TestTube: A System for Selective Regression Testing

Research Paper

Yih-Farn Chen David S. Rosenblum Kiem-Phong Vo

Software Engineering Research Department
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974 USA
{chen,dsr,kpv}@research.att.com

Abstract

This paper describes a system called TESTTUBE *that combines static and dynamic analysis to perform selective retesting of software systems written in C.* TESTTUBE *first identifies which* functions, types, variables *and* macros *are covered by each test unit in a test suite. Each time the system under test is modified,* TESTTUBE *identifies which entities were changed to create the new version. Using the coverage and change information,* TESTTUBE *selects only those test units that cover the changed entities for testing the new version. We have applied* TESTTUBE *to selective retesting of two software systems, an I/O library and a source code analyzer. Additionally, we are adapting* TESTTUBE *for selective retesting of nondeterministic systems, where the main drawback is the unsuitability of dynamic analysis for identification of covered entities. Our experience with* TESTTUBE *has been quite encouraging, with an observed reduction of 50% or more in the number of test cases needed to test typical software changes.*

1 Introduction

As software systems mature, maintenance activities become dominant. Studies have found that more than 50% of development effort in the life cycle of a software system is spent in maintenance; of that, a large percentage is due to testing [20, 19]. Except for the rare event of a major rewrite, in the maintenance phase changes to a system are usually small and are made to correct problems or incrementally enhance functionality. Therefore, techniques for selective software retesting can help to reduce development time.

There is a clear analogy between retesting and recompilation of software. Good examples of tools for selective recompilation are **make** [6] and **nmake** [8]. These tools implement a simple strategy whereby recompilation is carried out only on source files that have changed and on files that depend on the changed files. Similarly, a test suite typically consists of many test units, each of which exercises or *covers* some subset of the entities of the system under test. A test unit must be rerun if and only if any of the program entities it covers has changed. However, unlike system recompilation, where the dependency between a program and its source files is specified in build scripts or *makefiles*, it is not easy to identify the dependency between a test unit and the program entities it covers. Computing such dependency information requires sophisticated analyses of both the source code and the execution behavior of the test units. Fortunately, the requisite technologies for performing such analyses are now becoming available.

This paper describes TESTTUBE, a system for selective retesting that identifies which subset of a test suite must be rerun to test a new version of a system. The basic idea behind TESTTUBE is illustrated in Figure 1. In the figure, the boxes represent subprograms and circles represent variables. The arrows represent static and dynamic dependency relationships (e.g., variable references and function calls) among the entities in the system under test and the test units, with the entity at the tail of an arrow being dependent on the entity at the head. Suppose that the shaded entities were modified to create a new version of the system under test. Under a näive retest-all strategy, all three test units must be rerun in order to test the changes. However, by analyzing the relationships between the test units and the entities they cover, it is possible to eliminate test units 1 and 2 from the regression testing of the

0270-5257/94 $3.00 © 1994 IEEE

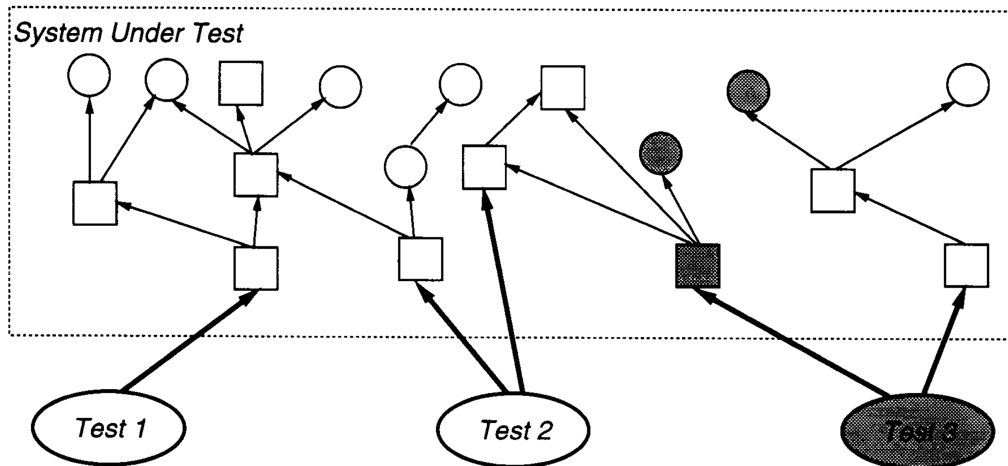

Figure 1: Selective Retesting of a New Version.

new version and rerun only test unit 3.

A number of selective retesting techniques have been previously described in the literature. Many of the early techniques were designed to work in tandem with a particular strategy for generating tests and ensuring test adequacy. Yau and Kishimoto describe a selective retesting technique for partition testing [24]. Ostrand and Weyuker [21] and Harrold et al. [12, 11] each describe selective retesting techniques for data flow testing. In these approaches, the system and its test units are analyzed to determine which test data adequacy criteria each test unit satisfies and which criteria are affected by a modification. While such techniques can be adapted to other kinds of test generation strategies such as mutation testing, such adaptation would require that the methods and tools that support the techniques be customized to the chosen strategy in each instance. Other techniques have been described that use data flow analysis independently of the chosen test generation strategy [7, 2, 13]. All of these data flow-based techniques employ *intra*procedural data flow analysis, which limits their usefulness to unit-level testing. Furthermore, it is conceivable that the computational complexity of data flow analysis could make data flow-based selective retesting more costly than the näive retest-all approach, especially for testing large software systems. Others have tried to avoid the costs of data flow analysis by employing slicing techniques instead [3, 10]. For example, the slicing technique described by Gupta et al. is used in conjunction with data flow testing to identify def-use pairs that are affected by program edits, without requiring the computation and maintenance of data flow histories of the program and its test units [10]. Finally, while TESTTUBE and all of the above techniques attempt to reduce testing costs by testing only the entities that are modified to create new versions, the INFUSE environment attempts to reduce testing costs as testing progresses from unit-level to acceptance-level [15]. In particular, INFUSE manages the integration test cycle and reruns lower-level test cases only if they exercised stub modules that have been replaced by real code.

TESTTUBE differs from these previous approaches in a number of ways. First, TESTTUBE can be used with any chosen test generation and test suite maintenance strategy; note, however, that we ignore the problems of generating adequate tests and maintaining the test suite as new versions are created. Second, the analysis employed in TESTTUBE is performed at a granularity that makes it suitable for both unit-level and system-level testing. Third, the analysis algorithms employed in TESTTUBE are computationally inexpensive and thus scale up for retesting large systems with large numbers of test units. Fourth, the analysis that is performed in TESTTUBE produces byproducts that can be used for purposes other than selective retesting. Fifth, while previous research in selective retesting has ignored the problem of testing nondeterministic systems such as real-time telecom-

munications software, we have begun to adapt TESTTUBE for use in selective retesting of such systems.

We begin the paper in Section 2 with a discussion of the conditions under which TESTTUBE is correct. That is, when these conditions are met, given a test suite and a new version to be tested, testing with only the test units selected by TESTTUBE will provide the maximum amount of information that can be gained from testing with the test suite. We describe our implementation of TESTTUBE for C programs in Section 3. In Section 4 we describe our experience using TESTTUBE on two software systems, one an I/O library and the other a source code analysis program. In Section 5 we discuss how TESTTUBE can be employed to test nondeterministic systems, and we present results for the I/O library under an assumption of nondeterminism. Our results demonstrate that there are tangible benefits in doing selective retesting with our approach. Furthermore, our results show that for some kinds of systems, the expected maintenance actions actually favor a selective retesting strategy. We conclude in Section 6 with a discussion of the time and space overhead of TESTTUBE and a discussion of our plans for future research.

2 The TestTube methodology

2.1 Basic method and terminology

The method underlying TESTTUBE is simple. First we partition a software system into basic code entities. These entities are defined in such a way that they can be easily computed from the source code and monitored during execution. We then monitor the execution of each test unit, analyze its relationship with the system under test, and in this way determine which subset of the code entities it covers. When the system is changed, we identify the set of changed entities and then examine the previously computed set of covered entities for each test unit and check to see if any has changed. If none has changed, the test unit need not be rerun. If a test unit *is* rerun, its set of covered entities must be recomputed. Note that the notion of what constitutes a change in the system is programming language-dependent.

This approach works well if the code entities are defined so that the partitioning of a software system can be done efficiently while still allowing effective reduction in the number of test cases that are selected. At one extreme of testing, the retest-all approach is simply that of considering the entire software system as a single code entity. On the other hand, the data flow approaches mentioned in Section 1 treat each statement as a code entity and thus obtain extremely precise information about which code entities are covered by each test unit. But the large cost of data flow analysis may overwhelm the benefits of test reduction. To strike a balance, we consider a *software system* S as being made up from two sets of entities: F, *functions* and V, *nonfunctions*.

- *Functions*: These are the basic entities that execute program semantics by creating and storing values. We assume that every action of a program must be carried out in some functions. An advantage of using functions as a basic code entity is that there are readily available profiling tools that can monitor program execution and identify the set of covered functions.
- *Nonfunctions*: These are nonexecuting entities in a program such as variables, types and preprocessor macros. Variables define storage areas that functions manipulate. Among other things, types define the storage extent of variables. We assume that every storage location that is potentially manipulated by a function can be statically or dynamically associated with some variable. Typically, these entities cannot be directly monitored during execution without great cost. However, they can be deduced from the source code and function call trace.

Next we define a program in the software system S as a composition of some subsets of F and V. A *test unit* T for the system S is defined as a program and some *fixed* input.[1] Fixing the input means that the set of functions covered by the test unit T can be determined by a single execution. This set of functions is called T_f. The set of non-functional entities that are used by these functions is called T_v.

2.2 Safe test skipping

The working of TESTTUBE relies on a premise that all value creations and manipulations in a program can be inferred from static source code analysis of the relationships among the functional and non-functional entities. This premise is valid for languages without pointer arithmetic and type coercion. In that case, we can summarize TESTTUBE as follows:

Proposition *Let T be a test unit for a software system S. When changes are made to S, if no elements*

[1] We view input as comprising both input data values and environment effects such as signals.

in T_f and T_v are changed, then T does not need to be rerun on the new version of S.

However, with languages such as C and C++, it is not always simple to infer all value manipulations just from analyzing the variables and pointers used by functions. For example, the following C code uses type coercion to convert an integer value to an address value so that the creation of the value 0 in the memory store is not associated with any visible variable.

```
*((char*)0x1234) = 0;
```

Another problem with languages like C and C++ that allow arbitrary pointer arithmetic is that pointers may violate the memory extents of areas to which they point. This means that values may be manipulated in ways that are not amenable to source code analysis. In the example below, the pointer expression `*(xp+1)` points beyond the memory extent defined by the variable `x` (whose address has been stored in the variable `xp`). On many hardware/compiler architectures, it may, in fact, point to the same memory area defined by y. So the value of y is changed without ever referring to y.

```
int x, y;
int* xp = &x;
...
*(xp+1) = 0;
```

To account for such memory violations, we assume the following hypotheses:

Hypothesis 1 (Well-defined memory)
Each memory segment accessed by S is identifiable by a symbolically defined variable.

Hypothesis 2 (Well-bounded pointer) *Each pointer variable or pointer expression must refer to some base variable and be bounded by the extent of the memory segment defined by that variable.*

For applications written in languages such as C and C++, the well-defined memory assumption is reasonable. This is because it is seldom the case that one needs esoteric constructs that would coerce a plain integer value to an address value, except for programs that require direct manipulation of hardware addresses such as device drivers. However, in cases where they are required, the addresses represented by such integer values are usually well separated from the memory space occupied by normal variables. Thus, we do not need to worry about values of variables that are changed without ever mentioning the variable names. In addition, for maintainability, such values are often assigned symbolic names using macros, which *are* amenable to source code analysis.

On the other hand, the well-bounded pointer hypothesis sometimes fails in real C and C++ programs due to memory-overwrite and stray-pointer faults. These faults are among the hardest to detect, isolate and remove. A number of research techniques and commercial tools are available to help detect these faults (e.g., see Austin et al. [1] and Purify [14]). Whenever such a fault is detected during testing, an attempt must be made to identify the entities that are affected by the fault; for example, memory overwrites are often confined to the functions that cause them. If the affected entities can be identified, then these entities must be added to the set of changed entities identified by TESTTUBE for the current round of testing. In extreme cases where the effects of such faults are too difficult to determine, it must be assumed that all parts of memory are potentially damaged, and hence all test units must be rerun in order to ensure thorough testing of any code that exercises the fault. The successful use of TESTTUBE depends on the quick removal of such faults so that they do not propagate from version to version. Once a software system has successfully grown into a maintenance phase, hopefully such faults will be few.

3 An implementation of TestTube for C programs

We have designed and implemented a version of TESTTUBE for C around a number of existing analysis tools. The collection of tools can be partitioned into three categories:

- *Instrumentation Tools*: The system source code is automatically instrumented by **app**, the Annotation Preprocessor for C [22]. The instrumentation causes each run of a test unit T to produce a *function trace list*, i.e. the set T_f of all functions covered by T as defined in Section 2.

- *Program Database Tools*: For each version of the system under test, a C program database is built using the C Information Abstractor, CIA [4, 5]. This database contains information about the C entities that the system comprises and the dependency relationships among them. Then, for each test unit, the TESTTUBE tool **ttclosure** uses the the program database to expand the function trace list T_f to an *entity trace list*, i.e. the set $T_f \cup T_v$ as defined in Section 2. When there are two versions of the source code, the TESTTUBE tool **ttdiff** uses the CIA tool **ciadiff** to analyze the

two corresponding program databases and produces an *entity difference list.*

- *Test Selection Tools*: Three tools are provided to assist selective retesting: **ttselect, ttidentify** and **ttcoverage**. The tool **ttselect** checks to see if there is an intersection between the entity difference list and the entity trace list of each test unit; test units with nonempty intersections must be rerun. To estimate retesting cost, the tool **ttidentify** computes the list of test units that need to be rerun if certain specified entities are changed. Finally, **ttcoverage** finds entities that are not covered by the existing test suite.

The C entities recognized by CIA are functions, variables (including pointers and their base variables), preprocessor macros, types and files; however, only global entities that can be used across entity declaration and definition boundaries are recorded. A C entity is considered to be changed if any token in the sequence of tokens that constitute the entity has changed. Thus, the CIA database provides the right code-entity partition required by the TESTTUBE methodology.

The rest of this section illustrates the use of the TESTTUBE tools on the test suite of **incl**, a program for detecting and removing unnecessary `#include` directives from C programs [23]; such directives are used to incorporate interface or "header" files into C programs. There are eight test units for **incl**. We first instrumented version 1 of **incl** with **app** and built a CIA database for it. We then generated the entity trace list for each test unit. Such lists are stored in files whose names end with `.clo`. Note that this initialization step is necessary only for the first version of **incl**. For later versions, only the entity trace lists of test units that are rerun need to be updated.

We then built a CIA program database for version 2 of **incl**. The example below shows that **ttdiff** was run to compare the two databases (contained in directories v1 and v2, respectively) and store the entity difference list in `tt.dfl`. Then, `tt.dfl` was printed out by **cat** to show that four program entities were changed from version 1 to version 2:

```
$ ttdiff -o v1 -n v2 > tt.dfl
$ cat tt.dfl
function;incl.c;dagprint
function;incl.c;dbeprint
function;incl.c;qexprint
macro;incl.h;N_LEVPRINT
```

We next ran **ttselect** to check for intersections between `tt.dfl` and the `.clo` files. The output of **ttselect** showed that four out of the eight existing test units for **incl** had to be rerun and that their `.clo` files had to be regenerated:

```
$ ttselect tt.dfl *.clo
rerun test t.3
rerun test t.5
rerun test t.6
rerun test t.8
```

While it is possible to use **ttselect** to predict the testing cost implied by a set of changes by manually constructing a hypothetical `tt.dfl` file, the tool **ttidentify** simplifies this process. Given a CIA entity pattern (which specifies an entity kind and a regular expression to match names of entities of that kind), **ttidentify** identifies which test units must be rerun if any matching program entities are changed. For example, the query below finds the list of all functions defined in `incl.c` whose names end with `print` and then lists the test units that cover them; the output of the query shows that changes to different functions incur different testing costs:

```
$ ttidentify 'function *print \
    file=incl.c' *.clo
incl.c dagprint:
        rerun test t.3
        rerun test t.6
incl.c dbeprint:
        rerun test t.8
incl.c dbvprint:
        rerun test t.8
incl.c exprint:
        rerun test t.1
incl.c levprint:
        rerun test t.2
incl.c qexprint:
        rerun test t.5
incl.c stprint:
        rerun test t.1
        rerun test t.2
        rerun test t.3
        rerun test t.6
incl.c subsysprint:
        rerun test t.3
        rerun test t.6
```

The entity trace lists provided by TESTTUBE can also be used as an aid in evaluating the adequacy of a test suite. Given a CIA entity pattern, **ttcoverage** finds entities matching the pattern that are not present in any entity trace list. For example, the following command finds all functions in **incl** that are not covered by any of the eight test units:[2]

[2] The specifier "-" in a CIA entity pattern is a wild card that matches all entity names.

```
$ ttcoverage 'function -' *.clo
NOT COVERED:
    incl.c subsys
    tree.c t_delete
    tree.c t_free
    util.c delNode
    util.c delSymbol
    util.c fatal
```

As seen below, the CIA tool **deadobj** finds that only `t_delete` is truly not used by any **incl** code, and therefore more test units should be added to the test suite:

```
$ deadobj function
tree.c function t_delete
```

This completes our illustration of the TESTTUBE tools for C code. The tools do not require redesigning existing test suites or manually modifying code. They are driven by data readily obtainable from program analysis tools such as **app** and CIA. Furthermore, even the dependency on such program analysis tools can be removed. For example, if a project already uses a tracing tool that can provide function trace lists from program executions, then we do not need to instrument code with **app**. With minimal textual transformation, the output from such a tool can be used by TESTTUBE. The same is true of CIA. Finally, outputs generated by TESTTUBE and the analysis tools find applications beyond selective retesting. In particular, CIA databases have been used by many software projects to study program structure, eliminate dead program entities, and skip unnecessary header files during compilation.

4 Experience

In this section we describe our experience in applying TESTTUBE in system-level regression testing of two software systems.

4.1 The library SFIO

Our first use of TESTTUBE was on a programming library called SFIO (Safe, Fast Input/Output), a replacement for the standard UNIX I/O library STDIO. SFIO contains a number of facilities for creating and manipulating I/O streams, including stacks of I/O streams and user-defined I/O disciplines [17].

The version of SFIO we studied contains roughly 11,000 lines of C code and exports 67 functions for application programs. Our analysis revealed that the library source code contains 481 preprocessor macro definitions, 175 function definitions, 18 definitions of global variables and 32 type definitions, for a total of 706 entities defined in the library. The library contains 3424 references of various kinds (function-to-function, variable-to-type, etc.) between pairs of these entities. A suite of 39 test units is used to test SFIO. Each test unit is a C program with a function `main` that makes sequences of calls to the library and covers a subset of entities in the library. For instance, analysis of the test `tdisc.c`, which is used to test I/O disciplines in SFIO, reveals coverage of 20 functions, 7 variables, 23 types and 102 macros in SFIO. To test SFIO, each test program is compiled, linked with the library and executed, and the execution results are then validated by the test units themselves.

The goal of our work on SFIO was to flesh out the details of the selective retesting method that TESTTUBE supports and to ensure that the TESTTUBE tools provide the required functionality. We instrumented the library and the test programs with **app** and then executed them to capture their function trace lists. In addition, we built a CIA database for the library to generate the entity trace lists. Since we did not have a second version of SFIO on which we could perform selective retesting, we used the tool **ttidentify** to simulate a minor maintenance of SFIO. This was done by determining for each function defined in SFIO, how many of the 39 test programs would have to be rerun if only that function were changed.

Table 1 presents a sample of our results for SFIO. The table presents data for 18 of the 67 functions exported by the library. As expected, the amount of retesting required after a single modification depends heavily on the function that was modified. However, what is most interesting about Table 1 is that the functions that require the most retesting are the functions that are least likely to be changed during maintenance. In particular, we observe that the exported functions in SFIO fall roughly into two groups, *core* functions and *feature* functions. The core functions provide the basic functionality of the library and are used by nearly all application programs, while the feature functions are more specialized and used less often. For example, `sfnew` is used in virtually all SFIO applications to create and initialize streams, but `sfstack` is used only in applications that stack them. Since core functions like `sfnew` are used heavily, their design and implementation stabilize early in development. On the other hand, feature functions like `sfstack` are often changed to adapt to new requirements, and their functionality may not always be thoroughly tested. Thus, much of the maintenance work on the library would tend to involve feature functions. As Table 1

Function Changed	Number of Tests	Function Changed	Number of Tests	Function Changed	Number of Tests
`sfsscanf`	1	`sfprints`	4	`sfreserve`	20
`delpool`	1	`sfsprintf`	4	`sfopen`	25
`sfpoll`	1	`sfstack`	5	`sfseek`	27
`sfpeek`	1	`sfpopen`	6	`sfwrite`	27
`newpool`	2	`sfdisc`	8	`sfclose`	28
`sfsize`	3	`sfread`	9	`sfnew`	35

Table 1: Retesting Required After Changing a Single Function in SFIO (out of 39 test units).

shows, such maintenance should require relatively little retesting and thus should be favorable to a selective retesting strategy. Further experience is needed to validate these hypotheses.

4.2 The tool Incl

Our second use of TESTTUBE involved the tool **incl** described in Section 3. We studied two versions of **incl**, each containing roughly 1700 lines of code and providing 8 command-line options, which are used to govern the output format and the extent of the search that is performed. Our analysis revealed that the source code for version 1 of **incl** contains 37 preprocessor macro definitions, 32 function definitions, 14 definitions of global variables and 8 type definitions, for a total of 91 entities defined in the tool. The tool contains 377 references of various kinds between pairs of these entities. A suite of 8 test units is used to test **incl**. Each test unit is a UNIX shell script that invokes **incl** with a number of command-line options. To test **incl**, each shell script is executed, and the execution results are then compared automatically against previously stored copies of expected results.

The goal of our work on **incl** was to determine how much reduction could be achieved in regression testing of version 2 of **incl**, which was created from version 1 to modify some of the output formats. As was illustrated in Section 3, creation of version 2 involved changing 3 function definitions and 1 macro definition; there were no additions or deletions of entities. We ran the tool **ttselect** and found that of the 8 shell scripts used to test **incl**, only 4 had to be rerun to test version 2, producing a 50% reduction in the number of test cases needed to test the new version.

As was also illustrated in Section 3, we ran the tools **ttidentify** and **ttcoverage** to gather statistics on the effects of minor maintenance of **incl** and on the entities in the **incl** source code that were not covered by any of the test scripts. The results from **ttidentify** confirmed the phenomena we observed in SFIO with regard to core functions and feature functions. Core functions such as `define` (which identifies header files that contain variable and function definitions) are exercised by nearly all of the tests, whereas feature functions such as `dagprint` (which is called only in the presence of the command-line option requesting output for the tool DAG [9]) are exercised by very few of the tests. Furthermore, **ttcoverage** revealed 5 key functions that were not covered by any test script; additional tests will be constructed to test these functions.

5 Retesting nondeterministic systems

A major issue that has not been addressed by the other work mentioned in Section 1 is how nondeterminism in a software system affects selective regression testing of the system. While it is certainly desirable to design test units to have predictable behaviors and outputs in order to control the testing process, it is very difficult to entirely eliminate the nondeterminism that results from occurrences of exceptions and signals, reading input from files or terminals, checking the values of shell environment variables, and so on. Furthermore, many hard real-time applications such as telecommunications switching software can produce a variety of subtle behavior differences between different runs of the same test unit. What this means for techniques that rely on runtime tracing of execution paths or function call activity is that analyzing a single trace of a test unit run may not be sufficient to identify all paths or entities that are *potentially* covered by the test unit. And the code instrumentation that provides such tracing can produce unacceptable interference with the required timing behavior of such systems. We have begun to study a number of techniques for overcoming these deficiencies in TESTTUBE. These techniques can be partitioned into two cases.

The simpler case involves unit testing with driver programs or system testing of libraries such as SFIO, where each test unit is a C program containing a func-

tion `main` that makes calls to the system under test. In this case, instead of generating the function trace list for the test unit via dynamic tracing, it is necessary to statically compute the transitive closure of references to all entities that are reachable from `main` in order to identify all functions that are potentially callable during a run of the test unit.

Since the library SFIO fits this simpler case, we studied it under an assumption of nondeterminism. We repeated the analysis described in Section 4.1, but with the potentially callable functions identified through a transitive closure search of all reference relationships starting from the function `main` of each test unit. This transitive closure search was performed using the CIA tool **subsys**, which identified the reachable variable, type and macro definitions in addition to the potentially callable functions. Table 2 presents the results for the same functions that were presented in Table 1. The numbers changed little from those presented in Table 1 for the functions we would identify as feature functions. However, the numbers increased dramatically for the functions we would identify as core functions. Thus, we interpret Table 2 as another confirmation of our hypothesis regarding core and feature functions, and we observe that modification of core functions in a nondeterministic system will often require full retesting of the system on all test units.

The more difficult case of testing nondeterministic systems is when the system under test contains the function `main`; performing the same transitive closure search that we perform in the simpler case would of course find every function in the system (except for dead code entities). However, the simpler case described above suggests some ways to handle this. For instance, for each test unit, a person knowledgeable about the system could identify a number of appropriate non-`main` "root functions" in the system under test from which to begin the transitive closure search. But since the whole point of selective retesting systems such as TestTube is to remove such ad hoc analysis from regression testing, this approach is undesirable. Because many systems use `main` as a deterministic driver for other root functions, the following may be a better approach: For each test unit, generate a trace of the function call activity of a single run of the test unit, and then perform the transitive closure search from each non-`main` function appearing in the trace. This approach at least automates the analysis, but it will still be inadequate for systems with certain kinds of nondeterministic features such as signal handling. For such systems it remains to be seen to what extent test cases can be safely eliminated.

6 Conclusion

We have described a system called TestTube that implements a selective regression testing method through dynamic and static analysis of a software system and its test units. Our experiences with SFIO and **incl** verify that there are tangible benefits to be derived from using a selective retesting technique. Furthermore, the data presented in Sections 4 and 5 suggest that the expected maintenance actions on some kinds of systems actually favor a selective retesting strategy. In particular, when the maintenance actions are typically perfection and enhancement of specialized feature functions, the resulting modifications can require relatively small amounts of retesting.

The choice of analysis methods used in any selective retesting strategy is governed by a spectrum of tradeoffs between the desired detail and accuracy of the analysis and the time/space costs required to perform the analysis. For instance, data flow analysis can provide information about a system at the granularity of a source code statement, but its relatively poor time complexity may make it prohibitive for analysis of large systems. In comparison with previous selective retesting techniques, TestTube employs relatively coarse-grained analysis of the system under test, producing a reasonable and practical tradeoff between granularity of analysis and time/space complexity.

Each phase of TestTube—instrumentation, program database construction, and test selection—contributes some amount of overhead to selective retesting. Instrumentation with **app** for dynamic tracing typically increases object code size by about 19%, although compilation and linking with uninstrumented system libraries usually reduces this space cost to around 2% of the fully-linked executable. Furthermore, the runtime cost of generating the trace is insignificant, although this may not always be the case if low-bandwidth I/O devices are used to collect the trace (such as in an embedded real-time system). Construction of a CIA database requires about 1.5 times as much disk space as the original source code and about 70% of the time needed to compile the source code. Note that projects that already use **app** for assertion processing and CIA for source code analysis are already incurring these costs. The rest of the overhead of TestTube comes from the test selection tools **ciadiff**, **subsys**, **ttselect**, **ttidentify** and **ttcoverage**. **Ciadiff** produces output that is linear in the size of the number of changed components. The test selection tools currently use näive algorithms for their computations, and thus we have not yet determined the minimum expected costs for these tools.

Function Changed	Number of Tests	Function Changed	Number of Tests	Function Changed	Number of Tests
`sfsscanf`	1	`sfprints`	4	`sfreserve`	39
`delpool`	2	`sfsprintf`	4	`sfopen`	39
`sfpoll`	1	`sfstack`	5	`sfseek`	39
`sfpeek`	1	`sfpopen`	6	`sfwrite`	39
`newpool`	2	`sfdisc`	8	`sfclose`	39
`sfsize`	3	`sfread`	39	`sfnew`	39

Table 2: Retesting Required After Changing a Single Function in SFIO (out of 39 test units)—Nondeterministic Case.

Phase	Space Overhead	Time Overhead
Instrumentation	19% of object code	≈ 0
Program Database	150% of source code	70% of compile time
Test Selection	O(no. of test units)	O(no. of test units $\times$ no. of changed entities)

Table 3: Overhead of using TestTube for Selective Regression Testing.

Table 3 presents a summary of the (current) overhead of using TESTTUBE for selective regression testing. Note that while it is easy to bypass the tracing code at minimal runtime cost in a "field-grade" release of a system, projects that choose not to release instrumented code to customers must incur an additional complete recompilation. In the future we plan to evaluate TESTTUBE using the cost model developed by Leung and White [18].

We are currently exploring a number of related research issues. First is the integration of TESTTUBE with other testing tools. Our experience with projects in AT&T is that each project's testing organization uses a unique set of test generation methods, test planning and management methods, and testing tools. Generalizing the interface between TESTTUBE and such a wide array of testing environments presents a significant challenge in making TESTTUBE usable by mainstream testing organizations.

We also intend to study ways of applying TESTTUBE's analysis methods to reduce the cost of testing, improve the adequacy of a given test suite, and enhance source code quality. For instance, the tool **ttcoverage** can be used to quickly identify source code entities that are not covered by a test suite so that new tests can be designed. In a different direction, for a mature system, if a large amount of retesting results from small code changes, then the code may not be well modularized. The information supplied by **ttselect** and **ttidentify** can be used to help identify ways in which the architecture and modularity of the code that could be improved.

In current practice, testing typically is carried out late in a development cycle by testers who are separated from code developers. One reason is that the prohibitively large cost of running an entire test suite whenever the code is changed precludes more frequent testing. An equally onerous reason is the lack of code sharing, which inhibits individual developers from having a complete view of the system. Test minimization technology such as TESTTUBE, in combination with code sharing technology such as the 3D File System [16], should make it possible to shift more of the burden of testing to software developers who will be able to test their changes against complete copies of the system under test with minimal cost. Toward this goal, we need to design software development environments and processes so that both code and tests can be shared easily.

Finally, we have two studies underway with large software projects in AT&T that are helping us to further evaluate the effectiveness of TESTTUBE and to gain experience with other kinds of testing problems, such as regression testing of real-time switching systems with manually-run call-processing scenarios.

Acknowledgments

The development of TESTTUBE was aided immensely by various discussions with Kent Clapp, Brian Enke, Rick Greer, Randy Hackbarth, Dave Korn, Todd Livesey, Steve Opferman, Merle Poller, Filip Vokolos, Elaine Weyuker, and Ray Zuniga. Dave Belanger, Kent Clapp, Brian Enke, Emden Gansner, and

Elaine Weyuker gave us many helpful comments on the manuscript.

REFERENCES

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. Technical Report TR 1197, Computer Sciences Department, University of Wisconsin-Madison, December 1993.

[2] 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. IEEE Computer Society, October 1988.

[3] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance 1992*, pages 41–50. IEEE Computer Society, November 1992.

[4] Yih-Farn Chen. The C program database and its applications. In *Proceedings of the Summer 1989* USENIX *Conference*, pages 157–171. USENIX Association, June 1989.

[5] Yih-Farn Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, SE-16(3):325–334, March 1990.

[6] Stuart I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9(3):255–265, March 1979.

[7] Kurt Fischer, Farzad Raji, and Andrew Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference*, volume 1, pages B6.3.1–B6.3.6. IEEE, November–December 1981.

[8] G. S. Fowler. The fourth generation make. In *Proceedings of the USENIX 1985 Summer Conference*, pages 159–174, June 1985.

[9] E. R. Gansner, S. C. North, and K. P. Vo. DAG—A program that draws directed graphs. *Software—Practice and Experience*, 18(11):1047–1062, November 1988.

[10] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance 1992*, pages 299–308. IEEE Computer Society, November 1992.

[11] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[12] Mary Jean Harrold and Mary Lou Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance 1988*, pages 362–367. IEEE Computer Society, October 1988.

[13] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, January 1990.

[14] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992* USENIX *Conference*, pages 125–136. USENIX Association, January 1992.

[15] Gail E. Kaiser, Dewayne E. Perry, and William M. Schell. Infuse: Fusing integration test management with change management. In *Proceedings of the 13th International Computer Software and Applications Conference (COMPSAC)*, pages 552–558. IEEE Computer Society, September 1989.

[16] David G. Korn and Eduardo Krell. A new dimension for the UNIX file system. *Software—Practice and Experience*, 20(S1):19–34, June 1990.

[17] David G. Korn and K.-Phong Vo. SFIO: Safe/fast string/file IO. In *Proceedings of the Summer 1991* USENIX *Conference*, pages 235–256. USENIX Association, June 1991.

[18] Hareton K. N. Leung and Lee White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance 1991*, pages 201–208. IEEE Computer Society, October 1991.

[19] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[20] Glenford J. Myers. *The Art of Software Testing*. Wiley-Interscience, 1979.

[21] Thomas J. Ostrand and Elaine J. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, September 1988.

[22] David S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104. Association for Computing Machinery, May 1992.

[23] Kiem-Phong Vo and Yih-Farn Chen. Incl: A tool to analyze include files. In *Proceedings of the Summer 1992* USENIX *Conference*, pages 199–208. USENIX Association, June 1992.

[24] Stephen S. Yau and Zenichi Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Proceedings of the 11th International Computer Software and Applications Conference (COMPSAC)*, pages 272–277. IEEE Computer Society, October 1987.