Using Fault Injection to Increase Software Test Coverage*

James M. Bieman Daniel Dreilinger Lijun Lin
Computer Science Department
Colorado State University
Fort Collins, Colorado 80523
bieman@cs.colostate.edu

To appear in *Proc. International Symposium on Software Reliability (ISSRE'96)*

Abstract

During testing, it is nearly impossible to run all statments or branches of a program. It is especially difficult to test the code used to respond to exceptional conditions. This untested code, often the error recovery code, will tend to be an error prone part of a system. We show that test coverage can be increased through an "assertion violation" technique for injecting software faults during execution. Using our prototype tool, Visual C-Patrol (VCP), we were able to substantially increase test branch coverage in four software systems studied.

Keywords: software testing, software test coverage, fault injection, error recovery, software reliability, Visual C-Patrol.

1. Introduction

Developing reliable and fault tolerant software is difficult and requires discipline both in specifying system functionality and in implementing systems correctly. Approaches for developing highly reliable software include the use of formal methods [11, 14, 9], and rigorous testing methods [2, 7, 13]. Testing cannot guarantee that software is correct [17], and verification requires enormous human effort and is subject to errors [6]. Automated support is necessary to help ensure software correctness and fault tolerance.

Fault injection has been proposed for use in *mutation testing* primarily as a mechanism for evaluating the adequacy of test data [3, 8]. Mutation testing injects faults by modifying program text. As a result, the testing process can generate enormous numbers of mutant programs, and each program must be recompiled and then tested. This process can be too labor and time intensive for practical use in the general testing of commercial software.

Our hypothesis is that fault injection can be effective when it is directed towards solving specific testing problems. In particular, we use fault injection to force the execution of difficult to reach program paths. To avoid the need for recompilation, we mutate program state rather than program text.

To be practical, any fault injection mechanism must be inexpensive and be able to model a wide variety of fault types. Much of the fault injection should be automated. Although a practical approach requires a limited fault model, we want to simulate as many faults as possible.

2. Fault Injection via "Assertion Violation"

Pre- and post-conditions contain logical assertions or invariants that specify the expected behavior of a program. A pre-condition is an assertion about the nature of the system state that must be true before a function is invoked to be sure that the function will run correctly. A post-condition is an assertion that describes the relationship between the input state and output state of a correct function. These conditions are usually not explicitly expressed in the program text, but rather implicitly assumed.

We can create a fault by dynamically changing the state of a running program so that a pre- or post-condition is not satisfied. Such an artificial fault can be used to model initialization faults, assignment faults, condition check faults, and even function faults.

The assertion violation scheme makes use of pre- and post-condition assertions. The first step is to define and state these using a new programming language construct. The second step is to automatically make them false in different ways and at different program locations while the program is executing. To create a fault, one of these invariants is violated or made false in some fashion. We can automatically

*Research partially supported by the Colorado Advanced Software Institute (CASI) and Storage Technology Inc. CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the state of Colorado. CATI promotes advanced technology teaching and research at universities in Colorado for the purpose of economic development.

generate a number of possible violations of a given assertion using language processing tools, such as *lex* and *yacc*.

The pre- and post-conditions provided by the tester or programmer are first order logic predicates about the function's input and output parameters, and global variables. They use a syntax similar to that used in the source language.

Example. Suppose the pre-condition for some function f(float x, y) is $\{x > 0 \wedge x \neq y\}$. We can derive several possible faults to inject. The injection is a textual insertion at the beginning of the function. For the example, the injection code might be the following (using C syntax):

```
switch (injection_status){
  case 0:  break;
  case 1:  x=0; break;
  case 2:  x=-1; break;
  case 3:  x=y; break;
  case 4:  y=x; break; }
```

The above block of code can be automatically generated by a code pre-processing tool. When injection is off, the injection_status variable is set to zero. When injection is on, the injection_status variable is set randomly to a value between 1 and 4. The number of injections and the probability of choosing a particular injection can be determined by the user. Typically, an injection will cause a short chain reaction of violated assertions. The invalid assertions should be restored by error recovery code in fault tolerant software.

The code mutation aspect of this scheme can be performed by a pre-processor, which transforms pre- and post-conditions into case injection statements and inserts them into a program.

An injector tool could accept statements from the propositional calculus. In our prototype system, we implemented a subset which specifically includes statements using the alphabet:

$$\{identifier, <, <=, ==, != , >=, >, \\ integer_literal, float_literal, \wedge\}.$$

Valid strings in the language are conjunctions of boolean terms (equalities or inequalities). Ideally, the language will be augmented with $\{\neg, \vee, \forall, \exists\}$. An example of a recognizable precondition in the system is:

$$(a < 3) \wedge (b! = c) \wedge (d > 1.5).$$

Many possible injections can be derived from this statement. In our prototype system, we inject violations that are near the boundary to simulate some common programming errors. Thus, $a < 3$ is mapped to $a = 3$, $b! = c$ gets mapped to two possible faults: $b = c$ and $c = b$, and finally, $d > 1.5$ is mapped to $d = 1.5$.

The timing impact of this scheme is quite small because the injected values can be computed statically. An optimizing C-compiler can compute the values to be injected, even if the source output of the injector is $y = 4 + 0.01$.

We can check the correctness of a system's behavior before and after injecting faults. A correct pre- or post-condition can be injected as code and checked at run time using a facility like C *assert* statements or C-Patrol *insertion directives*.

3. Implementing Assertion Violation Fault Injection

To demonstrate the proposed fault injection method, we extended the C-Patrol assertion insertion system [18] to support fault injection and built a visual X Window System interface.

3.1. C-Patrol

C-Patrol is a code insertion tool that can assist developers in the placement of software probes that are used in testing. Such probes may be implementations of data invariants, pre- and post-conditions, or other run time checks. An overall view of the C-Patrol system is shown in Figure 1.

C-Patrol allows developers to define and place assertions in *logical* locations in a C program. For example, a developer may prefer to place a data invariant along with an associated data declaration. A preprocessor under program control inserts these assertions into the correct locations for run time monitoring of a program. The data invariant is thus checked at the locations where the data structure is actually used, rather than where it is defined. Using C-Patrol, *virtual code* is defined within comments, and is inserted into regular code using possibly parameterized *directives*. A special labeling system matches directives to virtual code. When the probes are not needed, they remain with the program as comments that do not affect program execution.

Our initial intention was for C-Patrol software probes to be used to check the correctness of program behavior. However, probes can also be used to modify the system state and thus create any desired or undesired state. Probes can generate states that would result from software and hardware faults. Thus, C-Patrol can be used to inject faults at chosen points in a program.

C-Patrol is a pre-processor that inserts assertions, written as comments in *virtual C*, into specified locations in C programs. The user places virtual C code within special C-Patrol comments that are delimited by "`/*?`" and "`?*/`". These comments are skipped by the C compiler and do not affect the performance of the underlying system. Once the user activates the comments by running the pre-processor, the pre-processor translates virtual code into regular C and

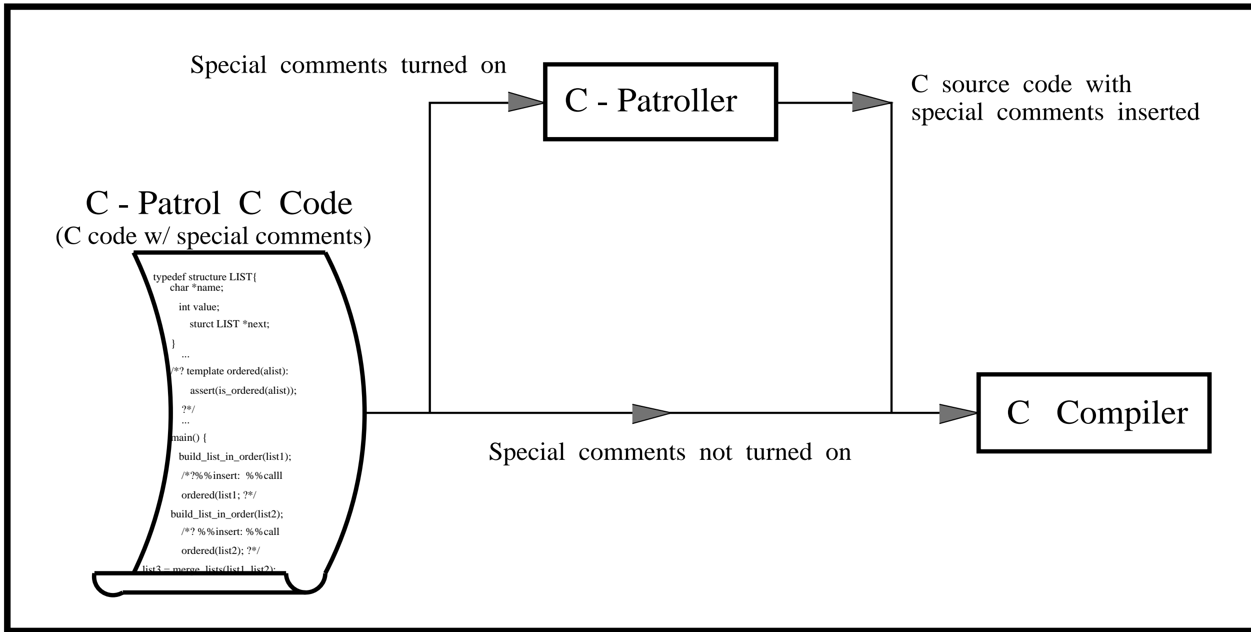

Figure 1. The C-Patrol System

inserts it based on a set of instructions called *directives*. The pre-processed program may then be compiled and run as normal C. Thus, a program with C-Patrol annotations can run with or without the inserted assertions as shown in Figure 1.

The C-Patrol annotation language includes insertion directives, call directives, label directives, template directives and bind directives. Insertion directives include `%%insert`, `%%pre` and `%%post`. The format for C-Patrol pre- and post-conditions is

```
/*?
    %%pre:  <C Code and Call Statements>
    %%post: <C Code and Call Statements>
?*/
```

The variable declarations in the block of code led by the `%%pre` directive have scopes that extend up to the function exit. Therefore, input arguments can be captured by assignments to temporary variables, which can be used to make comparison at function exit. A post-condition block is inserted before each exit and return statement. The post-condition block is also inserted before the function exit. Thus all function exits are guarded by the post-condition.

3.2. Visual C-Patrol (VCP)

To demonstrate the proposed fault injection method, we extended the C-Patrol system to support three major tasks:

1. Editing programs to add C-Patrol style assertions.
2. A run time visual interface to help monitor the execution of instrumented programs.
3. Fault injection based on assertion violation.

A user interface, shown in Figure 2, supports editing programs with C-Patrol style assertions. All function names in a file are displayed in a special window, along with their instrumentation status. The user can easily skip around and add, view, or change assertions. There is a menu, buttons for commonly used features, and an edit area in the interface. A function area on the left side of the window displays the names and instrumentation status of all functions of the current file.

To add pre- and/or post-conditions to functions of interest, a user first clicks on the displayed name of the desired function. The indicated function is displayed in an editing window. To add a pre- and post-condition pair, the user places the cursor right above the function body and below the function declaration and presses the */*? %%pre...* button. A template for the pre- and post-condition is inserted into the program text at the appropriate place, and the user then inputs the pre- and post-condition body. The function name in the function window will now show pre- and post-condition indicators.

To inject faults into a function, the user presses the pre-indicator of the function in the function window and then presses the *Instrument* button. The program must then be recompiled by pressing a button. The recompiled program contains the code to generate and inject faults. Now when the program is executed, the run time interface is display on

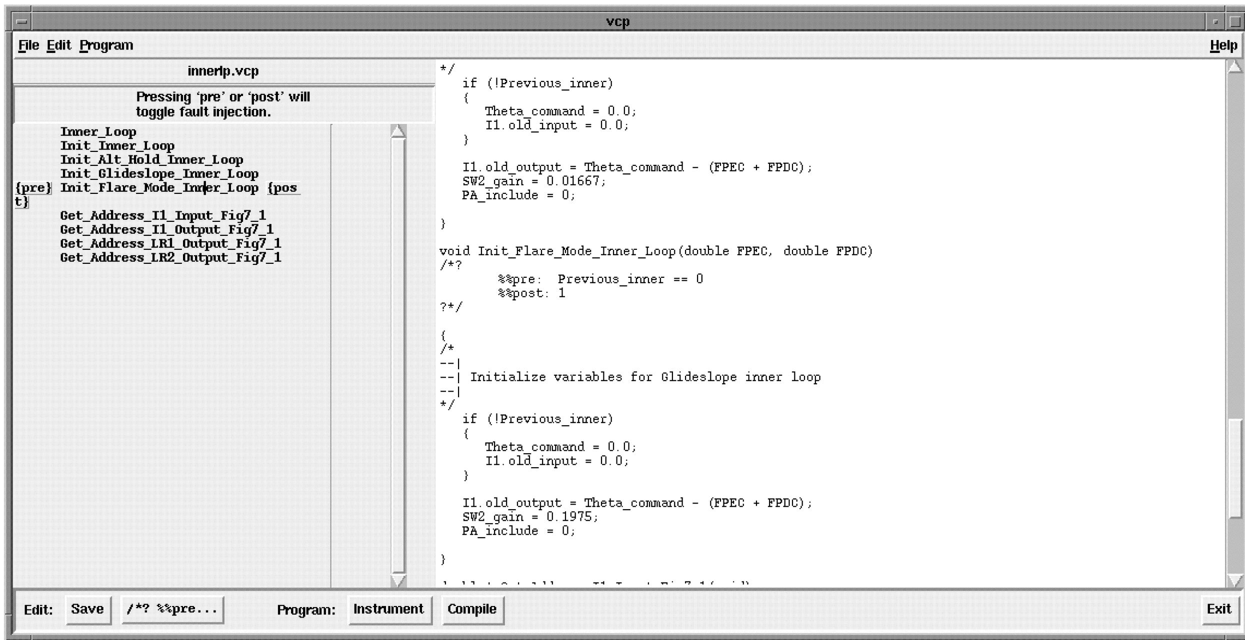

Figure 2. VCP user interface

the screen as shown in Figure 3.

The run time visual interface is automatically built for an instrumented program. Each instrumented function appears in the run time monitor interface with pass/fail indicators for each assertion. These show the programmer both the assertion pass/fail status and program flow of control. A pre- or post-condition indicator is green when the condition successfully executes and turns red when it fails. Execution can be slowed via a speed control bar and a step button.

Injecting a fault for a specific condition can be performed by using a step button to stop before entering the desired function. Then the user presses the *inject* toggle button and select a fault in the bottom window, and presses step. The fault will be injected. After fault injection, the pre-condition indicator turns purple. The tester can then check whether the results are as expected.

The third major VCP feature implements the assertion violation mechanism for fault injection. The state of an executing program is dynamically modified to an incorrect one with the goal of forcing recovery code to execute. The VCP editor allows the user to select specific assertions to instrument for fault injection, while the run time monitor allows the user to instantiate the injections and subsequently observe their effect on control flow and assertion status. A precondition indicator turns purple when a fault is injected.

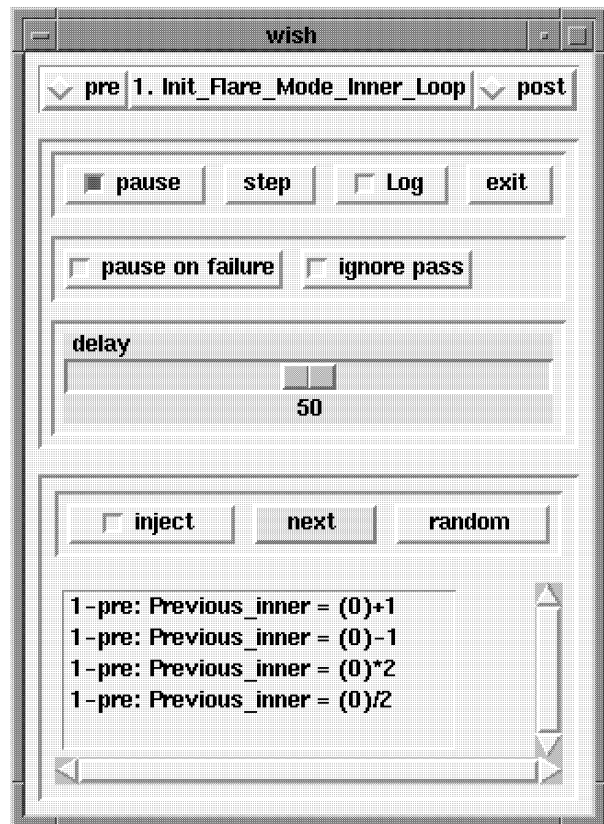

Figure 3. Run time interface of VCP

4. Effectiveness

We evaluated the assertion violation fault injection technique as implemented in Visual C-Patrol (VCP) by using it to test four systems or partial systems. We measured the test coverage—the ratio of the number of branches that were covered by a set of test cases to the total number of branches in a program—before and while using VCP.

4.1. Evaluation Data

We examined the application of VCP to all or part of four systems:

1. AUTOLAND, a fault tolerant software system developed at AT&T Bell laboratory. AUTOLAND consists of 3524 lines of C code in 12 C source files and 12 header files. We obtained a regression test set consisting of 5280 test cases.

2. C-Patrol, our software testing tool that we used to develop VCP. We evaluated 9 of the 13 C-Patrol source files consisting of 5834 lines of code and 677 branches. We used a regression test set consisting of 93 test cases.

3. AC, a compiler for the language A, a subset of C used in a compiler course. AC compiles source files written in A into the MIPS processor assembly code. We evaluated 7 of the 27 source files using 19 test cases.

4. Minix, a UNIX-like operating system used in education and research. We evaluated 5 Minix commands in early tests of VCP.

4.2. Evaluation Process

First, we selected a sample of individual functions and evaluated fault injection for these small code components. Then, we evaluated fault injection applied to source files which consist of many functions. For one system, Autoland, we evaluated fault injection applied to all files in the system.

Each evaluation was conducted in two steps:

1. We tested the software without using VCP and collected the test coverage, measured in terms of the number of control flow branches reached by the test data. We used the Generic Coverage Tool (GCT), a public domain testing tool, to collect test coverage information.

2. We used VCP to insert C-Patrol pre-conditions into the software, generate faults, and inject these faults into the program when it is tested. The injected faults change the run time state of the program and force new branches to be covered. We used GCT to collect the test coverage obtained with fault injection.

Table 1. Fault Injection Applied to Selected C Functions

Fun. Name	Lines	Branches	Test Cases	Init. Cover.	Final Cover.
AC.1	37	18	2	66.7%	94.4%
AC.2	72	46	2	23.9%	100%
AC.3	12	2	2	100%	100%
Minix.1	26	4	6	75%	100%
Minix.2	20	4	1	25%	100%
Minix.3	30	5	12	41.7%	41.7%
Minix.4	19	10	5	50%	90%
Minix.5	11	10	4	75%	100%
Ave.	28.3	11.6	4.3	41.9%	94.6%

4.3. Function Level Evaluation

In our initial evaluation, we selected 8 functions from 3 files. Table 1 shows the original branch coverage achieved during testing and the coverage achieved with fault injection. Our data included three functions from AC and five functions (commands) from Minix.

As Table 1 shows, we were quite successful in increasing test coverage of most of these functions. We were able to increase coverage to 100% in four of the functions (one function had 100% coverage before fault injection). Coverage ended up above 90% in all cases except for the one function that we failed to induce any coverage increase.

4.4. File and System Level Evaluation

We evaluated complete source files to expand the evaluation beyond individual functions. This evaluation included 27 files—9 files from AC, 9 files from C-Patrol, and all 12 files from Autoland. Tables 2, 3, and 4 show the evaluation data for these files.

Except for the two files with the highest coverage, 96.3% and 88.9%, we were able to increase the branch coverage in all of the files in AC (see Table 2). The largest change was in `measure.c` where coverage increased from 72.4% to 100%.

Branch coverage increases were more modest in the evaluation of C-Patrol coverage (see Table 3). We were able to increase coverage in only 4 of the 9 files. However, C-Patrol already had 84.5% branch coverage before fault injection, which is the highest initial test coverage of the 3 systems. The higher the initial coverage, the more difficult it will be to increase coverage. When there are fewer untested branches, the untested branches seem to be hard to reach.

Autoland is the one fault tolerant system that we evaluated. We were able to increase the test coverage of Autoland from 78.9% to 87.3% (see Table 4). This is the largest

Table 2. Fault Injection Applied to AC C Source Files

File	Lines	Branches	Test Cases	Orig. Cover.	Final Cover.
array.c	255	54	19	96.3%	96.3%
avasg.c	652	154	19	56.5%	63.0%
avexp.c	711	178	19	69.7%	71.3%
cfg.c	491	72	19	88.9%	88.9%
gcp.c	541	145	19	83.5%	89.0%
gcse.c	289	89	19	73.0%	77.5%
flat.c	877	204	19	86.3%	88.7%
meas.c	190	58	19	72.4%	100%
sem.c	637	213	19	82.6%	93.4%
Ave.	444.8	129.7	19	77.7%	83.6%

Table 3. Fault Injection Applied to C-Patrol Source Files

File	Lines	Branches	Test Cases	Orig. Cover.	Final Cover.
bind.c	426	64	94	89.0%	89.0%
call.c	595	110	94	90.9%	90.9%
inprt.c	498	16	94	93.8%	100%
insert.c	153	38	94	84.2%	84.2%
prepst.c	343	46	94	97.8%	97.8%
main.c	183	14	94	78.6%	78.6%
patrol.c	669	114	94	87.7%	88.6%
supprt.c	816	148	94	81.7%	88.5%
tmplt.c	691	127	94	73.2%	75.6%
Ave.	486	75.2	94	84.8%	87.0%

increase that we obtained out of the three systems. Since Autoland is fault tolerant, there are many branches that are designed only to respond to exceptional situations. These branches can be resistant to testing. This system came with a set of regression test data consisting of 5280 test cases, yet only 78.9% of the branches were reached. We were able to test many of these previously untested branches using the fault injection mechanism in VCP.

4.5. Potential Improvements

We were not able to increase coverage in several of the source files. Thus, we would like to improve our ability to force sections of code to run. We need to be able force the violation of more expressive assertions than the simple ones that we can now mutate. For example, now we cannot mutate local variables of a function, and, as a result, many faults cannot be modeled. The mutation of the state of local variables could be achieved by processing C-Patrol %%insert

Table 4. Fault Injection Applied to Autoland C Source Files

File	Lines	Branches	Test Cases	Orig. Cover.	Final Cover.
alt_hold.c	78	4	5280	100%	100%
autoland.c	296	24	5280	91.7%	91.7%
bae_gscf.c	414	22	5280	100%	100%
cmdmnt.c	117	6	5280	50.0%	83.3%
display.c	310	40	5280	85.0%	92.5%
flare.c	162	12	5280	83.3%	91.7%
glideslp.c	170	8	5280	100%	100%
innerlp.c	184	12	5280	75.0%	91.7%
interface.c	678	64	5280	60.9%	75.0%
mathutil.c	92	8	5280	75.0%	75.0%
mode.c	245	2	5280	100%	100%
racf.c	109	2	5280	100%	100%
Ave.	238	17	5280	78.9%	87.3%

statements with the injector.

Even with the ability to mutate local variable state, there are other faults that cannot be modeled. Consider the following code structure:

`if(!do_something())` then invoke recovery code

where the function do_something() is a library routine.

If `do_something()` is part of the program under test, the recovery code can be tested by violating the postcondition of `do_something()` or possibly the precondition. But, when the exception handling mechanism depends on some unmodifiable function, a different approach must be taken. For example,

```
if (p=malloc(sizeof(int))==NULL) recover();
```

is often found in C programs. It can be difficult to inject faults into `malloc`, because it is a library routine and we may have limited access to its source code. One possible solution is to automatically transform the `if` condition into something like:

```
if (!do_something()
OR active_injection==372)
then invoke recovery code.
```

Processing assertions with complicated mathematical expressions, rather than just simple inequalities, should be possible. For example, $(b^2 - 4ac) \geq 0$ could be processed algebraically to create invalid values of all three variables. An example injection of this sort is: $c = b^2/4a + \delta$. Our current system uses assertions that are executable when compiled as ordinary C code. That is, the pre-conditions work

after processing only by C-Patrol, as well as after processing with the injector and C-Patrol. If we relax this restriction, then we can write more descriptive assertions if we do not require that they match C syntax. For example, universal and existential quantifiers ($\forall$ and $\exists$) might be added to the recognized language.

5. Related Work

Fault injection can be used to modify either a program's source code text or the machine state of an executing program. The most common static fault injection is mutation testing. Much of the recent fault injection research is concerned with dynamic injection.

5.1. Static Fault Injection—Mutation Testing

Mutation testing involves testing modified or mutated program source text [8]. It is primarily applied to unit testing—testing which involves small individual modules of the program.

A mutant program is created by making a small syntactic change, a mutation, to the original program. For example, a greater-than operator, $>$, might be changed to greater-than-or-equal, $>=$. To help test error recovery code, we could mutate the operators in conditional statements to cause a program to branch into recovery code.

The output of the original program is compared with the output produced by the mutant. If the mutant and the original program produce the same output, then either the set of test cases is not adequate, or the mutant is a functionally identical to original. No automatic procedure can determine whether the original program and the mutant are equivalent or the set of test cases is inadequate.

Mutation testing can require the creation of a vast number of mutant programs. A program with N variable references can have N^2 mutant versions. In one study of a 30-line triangle program, 951 mutants were automatically created [16]. Massive computational resources can be required to repeatedly recompile and run all of the mutations.

Weak mutation testing is less rigorous but more efficient than strong mutation testing [12]. Using weak mutation testing, mutant and original program states are compared soon after the mutation is executed, rather than after entire programs are run. States can be compared soon after executing mutated expressions, statements, or basic blocks. Weak mutation testing is much less expensive than strong mutation testing, while almost as effective [16].

Weak mutation testing does not solve the problem of identifying equivalent mutants. Also, invalid program states in fault tolerant software should not propagate to the output. Thus, weak mutation testing may be difficult to apply to robust fault tolerant software.

The TAMER fault injection tool mutates source code to test fault-tolerant system [5]. TAMER injects possible faults at module interfaces using a "fault manager." The system is designed so that all mutants can be created using only one compilation, and iterates the execution of the mutants. An experiment demonstrated that fault injection is needed to test the fault tolerance of a program.

5.2. Dynamic Fault Injection

The dynamic or state changing forms of fault injection do not require multiple compilation. Dynamic fault injection is most commonly used to simulate hardware errors by modifying or injecting faults into memory bits and registers [15, 1, 4, 10]. The changed memory locations can contain both program instructions and data.

A wide variety of faults can be emulated. However, dynamic fault injection has been commonly used to modeling hardware faults such as bus errors or incorrect CPU instructions. Such hardware faults often have such a drastic effect on the system that human operator intervention is necessary, thus impacting an automated testing process. Testing only for hardware faults ignores potential software faults.

Four dynamic fault injection case studies are relevant to our work:

1. The DEF.Injector ('Defined Errors Fast Injector') injects a set of hardware faults by toggling memory bits or bytes, changing bus addresses, and changing machine instructions [10]. It is a hardware device that is physically connected to specific target machines. Because of the dedicated hardware involved, intermittent faults are easily modeled. The DEF.Injector can achieve thorough coverage of all defined tests because the address space of the target machines is quite small (8-64K). Exhaustive hardware fault injections would be infeasible on a machine with an address space of several megabytes.

2. FIAT ('Fault Injection-based Automated Testing') uses fault injection to evaluate the dependability of fault tolerant software [1]. FIAT, implemented entirely in software, changes bits in both program text and data regions of machine memory to simulate hardware faults.

3. Chillarege and Bowen manually injected 70 *overlay faults* into a large commercial transaction processing system [4]. Overlay faults occur when a program writes into an incorrect location due to a faulty destination operand. To decrease fault and error latency and increase the probability that a fault will cause an error, a large region of memory was corrupted with a single injection. Injection involved setting all bits in an entire page of physical memory to one. About 16% of

the faults immediately crashed the system; about 14% caused a partial loss of service; half of the faults did not cause failures.

4. FINE ('The Fault Injection and Monitoring Environment') was used to study fault propagation in UNIX operating system kernel [15]. This system modeled hardware faults including memory faults, CPU faults, bus faults, and I/O faults.

   Unlike the other studies, software faults were studied including initialization faults, assignment faults, condition check faults, and invalid function faults. Initialization faults can be detected by a compiler. Assignment and condition check faults are clearly relevant to the testing of error recovery code, since an incorrect assignment or condition can be a condition that should force the execution of recovery code. Here, invalid function faults are created by replacing function code with manually rewritten alternatives. For our purposes, invalid function code must be automatically generated, since manual rewriting of code is prohibitive in a large system.

These experiments show some seemingly inherent problems with dynamic fault injection. Some injected faults can crash the computer requiring operator intervention, so specialized hardware is needed for a completely automated process. Some injected faults will have a very high error latency. A system that continues to operate correctly for a long time after injection may or may not have recovered from the fault. It is very difficult to determine whether a latent error may still eventually occur.

5.3. Assertion Violation vs. Related Work

Our assertion violation technique generates faults by mutating program state at run time. Thus it does not incur the recompilation overhead of mutation testing. Correctness checking can be performed by comparing output or by comparing states. It can be used to model both hardware and software faults.

To model transient memory faults, the memory locations (variable aliases) characterized in an assertion can be corrupted. The degree of corruption will depend on the specific rules which control the violation. These faults could explicitly mimic the injections that others have used, for example, by setting and resetting bits and groups of bits.

Initialization faults related to function parameters are easily modeled. The parameters referenced in the precondition can simply be set to a random value, or some predetermined but incorrect value, such as zero.

Many assignment faults can be generated by modifying parameter values. An assignment that makes use of a mutated parameter or global variable will be faulty. However, not all assignment faults are covered. Like the assignment faults, condition check faults are modeled when the conditions make use of input parameters.

Invalid functions that result from complicated, abstract programming errors are modeled quite well by assertion violation, since there are many options for generating faults from complex assertions.

Some hardware faults cannot be easily modeled, such as incorrect CPU instructions or stuck-at bus faults. Modeling these faults may not be necessary, since they will probably crash most fault tolerant systems anyway (unless it is equipped with special hardware).

The software faults modeled by assertion violation will tend to accelerate the progression from fault to error. An injected fault is likely to be encountered soon after injection because the violated pre-condition is 'about' variables that are used in the function that runs immediately after execution. Similarly, the faulty return value changed by a post-condition injection will soon be detected by other pre- or post-conditions.

6. Conclusions

We have developed the assertion violation mechanism for inserting faults. The method mutates state by violating specified function pre- and post-conditions.

We have implemented assertion violation in our Visual C-Patrol system (VCP). VCP includes assertion violation fault injection in a visual interface to C-Patrol. The first step in using VCP is to use the VCP interface to add pre- and post- conditions to functions of interest. Then the software is tested by running the instrumented program. Faults can be injected and monitored at run time from the run time interface. Pre- and post-conditions will indicate if the fault-injected program satisfies specified constraints. A tester can also determine whether the error recovery code is correct from the output of the fault injected program.

Our evaluation demonstrates that fault injection can be effective in increasing the coverage of hard to reach parts of a program. When applying fault injection to individual functions, coverage reached 90% or higher in all but one function. Five functions reach 100% coverage, while the coverage of only one function is unchanged. Coverage also increased when we applied fault injection to entire source files. We applied fault injection to a total of 30 source files, and increased the coverage in 16 of them. Nine of the 14 files that showed no improvement already had over 90% coverage without fault injection. Fault injection increased the coverage in 15 of the 20 files (75%) that started with less than 90% coverage.

Table 5 shows that we were able to increase overall test coverage. In the file-level tests, coverage surpassed 80%. In industrial practice 80% branch coverage is considered quite

Table 5. Evaluation Summary Data

Evaluation Software	Original Coverage	Final Coverage	Increase over original
8 Functions from 3 files	41.9%	94.6%	125.8%
6 AC Compiler files	77.7%	83.6%	7.6%
9 C-Patrol files	84.8%	87.0%	2.6%
All 12 Autoland files	78.9%	87.3%	10.7%

good. Fault injection allowed us to increase test coverage without adding further test cases. Thus, we can get higher coverage with fewer test cases.

Assertion violation modifies the global variables or parameters values that appear in function pre- and post- conditions. Many other kinds of assertions could be used to generate faults, and thus increase the effectiveness of fault injection. Suggested future modifications to the assertion violation scheme are:

- Add the capability of local variable assertions,
- Enrich the assertion language,
- Remove some non-deterministic aspects.

References

[1] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. Fault injection experiments using FIAT. (Fault Injection-based Automated Testing). *IEEE Trans. Computers*, 39(4):575–583, April 1990.

[2] B. Beizer. *Black-Box Testing*. John Wiley & Sons, 1995.

[3] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 129–134. North-Holland, 1981.

[4] R. Chillarege and N. Bowen. Understanding large system failure – a fault injection experiment. In *Proc. 19th Int. Symp. Fault-Tolerant Computing (FTCS-19)*, pages 356–363, Chicago, IL, June 1989.

[5] R. DeMillo, T. Li, and A. Mathur. Architecture of TAMER: a tool for dependability analysis of distributed fault-tolerant systems. Technical Report SERC-TR-158-P, Software Engineering Research Center, Computer Science Dept., Purdue Univ., 1994.

[6] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):803–820, May 1979.

[7] R. DeMillo, W. McCracken, R. Martin, and J. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings, Menlo Park, CA, 1987.

[8] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offut, and K. N. King. An extended overview of the mothra sofware testing environment. *Proc. ACM SIGSOFT/IEEE Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, July 1988.

[9] S. Garland, J. Guttag, and J. Horning. Debugging larch shared language specifications. *IEEE Trans. Software Engineering*, 16(9):1044–1057, September 1990.

[10] J. Gerardin. The 'def.injector' test instrument, assistance in the design of reliable and safe systems. *Computers in Industry*, 11(4):311–319, Feb. 1989.

[11] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[12] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Engineering*, SE-8(4):371–379, July 1982.

[13] W. Howden. A functional approach to program testing and analysis. *IEEE Trans. Software Engineering*, SE-12(10):997–1005, October 1986.

[14] C. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, London, 1986.

[15] W.-L. Kao, R. Iyer, and D. Tang. Fine: a fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Trans. Software Engineering*, 19(11):1105–1119, Nov. 1993.

[16] A. Offutt and S. Lee. An empirical evaluation of weak mutation. *IEEE Trans. Software Engineering*, 20(5):337–345, May 1994.

[17] L. J. White. Basic mathematical definitions and results in testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 13–24. North-Holland, 1981.

[18] H. Yin and J. Bieman. Improving software testability with assertion insertion. In *Proc. Int. Test Conf.*, Oct. 1994.