. TSL specifications are processed by a tool, provided with our infrastructure, into test frames, which describe the requirements for specific test cases. Each test case is created and encoded in proper places within the object directory.

The second test process we have used involves coverage-based testing, in which we instrument the object program, measure the code coverage achieved by specification-based test cases, and then create test cases that exercise code not covered by them.

Employing these processes using multiple testers helps reduce threats to validity involving specific test cases that are created. Creating larger pools of test cases in this fashion and sampling them to obtain various test suites, such as test suites that achieve branch coverage or test suites of specific sizes, provides further assistance with generalization. We store such suites with the objects along with their pools of test cases.

In addition to two types of test suites just described, for our Java objects, we are also beginning to provide JUnit test suites (and have done so for ant, xml-security, jmeter, and jtopas). JUnit [23] is a Java testing framework that allows automation of tests for classes, and that is increasingly being used with Java systems. As noted in our description of Table 4, the JUnit test suites have been supplied with each Java program from its open source software host.

At present, not all of our objects possess equivalent types of test cases and test suites, but one goal in extending our infrastructure is to ensure that specific types of test cases and test suites are available across all objects on which they are applicable, to aid with the aggregation of findings. A further goal, of course, is to provide multiple instances and types of tests suites per object, a goal that has been achieved for the Siemens programs and space allowing the completion of several comparative studies. Meeting this goal will be further facilitated through sharing of the infrastructure and collaboration with other researchers.

Faults

For studies of fault detection, we have provided processes for two cases: the case in which naturally occurring faults can be identified, and the case in which faults must be seeded. Either possibility presents advantages and disadvantages: naturally occurring faults are costly to locate and typically cannot be found in large numbers, but they represent actual events. Seeded faults can be costly to place, but can be provided in larger numbers, allowing more data to be gathered than otherwise possible, but with less external validity.

To help with the process for hand-seeding faults, and increase the potential external validity of results obtained on these faults, we insert faults by following fault localization guidelines such as the one shown in Figure 3 (excerpted from our infrastructure documentation). These guidelines provide direction on fault placement. We also provide fault classifications based on published fault data (such as the simple one present in the figure), so that faults will correspond, to the extent possible, to faults found in practice. To further reduce the potential for bias, fault seeding is performed independently of experimentation, by multiple persons with sufficient programming experience, and who do not possess knowledge of specific experiment plans.

Another motivation for seeding faults occurs when experimentation concerned with regression testing is the goal. For regression testing, we wish to investigate errors caused by code change (regression faults). With the assistance of a differencing tool, fault seeders locate code changes, and place faults within those.

Although we have not yet incorporated it into our processes, fault seeding can also be performed using program mutation [8, 32]. Program mutation can produce large numbers of faults at low cost, and recent studies [1, 12] indicate that mutation faults can in fact be representative of real faults. If these results generalize, then we can extend the validity of experimental results by using mutation, and the large number of faults that result can yield data sets on which statistically significant conclusions can be obtained, facilitating the understanding of causality. Such studies can then be coupled with studies using other fault types to address external validity concerns.

Similarly, in the future, other researchers might be interested in other types of seeded faults, such as integration faults or faults related to side-effects. By defining appropriate mutation operators or tables of fault types, these researchers can simulate these fault types, allowing studies of their detection. For example, Delamaro et al. [11] have assessed the adequacy of tests for integration testing by mutating interfaces, in ways that simulate involve integration faults. Faults of these types can then also be incorporated into our infrastructure.

All of the objects in our infrastructure that are listed in Table 4 except space contain seeded faults; space contains 35 real faults that were identified during the testing and operational use of the program.

Figure 3: Fault localization guidelines for C programs.

1. When the goal is to seed regression faults in versions, use a differencing tool to determine where the changes occurred. If the changes between versions are large, the tool may carry differences forward, even when the code is identical. Verify the differences independently and use diff only as a guide.
2. The fault seeding process needs to reflect real types of faults. A simple fault classification scheme can be helpful for thinking about different types of faults:
   (a) Faults associated with variables: definition of variable, redefinition of variable, deletion of variable, change value of variable in existing assignment statement.
   (b) Faults associated with control flow: addition of new block of code, deletion of path, removal of block, redefinition of execution condition, change in order of execution, new call to external function, removal of call to external function, addition of function, removal of function.
   (c) Faults associated with memory allocation: allocated memory not freed or not initialized, or erroneous use of pointers.
3. When creating regression faults, assume that the programmer who made the modification inserted the fault, and simulate this behavior by inserting artificial faults into the modified code. There is no restriction on the size or type of fault at this stage. The only requirement is that the changed program can still be compiled and executed.
4. Since more than two programmers perform the seeding process independently, some faults may be repeated. Repeated faults must be removed after all faults have been inserted. Although the programmers could work together to avoid overlapping, this would hurt the validity and credibility of the process. The modified code from multiple programmers needs to be merged. This should be a simple and short (maximum 10 modifications) cut/paste process but needs to be done carefully. Make sure you compile, link and test your program.
5. Next, run the test suite on each version to perform further filtering. Filter out two types of faults:
   (a) faults that are exposed by more than 20% of the tests, because if they are introduced, they are likely to be detected during unit testing.
   (b) faults that are not detected by any tests.
6. Keep only those faults that have not been filtered out. Test the program and then move on to the next version.

4.2 Documentation and Supporting Tools

Documentation and guidelines supplied with our infrastructure provide detailed procedures for object selection and organization, test generation, fault localization, tool usage, and current object descriptions. The following materials are available:

- C (Java) Object Handbook: these documents describes the steps we follow to set up a typical, less than 30K LOC C program (or a typical 5-15K LOC Java program) as an experiment object. They are written primarily as sets of instructions to persons who have or wish to set up these objects, but they are also useful as an aid to understanding the object setup, and choices made in that setup.

- C (Java) Object Download: these web pages provide access to the most recent releases of each of our C (Java) objects, in tarred, gzipped directories.

- C (Java) Object Biographies: these web pages provide information about each of the C (Java) objects we currently make available, including what they are and how they were prepared.

- Tools: this web page describes and provides download capabilities for the tools. Short descriptions for supporting tools are as follows:

  - tsl: this program generates test frames based on a TSL specification.
  - javamts/mts: these programs generate test scripts of various forms and functionalities from universe files.
  - gen-fault-matrix: this module contains various scripts needed to generate a fault matrix file (a file relating faults to the test cases that expose them) for an object.
  - adiff: this tool produces a list of functions changed across two versions of a C object.

- Reporting Problems: this web page provides instructions regarding how to report problems in the infrastructure.

As suggested in Section 3, guidelines such as those we provide support sharing (and thus cost reduction), as well as facilitating replication and aggregation across experiments. Documentation and guidelines are thus as important as objects and associated artifacts.

Depending on the research questions being investigated, experiment designs and processes used for testing experiments can be complex and require multiple executions, so automation is important. Our infrastructure provides a set of testing tools that build scripts that execute tests automatically, gather traces for tests, generate test frames based on TSL specifications, and generate fault matrices for objects. These tools make experiments simpler to execute, and reduce the possibility of human errors such as typing mistakes, supporting replicability as well. The automated testing tools function across all objects, given the uniform

directory structure for objects; thus, we can reuse these tools on new objects as they are completed, reducing the costs of preparing such objects.

4.3 Sharing and Extending the Infrastructure

Our standard object organization and tool support help make our infrastructure extensible; objects that meet our requirements can be assembled using the required formats and tools. This is still an expensive process, but in the long run such extension will help us achieve sample representativeness, and help with problems in replicability and aggregation as discussed in Section 3.

In our initial infrastructure construction, we have focused on gathering objects and artifacts for regression testing study, and on facilitating this with faults, multiple versions and tests. Such materials can also be used, however, for experimentation with testing techniques generally, and with other program analysis techniques. (Section 5 discusses cases in which this is already occurring.) Still, we intend that our infrastructure be extended through addition of objects with other types of associated artifacts, such as may be useful for different types of controlled experiments. For example, one of our Java objects, nanoxml, is provided with UML statechart diagrams, and this would facilitate experimentation with UML-based testing techniques.

Extending our infrastructure can be accomplished in two ways: by our research group, and by collaboration with other research groups. To date we have proceeded primarily through the first approach, but the second has many benefits. First, it is cost effective, mutually leveraging the efforts of others. Second, through this approach we can achieve greater diversity among objects and associated artifacts, which will be important in helping to increase sample size and achieve representativeness. Third, sharing implies more researchers inspecting the artifact setup, tools, and documentation, reducing threats to internal validity. Ultimately, collaboration in constructing and sharing infrastructure can help us contribute to the growth in the ability of researchers to perform controlled experimentation on testing in general.

As mentioned earlier, we have been making our Siemens and space infrastructure available, on request, for several years. We have recently created web pages that provide this infrastructure, together with all more recently created infrastructure described in this article, and all of the programs listed in Table 4 with the exception of those listed as “near release”. This web page resides at http://esquared.unl.edu/sir/, and provides a password-protected portal to any researchers who request access to the infrastructure, with the proviso that they agree to report to us any experiences that will help us to improve the infrastructure.

4.4 Examples of the Infrastructure Being Used

In this section we present three examples of our infrastructure being used by other researchers and ourselves. The first two cases describe examples involving the C programs and their artifacts, and the third case is an example of the use of Java programs and their artifacts.

In presenting each example we address the following questions:

- What problem did the researchers wish to address?
- What types of artifacts were needed to investigate this problem?
- What did the researchers take from our infrastructure?
- What did the researchers learn from their study?

Example 1: Improving test suites via operational abstraction.

Harder et al. [19] present the *operational difference technique* for improving test suites using augmentation, minimization, and generation processes. The authors evaluated improved test suites by comparing them with other techniques in terms of the fault detection ability and code-coverage of the test suites.

To do this, the authors required objects that have test suites and faults. They selected eight such C objects from our infrastructure: the Siemens programs and space. They describe why they selected these programs for their experiment: the programs are well-understood from previous research, and no other programs that have human-generated tests and faults were immediately available.

Through their experiment the authors discovered that their technique produced test suites that were smaller, and slightly more effective at fault detection, than branch coverage suites.

Example 2: Is mutation an appropriate tool for testing experiments?

Andrews et al. [1] investigate whether automatically generated faults (mutants) can be used to assess the fault detection effectiveness of testing techniques.

To answer their research question, they compared the fault detection ability of test suites on hand-seeded, automatically-generated (mutation), and real-world faults.

For their experiment, they required objects that had test suites and faults. Similar to Example 1, they also used eight C programs: the Siemens programs and space. Since the Siemens programs have seeded faults and space contains real faults, the only additional artifacts they needed to obtain were automatically-generated faults (mutants). The authors generated mutants over the C programs using a mutation generation tool. The reason the authors chose the programs was that they considered the associated artifacts to be mature due to their extensive usage in experiments.

The authors compared the adequacy ratio of test suites in terms of mutants and faults. Their analysis suggests that mutants can provide a good indication of the fault detection ability of a test suite; generated mutants were similar to the real faults in terms of fault detection, but different from the hand-seeded faults.

Example 3: Empirical studies of test case prioritization in a JUnit testing environment.

Do et al. [13] investigate the effectiveness of prioritization techniques on Java programs tested using JUnit test cases. They measured the effectiveness of prioritization techniques using the prioritized test suite's rate of fault detection.

To answer their research questions, the authors required Java programs that provided JUnit test suites and faults. The four Java programs (ant, jmeter, xml-security, and jtopas) from our infrastructure have the required JUnit test cases and seeded faults.

Through their experiment, the authors found that test case prioritization can significantly improve the rate of fault detection of JUnit test suites, but also reveals differences with respect to previous studies that can be related to the language and testing paradigm.

4.5 Threats to Validity: Things to Keep in Mind When Using the Infrastructure

As mentioned in Section 4.3, sharing of our infrastructure can bring many benefits to both ourselves and other researchers, but may introduce other problems, such as that users of the infrastructure might misinterpret some artifacts or object organization mechanism. This, in turn, can generate experiments with misleading findings. We believe that our description of the infrastructure organization and its artifacts is detailed enough to limit misuse and misinterpretation, and in practice many researchers are using the infrastructure without major problems.[4] Using the infrastructure also demands users' caution; they must read documentation and follow directions carefully.

Extending the infrastructure by collaborating with other researchers also introduces potential threats to validity (internal and construct). First, to extend the infrastructure, we need to motivate others to contribute. It is not easy to convince people to do this because it requires extra effort to adjust their artifacts to our standard format and some researchers may not be willing to share their artifacts. We expect, however, that our direct collaborators will contribute to the infrastructure in the next phase of its expansion, and they, in turn, will bring more collaborators who can contribute to the infrastructure. (In Section 5, as a topic of future work, we propose possible mechanisms for encouraging researchers who use our infrastructure to contribute additions to it.) Second, if people contribute their artifacts, then we need a way to check the quality of the artifacts contributed. We expect the primary responsibility for quality to lie with contributors, but again by sharing contributed artifacts, we can reduce this problem since researchers will inspect artifacts as they use them.

Another potential problem with our infrastructure involves threats to the external validity of experiments performed using them, since the artifacts we provide have not been chosen by random selection. The infrastructure we are providing is not intended to be a benchmark; rather, we are creating a resource to support experimentation. We hope, however, to increase the external validity of experimentation using our infrastructure over time by providing a larger pool of artifacts through continuous support from our research group and collaboration with other researchers.

[4]There has been one reported problem reported to us regarding the use of the infrastructure to date, but it turned out that in this case, the user attempted to use the artifacts without reading the documentation carefully.

5 Conclusion

We have presented our infrastructure for supporting controlled experimentation with testing techniques, and we have described several of the ways in which it can potentially help address many of the challenges faced by researchers wishing to conduct controlled experiments on testing. We close this article by first providing additional discussion of the impact, both demonstrated and potential, of our infrastructure to date, and then by discussing anticipated future work.

Section 4.4 provided three specific examples of the use of our infrastructure. Here, we remark more generally on the impact of our infrastructure to date. Many of the infrastructure objects described in the previous section are only now being made available to other researchers. The Siemens programs and space, however, in the format extended and organized by ourselves, have been available to other researchers since 1999, and have seen widespread use. In addition to our own papers describing experimentation using these artifacts (over twenty such papers have appeared, see http://www.cse.unl.edu/˜grother) we have identified eight other papers not involving creators of this initial infrastructure that describe controlled experiments involving testing techniques using the Siemens programs and/or space [1, 7, 10, 19, 25, 31, 33, 50]. The artifacts have also been used in [17] for a study of dynamic invariant detection (attesting to the feasibility of using the infrastructure in areas beyond those limited to testing).

In our review of the literature, we have found no similar usage of other artifacts *for controlled experimentation in software testing*. On the one hand, the willingness of other researchers to use the Siemens and space artifacts attests to the potential for infrastructure, once made available, to have an impact on research. On the other hand, this same willingness also illustrates the need for improvements to infrastructure, given that the Siemens and space artifacts present only a small sample of the population of programs, versions, tests, and faults. It seems reasonable, then, to expect our extended infrastructure to be used for experimentation by others, and to help extend the validity of experimental results through widened scope. Indeed, we ourselves have been able to use several of the newer infrastructure objects that are about to be released in controlled experiments described in recent publications [12, 13, 14, 15, 29, 38], as well as in publications currently under review.

In terms of impact, it is also worthwhile to discuss the costs involved in preparing infrastructure; it is these costs that we *save* when we re-use infrastructure. For example, the emp-server and bash objects required between 80 and 300 person-hours per version to prepare; two faculty and five graduate research assistants have been involved in this preparation. The flex, grep, make, sed and gzip programs involved two faculty, three graduate students, and five undergraduate students; these students worked 10-20 hours per week on these programs for between 20 and 30 weeks. These costs are not costs typically affordable by researchers; it is only by amortizing the costs over the potential controlled experiments that can follow that we render the costs acceptable.

Finally, there are several additional potential benefits to be realized through sharing of infrastructure in

terms of challenges addressed; these translate into a reduction of threats to validity that would exist were the infrastructure not shared. By sharing our infrastructure with others, we can expect to receive feedback that will improve it. User feedback will allow us to improve the robustness of our tools and the clarity and completeness of our documentation, enhancing the opportunities for replication of experiments, aggregation of findings, and manipulation of individual factors.

Where future work is concerned, a primary consideration involves implementing mechanisms for further sharing and community development of the infrastructure. At the moment, the web address provided in Section 4.3 provides contact information by which users may obtain a user-id and password, and through which they can access the infrastructure. This small level of access control provides us with a limited method for identifying users of the infrastructure, so that we can inform them of modifications or enhancements, and contact them with questions about its use.

As future work, we are considering various additional mechanisms for encouraging researchers who use our infrastructure to contribute additions to it in the form of new fault data, new test suites, and variants of programs and versions that function on other operational platforms.

One possible approach is to use our web portal as the core mechanism for distributing, obtaining feedback on, and collecting new artifacts and documentation. In addition to providing a registration site for users, the web portal can let researchers provide comments on and evaluations of the downloaded artifacts. This information, together with usage statistics (e.g., number of downloads, number of experiments using artifacts), will help researchers determine what their peers are employing for experimentation, and the individual strengths and weakness of each artifact. We envision that each artifact could have an evaluation rating, a popularity index, and a set of compiled comments. This data could also help us guide further refinements of the available artifacts, and identify artifacts of interest that may be missing.

As a mechanism for longer-term support, a more refined management model involves forming a repository steering committee, that includes representation from the user community, whose role is to set policies that govern the operation, maintenance and expansion of the repository. Such a committee could have several permanent members to ensure appropriate institutional memory, and also a number of rotating members. While the steering committee's role would be purely policy oriented, the work to be performed in order to maintain, update and expand the repository would be organized in the spirit of the "open source" model.

Ultimately, through mechanisms such as this, we hope that the community of researchers will be able to assemble additional artifacts using the formats and tools prescribed, and contribute them to the infrastructure; this in turn will increase the range and representativeness of artifacts available to support experimentation. Furthermore, we hope through this effort to aid the entire testing research community in pursuing controlled experimentation with testing techniques, increasing our understanding of these techniques and the factors that affect them in ways that can be achieved only through such experimentation.

Acknowledgements

We thank the current and former members of the Galileo and MapsText research groups for their contributions to our experiment infrastructure. These and other contributors are acknowledged individually on the web pages through which our infrastructure is made available. We also thank the Siemens researchers, and especially Tom Ostrand, for providing the Siemens programs and much of the initial impetus for this work, and Phyllis Frankl, Filip Vokolos, and Alberto Pasquini for providing most of the materials used to assemble space infrastructure. Finally, this work has been supported by NSF awards CCR-0080898 and CCR-0347518 to the University of Nebraska - Lincoln, and by NSF awards CCR-9703108, CCR-9707792, CCR-0080900, and CCR-0306023 to Oregon State University.

References

[1] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Int'l. Conf. Softw. Eng.*, May 2005.

[2] V. Basili, R. Selby, E. Heinz, and D. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743, July 1986.

[3] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[4] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[5] J. Bible, G. Rothermel, and D. Rosenblum. Coarse- and fine-grained safe regression test selection. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, April 2001.

[6] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.

[7] D. Binkley, R. Capellini, L. Raszewski, and C. Smith. An implementation of and experiment with semantic differencing. In *Int'l. Conf. Softw. Maint.*, November 2001.

[8] T. Budd and A. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985.

[9] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Int'l. Conf. Softw. Eng.*, pages 211–220, May 1994.

[10] A. Coen-Porisini, G. Denaro, C. Ghezzi, and P. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of ACM Foundations of Software Engineering*, 2001.

[11] M. Delamaro, J. Maldonado, and A. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3), March 2001.

[12] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Int'l. Conf. Softw. Maint.*, September 2005.

[13] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Int'l. Conf. Softw. Rel. Eng.*, November 2004.

[14] S. Elbaum, D. Gable, and G. Rothermel. The Impact of Software Evolution on Code Coverage. In *Int'l. Conf. Softw. Maint.*, pages 169–179, November 2001.

[15] S. Elbaum, P. Kallakuri, A. Malishevsky, R. Rothermel, and S. Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Journal of Software Testing, Verification and Reliability*, 12(2), 2003.

[16] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Int'l. Conf. Softw. Eng.*, pages 329–338, May 2001.

[17] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Int'l. Conf. Softw. Eng.*, June 2000.

[18] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[19] M. Harder and M. Ernst. Improving test suites via operational abstraction. In *Int'l. Conf. Softw. Eng.*, May 2003.

[20] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.

[21] D. Hoffman and C. Brealey. Module test case generation. In *3rd Workshop on Softw. Testing, Analysis, and Verif.*, pages 97–102, December 1989.

[22] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Int'l. Conf. Softw. Eng.*, pages 191–200, May 1994.

[23] http://www.junit.org.

[24] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering: An International Journal*, 9(1), March 2004.

[25] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Int'l. Conf. Softw. Eng.*, May 2002.

[26] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, August 2002.

[27] B. Kitchenham, L. Pickard, and S. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, pages 52–62, July 1995.

[28] H.K.N. Leung and L. White. Insights into regression testing. In *Int'l. Conf. Softw. Maint.*, pages 60–69, October 1989.

[29] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Int'l. Conf. Softw. Maint.*, pages 230–240, October 2002.

[30] B. Marick. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, September 1999.

[31] M. Marre and A. Bertolino. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering*, 29(11), November 2003.

[32] A. Offutt, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2), April 1996.

[33] V. Okun, P. Black, and Y Yesha. Testing with model checkers: insuring fault visibility. *WSEAS Transactions on Systems*, 2(1):77–82, January 2003.

[34] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. ACM*, 41(5):81–86, May 1988.

[35] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.

[36] L. Pickard and B. Kitchenham. Combining empirical results in software engineering. *Inf. Softw. Tech.*, 40(14):811–821, August 1998.

[37] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[38] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *Int'l. Conf. Softw. Eng.*, May 2002.

[39] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Int'l. Symp. Softw. Testing Anal.*, August 1994.

[40] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[41] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[42] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[43] W. Tichy, P. Lukowicz, E. Heinz, and L. Prechelt. Experimental evaluation in computer science: a quantitative study. *Journal of Systems and Software*, 28(1):9–18, January 1995.

[44] W. Trochim. *The Research Methods Knowledge Base*. Atomic Dog, Cincinnati, OH, 2nd edition, 2000.

[45] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Int'l. Conf. Softw. Maint.*, pages 44–53, November 1998.

[46] E. Weyuker. The evaluation of program-based software test data adequacy criteria. *Comm. ACM*, 31(6), June 1988.

[47] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

[48] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. In *Proceedings of the Computer Software Applications Conference*, pages 522–528, August 1997.

[49] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Eighth Intl. Symp. Softw. Rel. Engr.*, pages 230–238, November 1997.

[50] T. Xie and D. Notkin. Macro and micro perspectives on strategic software quality assurance in resource constrained environments. In *Proceedings of EDSER-4*, May 2002.

[51] R. K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods, Vol. 5)*. Sage Publications, London, UK, 1994.

[52] M. Zelkowitz and D. Wallace. Experimental models for validating technology. *IEEE Computer*, pages 23–31, May 1998.

Appendix A

Table 5: Research articles involving testing and empirical studies in six major venues, 1994-2003 (articles by authors of this paper excluded).

Year	TSE			TOSEM			ISSTA			ICSE			FSE			ICSM		
	P	T	EST	P	T	EST	P	T	EST	P	T	EST	P	T	EST	P	T	EST
2003	74	8	7	7	0	0	-	-	-	74	6	2	33	4	4	39	4	3
2002	73	7	3	14	2	0	23	11	4	55	2	2	17	0	0	60	8	7
2001	53	4	3	8	1	0	-	-	-	60	4	2	28	3	3	67	3	2
2000	62	5	2	14	0	0	20	9	1	66	4	1	17	2	2	24	0	0
1999	46	1	0	12	1	0	-	-	-	58	4	2	29	1	1	35	3	1
1998	72	3	2	12	1	1	16	9	1	37	0	0	22	2	1	31	2	2
1997	50	5	4	11	0	0	-	-	-	52	4	2	27	3	1	34	2	1
1996	59	8	2	12	4	1	29	13	1	53	5	3	17	2	1	34	1	1
1995	70	4	1	10	0	0	-	-	-	31	3	1	15	1	1	30	4	1
1994	68	7	1	12	3	1	17	10	1	30	5	2	17	2	0	38	3	1
Total	627	52	25	112	12	3	105	52	8	516	37	17	222	20	14	396	34	23

Table 6: Further classification of published empirical studies (articles by authors of this paper excluded).

Publication	Empirical Papers	Example	Case Study	Controlled Experiment	Multiple Programs	Multiple Versions	Tests	Faults	Shared Artifacts
TSE (1999-2003)	15	0	9	6	12	5	13	4	2
TSE (1994-1998)	10	1	9	0	5	1	7	1	0
TSE (1994-2003)	25	1	18	6	17	6	20	5	2
TOSEM (1999-2003)	0	0	0	0	0	0	0	0	0
TOSEM (1994-1998)	3	1	1	1	2	1	3	2	0
TOSEM (1994-2003)	3	1	1	1	2	1	3	2	0
ISSTA (1999-2003)	5	0	4	1	4	1	5	0	0
ISSTA (1994-1998)	3	0	2	1	3	1	3	1	0
ISSTA (1994-2003)	8	0	6	2	7	2	8	1	0
ICSE (1999-2003)	9	1	4	4	6	3	9	4	3
ICSE (1994-1998)	8	0	6	2	4	6	8	3	1
ICSE (1994-2003)	17	1	10	6	10	9	17	7	4
FSE (1999-2003)	10	2	6	2	5	2	8	1	0
FSE (1994-1998)	4	0	2	2	3	2	4	2	0
FSE (1994-2003)	14	2	8	4	8	4	12	3	0
ICSM (1999-2003)	13	2	10	1	9	8	7	1	2
ICSM (1994-1998)	6	3	3	0	2	2	1	1	0
ICSM (1994-2003)	19	5	13	1	11	10	8	2	2
Total (1999-2003)	52	5	33	14	36	19	42	10	7
Total (1994-1998)	34	5	23	6	19	13	26	10	1
Total (1994-2003)	86	10	56	20	55	32	68	20	8