The Stata Journal (2001)
1, Number 1, pp. 29–50

Statistical software certification

William Gould
Stata Corporation
wgould@stata.com

Abstract. In the first part (sections 1 and 2), this article describes the automated process that is used to certify Stata prior to shipment of new releases or updates. The second part (section 3) describes how these techniques can be adopted to your own work.

Keywords: pr0001, certification scripts, ado-files, cscript

1 Introduction

Whenever a new version, release, or update of Stata is created, it is tested before being put into production or being posted at *www.stata.com*. Many users suspect that this testing is done by other users (beta testing) or by Stata employees using Stata just as a user would. In fact, we seldom use beta testing and, while Stata developers will of course use Stata to try out their changes and additions, as a testing method it is not trustworthy because it is not reproducible and neither is it sufficient.

In our experience, a person sitting at a computer can, for short bursts, execute one Stata command every three seconds. Even if they could keep that rate up for 8 hours, that would amount to issuing "only" 9,600 Stata commands (invoking, via ado-files, the execution of around 59,000 total commands), and the tester could not, in the three seconds allotted to each command typed, look carefully enough at the output to detect any problems that might arise.

Stata is instead tested using an automated procedure that involves running 1,064 do-files containing 158,391 lines that cause Stata to execute 38,343,139 commands and produces just over 16 megabytes (473,859 lines) of output. The whole process, from beginning to end, requires typing "`do testall`". While this automated procedure solves the input problem, reviewing 16 megabytes of log files would require a herculean effort. Moreover, there are currently 28 different versions of Stata once one counts Windows, Macintosh, and Unix platforms, and running tests on all of them results in $28 \times 16 = 448$ megabytes of log files.

We maintain certified versions of these logs and use computer techniques to compare the certified logs to the output produced, but it is rare indeed that this method actually detects problems. The primary way Stata is certified involves the construction of the test scripts themselves: The scripts not only set up problems and run them, but also include code to verify that the results are as expected. If a result deviates from what is expected, the test script produces error messages and stops.

Since Stata is programmable, Stata is used to certify itself. The test scripts exploit Stata's ability to save and later to reaccess calculated results and couples that with Stata's `assert` command. `assert` allows one to state what should be true. If the statement is true, `assert` does nothing. If the statement is false, `assert` issues an error. For instance, if one has a dataset in which the mean of `x` should be 3, one can type

```
. use dataset, clear
. summarize x
. assert r(mean)==3
```

If one places those commands in a do-file, and if the assertion applied to these data should turn out to be false, Stata will issue an error message and the do-file will stop:

```
. do testit
. use dataset, clear
. summarize x
    Variable |      Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
           x |       74    21.43243     6.05473        12         41
. assert r(mean)==3
assertion is false
r(9);
end of do-file
r(9);
. _
```

That the test failed cannot be missed.

As mentioned, we use 1,064 do-files to test Stata, but the do-files are nested, with do-file calling do-file, and all that is required to perform the tests is to launch Stata, change to the directory containing the test scripts, and type

```
. do testall
```

The result of that is either to produce

```
(output omitted)
end of do-file
. _
```

or

```
(output omitted)
assertion is false
r(9);
end of do-file
r(9)
. _
```

In the first case, Stata passes certification. In the second, there is a problem.

2 Organization and procedures used at StataCorp

The test scripts are nothing more than a collection of do-files. The top-level `testall.do` file reads

top of testall.do

```
clear
discard
set more off

cd base
quietly log using test, replace
do test
quietly log close

cd ../ado1
quietly log using test, replace
do test
quietly log close

(lines omitted)

* end of testall.do
```

end of testall.do

The purpose of this do-file is to run a sequence of do-files located in subdirectories of which, currently, there are 46. Each subdirectory contains a file `test.do`. Here is the `test.do` file from the subdirectory named `base`:

top of base/test.do

```
about
query compilenumber

do assert           /* assert */
do merge            /* merge/append */
(lines omitted)
do nchi2            /* nchi2() function */
```

end of base/test.do

`test.do` is another file that simply causes other do-files to run. It is the files `assert.do`, `merge.do`, and `nchi2.do` that are the actual test scripts. There are 1,064 files for testing Stata, one of which is `testall.do` and 46 of which are the `test.do` files in the 46 subdirectories, so there are 1,017 files like `assert.do`, `merge.do`, and `nchi.do`.

The division into the 46 subdirectories is based on nothing more than convenience. As previously mentioned, the result of running all the tests produces 16 megabytes of log files. A single 16 megabyte file is difficult to deal with. We instead produce a separate log in each subdirectory, so the 16 megabytes are spread over 46 separate logs with the average log being 348 kilobytes long. The actual logs vary between 4,978 bytes and 1,105,298 bytes.

Before shipment, a formal round of certification runs are made and the logs are collected and archived, but Stata is also continually subjected to certification. Every

developer at StataCorp has the test scripts on his or her computer and runs the test scripts, in whole or in part, often.

Developers can run the whole suite by typing "`do testall`", and this takes about an hour on the median computer at StataCorp, with the actual times varying from just over half an hour on the fastest computers to 8 hours on the slowest. A developer can also change to one of the subdirectories and type "`do test`" to run one of the 46 subsets of tests which, on the median computer, can be run on average in about 1.3 minutes. Alternatively, a developer can run one of the particular tests. A developer could type "`do assert`". The particular tests can be run in a few seconds on the median computer.

This constant access to the test suite, both in whole and in part, actually speeds development. Stata is a tightly integrated system where commands call other commands to act as their subroutines, and changes to a command can have unintended side effects, both positive and negative. It is, in general, not sufficient for a developer to convince himself or herself that the changes to a particular command, whether in internal C code or the ado-files, are as desired. He or she must also establish that the changes do not interact with some other feature of some other command in unexpected ways.

From this description, the test suite may seem to be a static, carefully thought out collection of files. Nothing could be further from the truth. As of the instant of this writing, the test suite contains 1,064 files in 46 directories, but as of tomorrow, the number of files or even number of subdirectories might grow. Every time a new feature is added to Stata, or a bug reported and fixed, or a change made to the code, additions are made to the test scripts. Every developer at StataCorp has a copy of the test suite on their computer and, because the test suite is nothing more than a collection of files, they can change or make additions to any part of it. And they do. There is then a procedure that they go through to submit their changes so that the changes become part of the official certification scripts, just as there is a procedure for changes to the Stata code itself. In fact, the submission of changes and additions to the test script is a required part of submission of code.

Thought is given to the individual test scripts, but the overall design and thrust of the suite simply evolves as changes and additions are made to its individual components.

2.1 Test scripts

The first test script of the first test suite directory is `assert.do`. It is the purpose of this script to establish that `assert` is working. This makes sense because all of the remaining test scripts use `assert` to establish that the results are correct.

It is, however, not important that `assert` be tested first; it is merely important that, at some point, `assert` be tested.

Each particular test script focuses on some aspect of Stata, typically a command. The purpose of a test script is to establish that the feature under consideration works as intended and, in the test, the script may assume that all the remaining features of Stata work just as they should. Thus, in testing feature A, one writes the script assuming

features B and C work. Other test scripts will establish that B works conditional on A and C, and that C works conditional on A and B. The result of running the entire suite will be to establish that A, B, and C all work unconditionally.

The only caution that must be exercised in this approach is that, if the test of A conditional on B and C working fails, one cannot jump to the conclusion that the problem is with A. The problem might be with B or C, and that is often the case. Say that, as a developer, I am working on a preexisting ado-file `xyz.ado`. Somewhere in the test scripts, there is a corresponding `xyz.do` file that tests `xyz.ado`. As I work on the `xyz.ado`, I either add to the existing `xyz.do` file, or perhaps create a new, additional test script, `xyz2.do` that I insert somewhere in the suite.

There is no rule at Stata that there be a one-to-one correspondence between commands and test scripts; the rule is merely that there be at least one test script for each and every command. In fact, Stata currently has 630 user commands (ignoring ado-files that are subroutines to commands users type). Given that there are 1,017 test scripts, there are, on average, 1.6 test scripts per command.

Anyway, I am working on `xyz.ado` and continually rerunning the particular test script `xyz.do` and perhaps `xyz2.do` and, one assumes, adding to it. Satisfied, I think that all is working. It is now of vital importance that I run the whole suite. If a problem arises, and if `xyz.ado` is the only file that has changed, I can be certain that the problem is with `xyz.ado`, even if the test that fails intends to test `abc.ado` under the assumption that `xyz.ado` is working.

The tight integration of Stata which, at first blush, makes it seem that Stata will be more difficult to test and certify, in fact makes it more likely that bugs will be uncovered by tests that were never intended to uncover bugs where they are found.

Returning to the issue of StataCorp procedures, there is no danger that a developer will forget to run full certification. Individual developers use and continually run the test scripts, but those private actions are separate from the official runs made before software is shipped, at which point the process becomes very officious and bureaucratic. Logs are then collected and compared to certified logs in another lengthy procedure, the entirety of which takes almost a week. As an aside, individual developers run the tests scripts privately not only because they care or it makes software development easier, but also to avoid embarrassment. The worst shame a developer can experience is to turn in changes and have them fail in the official certification round. This is not policy but is simply a result of the work that all developers must go to during an official certification round. If certification fails at this point, everyone at StataCorp must back up, and the other developers make their feelings about the extra work well known to the culprit. There is not one developer at StataCorp who has not made this mistake, but most make it only once.

2.2 Log comparison

Running the test scripts produces 46 log files containing a total of 473,859 lines or 16,248,046 characters. In an official certification round, which occurs prior to the release of an update or new version of Stata, these logs are collected, compared to official versions of the logs, and archived. The comparison is performed on a Unix computer using the Unix `diff`(1) command.

As mentioned previously, problems are seldom uncovered at this step because most problems are detected by the test scripts themselves; the certification run fails with the error message "assertion is false"; r(9). The only problems that should be detected at this stage are problems with Stata's output engine. The calculation has been made correctly and the result has been stored correctly, but somehow, in the presentation of that correct result on the page, it became corrupted. Log comparison did detect problems during the development of SMCL in Stata 7, when the output engine itself was being rewritten. Log comparison was also useful during the development of Stata 7 with the introduction of long variable names. Test scripts could detect problems with the use of long variable names, but they could not detect problems with tables that no longer lined up properly because a long variable name violated the column-width assumptions.

Putting those two cases aside, the detection of problems at the log-comparison step is rare and, in most instances, indicates a problem with the test script in that the test script should have detected the problem and stopped the certification run. When such problems are detected, the test script is changed. Sometimes, the test script did not check something because the command did not save it. In that case, the command is modified, and that is why, if you type `return list` or `estimates list` after a command, you will sometimes find results saved that are not documented in the manual. The extra, "uninteresting" results were included to allow certification.

3 Adoption

Stata is a unique package in that users can and do make additions to Stata, and there is a large and active user community not only doing that but also making those additions available to other Stata users over the Internet. The methods that we use to certify Stata are easily adopted to personal use.

The materials that we use to construct our official test scripts are available and are installed in every copy of Stata, although they are not documented in the manual. Documentation can be found by typing "`help cscript`". Those tools, however, are not a necessary ingredient; they just make writing a test script a little easier. What is required is a little organization and an understanding of what should appear in an individual test script.

Program authors will find test scripts to be well worth the effort. It is not uncommon that authors need to go back and make changes to their programs to improve the output or improve the input, and it is convenient to have a quick way to verify that the changes just made did not break the program. In this regard, even short, incomplete test scripts work well.

3.1 Organization

Begin by duplicating our organization:

1. Create a directory called `bench`, placing the directory wherever you find convenient. The directory can be `c:\bench`, `d:\mystata\bench`, `/home/me/wherever/bench`, `:mydrive:analysis:bench`, or anything else that appeals. In the future, when you want to run your certification scripts, you will change to that directory and type "`do testall`".

2. Create a subdirectory from `bench` for your first (and perhaps only) collection of tests. Call this first subdirectory `bench/first`.

   Notice our use of the forward slash character (`/`). Paths in Windows are typically written using backslashes and paths in Macintosh using colons, but Stata itself understands the forward-slash notation and so that single notation can be used on all operating systems. When writing Stata do-files, using the forward-slash notation will allow your scripts to run on any operating system should the need ever arise.

3. Create file `testall.do` in the `bench` directory that contains

```
top of testall.do
        clear
        discard
        set more off

        cd first
        quietly log using test, replace
        do test
        quietly log close

        cd ..
end of testall.do
```

   or

```
top of testall.do
        clear
        discard
        set more off

        cd first
        do test

        cd ..
end of testall.do
```

   depending on whether you plan to collect logs. Although we at StataCorp collect logs, it is not nearly as important that you collect logs since no ado-file you write could modify the Stata output engine. If you collect logs and examine them, it will be more likely that you will detect problems with misformatted tables, but misformatted tables do not impose great costs on users. They are inconvenient, but no one is misled into accepting a result as correct that is not.

If later you want to add a second directory of tests, you can edit `testall.do` and add "`cd second`" and "`do test`".

4. Change to the `first` subdirectory and create `test.do`:

```
------------------------------------------------------- top of first/test.do -------
        about

        do firsttest
------------------------------------------------------- end of first/test.do -------
```

   This is nearly identical to the `test.do` style we use, with the omission of the line `query compilenumber` after the `about`. `query compilenumber` is not documented in the manuals, but every compile of Stata has a number that we use in tracking the logs we collect.

5. Finally, create file `first/firsttest.do`, which we will make just a stub of a program that we will fill in later:

```
------------------------------------------------------ top of first/firsttest.do -----
        cscript "stub of test script"
        display "to be filled in"
------------------------------------------------------ end of first/firsttest.do -----
```

Doing all of this, you should be able to launch Stata, change to the `bench` directory, type "`do testall`", and see the following result:

```
. do testall

. clear

. discard

. set more off

. cd first
/home/wwg/bench/first

. do test

. about

Intercooled Stata 7.0 for Unix
Born 17 Sep 2001
Copyright (C) 1985-2001

2-user Stata for Linux (network) perpetual license:
       Serial number:  1034
         Licensed to:  William Gould
                       StataCorp

. do firsttest

. cscript "stub of test script"
-------------------------------------------------------BEGIN stub of test script
```

```
. display "to be filled in"
to be filled in

.
end of do-file

.
end of do-file

.
. cd ..
/home/wwg/bench

.
end of do-file
```

3.2 Test scripts

While the goal of a test script is to establish that a command works—that it gives the correct answers in all cases—that is a goal that no test script can achieve. Real tests scripts do nothing more than establish that the command works in some cases and, because the test script will be run repeatedly in the future, that the command continues to produce the same answers as changes are made to the command, to the environment in which it runs, and to Stata itself.

A test script is nothing more than a do-file, and most test scripts are inelegant collections of lines that use datasets, make calculations, and verify that results are as anticipated. A few of the test scripts used at StataCorp are elegant programs themselves—the do-file includes the definitions of tens of programs that then run the command being tested over and over and verify the consistency of the results—but that is very much the exception. The typical test script looks like

top of summarize.do

```
cscript summarize
which summarize

use auto
summarize mpg
assert r(N)==74 & r(sum_w)==74 & r(min)==12 & r(max)==41 & r(sum)==1576
assert reldif(r(mean), 21.29729729729730) < 1e-14
assert reldif(r(Var),  33.47204738985561) < 1e-14
assert r(sd) == sqrt(r(Var))

summarize mpg if foreign
assert r(N)==22 & r(sum_w)==22 & r(min)==14 & r(max)==41 & r(sum)==545
(lines omitted)
```

end of summarize.do

All test scripts begin with the `cscript` command. This command merely clears things that previous test scripts might have left in memory such as values labels, macros, and the like.

Note that test scripts do not start with a `version` command. The purpose of a test script is to establish that the command works under whatever is the current version of Stata. An exception to this rule would be a test script that intended to test a command that changed from one version to the next. In that case, the older test script might very well start with a `version` command to verify that, in backward compatibility mode, the command still works as it used to work. At the time of the change, you would take the test script and make two test scripts from it. In one copy, you add a `version` command at the top and use the resulting script to test backward-compatibility mode. In the other, you add no `version` statement but you update the script to reflect the more modern usage. In neither case do you worry that some parts of the two test scripts are covering the same ground.

An important rule of test-script writing is never to delete. Instead, you should add. The more you test a command, the better. It is not important that a test script look pretty, be elegant, or appear well organized. In fact, if you are collecting and comparing logs, organization can make comparison more prone to error. At StataCorp, when a command incorporates a new feature, we never change the middle of a script. We add to the bottom of the script or create a new script so that, when the computer lists the log differences, we can see the new output collected together and know that the old output did not change.

Finding correct answers with which to compare

In writing real test scripts, finding answers with which to compare can be daunting, difficult, or impossible. Sometimes, you can do pencil-and-paper calculations for some simple cases. Sometimes, you can find examples in books, although just because results have been published does not mean that they are right. Sometimes, there is simply nothing with which to compare.

The problem of determining whether results are right is nothing new and is independent of test scripts. What is new is that you will codify some answer into the test script, and some programmers simply cannot bring themselves to codify an answer that they are not absolutely certain is right. To the extent that this causes them to work harder to verify they have correct answers to which to compare, that is good. To the extent that it prevents them from creating a test script because of their uncertainty, that is bad. Test scripts can be changed just as the underlying program can be changed. You code the answers you think are right and if, later, it turns out you were wrong, you change the test script just as you change the software.

Test scripts do not so much verify that results are right as they verify that results continue to be the same in the future. With age, the tests in your certification scripts become verification of correctness.

You convince yourself that you have right answers the first time just as you have always convinced yourself that you have right answers. If there is nothing with which to compare, you look at consistency—do the residuals sum to zero, are the predictions in line with the actual values, etc.—and having done that outside the test script, you

declare the answer produced by your code to be "correct". This is exactly what was done in the sample `summarize.do` above. Working outside the test script, the developer convinced himself that `summarize` was producing the correct answers and then, in producing the test script, the developer simply took answers produced by `summarize` and codified them into `assert` statements.

In the case of estimators and tests, we at StataCorp often perform simulations and check that coverage is correct. That is not part of the test script because we do not want to wait for simulations to run in the future. We check those things once and then, convinced that the routine is producing correct answers, we accept the answers being produced by the routine at that instant as correct and codify them.

You do the work to obtain correct answers just as you ordinarily would and with the same care and caution that you do ordinarily.

Precision and false precision

It is very common that, in the production of test scripts, after convincing oneself that the results are correct, one simply copies the results from the command into the script. In testing `summarize`, the developer interactively typed

```
. summarize mpg
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |        74     21.2973    5.785503         12         41
. return list
scalars:
                  r(N) =  74
              r(sum_w) =  74
               r(mean) =  21.2972972972973
                r(Var) =  33.47204738985561
                 r(sd) =  5.785503209735141
               r(min) =  12
               r(max) =  41
               r(sum) =  1576
```

and then copied the results from `return list` into the script, editing them to produce the appropriate `assert` statements:

```
assert r(N)==74 & r(sum_w)==74 & r(min)==12 & r(max)==41 & r(sum)==1576
assert reldif(r(mean), 21.29729729729730) < 1e-14
assert reldif(r(Var),  33.47204738985561) < 1e-14
assert r(sd) == sqrt(r(Var))
```

Assume that you have a new command `xyz.ado`, that it produces the result .13698414276846993, and that you are reasonably convinced that .13698414276846993 is "correct". You might then be tempted to code in your do-file

```
assert r(result)==.13698414276846993
```

You argue thusly: I am reasonably certain that .13698 is correct to -5 digits. I just ran my program on the data and it produced .13698414276846993, which could be correct

and presumably is more precise than .13698. I want to be certain that my program continues to produce the same answers in the future, so I will code in my script

```
assert r(result)==.13698414276846993
```

You do that and all works well. Next year, you buy a new computer and perhaps change operating systems. You run your certification scripts and Stata reports, "assertion is false"; r(9).

How can it be that the answer changed? If you changed operating systems, you will probably decide that there must be a bug in Stata. In the case where you did not change operating systems, you begin to wonder if all the files copied correctly.

You have just encountered the problem of false precision. Given the way modern computers work, the same code can produce slightly different answers due to the way coprocessors are designed and employed. Double precision floating point numbers are stored using 64 bits. Coprocessors, however, use 80 bits, providing extra guard bits to improve accuracy. On the coprocessor, calculations are made using 80 bits and are then handed back to the CPU rounded to 64 bits. If you were to calculate $(a+b)/(c+d)$, part of it or all of it is performed at 80-bit accuracy, and whether it is all or part makes a difference. On some computers, the result will be to calculate $(a+b)/(c+d)$ using 80 bits and then to round the result to 64 bits. On others, $a+b$ will be calculated at 80-bit accuracy and the result then rounded to 64 bits, $c+d$ calculated similarly, and then the two 64-bit results divided (with 80 bits of precision) and rounded to 64 bits. The problem can be even worse than that. If your code calculates $z = (a+b)/(c+d)$ and later uses z in a subsequent calculation, with some computer/compiler combinations, z will continue to reside on the coprocessor in its full 80-bit glory and, on others, the 64-bit rounded version will be used.

The outcome is that the same calculation will yield slightly different results.

There are other, more traditional problems that can affect accuracy. Try the following experiment:

```
. use auto, clear
. sort mpg
. probit foreign mpg weight
. mat b1 = e(b)
. sort make
. probit foreign mpg weight
. mat b2 = e(b)
. display mreldif(b1, b2)
```

You will discover that b_1 differs from b_2 by about 1.915e–16. `probit` went through the same calculation in both cases, but the dataset was in a different order, and the calculation of the probit estimator involved the calculations of sums. Although addition is associative mathematically, in finite-precision floating point arithmetic, it is not. A small difference arose because the sums over the data were performed in a different order.

The result of all of this is that you do not want to code

```
assert r(result)==.13698414276846993
```

You want to code

```
assert reldif(r(result), .13698414276846993) < 1e-5
```

or

```
assert reldif(r(result), .13698414276846993) < 1e-6
```

Judging how many digits to include is virtually impossible unless you have lots of different computers and run the script on all of them, and even then, you have to make a judgment because you know there are computers on which you have not tested. At StataCorp, we would probably code

```
assert reldif(r(result), .13698414276846993) < 1e-14
```

and wait for a computer to complain, in which case we might trim the comparison back to 1e–13. We tend to require lots of precision, even false precision, because we want to know when an answer changes, even if it is just a little bit. As the code changes, the answer will change a little and we will have to modify our test script. In our case, we want to exercise lots of control, and we are willing for test scripts to raise lots of false alarms. In your case, it would not be unreasonable to restrict the comparison to the number of digits you actually believe.

Testing extreme cases

No test script can prove that a command works by verifying just a few examples, even when the answers are known with certainty. One of the best ways to test routines is with extreme cases. Extreme cases put considerable stress on your code and, if the code can handle that, the chances that it is getting interior cases correct are greatly improved.

$R^2 = 1$ regressions; `xt` estimators run on a single panel or multiple, single-observation panels, or on dependent variables that do not vary; all form fruitful examples worth exploring. Many estimators, in such extreme contexts, reduce to other, simpler estimators to which results may be compared. Moreover, it is worth remembering that users will use your program in inappropriate circumstances if, due to nothing else, by accident.

These problems are easily set up, and easily run, and nearly always detect problems in the underlying code.

In `summarize.do`, another part of the test script reads

top of summarize.do

```
(lines omitted)
clear
set obs 1000
gen double x = 1/9
summarize
assert r(mean)==1/9 and r(Var)==0
(lines omitted)
```

end of summarize.do

This particular test is extremely taxing and few packages could pass it. One-ninth is both a repeating decimal and a repeating binary; that the mean comes out to be exactly one-ninth (to 64-bit precision) and the variance to be zero establishes that Stata's quad-precision (128 bit) routines are working.

Testing for errors

It is not sufficient to test that your code works in circumstances where it should. You must also test that your code does not work in circumstances where it should not. This is not unlike testing extreme cases except that, in extreme cases, you are expecting an answer and, in this case, you are simply expecting that your code will refuse to run. For example, in testing `probit`, one might code

```
use auto, clear
gen outcome = 1
rcof "noisily probit outcome mpg weight" == 2000
replace outcome == 0
rcof "noisily probit outcome mpg weight" == 2000
```

`rcof` is one of the more useful commands in the `cscript` suite; see `help rcof`. Test scripts stop when an error arises, which is to say, when the return code is not zero. Thus, it would seem impossible to include intentional errors in test scripts. `rcof` solves that problem. `rcof` runs a command and verifies that the return code is as stated. If it is, `rcof` is satisfied and there is no error, `rcof`'s return code is zero. If the return code is not as stated, `rcof` itself returns a nonzero return code, stopping the test.

`rcof` can be used to test that the return code is a particular number, as we did above, or simply to establish that some error did indeed occur:

```
rcof "noisily probit outcome mpg weight" ~= 0
```

Testing for syntax errors

If the command has complicated or nonstandard syntax, verify that the command gives the appropriate error message if used inappropriately:

```
rcof "noisily summarize mpg, detail meanonly" ~= 0
```

The above line appears in the official test script for `summarize`. Options `detail` and `meanonly` are mutually exclusive.

Testing data-management commands

Not just statistical commands require testing. Statistical commands return results in `r()` or `e()`, and it is simply a matter of verifying that saved results are as expected. In data-management commands, results are seldom returned. Instead, the dataset itself is modified.

Determining correct answers in data-management commands is easy. Asserting those results sometimes requires more code than the testing of statistical commands. For instance, verifying that `sort` works with one variable is easy enough:

```
sort x
assert x>=x[_n-1] if _n>1
```

Establishing that `reshape` works takes more lines:

```
input id sex inc80 inc81 inc82
1 0 5000 5500 6000
2 1 2000 2200 3300
3 0 3000 2000 1000
end

reshape groups year 80 81 82
reshape vars inc
reshape cons id sex
reshape q
reshape long
sort id year
assert _N==9
assert id==1 in 1/3
assert id==2 in 4/6
assert id==3 in 7/9
assert sex==0 in 1/3
assert sex==1 in 4/6
assert sex==0 in 7/9
assert inc[1]==5000 & inc[2]==5500 & inc[3]==6000
assert inc[4]==2000 & inc[5]==2200 & inc[6]==3300
assert inc[7]==3000 & inc[8]==2000 & inc[9]==1000
by id:  assert year==80+_n-1
```

The above are some of the 161 lines that appear in the official `reshape.do` file for testing `reshape.ado`.

Testing graphical commands

Most graphical commands return nothing except a graph on the screen. All graphical commands, however, include option `saving(`*filename*`, replace)` for saving the graph into a file, and that provides the key for comparing graphs created in test script with graphs that have been carefully reviewed and previously saved.

Test scripts for graphical commands should specify the `saving(`*filename*`,replace)` option. In addition, so that test scripts can be run without having to clear the graph before the script continues, one should `set more off` in the test script before drawing graphs.

Review the graphs created by the test script and then copy them to another directory so that, next time, you can verify that the `.gph` files are the same by comparing the files the test script produces with the certified copies. This comparison must be done outside of Stata using something like the Unix `cmp`(1) command.

At StataCorp, we copy files to a Unix computer from our Windows and Macintosh certification runs. Unix-style tools are available for free for Windows from the Cygwin project (Cygwin Group 2001, *http://www.cygwin.com*). The modern Macintosh OS X operating system is based on Unix and includes the `cmp` command.

Testing if and in

Errors in honoring `if` *exp* and `in` *range* can easily occur in Stata code, and so it is of importance that you test these components. In `summarize.do` appears the following lines:

```
summarize mpg if foreign
assert r(N)==22 & r(sum_w)==22 & r(min)==14 & r(max)==41 & r(sum)==545
assert reldif(r(mean), 24.77272727272727) < 1e-14
assert reldif(r(Var),  43.70779220779221) < 1e-14
assert r(sd)==sqrt(r(Var))
```

It is sometimes useful to place the assertions for a particular case into a program so that they can be used repeatedly. A piece of `summarize.do` could have read

```
program define check_if_foreign
        assert r(N)==22 & r(sum_w)==22 & r(min)==14 & r(max)==41 & r(sum)==545
        assert reldif(r(mean), 24.77272727272727) < 1e-14
        assert reldif(r(Var),  43.70779220779221) < 1e-14
        assert r(sd)==sqrt(r(Var))
end

use auto, clear
summarize mpg
rcof "check_if_foreign" ~= 0              /* verify test script works */

keep if foreign
summarize mpg
check_if_foreign

use auto, clear
summarize mpg if foreign
check_if_foreign
```

Many Stata test scripts read like this.

Simultaneous authoring

The writing of test scripts can actually speed development if you write the test script as you write the code. Most complicated commands do not come into existence all at once. You begin writing the command and, early on, it handles some simple problem while the parts of the code to handle other parts of the problem are still missing, then it handles more parts, and so on.

Write test scripts as each piece of the program is made to work. The advantage of this is that, as you delve deeper into the problem, you will sometimes need to rewrite portions of the code to make those parts more general. As the program grows in complexity, you can run your partial test script to verify that those portions still work. Moreover, when a piece fails certification, you know exactly where to look in the code to address the problem.

At StataCorp, most developers work with three windows open at once. In one window is the command being developed. In another is the test script. In the third is Stata where both the test script and the command can be run.

A test script for xttest3

Let us write a test script for `xttest3`, a program by Christopher F. Baum of Boston College. `xttest3` can be found in the Boston Archive. You can most easily find and install the program by typing "`findit xttest3`". (*Editor's note: unexpectedly and randomly,* `xttest3` *also appears in this issue of the Journal, see* Baum (2001).)

We are going to write the new test script `first/xttest3.do`. As we write the test script, we will test it by launching Stata, changing to the `bench/first` directory, and typing "`do xttest3`". Once we are satisfied with the script, we will add the line "`do xttest3`" to `first/test.do`, so that when we type "`do test`" from `bench/first` we run all the tests in the `first` subdirectory. When we type `do testall` from the `bench` directory, test script `xttest3.do` will be run automatically.

`xttest3` was chosen because the author provided a well-written help file that included a reference to a dataset from a published source, (Greene 2000, p. 598), which can be used to establish that `xttest3` gives correct answers. In the help file, the author gave as an example

```
. use http://fmwww.bc.edu/ec-p/data/greene2000/tbl15-1.dta,clear
. iis firm
. xtreg i f c if firm~=2, fe
. xttest3
```

and that, as a matter of fact, will become the basis of the first part of the test script. The first thing to do, however, is to type

```
. copy http://fmwww.bc.edu/ec-p/data/greene2000/tbl15-1.dta tbl15-1.dta
```

so that you have a copy of the dataset in the `first` subdirectory. This is a rule of test-script writing: test scripts should not refer to files outside of the directory in which the test script resides. They should not even refer to files in other subdirectories of the test suite. Each subdirectory of the test script forms an independent component, and all the materials for that component should appear in the directory along with the do-file. At StataCorp, this rule extends even to datasets shipped with Stata: if you want to use `auto.dta`, copy the file from the Stata directory into the test subdirectory. You never know when or how StataCorp might change the file, and you want your test script to continue working in the future.

I began `xttest3.do` as follows:

top of xttest3.do

```
cscript xttest3
which xttest3

use tbl15-1, clear
iis firm
quietly xtgls i f c, panels(heteroskedastic)
xttest3
```

end of xttest3.do

The published source indicated that `xttest3` should report a χ^2 value of 14,681.3 with 5 degrees of freedom. In fact, `xttest3` reported a value of 14,948.19. This difference was caused by `xttest3` using the FGLS estimate of the overall variance, whereas in the published example, the source used the OLS estimate. The source also noted that either variance estimate could be used and, one would suspect, the FGLS estimate should be more general. In addition, but of no importance in this case, `xttest3` obtained variance estimates by dividing by n, whereas the published source used $n-1$. Pencil-and-paper calculation certified the value of 14,948.19, and so the test script was modified to read

top of xttest3.do

```
cscript xttest3
which xttest3

use tbl15-1, clear
iis firm
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==5 & reldif(r(wald), 14948.19)<.000001
assert r(p)==chi2tail(r(df), r(wald))
```

end of xttest3.do

The help file that accompanied `xttest3` suggested typing

```
. xtgls i f c if firm~=2, panels(heteroskedastic)
. xttest3
```

and that was next added to the script, but twice. In the first rendition, added was

```
keep if firm~=2
xtgls i f c
xttest3
```

and the answer produced was declared to be correct under the assumption that if `xttest3` worked with one dataset in which the answer was carefully verified, it would work equally well with a similar dataset. Note, however, that I did not assume that `xttest3` could correctly restrict itself to the estimation subsample rather than using all the data in memory. In the code above, all observations are used. Then the answer just produced was used to certify

```
xtgls i f c if firm~=2, panels(heteroskedastic)
xttest3
```

The test script now read

top of xttest3.do

```
cscript xttest3
which xttest3

use tbl15-1, clear
iis firm
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==5 & reldif(r(wald), 14948.19)<.000001
assert r(p)==chi2tail(r(df), r(wald))

keep if firm~=2
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==4 & reldif(r(wald), 5903.67)<.00001
assert r(p)==chi2tail(r(df), r(wald))
use tbl15-1, clear
iis firm
quietly xtgls i f c if firm~=2, panels(heteroskedastic)
xttest3
assert r(df)==4 & reldif(r(wald), 5903.67)<.00001
assert r(p)==chi2tail(r(df), r(wald))
```

end of xttest3.do

This was an important test to add. While `xttest3` itself does not explicitly allow an `if` *exp*, it is implicitly a function of the estimation sample. It would be very easy for the programmer of `xttest3` to have not properly restricted the calculation to the estimation subsample. We have now verified that the `e(sample)` is indeed respected because we got the same result both ways.

Concerning the subsample, I also decided to test that `xttest3` would respond gracefully when the data had disappeared altogether. Interactively, I typed

```
. drop _all          (to drop the data)
. xtgls              (to verify that estimation results were still present)
. xttest3            (to see what would happen)
```

and discovered that `xttest3` reported "f not found"; r(111), which I viewed as satisfactory. I then added to the test

```
drop _all
xtgls       /* verify estimation results still present */
rcof "noisily xttest3" == 111
```

`xttest3` can be used after `xtgls` and `xtreg, fe`. I had no published results with which to compare, so on the side I ran a simulation of the test. In the process I discovered that the test performed very poorly when T is small relative to the number of panels but, as T increased, the coverage converged to what was expected. I then used some of the code from my simulation to add to the test script:

```
* test after xtreg, fe

* make dataset
clear
set seed 10394
set obs 50
gen id = _n
gen u1 = invnorm(uniform())
expand 120                  /* if T=20, test performs poorly */
gen u2 = invnorm(uniform())
corr2data x1 x2, mean(1 2) corr(1 .4  .4 1)
gen y = x1 + x2 + u1 + u2
quietly xtreg y x1 x2, fe i(id)

xttest3
assert r(df)==50 & reldif(r(wald), 60.95)<.0001
assert r(p)==chi2tail(r(df), r(wald))
```

In the above, I simulated a dataset under the null hypothesis. Note that I set the random-number seed so that, each time the script is run, I will get the same results. This then allowed me to assert the answer produced by `xttest3`.

That done, I added one more test to verify that `xttest3` would not run after `xtreg, re`. I knew that `xttest3` would run after `xtreg, fe`, and I was concerned that it would inappropriately run after any `xtreg` command. I added the lines

```
use tbl15-1, clear
iis firm
quietly xtreg i f c, re
rcof "noisily xttest3" == 301     /* last estimates not found */
```

Finally, this being an `xt` estimator, I wanted to try extreme cases: one example of there being one panel with lots of observations, and the other being lots of panels, each with one observation. Interactively, to see how `xttest3` would respond, I typed

```
. use tbl15-1, clear
. gen id1 = 1
. iis id1
. xtgls i f c, panel(hetero)
. xttest3
```

I discovered that in this case `xttest3` reported a χ^2 of zero but, when I did a `return list`, I discovered that the actual value was about 2e–30. Looking at the code, I could see how such a small number could arise.

I then tried the many panels, each with one observation case:

```
. gen id2 = _n
. iis id2
. xtgls i f c, panel(hetero)
. xttest3
```

`xttest3` again reported a χ^2 of zero and this time, `return list` reported that the result was exactly zero.

In any case, I was greatly reassured about the accuracy of `xttest3` by these two tests and added them to the test script:

```
use tbl15-1, clear
gen id1 = 1
gen id2 = _n
iis id1
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==1 & r(wald)<1e-29

iis id2
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==100 & r(wald)==0
```

The final test script read

top of xttest3.do

```
cscript xttest3
which xttest3

use tbl15-1, clear
iis firm
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==5 & reldif(r(wald), 14948.19)<.000001
assert r(p)==chi2tail(r(df), r(wald))

keep if firm~=2
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==4 & reldif(r(wald), 5903.67)<.00001
assert r(p)==chi2tail(r(df), r(wald))
use tbl15-1, clear
iis firm
quietly xtgls i f c if firm~=2, panels(heteroskedastic)
xttest3
assert r(df)==4 & reldif(r(wald), 5903.67)<.00001
assert r(p)==chi2tail(r(df), r(wald))

drop _all
xtgls       /* verify estimation results still present */
rcof "noisily xttest3" == 111

* test after xtreg, fe

* make dataset
clear
set seed 10394
set obs 50
gen id = _n
gen u1 = invnorm(uniform())
expand 120                  /* if T=20, test performs poorly */
gen u2 = invnorm(uniform())
corr2data x1 x2, mean(1 2) corr(1 .4  .4 1)
gen y = x1 + x2 + u1 + u2
quietly xtreg y x1 x2, fe i(id)
```

```
xttest3
assert r(df)==50 & reldif(r(wald), 60.95)<.0001
assert r(p)==chi2tail(r(df), r(wald))

* verify does not work after random-effects estimation
use tbl15-1, clear
iis firm
quietly xtreg i f c, re
rcof "noisily xttest3" == 301     /* last estimates not found */

* extreme cases
use tbl15-1, clear
gen id1 = 1
gen id2 = _n
iis id1
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==1 & r(wald)<1e-29

iis id2
quietly xtgls i f c, panels(heteroskedastic)
xttest3
assert r(df)==100 & r(wald)==0
```

end of xttest3.do

This is an adequate test script. Although it is long, the entire assembly time was about five minutes, ignoring the problems with the mismatch with the source (which took about half an hour to work out) and the side production of a simulation to verify coverage (another hour). Had I instead just done a pencil-and-paper calculation with a small dataset, I could have skipped the simulation, but since I was not familiar with this test, I wanted to do more than simply verify that a calculation had been carried out correctly.

4 References

Baum, C. F. 2001. Residual diagnostics for cross-section time series regression models. *The Stata Journal* 1: 100–103.

Greene, W. 2000. *Econometric Analysis*. Upper Saddle River, NJ: Prentice–Hall.

About the Author

William Gould is President of Stata Corporation and, more importantly, head of technical development.