ACM Transactions on Software Engineering and Methodology (to appear).

A Comparative Study of Coarse- and Fine-Grained Safe Regression Test Selection Techniques

John Bible
Department of Computer Science
Oregon State University
Corvallis, OR 97331
bible@cs.orst.edu

Gregg Rothermel
Department of Computer Science
Oregon State University
Corvallis, OR 97331
grother@cs.orst.edu

David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
dsr@ics.uci.edu

January 6, 2001

Abstract

Regression test selection techniques reduce the cost of regression testing by selecting a subset of an existing test suite to use in retesting a modified program. Over the past two decades, numerous regression test selection techniques have been described in the literature. Initial empirical studies of some of these techniques have suggested that they can indeed benefit testers, but so far, few studies have empirically compared different techniques. In this paper, we present the results of a comparative empirical study of two safe regression test selection techniques. The techniques we studied have been implemented as the tools `DejaVu` and `TestTube`; we compared these tools in terms of a cost model incorporating *precision* (ability to eliminate unnecessary test cases), *analysis cost* and *test execution cost*. Our results indicate that in many instances, despite its relative lack of precision, `TestTube` can reduce the time required for regression testing as much as the more precise `DejaVu`. In other instances, particularly where the time required to execute test cases is long, `DejaVu`'s superior precision gives it a clear advantage over `TestTube`. Such variations in relative performance can complicate a tester's choice of which tool to use. Our experimental results suggest that a hybrid regression test selection tool that combines features of `TestTube` and `DejaVu` may be an answer to these complications; we present an initial case study that demonstrates the potential benefit of such a tool.

Keywords: comparative study, empirical study, regression testing, safe regression test selection, software testing, `TestTube`, `DejaVu`

1 Introduction

Regression testing is an expensive testing process used to validate modified software and detect whether new faults have been introduced into previously tested code. A typical approach to regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. One regression testing strategy reruns all such test cases, but this *retest-all* technique may consume excessive time and resources. *Regression test selection* (RTS) techniques (e.g. [1, 3, 4, 5, 6, 8, 11, 14, 22, 24, 26]), in contrast, attempt to reduce the cost of regression testing by selecting a subset of the test suite that was used during development and using that subset to test the modified program.

Most RTS techniques are not designed to be *safe*. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In this paper, we are interested in safe RTS techniques. If an explicit set of safety conditions can be satisfied, safe RTS techniques guarantee that the subset selected contains all test cases in the original test suite that can reveal faults in the modified program.

In this paper, we present the results of a comparative empirical study of two safe RTS techniques. The two techniques we studied have been implemented as the tools `DejaVu` [22] and `TestTube` [6]. These two tools exemplify the classic tradeoff of precision versus efficiency that implementers of safe RTS techniques must consider. `TestTube` uses a coarse level of granularity in examining code, viewing functions, global variables, macros and type definitions as its most primitive elements, whereas `DejaVu` focuses on statements. `TestTube` does not consider the temporal ordering of code components, whereas `DejaVu` relies on control flow. `TestTube` derives change information from an entity database, whereas `DejaVu` uses textual equivalence to draw conclusions about semantic equivalence. Given these differences, these two tools together incorporate many of the fundamental ideas existent in other safe RTS techniques, such as `Pythia` [24], Laski and Szermer's safe RTS algorithm [14], and Ball's family of safe RTS algorithms [3].

To examine the tradeoffs between `TestTube` and `DejaVu`, we empirically compared them using three subject groups of programs having many versions and test cases; the two smaller subject groups also possess a large number of test suites. The three subject groups consist of the Siemens programs [12] (100-516 lines of code), the `space` program (6,218 lines of code) and the `player` component of the Internet-based game `Empire` (49,316 lines of code). On the Siemens programs, `DejaVu` often selected subsets of the original test suite that were substantially smaller than the full test suite, as did `TestTube`. Both tools, however, required excessive analysis time, and proved less efficient than simply re-executing all test cases. On `space`, `DejaVu` once again selected fairly small subsets of the original test suite, but proved inefficient given the short running times of the test cases. `TestTube` nearly equaled the precision of `DejaVu` on `space` for substantially less cost; again, though, short running times for test cases made the retest-all technique slightly more efficient. On `player`, however, both tools produced substantial gains in efficiency. `DejaVu` always significantly decreased the cost of retesting the software. Similarly, `TestTube` often reduced the cost of retesting the software to the same degree as `DejaVu`, although sometimes it saved no time.

These are the main results of our comparative study, and they further suggest that a hybrid approach, composed of a pass by `TestTube` followed by a pass by `DejaVu`, might provide test selection costs closer to the lesser costs of the former, while maintaining the higher precision of the latter. More precisely, such a hybrid approach first employs a `TestTube`-like technique on the test suite and program information and then employs a `DejaVu`-like analysis on a truncated set of control-flow graphs to further reduce the subset of test cases to re-execute. We present results of an initial case study in which we applied an implementation of this approach to the `space` program. The results demonstrate the potential benefit of the approach.

In the next section, we describe the safe RTS techniques underlying `DejaVu` and `TestTube`; we also analyze the expected trade-offs in choosing one technique over the other. In Section 3, we describe our empirical studies comparing `DejaVu` and `TestTube`, and their results. In Section 4, we present the details of the hybrid safe RTS technique, and the results of our case study using it. Section 5 presents our conclusions.

2 Background and Related Work

In this section, we discuss safe regression test selection, describe the specific RTS techniques and tools on which our studies focus, present the cost model that we use to evaluate results, and review related work.

In the rest of this paper, let P be a program, let P' be a modified version of P, and let S and S' be the specifications for P and P', respectively. Let T be a test suite developed initially for P.

2.1 Safe Regression Test Selection

In general, regression testing involves several subproblems, including the problems of selecting a subset of the original test suite to run, generating new test cases to exercise new code or functionality, executing old and new test cases, and constructing a new test suite for use on future versions. While each of these problems is important, in this work we focus on the regression test selection problem: the problem of selecting a subset of an existing test suite T to execute on P'.

Regression test selection (RTS) techniques (e.g. [1, 3, 4, 5, 6, 8, 11, 14, 22, 24, 26]) address this problem. In this paper we are primarily interested in *safe* RTS techniques (e.g. [3, 6, 14, 22, 24]), which by definition eliminate only test cases that are provably not able to reveal faults. This notion of safety is defined formally and in detail in [20, 21]; we summarize relevant points of that discussion here.

A test case t detects a fault in P' if it causes P' to fail: in that case we say t is *fault-revealing for* P'. A program P fails for t if, when P is tested with t, P produces an output that is incorrect according to S. There is no effective procedure for finding the test cases in T that are fault-revealing for P' [20]. Under certain conditions, however, an RTS technique can select a superset of the set of test cases in T that are fault-revealing for P'. Under those conditions, such a technique omits no test cases in T that can reveal faults in P'. It is these conditions, and these techniques, that interest us.

Consider a second subset of the test cases in T: the modification-revealing test cases. A test case t is *modification-revealing for* P *and* P' if and only if it causes the outputs of P and P' to differ. Given two assumptions, the modification-revealing test cases in T form a set equivalent to the set of fault-revealing test cases in T:

Assumption 1: For each test case $t \in T$, when P was tested with t, P halted and produced correct output.

Assumption 2: For each test case $t \in T$, t is not obsolete for P'. Test case t is *obsolete* for P' if and only if t specifies an input to P' that, according to S', is invalid for P', or t specifies an invalid input-output relation for P' (i.e., the expected output of t has changed for S').

Where regression testing is concerned, from a practical standpoint these assumptions are not unreasonable. With respect to Assumption 1, in practice, test suites often retain test cases that are known to expose faults; however, such test cases can be excluded from T initially and then re-included in a selected subset of T after selection has been performed. With respect to Assumption 2, if we cannot effectively determine test obsolescence, we cannot effectively judge test correctness. Thus, this assumption is necessary in order to reuse test cases in any manner. Assumption 2 does imply, however, that safe RTS algorithms can actually be applied only to the subset of T that consists of non-obsolete test cases. When specification changes have occurred, test cases that have become obsolete due to those changes (i.e., test cases for which the input and/or output expectations have changed) are not candidates for selection.

In any case, when either of the two assumptions does not hold with respect to T, if we can identify and remove from T those test cases for which the assumptions are violated, we can apply an RTS technique to the remaining test cases. For simplicity, we shall henceforth assume that such removal has (if necessary) occurred, and that T contains no test cases that violate the assumptions.

In this case, we can find the test cases in T that are fault-revealing for P' by finding the test cases in T that are modification-revealing for P and P'; we call a technique that selects all modification-revealing

test cases *safe*. Unfortunately, even when assumptions 1 and 2 hold, there is no effective procedure for precisely identifying the set of test cases in T that are modification-revealing for P and P' [20]. Thus, safe RTS techniques attempt to conservatively approximate this set. A typical approach is to select (or in practice, conservatively approximate) a third subset of T: the modification-traversing test cases. Informally, a test case $t \in T$ is *modification-traversing for P and P'* if and only if it (a) executes new or modified code in P', or (b) formerly executed code that has since been deleted from P. To capture this notion more formally, Reference [20] defines the concept of an *execution trace* $ET(P(t))$ for t on P to consist of the sequence of statements in P that are executed when P is tested with t. Two execution traces $ET(P(t))$ and $ET(P'(t))$, representing the sequences of statements executed when P and P', respectively, are tested with t, are nonequivalent if they have different lengths, or if when their elements are compared from first to last, some pair of lexically nonidentical elements is found. Test case t is modification-traversing for P and P' if and only if $ET(P(t))$ and $ET(P'(t))$ are not equivalent.

The modification-traversing test cases are useful to consider because with a third assumption (in addition to Assumptions 1 and 2), a test case $t \in T$ can only be modification-revealing for P and P' if it is modification-traversing for P and P':

Assumption 3 (*Controlled Regression Testing*): When P' is tested with t, all factors that might influence the output of P', except for the code in P', are held constant with respect to their states when P was tested with t.

If factors other than the program (such as the operating environment, the nondeterministic ordering of statements in concurrent programs, or databases and files that contribute data) can affect test execution but cannot be controlled, then the set of fault revealing test cases might vary between executions. Safe techniques require the environment to be controlled in order to prevent this phenomenon. Note that this problem is hardly unique to regression testing—any testing methodology relying on systematic code execution demands the same level of control. Moreover, in the absence of controlled regression testing, even a retest-all technique may "miss" important faults that it is expressly designed to be able to expose if, for example, lack of control causes its test cases to behave in unintended ways and no longer exercise the code or features they are meant to exercise. Further ramifications of this assumption are discussed in [20, 21].

The techniques we examine in this paper — techniques implemented as `DejaVu` and `TestTube` — are both safe and depend for their safety on the set of assumptions just described. Both techniques attempt to identify, though at different levels, the test cases in T that are modification-traversing for P and P'.

Reference [21] presents a framework, that we employ in this work, for use in comparative analyses of safe RTS techniques. The framework consists of four criteria: inclusiveness, precision, efficiency, and generality. *Inclusiveness* measures the extent to which an RTS technique selects test cases from T that are modification-revealing for P and P'—for the safe techniques that we consider, 100% inclusiveness is required. *Precision* measures the extent to which a technique omits test cases in T that are not modification-revealing for P and P' and are thus unnecessary. *Efficiency* measures the time and space requirements of a technique. *Generality* measures the ability of a technique to function in a practical and sufficiently wide range of situations: in particular, a general safe technique does not require strong or difficult-to-satisfy preconditions for safety.

2.2 A Cost Model for Comparing Safe RTS techniques

To quantify the comparative efficiency of two safe RTS techniques, we adapt a cost model presented by Leung and White [16]. For a safe RTS technique M to be cost-effective, the cost A_M of applying M and the cost of executing and validating the selected test cases T_M on P' must be less than the cost of executing and validating the original test suite on P':

$$A_M(T, P, P') + E(T_M, P') + V(T_M, P') < E(T, P') + V(T, P')$$

The left side of this inequality represents the cost incurred if we use a safe RTS technique. $A_M(T, P, P')$ is the cost of analysis required for safe regression test selection. The E and V terms represent the costs of executing and validating the test suites, respectively.

For a comparative analysis of two techniques, we consider whether the left side of the above inequality for one technique M_1 is less than the left side of the inequality for the second technique M_2:

$$A_{M_1}(T, P, P') + E(T_{M_1}, P') + V(T_{M_1}, P') < A_{M_2}(T, P, P') + E(T_{M_2}, P') + V(T_{M_2}, P')$$

where A_{M_1} and A_{M_2} denote the analysis costs for the two techniques, T_{M_1} is the set of test cases selected by the first technique, and T_{M_2} is the set of test cases selected by the second technique.

Note that, given the terms in the foregoing inequality, the outcome of a comparative analysis is affected by both the percentages of the test suite selected by the techniques (precision), and the analysis times required by the techniques (efficiency). Neither of these factors is simply a function of the other: a technique can be both highly inefficient and highly precise, or highly efficient and highly imprecise.

In using the above cost model, we distinguish between costs incurred in the *preliminary phase* of regression testing and those incurred in the *critical phase*. Preliminary phase costs are incurred incrementally as the project develops. For instance, source code control and configuration systems can incrementally maintain information used later by the analysis portion of a safe RTS technique. In our analyses, we will almost always consider preliminary costs to make no contribution toward the total cost of the analysis technique. We will, however, point out situations in which preliminary phase costs might be an issue in choosing a safe RTS technique.

Critical-phase costs, on the other hand, occur in the time period designated specifically for regression testing. At this point, product development is essentially complete. The analysis of critical-phase costs can be complicated, and in general may involve a variety of costs such as computing time, human effort, and investment in equipment and personnel. In our experiments, we use the cost of computing time as the sole measure of the cost of the critical phase. This simplification is appropriate for the particular subject programs and test suites we studied and hence does not affect the validity of our results; in general, however, other factors such as human costs might also be important.

2.3 Two Safe RTS Techniques

2.3.1 DejaVu

Rothermel and Harrold developed a family of safe RTS techniques, one of which is implemented as the tool `DejaVu`. We provide an overview of `DejaVu` here; Reference [22] presents the detailed algorithm, with cost analyses and examples of its use.

`DejaVu` constructs control-flow graph (CFG) representations of the procedures in two program versions P and P', in which individual nodes are labeled by their corresponding statements. `DejaVu` assumes that a *test history* is available that records, for each test case t in T and each edge e in the CFG for P, whether t traversed e. This test history is gathered by instrumentation code inserted into the source code of the system under test. Construction and storage of the CFGs for the base program P and the test history are preliminary phase costs.[1] Construction of the CFGs for P', however, is a critical-phase cost.

`DejaVu` performs a simultaneous depth-first graph walk on a pair of CFGs G and G' for each procedure and its modified version in P and P', keeping pointers $\uparrow N$ and $\uparrow N'$ to the current node in each graph. The tool begins with the entry nodes of G and G'; it then executes a recursive depth-first search on both CFGs. Upon visiting a pair of nodes N and N' in G and G', the tool examines each edge e leaving N to determine whether there is an equivalently labeled edge in G'. If not, then the tool places e into a set of *dangerous edges*. If there is a corresponding edge in G', and that edge enters an already traversed portion of the graph at the same node in both CFGs, then the current recursion is terminated. Otherwise, $\uparrow N$ and $\uparrow N'$ are moved forward to point to a new pair of nodes. The tool then checks to see if the labels on the nodes pointed to by $\uparrow N$ and $\uparrow N'$ are lexically equivalent (textually equivalent after non-semantically meaningful characters such as white space and comments have been removed). If they are not, the tool places the edge it just followed into the set of dangerous edges and returns to the source of that edge, ending that trail of recursion.

After `DejaVu` has determined all the dangerous edges that it can reach by crossing non-dangerous edges, it terminates. At this point, any test case $t \in T$ is selected for retesting P' if the execution trace for t contains a dangerous edge.[2]

`DejaVu` guarantees safety as long as equivalent execution traces for identical inputs imply equivalent behaviors. As long as the three assumptions discussed in Section 2.1 hold, `DejaVu` selects a superset of the fault-revealing test cases. However, when the assumptions do not hold, as with programs in environments that cannot be controlled, `DejaVu` can provide only a guideline for safety. For instance, in a system in which the state of the operating system can nondeterministically affect program behavior, and in which the state of the operating system cannot be controlled or simulated, `DejaVu` might omit a potentially fault-revealing test case.

2.3.2 TestTube

Chen, Rosenblum and Vo developed the safe RTS technique implemented as `TestTube` specifically to address considerations of cost-effectiveness. We provide an overview of `TestTube` here; for complete details, see [6].

`TestTube` uses a general technique of regression test selection based on coarse-grained analysis of the coverage relationship between test cases and the system under test; it has been implemented for the C programming language. It was originally developed using the CIA database program, the APP instrumenter, and the AST libraries [13].[3] In the current implementation of `TestTube` for C, each test case t is associated with a subset of program code involved in the execution of that test case: this subset is composed of units

[1] `DejaVu` relies on the `Aristotle` analysis system [9] for CFG construction, code instrumentation, and test history construction.

[2] The version of the `DejaVu` algorithm presented in [22] selected test cases during its graph walk, each time it located a dangerous edge. The version of the algorithm that we discuss here includes an optimization suggested by Ball [3] that improves on the worst-case runtime of the original version by postponing test case selection until completion of the walk.

[3] CIA, APP, and the AST libraries were all developed by AT&T Bell Laboratories.

at the granularity of function definitions, global variables, types, and preprocessor definitions. To gather this coverage information, P is instrumented to generate traces of all functions called during its execution. Execution of a test case t on the instrumented system thereby generates a trace of the functions executed by t. The remaining coverage information is generated from a source code database created from P. The database records all of the (static) references between the entities in the source code of P. For each function associated with t, the transitive closure of the function's references to global variables, types, and macros is computed and associated with t, thereby conservatively identifying all global namespace entities that t potentially covers.

When a new version P' is created, a source code database is created for P'. The databases for P and P' are compared to identify the entities that were changed to create P'; this comparison is based on checksums that are computed over the definition of each entity. `TestTube` then compares the list of changed entities with the coverage information preserved for each test case. Finally, `TestTube` selects all test cases that cover changed entities for retesting P'.

Coverage information must be generated for all test cases in the first version of the system under test, but as changes are made to create new versions, coverage information is re-computed only for test cases that are selected to test the new versions. Coverage information can be re-computed for the test cases run on a particular version during the preliminary phase of the next version. However, creation of the source code database and selection of test cases for the next version are critical-phase activities.

The safety of `TestTube` follows from the fact that, because a change to an entity triggers selection of all test cases (potentially) covering that entity, `TestTube` selects all test cases associated with changed entities. Of course, this safety, like `DejaVu`'s, depends on Assumptions 1-3 (Section 2.1); thus, for example, for nondeterministic systems in which multiple executions of the same test case may produce different execution paths, `TestTube` is not safe.

It is easy to show that when Assumptions 1-3 hold, the set of test cases selected by `TestTube` is a superset of the set of test cases selected by `DejaVu`. Although we do not provide a formal proof, informally the argument is as follows. If `DejaVu` selects a test case t, it is due to a difference within a pair of functions f and f', or in a data item used in f or f'. In either case `TestTube` will select all test cases that executed f because f is different from f'; this set necessarily includes t, but may include other test cases that did not execute the code identified as changed by `DejaVu`. One goal of this work is to empirically investigate the effect that this analytically demonstrable difference may have in practice.

2.4 Related Work

Selective retest techniques have been evaluated and compared analytically in [21], but only recently have attempts been made to evaluate or compare them empirically [5, 6, 7, 10, 18, 19, 22, 23, 27].

Of these studies, those reported in [5, 6, 22, 23, 27] focus on single techniques. Reference [27] reports a study of the time and space requirements for the subsystem of a tool implementing the "Firewall" RTS technique [15] that implements the underlying database required by that technique; the study does not, however, examine the Firewall technique itself. Reference [5] studies the application of a slicing-based RTS technique to five programs, measuring test suite size reduction and analysis tool run-time. Reference [6] examines the results of applying `TestTube` to two systems, measuring the amount of test suite reduction that was achieved. References [22] and [23] describe studies in which `DejaVu` was applied to several programs,

measures of its efficiency and precision were collected, and its cost-effectiveness is evaluated taking into account the costs of test execution and validation. With the exception of [27], which does not directly examine an RTS technique, these studies all illustrate, to some extent, potential for cost-effectiveness. None of the studies, however, compares different RTS techniques, and only [22] and [23] consider test execution time in any comparisons of overall costs and savings to those achieved by the retest-all technique. In contrast to these other studies, the work described in this paper focuses on safe RTS techniques, on comparative analyses, and on collection of test execution costs, facilitating a more complete view of cost-benefits tradeoffs.

References [10] and [19] also present empirical studies of RTS techniques, considering `TestTube` in the latter case, and `TestTube` and `DejaVu` in the former. This work focuses, however, on the problem of predicting RTS technique effectiveness, empirically investigating the effectiveness of coverage-based effectiveness predictors. The studies measure test suite size reduction, but not analysis or test execution costs.

In a recent study [7], three varieties of techniques, including one safe technique, one minimization technique, and one other non-safe technique, were empirically compared to random test selection for relative precision and inclusiveness. Our studies differ from this one in that they focus on comparisons of safe techniques; in addition, our studies directly examine efficiency tradeoffs.

In [18], we reported results of a preliminary study of `DejaVu` and `TestTube`, in which we compared the two tools for relative precision on two subject groups of programs. That study, however, did not measure the relative costs of the tools, either in terms of analysis or test-execution costs. Consideration of these costs is necessary for a complete cost-benefits analysis. The work reported here represents results of a complete study, applying actual implementations of `DejaVu` and `TestTube` to three subject groups of programs, measuring both precision and efficiency, and considering both analysis and test execution/costs in the evaluation of efficiency. This work also builds on the results of the studies to provide initial data on a hybrid coarse- and fine-grained RTS technique.

3 Empirical Studies of `DejaVu` and `TestTube`

As discussed in Section 2.3.2, we expect `TestTube` always to select a superset of the test suite selected by `DejaVu`. Conversely, we expect `DejaVu` to be more precise in its selection of test suite subsets than `TestTube`. Furthermore, we expect `DejaVu`'s analysis to be more costly than `TestTube`'s, but since `DejaVu` might create smaller test suite subsets, this extra analysis cost might be offset by savings in test execution cost. The empirical studies described in this section are meant to investigate these intuitions.

3.1 Objectives

Our primary objective was to empirically investigate the relative cost-benefit tradeoffs involved in utilizing `TestTube` or `DejaVu` for regression test selection. We also wished to compare the cost-benefits of using those tools to those of not using regression test selection at all. The independent variables we studied for this comparison are our subject programs, their test suites, and the modifications that were made to these programs. The dependent variables are the percentage of test cases selected by each technique from the full test suites under consideration, and the cost of performing that test selection. We describe and discuss these quantities in greater detail in the sections that follow.

Program Name	*Number of Functions*	*Lines of Code*	*Number of Versions*	*Test Pool Size*	*Test Suite Avg Size*	*Description of Program*
printtok1	21	402	7	4130	16	lexical analyzer
printtok2	20	483	10	4115	12	lexical analyzer
replace	21	516	32	5542	19	pattern replacement
schedule1	18	299	9	2650	8	priority scheduler
schedule2	16	297	10	2710	8	priority scheduler
tcas	8	138	41	1608	6	altitude separation
totinfo	16	346	23	1054	7	information measure
space	270	6218	38	13520	155	ADL interpreter
player	766	49316	5	—	1033	transaction manager

Table 1: Experiment subjects.

3.2 Subject Programs, Versions, and Test Suites

Table 1 describes the programs, modified versions, and testing related materials used as subjects in our studies.[4] The programs are all written in C; for each program, the table lists the number of functions, the number of lines of code in the base versions of the C source and header files (measured as nonblank, noncomment lines), the number of versions available, and a brief description. The first eight programs have large test pools from which we generated test suites by a process described below; the table lists the test pool and average test suite sizes for each program. The ninth program, `player`, has only one test suite of 1033 test cases.

The Siemens Programs. The base programs, modified programs, and test pools described on the first seven lines of Table 1 were assembled originally by researchers at Siemens Corporate Research for use in experiments with data-flow- and control-flow-based test adequacy criteria [12]. We refer to these as the *Siemens programs*. The programs perform a variety of tasks: `tcas` is an aircraft collision avoidance system; `schedule1` and `schedule2` are priority schedulers; `totinfo` computes statistics over input data; `printtok1` and `printtok2` are lexical analyzers; and `replace` performs pattern matching and substitution.[5]

The researchers at Siemens sought to study the fault-detecting effectiveness of coverage criteria. Therefore, they created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [12, p. 196]. Only non-identical faults were retained; however, the researchers did not control for location or type of fault, and thus, in some cases some faults appear similar to, or occur in the same statements as, other faults. For the purposes of our study, we view these faulty modified versions of the base programs as ill-fated attempts to create new versions.

For each base program, the researchers at Siemens created a large test pool containing possible test cases for the program. To populate these test pools, they first created an initial suite of black-box test cases "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code" [12, p. 194], using the *category partition method* and the Siemens Test Specification Language tool [2, 17]. They then augmented

[4] The programs, program versions, test cases, test suites, tools, and other materials used in these studies, as well as all data collected in the studies, can be accessed by contacting the second author.

[5] The programs `printtok1` and `printtok2`, and `schedule1` and `schedule2`, appear to be pairs of independently written programs each computing the same function but using different algorithms and sharing no code.

this pool with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control-flow graph was exercised by at least 30 test cases.

Different test suites — even test suites that achieve equivalent overall coverage of specific code components — may possess different characteristics (e.g., they may have different fault-revealing capabilities, or trace different sets of paths through the program under test with varying frequencies). Therefore, to experiment with test suites, researchers typically control for such differences by using multiple distinct test suites in their studies. We have taken this approach in these studies.

To obtain such test suites for the Siemens programs, we used test suites previously generated for a previous study of regression testing by Rothermel and Harrold [22]. Rothermel and Harrold had used the test pools for the base programs, and test-coverage information about the test cases in those pools, to generate 1000 branch-coverage-adequate test suites[6] for each program. For each base program P, each branch-coverage-adequate test suite T was generated by randomly choosing test cases from the test pool for P, and adding a test case to T only if it added to the cumulative branch coverage of P achieved by the test cases added to T thus far. Test cases were added to T until T contained, for each dynamically exercisable branch in the base program, at least one test case that would exercise that branch. Any duplicate test suites were discarded; thus, each of the 1000 test suites is unique. Table 1 lists the average sizes of these test suites.

The Space Program. The `space` program consists of 9564 lines of C code (6218 executable). The program was designed by the European Space Agency (ESA) to interpret an array definition language (ADL), which simplifies the alignment of antennae on satellites. `Space` reads a file that contains ADL statements, checking the contents of the file for syntactic correctness. If the ADL file is correct, `space` outputs a data file containing a list of array elements, positions, and excitations; otherwise, the program outputs an error message.

The `space` subject consists of a single base program and 38 associated versions; each version consists of the base program, augmented with the code associated with one fault previously found in the program. (The first 33 faults originated with the actual development team, we found five additional faults while developing test cases for the program.) Note that the versions of `space`, like the Siemens program versions, are not sequential, in that they do not represent any ordering in which faults were found.

We constructed a test pool for `space` in two stages. We obtained an initial pool of 10,000 randomly generated test cases, created by Vokolos and Frankl for an earlier study [25]. Beginning with this initial pool, we instrumented the program for branch coverage, measured coverage, and then added additional test cases to the pool until it contained, for each executable branch in the control flow graph for the program,[7] at least 30 test cases that exercised that branch. This process yielded a test pool of 13,585 test cases. Finally, we used `space`'s test pools to obtain branch-coverage-adequate test suites for the program, following the same process utilized for the Siemens programs. This process yielded 1000 distinct test suites of sizes ranging from 141 to 169 test cases, and averaging 155 test cases.

The Player Program. The program described on the last line of Table 1, `player`, is a component of the software distribution for the Internet-based game, `Empire`. Several modified versions of `player` have been created to fix bugs or add functionality. For the base version listed in Table 1, five versions,

[6] A test suite T is branch-coverage-adequate for program P if every dynamically feasible outcome of every predicate in P is exercised by at least one test case in T.

[7] We also treated 17 branches executed only following the failure of a `malloc` system call as nonexecutable.

created by different programmers for different reasons, were collected. Like `space`, then, `player` reflects real modifications in distinct, non-sequential, actual program versions; however, unlike `space`, differences between versions represent significant changes to the base version, as summarized in Table 2. Also, unlike `space` and the Siemens programs, the modifications do not involve any (known) faults.

Version	*Functions Modified*	*Lines of Code Changed*
1	3	114
2	2	55
3	11	726
4	11	62
5	42	221

Table 2: Modified versions of `player`.

The single test suite for `player`, created for use in an earlier study [22], contains "black box" test cases. These were constructed from the `Empire` information files, which describe 154 commands recognized by `player`, and the parameters and special side effects associated with each command. The information files were treated as informal specifications; for each command, test cases were created to exercise each parameter and special feature, as well as to test erroneous parameter values and conditions. Because the complexity of commands and parameters varies widely over the `player` commands, this process yielded between 1 and 30 test cases for each command, ultimately producing a test suite of 1033 test cases.

Further Discussion. These three sets of programs provide a range of subjects whose sizes vary by successive orders of magnitude. The Siemens programs are small, sequential programs that perform well-defined tasks. `Space`, at 6000 lines of code, is somewhat more complex, and represents a real application developed by the ESA; furthermore, its versions contain real faults. Finally, `player` forms one module of a larger software system that consists of several concurrently executing tasks, although the `player` component performs only one of these tasks.

This diversity among subject programs lets us sample the performance of `DejaVu` and `TestTube` in different situations. As we expect that neither tool would always prove superior to the other, the programs let us probe the relative strengths of each. For instance, because of the small size of the Siemens programs and the correspondingly small size of their test suites, we expect that neither `TestTube` nor `DejaVu` will prove more efficient than the retest-all technique on these subjects. On the other hand, these small programs do provide a base against which to measure the tools' successes. Furthermore, the subjects provide a large number of versions to run both safe RTS techniques against. Like the Siemens programs, `space` also possesses small, quickly executed test suites. As such, we also expect both tools to prove inefficient when run on `space`. `Space`, however, does provide us with comparative precisions for `DejaVu` and `TestTube` on real program versions; furthermore, `space` should reveal the potential advantages of `TestTube`. `Player`, on the other hand, possesses the potential to reveal the strengths and limitations of the two methods relative to a more realistically sized program. This three-step staircase of programs should suggest a trend for the behavior of safe RTS techniques on larger programs.

3.3 Method

For these studies, we constructed an implementation of `TestTube` using the latest versions of the CIA database program, the APP instrumenter, the AST libraries, and scripts written in Perl.[8] To collect data for `TestTube`, we instrumented the executables for the base versions and executed them against the original test suites to generate test execution traces. The logs so generated contained a record of the function executions, along with a list of the files in which the functions are located. We then analyzed the logs to produce a formatted list of functions executed at least once. (More efficient mechanisms are available for computing function traces; however, because we consider trace collection to be a preliminary phase cost, inefficiencies in our computation of these traces do not affect the results of these studies.) We then ran a `TestTube` *closure* on the resulting function list, which involved adding to the list of functions all macros, global variables, and user-defined type names reachable from those functions without going through another functional entity. All of these tasks were considered to occur in the preliminary phase, because they run on the original code base only. In addition, the creation of a CIA database for the unmodified portions of the program was considered a preliminary phase cost because `TestTube` can just reuse pre-existing information.

The critical phase of `TestTube`'s activities began with the construction of new database entries for modified code files, and generation of a difference database. The difference database was built from the CIA databases for the two versions of the program. A difference file was then extracted from the difference database: this file lists all entities that are reachable through code changes. Finally, a script was run on the closure files, selecting any test case that included one or more of the entities discovered in the difference file.

For `DejaVu`'s we used an existing implementation [22], and a script to execute that implementation and select test cases on the program versions. As with `TestTube`, we considered the creation of execution traces and the construction of base CFGs for the program to be preliminary costs for `DejaVu`, but we considered construction of CFGs for the modified program to be entirely critical-phase. Alternatively, one could construct CFGs for unchanged portions of the modified code in advance; however, the implementation as it exists does not support this option. If it did, `DejaVu`'s critical-phase costs could be reduced. We also considered the process of walking the CFGs and selecting test cases for re-execution to be entirely critical-phase costs.

For both techniques, after the analysis was performed, all selected test cases were executed, and the time required to execute and validate the output of the test cases was recorded. We also executed the original test suite, to measure the cost of the retest-all technique. We then calculated the cost of each individual run of an RTS technique in terms of the cost model of Section 2.2, as the sum of the critical-phase cost of the analysis plus the cost of executing selected test cases.

We applied this method for the Siemens programs and `player` on a Sun Microsystems SPARCstation 1+ with 24MB of physical memory.[9] We used a Sun Microsystems UltraSparc 1 with 50 MB of physical memory for the runs of `DejaVu` and `TestTube` on `space`.[10] To control factors that could influence runtime, access to the machines was limited to our processes, and all data was written to local storage during the period in which timing data was being collected.

[8] An older version of `TestTube` based on KornShell scripts exists, but changes in newer versions of the CIA software have rendered it inoperable.

[9] SPARCstation is a trademark of Sun Microsystems, Inc.

[10] Differences in architectures were unavoidable due to differences in the platforms on which the subject programs could execute; we take this into account when comparing results.

Program	*Mean Savings for DejaVu*	*Median Saving for DejaVu*	*Standard Deviation for DejaVu*	*Mean Savings for TestTube*	*Median Savings for TestTube*	*Standard Deviation for TestTube*
printtok1	0.44	0.36	0.05	0.12	0.13	0.27
printtok2	0.67	0.69	0.09	0.18	0.18	0.50
replace	0.56	0.55	0.07	0.40	0.42	0.20
schedule	0.47	0.49	0.10	0.39	0.46	0.14
schedule2	0.25	0.18	0.08	0.25	0.18	0.08
tcas	0.34	0.36	0.04	0.27	0.18	0.10
totinfo	0.26	0.27	0.09	0.16	0.07	0.14

Table 3: Means, medians, and standard deviations in precision results for `TestTube` and `DejaVu` over all program version statistics on the Siemens programs (table entries represent fractions of the original test suite excluded from re-execution).

3.4 Study 1: Results for `TestTube` and `DejaVu` on the Siemens programs

Study 1 compared the precision of `TestTube` and `DejaVu`, as well as the costs of using `TestTube`, `DejaVu`, and the retest-all technique, on the Siemens programs.

3.4.1 Precision Results for `DejaVu` and `TestTube`

Figure 1 depicts the average precision results obtained in this study. The figure contains a separate graph for each of the seven programs. In each graph, each version of the program occupies a position along the x-axis and is represented by an "X", an "O", and (in some cases) a directed edge between them. The position of the (center of the) "X" denotes the percentage of test cases selected by `DejaVu`, averaged over the 1000 test suites for the given version. The position of the (center of the) "O" denotes the percentage of test cases selected by `TestTube`, averaged over the 1000 test suites for the given version. The length of the directed edge between the "X" and the "O" denotes the size of the difference between the two precisions. In cases where the two tools had equal average precisions, this edge is absent.

By their nature, `TestTube` and `DejaVu` necessarily exhibit a positive treatment effect for precision if they remove test cases (the size of this treatment effect is, of course, of interest). Thus, we do not analyze our data to prove or disprove that assertion. Similarly, the test cases removed by `DejaVu` are not independent of the test cases removed by `TestTube` (they are necessarily a subset). Thus, if `DejaVu` ever removes more test cases than `TestTube`, it exhibits a greater positive treatment effect than `TestTube` — Figure 1 demonstrates that `DejaVu` can remove greater numbers of test cases. Thus, we instead focus on measures that expose characteristics of the raw data itself. In Table 3, we present the averages of the means, medians, and standard deviations of the precisions achieved by `TestTube` and `DejaVu` over all program versions, for each of the programs. The values in the table represent the fraction of test cases excluded by the tools. Of course, using an average of standard deviations is not a precise statistical measure, but it does provide some sense of the distributions of the individual data sets.

The results show that `DejaVu`'s greater expenditures on analysis often allowed it to select more precise subsets that `TestTube`, but in many cases, `TestTube` matched `DejaVu`'s precision. We also found that the relative precision of the two tools varied with respect to the subject program under test. On `schedule2`, for instance, the two methods exhibited virtually identical precision on all versions. In contrast, on `printtok2`, `DejaVu` was always substantially more precise than `TestTube`. A reasonable conclusion follows: the relative success of `DejaVu` and `TestTube` is due in part to the structure of the program under test. In particular, we

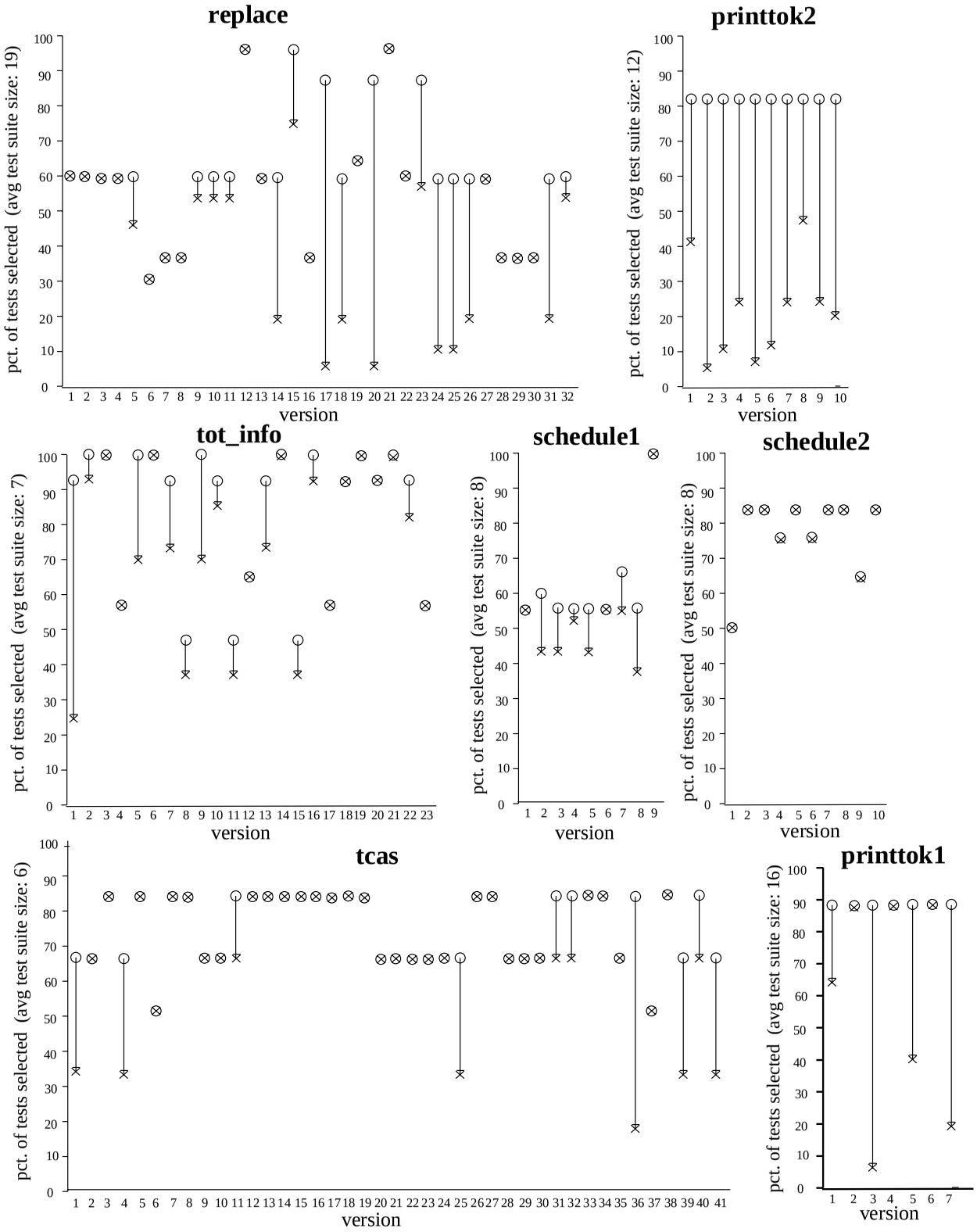

Figure 1: Test selection precision for `DejaVu` ("X" marks) vs `TestTube` ("O" marks) on the Siemens programs.

Program	*TestTube Analysis Time* ($A_{ttube}(T,P,P')$) ($A_{ttube}(T,P,P')$)	*TestTube Test Execution Time* ($E(T_{ttube},P')$ $+V(T_{ttube},P')$)	*DejaVu Analysis Time* ($A_{dvu}(T,P,P')$) ($A_{dvu}(T,P,P')$)	*DejaVu Test Execution Time* ($E(T_{dvu},P')$ $+V(T_{dvu},P')$)	*Retest-all Time* ($E(T,P')$ $+V(T,P')$)
printtok1	4.6	2.4	8.2	1.5	2.7
printtok2	4.4	1.6	5.9	0.6	1.9
replace	4.0	3.3	10.5	2.2	5.0
schedule	3.4	1.4	7.2	1.3	2.3
schedule2	3.3	1.7	7.4	1.7	2.2
tcas	3.1	1.2	8.5	1.1	1.7
totinfo	5.4	1.9	8.4	1.7	2.3

Table 4: Average timing results (in seconds) for `TestTube`, `DejaVu`, and the retest-all technique, on the Siemens programs.

postulate that programs with mostly linear control flow are less amenable to safe RTS techniques such as those embodied in `DejaVu` and `TestTube`, primarily because a single change can cause the entire test suite to be selected. Programs with more complex control flow, on the other hand, give `TestTube` and `DejaVu` greater opportunities for reducing test suite size, with more of these opportunities accruing to `DejaVu`.

3.4.2 Efficiency Results for `DejaVu`

Table 4 depicts the average costs of employing `DejaVu` on the Siemens programs and versions, relative to the costs of employing `TestTube` and the retest-all technique. The table lists analysis and execution/validation times in terms of the components of the cost model given in Section 2.2 (with column headings labeled accordingly). As the data show, `DejaVu` was always more expensive to run on these small programs than `TestTube` or the retest-all technique. In fact, due to the short execution times for test cases, `DejaVu`'s analysis time alone exceeded the total time required to re-execute the entire test suite. Furthermore, many of the small programs in this study have relatively few branches in their control flow; thus, even small numbers of changes infect large numbers of test cases, limiting reductions in test suite size, and limiting savings in test case execution and validation time. On the other hand, while `DejaVu` was not cost-effective for the small programs of this study, its selected subsets required less time to execute and validate (by a percentage ranging from 23 to 68 percent across the seven programs) than the entire test suite. Furthermore, for six of the seven programs (i.e., all but `schedule2`), `DejaVu` selected (on average) test suites that had shorter execution times than the test suites selected by `TestTube`.[11]

We will show in Study 3 that `DejaVu`'s expenditure of time on analysis can begin to pay off with larger and more complicated programs. Of course, even with small programs `DejaVu` might provide benefits if test cases take more significant amounts of time to complete. For instance, if the Siemens test suites had required greater time to execute and validate (e.g. if human effort were required for validation) the savings resulting from selection of more precise subsets could have offset the analysis costs.

3.4.3 Efficiency Results for `TestTube`

Table 4 also shows the average analysis time, and the reduced test suite execution and validation times, for `TestTube`. As the data show, the analysis times for `TestTube` were substantially lower than the analysis times for `DejaVu`. Furthermore, part of this cost arises from the total recompilation of the `CIA` database.

[11] As can be seen in Figure 1, on three versions of `schedule2` (4, 6, and 9), the average percentage of test cases selected by `DejaVu` was lower than the average percentage of test cases selected by `TestTube`, but this difference was too small to account for any measurable difference in test suite execution times.

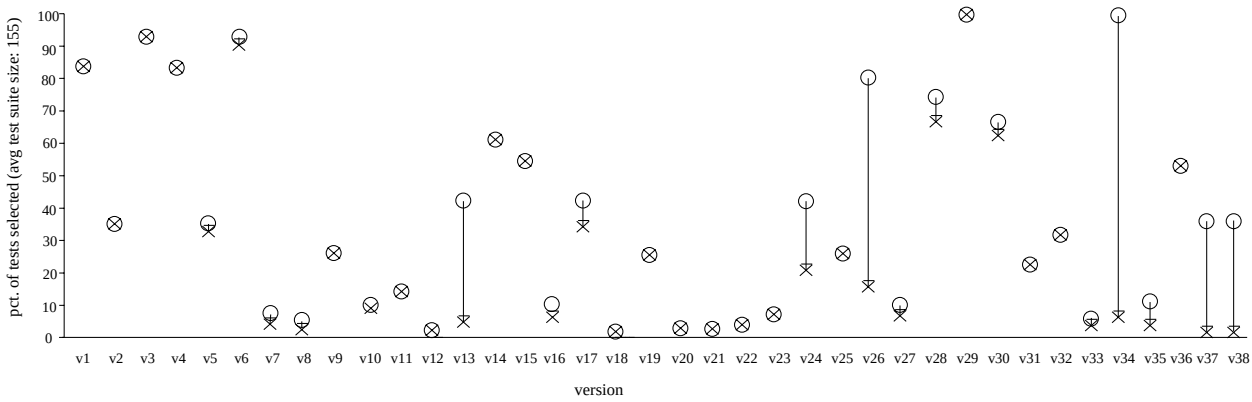

Figure 2: Test selection precision for `DejaVu` ("X" marks) and `TestTube` ("O" marks) on `space`.

With these small programs, we must recompile the entire database for each changed version, because the programs consist of single files; on larger programs, we would expect to need to recompile only a portion of the database. This factor slightly inflates the analysis time of `TestTube` for the Siemens programs (but not enough to make `TestTube` cost effective if total recompilation was not a factor).

In this study, `TestTube`'s critical-phase cost was always less than `DejaVu`'s critical-phase cost. However, the execution and validation times for test cases selected by `TestTube` equaled or exceeded the execution and validation times for test cases selected by `DejaVu`. Like `DejaVu`, `TestTube` always lost to the retest-all technique. Also like `DejaVu`, however, `TestTube` might have provided benefits with these programs if test case execution and validation had been more expensive. Moreover, if that had been the case, then given the lower cost of `TestTube`'s analysis, such benefits might have been possible for `TestTube` in cases where they would not have been possible for `DejaVu`, depending on whether `TestTube` could be sufficiently precise in those cases.

3.5 Study 2: Results for `TestTube` and `DejaVu` on `space`

Study 2 employed `DejaVu` and `TestTube` on the larger `space` program. This study provides an important step between the Siemens programs and `player`. First, it involves a program that is an order of magnitude larger than the Siemens programs and an order of magnitude smaller than `player`. Moreover, while `space` is still small, it was developed by engineers to meet real-world needs, thus taking it out of the realm of toy programs.

3.5.1 Precision Results for `DejaVu` and `TestTube`

Figure 2 depicts the precision results for this study. The figure contains a graph similar to those presented in Figure 1, depicting the percentages of test cases selected by `DejaVu` ("X" marks) and `TestTube` ("O" marks) for each version, *on average over the 1000 test suites.*

Applied to `space` and its versions and test suites, `DejaVu` again always equaled or exceeded `TestTube` with respect to precision. On the other hand, `TestTube` did often manage to maintain precision levels close to those of `DejaVu`. In fact, for half of the program versions, `TestTube` equaled `DejaVu` in precision. Furthermore, `TestTube` proved more than 50% less precise than `DejaVu` on only 7 versions.

Later, in Study 3, we will see that `TestTube` lags significantly behind `DejaVu` in terms of precision on

Measure	*TestTube Analysis Time* $(A_{ttube}(T,P,P'))$	*TestTube Test Execution Time* $(E(T_{ttube},P')$ $+V(T_{ttube},P'))$	*DejaVu Analysis Time* $(A_{dvu}(T,P,P'))$	*DejaVu Test Execution Time* $(E(T_{dvu},P')$ $+V(T_{dvu},P'))$	*Retest-all Time* $(E(T,P')$ $+V(T,P'))$
Min	3.9	0.0	14.0	0.0	3.0
Median	4.7	1.5	15.6	0.2	5.9
Max	5.4	7.6	20.0	6.5	7.6

Table 5: Timing results (in seconds) for `TestTube`, `DejaVu`, and the retest-all technique on `space`, given as minimum, median, and maximum costs over all versions and test suites.

`player`. Also, as we saw in Study 1, on some of the Siemens programs (such as `printtok2`) `DejaVu` tended to be significantly more precise than `TestTube`. These results suggest that something fundamental in the design of `space` stymies `DejaVu`. The code for individual procedures in `space`, for example, tends to be relatively linear (i.e., employing few or no branches), particularly when compared with the code in the `player` program. A high degree of linearity suggests that a high percentage of test cases executing that procedure will in turn execute the changed code. In such cases, selecting at the function level can be nearly as precise as selecting at the statement level.

3.5.2 Efficiency Results for `DejaVu` and `TestTube`

Table 5 depicts the minimum, median, and maximum costs (among all version/test-suite combinations) for employing `DejaVu`, `TestTube`, and the retest-all technique on the versions of `space`. Results for `DejaVu` and `TestTube` are again separated into analysis and test case execution/validation costs. The results exhibit a great deal of variability, especially in test execution time. Once again, however, `DejaVu` costs significantly more than `TestTube` or the retest-all technique.[12] Also, note that `TestTube`'s analysis time is strongly correlated with the execution time of the test subset produced. In particular, when most of the test suite was selected (as occurred for the maximum time case), the analysis cost was relatively low. When few test cases needed to be executed, analysis costs tended to be relatively high. The likely reason for this behavior is rather simple: `TestTube` must examine the entire entity trace for each test case that is excluded, but can immediately terminate examination of the entity trace of a test case as soon as it finds a dangerous entity.

As with the Siemens programs, the relative efficiency of the tools is not surprising. `DejaVu` must create control-flow graphs and then walk them in order to select test cases; this step alone usually required about 16 seconds of analysis. At the same time, the average test suite for `space`, possessing only 155 test cases, can be executed and validated in about 6 seconds. `DejaVu` simply cannot compete with the retest-all technique if test suites contain small numbers of low-cost test cases.

`TestTube` also suffers from this limitation; however, since `TestTube`'s analysis cost (5.4 seconds median) is slightly less than the median cost of running all test cases, `TestTube` almost reaches the break-even point. Had the test cases taken slightly longer to execute, `TestTube` would have been cost-effective for `space`.

Although we have not yet presented the results of our third study, we can at this point make a preliminary observation which is subsequently borne out in Study 3. `DejaVu` proves cost-effective only when large percentages of the test suite can be excluded or when the cost of a single test case is large in comparison to `DejaVu`'s analysis time. Thus, `DejaVu` appears best when few changes have been made to the code, or when

[12] Recall that `space` executed on a different (and faster) architecture than the Siemens programs and `player`; this caused cost measurements on `space` to exhibit faster times than they would have had they been gathered on the same architecture as was used for the Siemens programs. This should be considered in any inter-study cost comparisons.

significant human cost or large amounts of computation are required per test case. TestTube, on the other hand, appears to be more resilient when faced with significant changes to the code, or when test cases are not too expensive relative to TestTube's analysis cost (which is lower than that of DejaVu).

3.6 Study 3: TestTube and DejaVu on player

Study 3 applied TestTube and DejaVu to the significantly larger player program. As previously discussed, the player subject set consisted of a base version of player and five modified versions. Since this subject has nearly 50,000 lines of code, player gives us an opportunity to study how scaling up the program size affects the behaviors of TestTube and DejaVu. Recall, however, that unlike the Siemens programs and space, player possesses only a single test suite; results are thus not averages over a larger set. Nonetheless, the data for this single test suite on the five versions of player may plausibly be seen to suggest a trend.

3.6.1 DejaVu Implementation Considerations

As described in Section 2.3.1, DejaVu uses a code instrumenter to obtain test history information. Because of limitations in our code instrumenter, we could not instrument all of the player code; thus, we could not obtain test history information for player.[13] Because DejaVu requires test history information to perform test selection, we could not use the original DejaVu tool to select test cases for player. We were able, however, to create (1) a method for determining the exact test set that the original tool would have selected, yielding precision and test execution cost measurements equivalent to those that the original tool would have produced, and (2) a version of DejaVu that provides a close upper bound on the analysis costs that the original DejaVu tool would incur. Because the validity of our results depends on the fairness of our approach to simulating DejaVu's behavior in this fashion, we describe the approach in detail.

To determine the exact test sets that the original DejaVu tool would have selected, we proceeded as follows. For each modified version player′ of player, we used the Unix diff utility and inspection of the code to locate differences between player and player′. We then edited player′ and instrumented it as follows. At each executable code location where the two versions differed, we inserted the function call "dejavu()". In cases where variable declarations differed, we found the locations in the code where those variables occurred and instrumented those locations as if they contained modified code. This approach produced results equivalent to those that would be produced by the implementation of DejaVu that we utilized in Studies 1 and 2.[14]

The dejavu() function, which we supplied, opens a file result, writes the phrase "selected" to that file, and closes it. We also edited function main in player′ and inserted, as the first executable statements in that function, code that opens the result file, writes the phrase "not selected" to that file, and closes it. We then compiled and linked player′. Given these edits, when our instrumented player′ is invoked, it immediately writes "not selected" to the result file; subsequently, if and only if modified code is encountered does the program invoke the dejavu() function. In that case, the dejavu() function overwrites the result file with

[13] This limitation was due only to the particular code instrumenter used, which was still in the prototype phase and not robust enough to accommodate all of the C usages in player. The newer instrumenter used for DejaVu on space in the second study can handle the various C usages, but does not function on the older SunOS4 system on which the player program operates.

[14] In general, we would not expect diff to produce results equivalent to those produced by DejaVu, due to differences in the algorithms used by the two tools. However, coupling the use of diff with manual inspection allowed us to correctly identify change points. We later verified the correctness of that identification using the modified DejaVu tool described in the following paragraphs.

the phrase "selected". Because the file is opened and closed anew on each call to the `dejavu()` function, the file always contains exactly one line; once the file contains the phrase "selected", it retains that phrase until the program terminates.

Given the foregoing instrumentation, when we run a test case t on `player`$'$, we know that t would be selected by `DejaVu` if and only if the `result` file contains the phrase "selected" when `player` terminates. The approach requires us to run all test cases in order to determine which test cases `DejaVu` would select, and thus is of use only for experimentation; however, the approach lets us determine exactly the test cases that that the original `DejaVu` tool would select. Given this result, we can then determine exactly the execution and validation time of the selected test cases.

We also needed to measure the costs associated with `DejaVu`'s test selection analysis for `player`. We were able to directly and exactly measure two of the three components of that cost. First, because `DejaVu` relies on `Aristotle` for control flow graph construction, and `Aristotle` successfully builds control flow graphs for `player`, we were able to precisely measure the cost of constructing control flow graphs for the modified versions of `player`; this cost dominates the overall cost of the analysis performed for `DejaVu`. Furthermore, we were able to use most of the original `DejaVu` implementation, including all code involved in performing the graph walk, to obtain a precise measurement of the graph traversal cost of the tool.

The one portion of the original `DejaVu` tool's cost that we could not measure directly, in the absence of test history information, was the time required to select test cases following identification of changes. To estimate this portion of the tool's cost, we modified the test selection portion of the `DejaVu` tool so that, on completion of its traversal, the tool performs n set union operations on sets containing k test cases, where n and k are parameters supplied at invocation. By setting n to the number of `dejavu()` calls present in the modified version, and k to the number of test cases known to be selected for that version, we forced the `DejaVu` tool to perform the maximum number of set operations that it could have performed given test history information.

It is important to note that the resulting tool can err only in overestimating the time required for `DejaVu`'s test selection phase. In our experiments with `player`, we determined that this test selection time never accounted for more than 7% of the time required by `DejaVu` to perform its graph-walk and test-selection activities (this time as measured for `player` is reported separately in Section 4 of this paper, in Table 7). This bounds the possible overestimate, with respect to the `player` versions that we considered, at 17 seconds. This overestimate is dwarfed by other components of the tool's cost, and it will be clear from our presentation of timing results that such an overestimate has no material effect on the conclusions that we are able to draw about the relative cost-effectiveness of the two tools.

3.6.2 Precision Results for `DejaVu` and `TestTube`

Figure 3 depicts the precision results for `DejaVu` ("X" marks) and `TestTube` ("O" marks) on `player`. As the figure shows, `DejaVu` always selected a more precise subset than `TestTube`. In some cases, the differences were extreme, as in versions 4 and 5, where `DejaVu` reduced the test suite by 90%, while `TestTube` selected the entire suite. On the other hand, `TestTube` came relatively close to `DejaVu` on versions 1, 2 and 3.

The large differences for versions 4 and 5 arose because changes in these versions were made to the `main` function in both versions. Thus, `TestTube` selected all test cases that executed `main`, which necessarily includes all test cases in the test suite. `DejaVu` gained here by analyzing the code and determining which

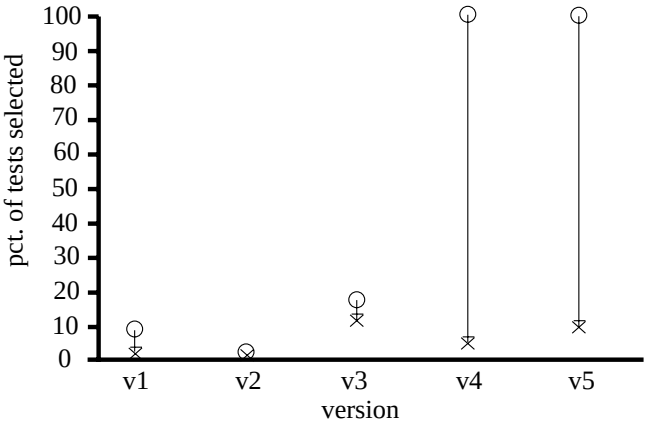

Figure 3: Test selection precision for `DejaVu` ("X" marks) and `TestTube` ("O" marks) on `player`.

Version	*TestTube Analysis Time* ($A_{ttube}(T,P,P')$) ($A_{ttube}(T,P,P')$)	*TestTube Test Execution Time* ($E(T_{ttube},P')$ $+V(T_{ttube},P')$)	*DejaVu Analysis Time* ($A_{dvu}(T,P,P')$) ($A_{dvu}(T,P,P')$)	*DejaVu Test Execution Time* ($E(T_{dvu},P')$ $+V(T_{dvu},P')$)	*Retest-all Time* ($E(T,P')$ $+V(T,P')$)
1	0:17:13	0:21:03	0:15:48	0:05:23	7:42:47
2	0:18:16	0:01:43	0:16:07	0:00:54	7:42:58
3	0:17:44	1:14:23	0:16:12	0:49:00	7:40:09
4	0:02:42	7:33:18	0:16:13	0:15:48	7:33:18
5	0:02:43	7:43:27	0:15:55	0:40:20	7:43:27

Table 6: Timing results (hours:minutes:seconds) for `TestTube`, `DejaVu`, and the retest-all technique on `player`.

test cases potentially executed changed code; it was able to determine that the modifications in `main` were on a subpath traversed by relatively few test cases. In essence, certain changes to code present larger obstacles for `TestTube`, while `DejaVu` can safely work its way around these obstacles.

3.6.3 Efficiency Results for `DejaVu`

Table 6 presents the costs of employing `DejaVu`, `TestTube`, and the retest-all technique on `player`, for each of the five versions. For the moment, we focus on `DejaVu` as compared with the retest-all technique. The cost of `DejaVu` is represented in the table by the columns third and second in from the right, which present analysis time and test execution/validation time, respectively.

Clearly, `DejaVu`'s behavior on `player` differs substantially from its behavior on the Siemens programs and `space`: `DejaVu` now consistently yields substantial savings across the versions, with an average saving of around 7 hours. Code size and test expense are likely causes of this change, but differences in the complexity of `player` and the types of test cases in `player`'s test suite might account for some of the variations. For instance, the Siemens programs are compact programs meant to accomplish single, small tasks. As such, one might expect test cases for the Siemens programs to execute large percentages of the program code. `Player`, on the other hand, has a large number of commands, each of which executes relatively small subsets of the code. Thus, a change would be much less likely to affect a majority of its test cases.

Another difference worth considering is the behavior of the test cases for the different programs. Although the number of test cases in a test suite divided by the number of lines of code is comparable for both `player`

and the programs in studies 1 and 2, the test cases for `player` required a bit less than 30 seconds each on average to complete; the Siemens test cases, on the other hand, required at most 0.5 seconds to complete, while the test cases for `space` required on average only 0.04 seconds each. As we discussed in regard to studies 1 and 2, if these test cases had required more time to execute, `DejaVu` might have been cost-effective on those programs. Similarly, if the `player` test cases had required significantly less time to execute, the retest-all technique would have been more cost effective on `player`.

3.6.4 Efficiency Results for `TestTube`

Table 6 also depicts the costs of regression testing using `TestTube`. As the table shows, `TestTube` exhibited diverse behaviors. For the first three versions, `TestTube` behaved in a fashion comparable to `DejaVu`, and thus cost significantly less than the retest-all technique. Conversely, for versions 4 and 5, the use of `TestTube` was as costly as the use of the retest-all technique. The reasons for this are relatively clear: `TestTube` could not detect the subtle impacts of individual changes in the code in these two versions.

On the other hand, the cost of using `TestTube` never greatly exceeded the cost of using the retest-all technique. The most expensive portion of `TestTube`'s task was analyzing the code to select test cases, and since (as described in Section 3.5) the cost of that task tended to decrease with the percentage of test cases selected, `TestTube`'s test selection analysis required only about two minutes for the `player` program in versions 4 and 5. Thus, the critical-phase cost of using `TestTube` was less than about 1% higher than the cost of using the retest-all technique. For the other three versions, `TestTube`'s test selection process required from seventeen to eighteen minutes; however, since this analysis led to selection of test suites that were substantially smaller than the original suite, it led to significant savings.

The data shows that in the three cases in which `TestTube` did succeed, it nevertheless cost more than `DejaVu`: 0:17:15 more on version 1, 0:02:58 more on version 2, and 0:26:55 on version 3. In fact, on these three versions, `TestTube`'s analysis cost alone always exceeded `DejaVu`'s cost, and `TestTube`'s lesser precision exacerbated this difference. In relation to the total testing time of over 7 hours, however, these differences are not great. Furthermore, `TestTube` does possess the advantage that it can be implemented with only a set of simple scripts in most software environments. `DejaVu` requires more substantial effort to implement; however, its results on versions 4 and 5 show that it has the potential to produce savings considerably more substantial than those yielded by `TestTube`.

3.7 Summary

In summary, in our studies, we found that `TestTube` and `DejaVu` were often comparable in their costs, although for some programs, one tool or the other could prove superior. For `space`, for example, `TestTube` proved the more efficient choice. For `player`, on the other hand, `DejaVu` consistently reduced reverification time by a significant amount, whereas `TestTube` varied in its behavior.

Our results suggest two main factors that can contribute to the success or failure of the two safe RTS tools. First, the program's size (or more probably the complexity of the program's control structure) seems to be an important factor. Presumably, larger programs may decompose more easily into different control paths, which, in turn, will allow `DejaVu` and `TestTube` to more readily separate out test cases with potentially changed behaviors. Second, the three studies indicate that the costs of executing and validating test cases can have a significant effect on the cost-effectiveness of both methods. Where test execution is inexpensive,

analysis will generally assume a larger percentage of the reverification cost. For a tool such as `DejaVu`, for which the bulk of the cost is in analysis of the code itself, large test suites can amortize the cost of analysis, in which case the tool might be cost-effective even for test suites with test cases that require only seconds to execute and validate. Most of `TestTube`'s cost, on the other hand, is derived from the time required to select test cases using the difference file; thus it is likely that `TestTube` will suffer if test cases are inexpensive to execute and validate. On the other hand, when test cases are sufficiently expensive to execute and validate, our results suggest that both methods can make substantial gains in efficiency. Further research is necessary to determine if these are the only factors contributing to the success or failure of these tools, or even the most important ones.

3.8 Threats to Validity

For completeness, in this section we discuss the potential threats to the validity of our studies.

3.8.1 Construct Validity

Construct validity deals with the issue of whether or not we are measuring what we purport to be measuring, that is, whether our measurements of independent and dependent variables are valid.

In these studies, our measurements of precision and cost partially capture the factors important to assessing the cost-benefits of regression test selection. However, our measures do not account for the possibility that test cases may have costs and benefits beyond those measurable in terms of time.

A further source of threats lies in the way the timings were obtained. As discussed in Section 3.3, we attempted to control for these threats by limiting access to the machine and limiting network-dependent behavior. Further control comes from the fact that on the Siemens programs and `space` our results average the runs of 1000 test suites, and on `player` our final timing results were obtained by averaging results collected in five separate runs.

Finally, as discussed in Section 3.6.1, our measurements of cost in Study 3 are conservative due to the need to simulate `DejaVu`'s behavior.

3.8.2 Internal Validity

Threats to internal validity are causal influences other than the phenomena underlying the independent variables that can affect the phenomena underlying the dependent variables without the researcher's knowledge. Our greatest concern with respect to internal validity in these studies are instrumentation effects that can bias our results. One source of such effects are faults in the tools and scripts utilized. To reduce the likelihood of such effects, we performed code reviews on all tools, and validated tool outputs on a small but tractable program (130 lines, 12 versions, 15 test suites), and on several of the Siemens test suites.

3.8.3 External Validity

Threats to external validity are conditions that limit our ability to generalize the results of our studies. The threats to external validity of our studies are centered around the issue of how representative the subjects of our studies are. The Siemens programs, although nontrivial, are small, and larger programs may be subject to different cost-benefit tradeoffs. `Space` and `player` are "real" programs; however, they are only two such programs. Furthermore, peculiarities such as the complexity of the control- and data-flow in the code of

the subject programs, the kind and locality of changes to the subject programs, and the composition of the subject test suite, all contribute threats to the external validity of the results. To reduce this threat, we used a factorial design to apply each test selection tool to each test suite and each subject program.

The fact that the modifications in the Siemens programs involved synthetic (seeded) faults, some of which are similar and some of which affect the same program statements, may also affect our ability to generalize our results. The faults in `space` were not artificially injected; still, we cannot claim that these faults are representative of typical faults encountered during software development. Furthermore, unlike real software, the `space` versions each contain only one fault with respect to the base version; thus, the difference between the base and modified versions often amounts to only two or three lines of code. The study with `player`, however, used real and larger modifications.

Our branch-coverage-adequate test suites are reasonable test suites that could be generated in practice if coverage-based testing of the programs were being performed; however, many other varieties of test suites are possible.

In general, however, such threats to external validity as these can be addressed only by additional studies on additional subjects.

4 Further Exploration

We had conjectured that a hybrid coarse- and fine-grained technique might be able to yield the precision of the fine-grained technique, but at a cost closer to that of the less expensive coarse-grained technique. We were interested in exploring that conjecture further.

Initially, the results of our third study appeared to contradict our conjecture. `TestTube`'s timings on `player` shown in Table 6 always exceed – sometimes significantly – `DejaVu`'s. Further analysis of individual cost factors operating in study 3, however, proved illuminating. Table 7 separates out the various contributors to `TestTube`'s and `DejaVu`'s critical-phase costs when run on `player`. In the table, *pre-selection analysis* costs include costs that occur prior to test selection: for `DejaVu` this involves only control flow graph construction, and for `TestTube` it involves all activities up through creating a difference database and extracting the differences. In the table, *test-selection analysis* costs included costs of the remaining analysis: for `DejaVu` this involves walking control flow graphs and selecting test cases and for `TestTube` it involves obtaining closures and selecting test cases. The final cost component, *test execution* cost, is the time required to run and validate the outputs of the selected test cases.

Note that the bulk of `TestTube`'s cost comes from test-selection analysis and test execution, as opposed to pre-selection analysis of code differences. In fact, pre-selection analysis time contributes little to `TestTube`'s total execution time. `DejaVu`, on the other hand, incurs over half of its cost from its pre-selection analysis, in creating CFGs for the functions in the programs. But suppose we knew which functions might contain regression faults; then, with slight modifications, `DejaVu` could omit construction of CFGs for other functions (as well as the traversal of those CFGs) thus drastically reducing execution time. More to the point, if `DejaVu` could bound the set of functions that possibly contain regression faults, `DejaVu` could save time in proportion to the functions excluded. `TestTube`, as we have seen, can often create drastically reduced test suite subsets; furthermore, even when `TestTube` does not create reduced subsets, it virtually always localizes changes to a highly constrained set of functions and global entities. A tool like `DejaVu` could use this information as

	DejaVu			TestTube		
Version	*Pre-Selection Analysis*	*Test Selection Analysis*	*Test Execution*	*Pre-Selection Analysis*	*Test Selection Analysis*	*Test Execution*
1	0:15:48	0:02:50	0:05:22	0:01:43	0:15:30	0:21:03
2	0:16:07	0:03:58	0:00:52	0:01:46	0:16:30	0:01:43
3	0:16:12	0:02:43	0:49:10	0:01:44	0:16:00	1:14:40
4	0:16:13	0:03:10	0:11:37	0:01:42	0:01:00	5:33:36
5	0:15:55	0:03:13	0:40:20	0:01:43	0:01:00	7:43:27
average	0:16:03	0:03:13	0:21:44	0:01:44	0:10:00	2:58:48

Table 7: Costs (in hours:minutes.seconds) of executing `DejaVu` and `TestTube` on each version of `player`, partitioned into costs of preliminary analysis necessary to support test selection, test selection analysis, and test execution.

input and operate only on this reduced set.

To further investigate this notion, we created a prototype hybrid coarse- and fine-grained regression test selection tool using pieces of the `TestTube` and `DejaVu` implementations. Given program P and changed version P', A "`TestTube`-like" front end of the prototype proceeds exactly like `TestTube` up to and including the point of creating a difference file (where the difference file contains the list of functions, global variables, macros and type definitions revealed by code differencing as changed). During the preliminary phase, this front end constructs `CIA` databases for P and P' and creates the same test histories and closures used by `TestTube`. Then, in the critical phase, just like `TestTube`, the front end creates a difference database and extracts a difference file.

Now, in a real hybrid tool, a "`DejaVu`-like" back end would simply pick out the changed entities from the difference file and operate only on the functions associated with those particular entities. Information on P could either be in memory or stored in a database, where entities in P's global namespace would already be individually available for easy access. Unfortunately, the `CIA` databases are designed for purposes other than our hybrid tool, so we were not able to create a tool that operated in this fashion. Instead, we created an interface program that converts the information in the difference file to a form usable by the `DejaVu`-like back end. The interface program, `Hybrid.java`, parses P and P' and outputs truncated programs containing functions mentioned in the difference file plus other global entities.

Given our interface program, the `DejaVu`-like back end of our hybrid tool can be run on the truncated files as if they were whole programs. As with the regular usage of `DejaVu`, the `DejaVu`-like back end requires that traces exist for the entire program; in the prototype, these traces are distinct from those used for the `TestTube`-like front end, but a real tool could easily generate traces for both parts. The back end of the hybrid tool then outputs the final set of test cases to be re-executed.

To obtain some initial data on the costs of using this tool, we applied it to `space`. The hybrid tool is necessarily at least as precise as `DejaVu`, and so we did not expect to see any significant changes in precision results. However, we were surprised to discover that the hybrid tool was actually *more efficient* than `TestTube` alone had been on `space`.

Figure 4 shows the costs for running the hybrid tool on the 38 versions of `space`, partitioned into the components of those costs attributable to the `TestTube`-like front end, the `DejaVu`-like back end, and the execution and validation of the selected test cases. The graph at upper-left shows these costs for versions 1-19, and the graph at lower-left shows these costs for versions 20-38. These costs can be compared with the data in Table 5; to aid in this comparison, however, the graph on the right in the figure shows the median

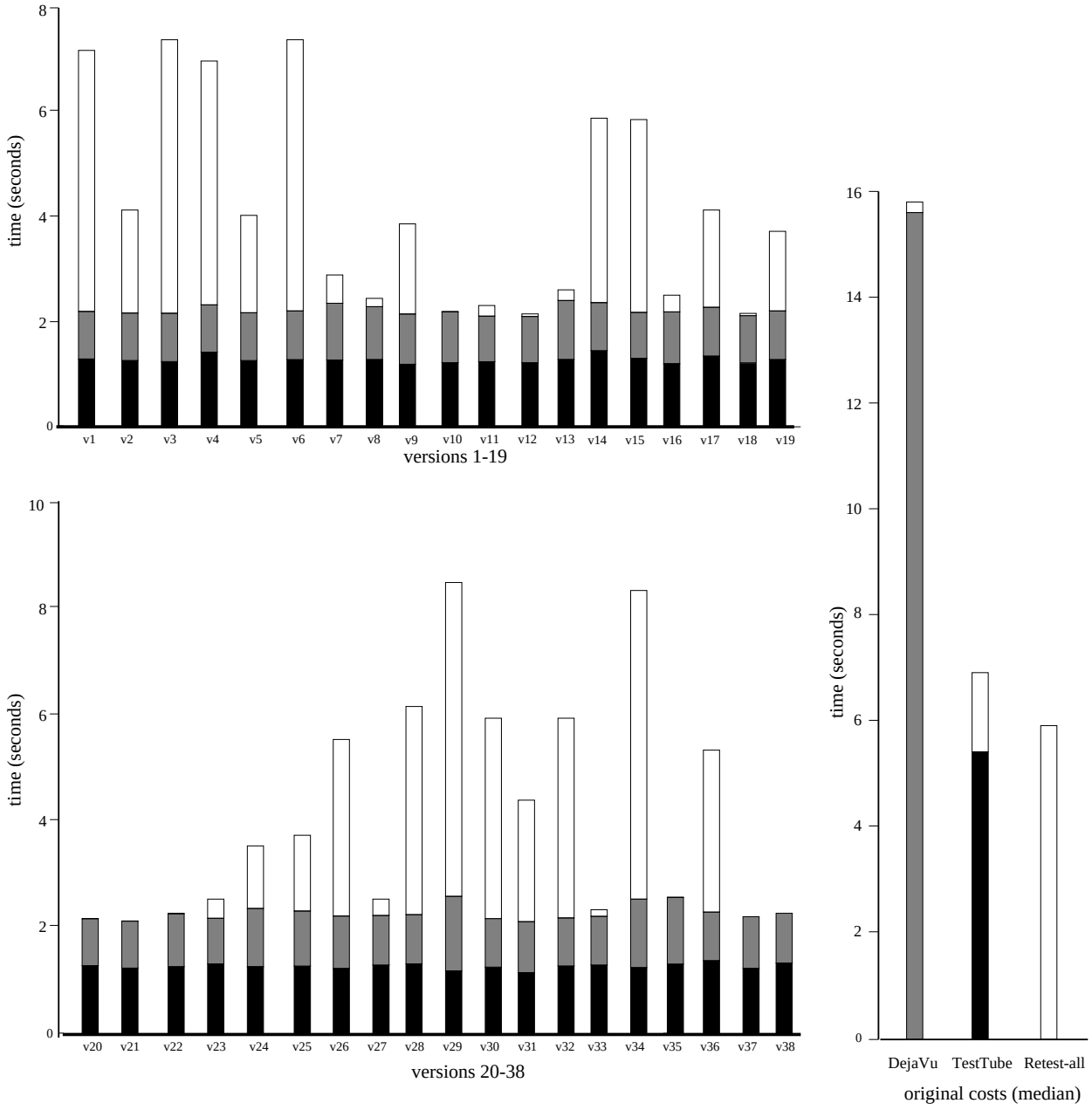

Figure 4: Left upper and lower: costs for running the hybrid tool on `space`, divided into costs of `TestTube`-like front end (black bars), `DejaVu`-like back end (grey bars), and execution and validation of selected test cases (white bars). Right: median costs measured in Study 2 for `DejaVu`, `TestTube`, and the retest-all technique, divided into costs of analysis and test execution. Vertical axes depicting costs share the same scale across all graphs.

costs of applying `DejaVu`, `TestTube`, and the retest-all technique to `space`, using white bars to indicate test execution/validation costs, and grey or black bars to indicate the `DejaVu` and `TestTube` analysis costs. Visual inspection of the graphs show that the hybrid tool always proved superior to the median case of `DejaVu`, proved superior to the median case of `TestTube` on all but 6 versions, and proved better than the retest-all median on all but 11 versions. Inspection of the data for individual versions confirms that the

hybrid tool was less costly than TestTube on 36 of the 38 versions, significantly less costly than DejaVu on all of the 38 versions, and less costly than the retest-all technique on 27 of the 38 versions.

The counter-intuitive result in which the hybrid tool out-performed TestTube can be further explained by analyzing the work performed by TestTube. After TestTube generates a difference file, it then must read through the closures for each test case, searching for entities in the difference file. These closures can be quite large, particularly for large programs. TestTube needs only to read the files line by line, but there can be quite a bit of text to scan. The truncated programs, on the other hand, often contain very little code; thus, even though DejaVu does more analysis of that code than TestTube, the small size of the truncated programs allows it to be very efficient.

These comparisons should be partly qualified. The times for the hybrid tool shown in Figure 4 include only the cost of the TestTube-like front end and DejaVu-like back end of that tool. We excluded the time required by Hybrid.java, because as we discussed earlier, this portion of our prototype hybrid tool would not be required in a production-quality implementation; thus, including its runtime inappropriately inflates the timing results one might expect for such a tool. Nevertheless, it seems reasonable to suggest that our results support the conjecture that a production-quality hybrid tool could incur no greater critical-phase cost than TestTube, and also gain a great deal of precision.

Note further that, while the hybrid tool can incur less critical-phase cost even than TestTube, the hybrid tool can also incur greater preliminary phase costs than either DejaVu or TestTube. Trace information will necessarily need to be more detailed for the hybrid tool than for TestTube alone, as DejaVu requires traces at the source statement or basic block level. At the same time, DejaVu does not require any pre-existing information other than traces, but the hybrid tool, like TestTube, requires the creation of a program database. Of course, much of the information stored for the hybrid tool can be used for other purposes, but even if the hybrid tool requires greater initial investments, preliminary phase costs are typically not as important to regression test selection as critical-phase costs, as discussed in Section 2.2.

This initial study suggests that further exploration of a hybrid coarse- and fine-grained regression test selection tool may be beneficial; we are continuing to pursue this avenue.[15]

5 Conclusion

In this paper, we have described the results of the first comparative empirical study of two safe regression test selection techniques. In particular, the study compared the cost-effectiveness of TestTube and DejaVu. We evaluated the precision of both tools, but more importantly, we described the tools' relative costs, as well as the costs of employing the tools as compared to using the retest-all technique.

The results of our studies suggested that a hybrid technique combining portions of the algorithms underlying TestTube and DejaVu has the potential to provide cost-effective safe regression test selection. As a result, we created and performed an initial study using a prototype hybrid tool. The results of that study suggest that the tool has the potential to equal DejaVu in precision, to greatly exceed DejaVu in efficiency, and to frequently exceed TestTube in efficiency. Further study of this approach is ongoing.

[15] A natural and desirable next step in this work would be to study the application of the hybrid tool to player; however, by the time our research reached this juncture, we no longer had access to a machine on which player could execute. To function on newer OS's, player would require substantial changes; moreover, that version of player is no longer supported by its creators. Thus, we are seeking additional and larger experiment subjects on which to continue experimentation.

Our results also suggest several additional avenues for future work. As outlined above, our studies have suggested that program complexity and test execution cost are two factors affecting the costs and benefits of regression test selection techniques. We have suggested some possible factors in program and test suite design and modification patterns that might underlie this, but further study of these factors and their effects on regression test selection techniques are necessary. Such studies should also consider the effects of varying test and fault costs.

It should also be evident from this work, however, that empirical evaluations of regression testing techniques require substantial infrastructure in the form of programs, modified versions, fault data, and test suites. Assembling such an infrastructure is difficult, yet once assembled, the infrastructure could serve as a basis for benchmark experiments. Our experiment subjects, with their versions and test suites, could conceivably constitute members of such a benchmark suite. Of course, construction and acceptance of such a benchmark suite would also require an understanding of the degree to which such a suite represents the general population of testing-related artifacts, and hence would require a certain level of agreement among software testing researchers and practitioners. Moreover, in practice, as our experience with the obsolescence of `player` indicates, such an infrastructure would need to be continually maintained, and the cost of such maintenance is non-trivial. Nevertheless, construction of such an infrastructure would greatly improve the prospects for obtaining meaningful empirical results with respect to regression testing techniques in particular, and for empirical studies of testing in general.

Beyond the in-vitro studies that such an infrastructure could support, another pressing need is for in-vivo case studies, in which techniques are applied to mature software products within realistic settings. Only through the results presented in this paper, coupled with results of such additional case studies, can we hope to demonstrate the economic viability of employing regression test-selection methods to reduce the exorbitant costs of software maintenance.

Acknowledgments

We thank Yih-Farn Chen of AT&T Labs - Research for explaining many details of the CIA database program. We also thank the anonymous reviewers and associate editor for suggestions (including, in particular, suggestions for graphically depicting precision data) that materially improved the paper. This work was supported in part by National Science Foundation under Faculty Early Career Development Award CCR-9703108 to Oregon State University. Siemens Corporate Research supplied the Siemens subject programs. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the `space` program and many of its test cases. Chengyun Chu assisted with further preparation of the `space` program and its test cases.

References

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 348–357, September 1993.

[2] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification*, pages 210–218, December 1989.

[3] T. Ball. On the limit of control flow analysis for regression test selection. *ACM International Symposium on Software Testing and Analysis*, pages 134–142, March 1998.

[4] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 352–361, October 1988.

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

[6] Y.-F. Chen, D. Rosenblum, and K.-P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.

[7] T.L. Graves, M.J. Harrold, J-M Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *The 20th International Conference on Software Engineering*, April 1998.

[8] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 299–308, November 1992.

[9] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, March 1997.

[10] M.J. Harrold, D.S. Rosenblum, G. Rothermel, and E.J. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, (to appear).

[11] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 362–367, October 1988.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering-1992*, pages 191–200, May 1994.

[13] B. Krishnamurthy, editor. *Practical Reusable UNIX Software*. Wiley, 1995.

[14] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintentance. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 282–290, November 1992.

[15] H.K.N. Leung and L.J. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance - 1990*, pages 290–300, November 1990.

[16] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance-1991*, pages 201–208, October 1991.

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

[18] D. Rosenblum and G. Rothermel. A comparative study of regression test-selection techniques. In *Proceedings of the International Workshop for Empirical Studies of Software Maintenance*, pages 89–94, October 1997.

[19] D.S. Rosenblum and E.J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, March 1997.

[20] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques.* Ph.D. dissertation, Clemson University, May 1996.

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

[22] 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.

[23] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineerin*, 24(6):401–419, June 1998.

[24] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proceedings of the 3rd International Conference on Reliability, Quality & Safety of Software-Intensive Systems (ENCRESS '97)*, May 1997.

[25] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53, November 1998.

[26] L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 262–270, November 1992.

[27] L.J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 338–347, September 1993.