Paper 003-2009

# The Science of Application Debugging

David A. Scocca, Rho, Inc., Chapel Hill, NC

## ABSTRACT

An old maxim holds that every nontrivial computer program has bugs. As a developer, you want to minimize the number and severity of the bugs that appear in your released application. When a bug is reported, you need to be able to identify and resolve it as efficiently as possible.

Effective debugging requires a scientific approach and an understanding of the tools available to help you identify, understand, and solve the problems that cause bugs.

## INTRODUCTION

### WHAT IS A BUG?

First, what is a “bug”? A short answer would be that a bug is when an application does not behave as expected. The obvious question: who is doing the expecting?

The users running the application are the ones most likely to see their expectations violated directly. But stakeholders like the developer, the author of the specification, the user’s managers, and the application’s “customers” all have expectations which the application may fail to meet. In fact, it’s possible (and maybe even likely) that different users or stakeholders have different and incompatible expectations.

At one extreme, a “Joel on Software” article argues that every call to technical support can be treated as evidence of a bug (Spolsky 2001). Whenever you provide customer support, it means your application has failed to allow the user to get the job done without your assistance. The issue may be a failure of your documentation—rather than what you think of as “the application”—but resolving it can improve your user’s experience and avoid repeating the support request.

A bug can also be an issue of performance: the application does exactly what is expected, but is slow enough to limit user productivity or consumes so many resources that other users, processes, or applications suffer.

### ELIMINATING BUGS

If we believe the old maxim that every nontrivial computer program has bugs, then completely eliminating bugs is impossible. (The corollary to the maxim is that any bug-free computer program is trivial.) What we want to do, first and foremost, is reduce the impact of the bugs on the application’s users and customers. This requires prioritizing bug handling: things that affect many users or present risks of data loss need a higher priority than minor interface issues or conditions for which an obvious work-around is available.

Reducing bugs is always a goal, but avoid giving bug counts too big a role in driving your procedures. There are two potential pitfalls: first, basing programmer rewards on bug count metrics can lead to more time being spent arguing about what “counts as a bug” than on actually improving the application. Second, offering “bug bounties” for testers can produce an underground economy that produces many “fixed bugs” without moving application development forward.

How aggressive should you be in fixing bugs? That depends in part on the nature of the application. If your application is sold for external use or deployed through the web to a wide range of users, you will need to make it “bulletproof” against a relatively wide variety of user expectations and computer environments.

If you are instead developing an application for internal use, your company has much greater control over the user environment and can establish a standard configuration on which your application can depend. You can also use your training programs to set user expectations and procedures.

There are limits to the ability of training to reduce conflicts between user expectations and program behavior. For example, we once encountered limitations in SAS® that prevented a SAS/AF® application from properly saving data when users quit the application by clicking on the main SAS close box from a data entry screen. While attempting to develop a work-around, we asked users to avoid exiting via the close box: this request proved an almost total failure, as the close box habit is too deeply ingrained to be trained around. (It was necessary to disable the close box entirely with the AWSCONTROL system option to prevent the loss of data.)

### FOCUS

The focus of this paper will be on general principles and strategies for application debugging and on tools and tactics that are useful in determining why an application misbehaves and how to fix it. I will generally be assuming that SAS at least parses your code and compiles your application successfully; when compilation fails, papers from Frank Dilorio (2001) and Peter Knapp (2004) provide a good resource for debugging syntax problems.

## PRINCIPLES OF DEBUGGING

### SOFTWARE IS DETERMINISTIC

That software is deterministic is, I believe, the first principle to remember when attacking truly difficult bugs. Unless you are intentionally introducing randomness, providing the software with the same inputs in the same circumstances should always produce the same outputs. If a user is encountering a bug in your application, you should be able to replicate it using the same data and the same version of the application.

If you are getting inconsistent behavior from your application with a bug manifesting itself only intermittently, there are three possibilities:

First, the inputs may not actually the same. If the user is not doing exactly the same thing each time, or if the starting data is not identical, then the application behavior and results may differ.

Second, the circumstances or environment may not be consistent. If the application accesses a data server, the server load will vary and the server may be intermittently unavailable. The operating system version, the installed SAS hot fixes, and the version of other applications such as Microsoft Office may vary across user workstations.

Finally, the problem may be outside your software. Because software is deterministic, a truly intermittent problem isn't in the software. Rather, it is likely an issue in the computer or network hardware or synchronization, or within the operating system layers that interact with the hardware. (I am assuming here that you are not writing an operating system or low-level drivers in SAS.)

Particularly if you are faced with a probabilistic set of failure conditions, where an identical job can run successfully dozens or hundreds of times and then fail without anything apparently having changed, it is more likely that the problem lies in the underlying operating system, drivers, and hardware than in your program.

Of course, this conclusion is unsatisfying. While it means you aren't "at fault", it also leaves you without a clear way to resolve the problem. This is where additional testing and debugging effort is required to see if you can make your application work around the underlying error.

For example, we encountered an operating system and network storage interaction that caused a failure a very small fraction of the time when writing SAS permanent datasets. Most of the time this was detected by the application and could be worked around by checking for the existence of the new dataset. But in cases where the failure occurred during the in-place sort of a dataset, it lost the data and caused both the application and the data server to crash. The work-around was to eliminate in-place sorting, allowing the application to check for the successful creation of a sorted copy of the dataset before modifying the existing dataset.

### SOFTWARE DOES NOT RUST

This is a variation on the previous principle. Left unattended, software does not rust or decay. If a new problem arises—if a user finds that your application misbehaves today in a way it did not misbehave yesterday or last week—then the key to solving the problem lies in something that has changed in the interval. This is particularly important to recognize when the application has been behaving correctly without modification for a long period of time.

In cases where network settings have been changed, or a new file server or data server has been deployed, such changes can cause an application that works normally to encounter errors or performance issues. While this could reflect an incorrect implementation of the network or server change, it is also possible that a change revealed an underlying bug in the application.

For example, consider the case where one component of your application opens files without properly closing them, and a new replacement server is set up with a lower limit on the number of simultaneous open files than was in place on the old server. While the server change would cause the user to see an error message than had not previously appeared, the underlying bug is in fact in the application code even though it ran on the old server without presenting an error message.

Another possibility is that the amount or variety of data has grown and has passed some threshold beyond which the application no longer behaves properly. We find that maybe 80 percent of our application issues are found while working on about ten percent of our clinical trials—specifically the ones that push the capabilities of the application farthest beyond what has been previously done.

Because you can never anticipate everything that production data will contain, it is important to be able to test and debug using a copy of the real data your application reads.

### DEBUGGING IS A SCIENCE

What do I mean by saying debugging is a “science”?  As a biochemist of my long acquaintance was known to tell his graduate students (Scocca, 2008):

**Science is the art of changing one thing at a time.**

You start by observing some phenomenon in the world—in the case of debugging, that will be your application’s misbehavior. You look at it in the light of a theory, or a model of how the world works; for an application, that would be your understanding of what the code is and what it is supposed to be doing.

Based on that model, you develop a hypothesis that explains the phenomenon: what part of the application might be responsible for the behavior you are observing? What parts of the application or the supporting environment might not be responsible? And given your hypothesis, what one thing can you change that would allow you reject the hypothesis?

Many simple hypotheses about your application’s misbehavior can be easily tested and either supported or refuted. Is the misbehavior specific to a particular user, or to a particular workstation? Simple attempts to replicate the problem, by yourself or on your machine, can be informative and allow you to rule out many possibilities.

| Simple Hypothesis | Simple Test |
|---|---|
| Issue is related to user profile | Does another user encounter the problem? |
| Issue is related to workstation configuration | Does the user encounter the problem on another machine? |
| Issue is related to performance of data server | Does the problem occur if a copy of the data is accessed from the local drive instead? |

**Table 1: Simple Hypotheses and Tests**

If testing allows you to reject one or more of these simple hypotheses, you can focus your investigation more narrowly on your application code.

For a software developer, the nice thing about experimentation is that it can be fast. The experimental loop: modifying your code or data, re-compiling any changed code, and running the application to see the results can take a matter of minutes.

While scientific journals most frequently publish results that support interesting hypotheses, in debugging the ability to reject hypotheses is particularly useful. It allows you to simplify the problem you are facing. If you can remove variables specific to a user, a workstation, or a data server from consideration, then you can focus more directly on your application code and on the data used.

## DEBUGGING STRATEGY

The full debugging strategy outlined here will allow you to attack complicated or difficult bugs. Simple bugs like fixing a typing error in the text of a user dialog may have obvious solutions—but it is possible to fix a typing error and then discover that the problem persists because you changed a dialog that was similar but not identical to the one in which the user encountered the original error.

## REPLICATE THE PROBLEM

The first step is to replicate the problem the user experienced. If the problem cannot be replicated, it may be related to a server load or multiple users contending for access to the same resources. You may need to observe the problem at the user's machine, either in person or using a network access tool like Virtual Network Computing (VNC).

Often it will be necessary to copy the data that was in use when the error was encountered, and to replicate the issue using the copy of the data. This will allow you to perform experiments that require modifying the data or the application.

This is particularly important if you are running in a validated environment, as when dealing with clinical trial data. If your application is validated, your debugging process cannot involve modifying production data or deploying a modified version of the application. A validated application will usually have a defined formal change process, and deploying a change that fixes a bug in your code will require developing a test case to establish that the change is properly implemented and meets the requirement defined in the change request.

## OBSERVE THE SITUATION

Once you can replicate the problem—or, failing that, if you can watch a user who is encountering it—observe as much information as you can about both the situation under which the issue arises and the application's exact behavior. Check the SAS log and inspect the code that you believe to have been running when the error occurred.

It may be important to **see** the problem in action. Sometimes you can observe something that you had not thought to ask about, for example a field that should be displaying a data value instead appears blank. When the system generates multiple error messages, sometimes users will only report the last one, or the one they feel is most meaningful. By carefully observing the situation in which the error occurs, you can find additional clues that will help you track down the problem.

## DEVELOP A HYPOTHESIS

Looking at the evidence, think about what might have caused the application's unwanted behavior. This step is less difficult than you might expect: if you are intimately familiar with the construction of the application and the algorithms it uses, it may be obvious that a particular form of misbehavior can be caused by only one thing. There are many bugs for which a clear articulation of the exact problem effectively gives the developer precise directions to the location and nature of the error.

## EXPERIMENT

Once you have a hypothesis about the cause of the bug, make **one** change to the application or data—ideally a change that will prevent the problem from occurring—and repeat your earlier attempt to replicate the issue. If the bug is gone, this supports your hypothesis about the cause of the bug.

You can also experiment by making changes which you expect will **not** alleviate the problem. If you feel you need more information about the state of the application, you can add lines to write status messages and values to the SAS log; you would obviously not expect this logging to prevent the error, but when you replicate the error you will be able to more fully observe what your code is doing.

You can also make changes to eliminate parts of the application code that you hypothesize are **not** involved with the problem. If you can remove portions of the code and still replicate the problem, then you can focus on the remaining code in your search for a solution.

Your experiment should be able to either support or refute your hypothesis. If your hypothesis is supported, then either you have eliminated the bug or have successfully narrowed the field of inquiry. If the experiment leads you to reject your hypothesis, then use the results you observe to modify your model of the way the application is working and to develop a new or modified hypothesis on which to base another experiment.

Sometimes the process of tracking down the bug will lead you to the point where you have identified a particular line or small block of code as being at fault—the code clearly causes the error—but where there is no obvious way to make your application do what it is supposed to without encountering the issue. This could result from a case where either the user's actions or the data violate your assumptions, or a SAS operation that behaves in a way inconsistent with your expectations about how it works.

### A STRATEGY: TEAR IT DOWN

If you are faced with a difficult bug in a large, complex application in which a lot of different things are going on, a good first step is to cut the application back to see how little of its framework is actually necessary to reproduce the issue. Among the things you could eliminate:

- data server access (by reading data files directly rather than through a server)
- code that runs after the point where the error is encountered
- code you test and find not relevant to the process involved in the area
- any application-wide frameworks or services that are unrelated to the process you are studying

In dealing with an application written in Base SAS, you should know what data you expect to exist before and after every step. Ideally, you can find one DATA step or one procedure that has the expected input but fails to produce the expected output. If the block of code is inside a macro, create the necessary starting conditions and strip away the macro code. Focusing on the smallest block of code that exhibits the problem will make it easier to understand the cause and either correct the error or develop a work-around for the situation.

Once you have identified a small chunk of code that seems to be responsible for the problem, it is worth trying to copy that code and its minimum prerequisites into a new program or an empty application framework to see if the problem still appears. If you can reproduce the issue in a small program that runs outside your main application, you greatly reduce the potential interactions and complicating factors.

This is also useful when you seek help from SAS technical support. The smaller the program you need to send to SAS, and the simpler the setup required, the easier it is for the support team to understand and test your code.

### A STRATEGY: BUILD IT UP

Another strategy is to start not with code that encounters the problem but instead with a stripped-down version of the application that runs without errors. This stripped-down application may not contain the functionality that encounters the bug, or may display a trivial screen with no actual content. Bit by bit, you can add back in the code to restore the functionality and see at what point the bug re-appears.

## DEBUGGING TOOLS

### DESIGN FOR DEBUGGING

Decisions you make when designing the basic framework of your application can be rewarding when you need to track down a bug. Here are a few examples of design strategies that can make your debugging process easier.

- For a particular dataset or a set of similar datasets, isolate access to that data in a single object or code module. This will allow you to know specifically which code to examine when there is a problem related to reading or writing that data.
- Avoid code duplication. Duplicate code often means that you have to spend time tracking down multiple occurrences of what is essentially the same bug.
- Handle run-time exceptions. An exception-handling framework (Scocca, 2004) allows you to control the flow of the application when an error is encountered and to log information about the cause of the error.

In a well-designed application, it is often the case that when the user encounters an error, their description of the error will be enough to point you to specific small section of code rather than requiring you to look in many different places.

### THE SAS LOG

The SAS log is great resource for debugging, not so much for what SAS puts in it as because of the ways you can use it to easily record application state, user choices, and other relevant information.

Among the things worth recording in the SAS log are:

- Return codes from operations that are unsuccessful, and any associated error messages.
- The location of data libraries that are assigned, particularly if the data location is dependent on the user's choice.
- User choices. If they user says they did X and the log says they did Y, and the output is consistent with Y, then you have a user relations problem and the solution is not in your code (though it might be in your documentation or your user interface).
- Trackback listings of the call stack when an exception is encountered.
- The values of any variables or lists that you suspect might be involved in a problem.

If your application stores the SAS log in a file and checks that file for errors, it is also useful to set up a way to write your own error or warning messages to the log and have these messages detected by your log-checking code. (A simple way is to begin the log messages you want detected for the application "MyApp" with the words "MYAPP WARNING" or "MYAPP ERROR", and to have your log-checking code look for these specific indicators.)

If you are doing any sort of SAS log redirection, or if your bug results in a system halt that prevents you from reviewing the log, the SAS system option ALTLOG (short for "alternate log") can be exceptionally useful. The ALTLOG option, which is specified at SAS startup, designates a file which will contain a copy of the SAS log contents for the entire session. If a crash or redirection error prevents you from seeing the log window, or if your application is unable to write its normal log files, the ALTLOG file provides you with another source for the logged information.

Log redirection using the PRINTTO procedure or the SAS-provided %OUT2HTM macro does not affect the echoing of SAS log information to the ALTLOG file.

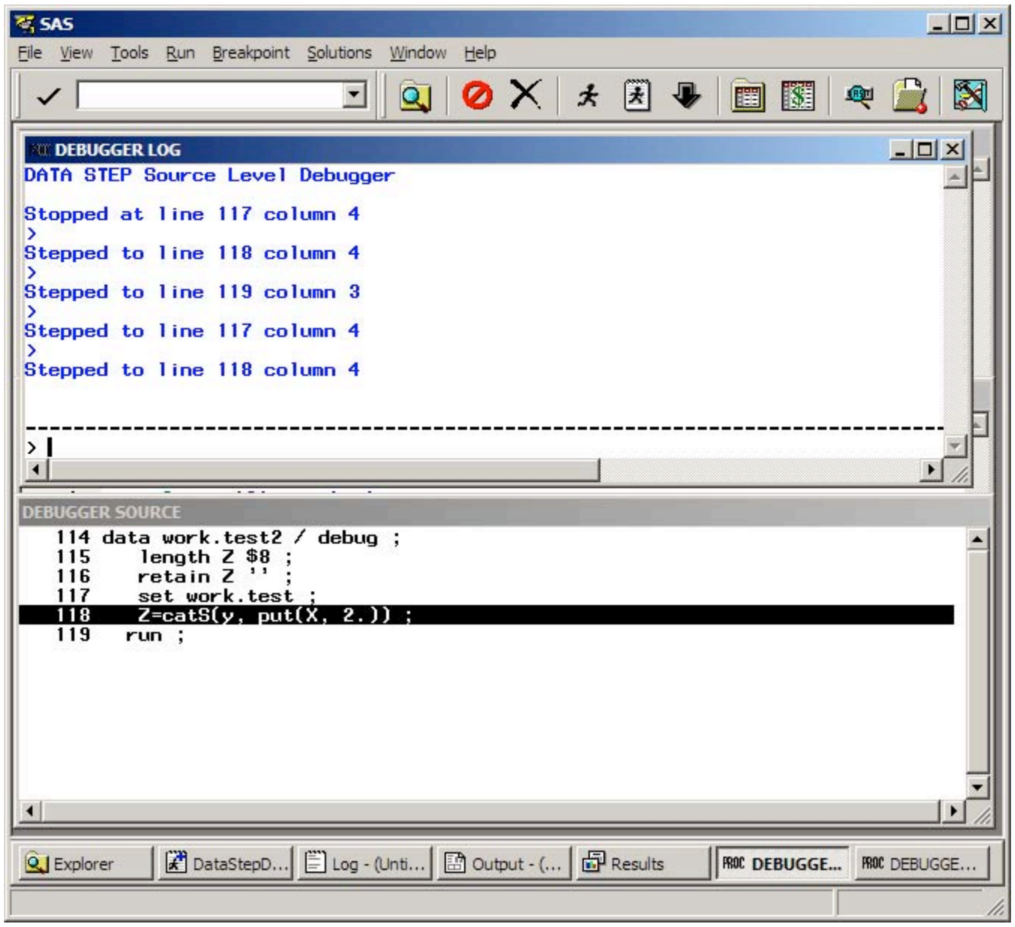
## STEPWISE DEBUGGERS

A stepwise "debugger" facility allows you to run your program one statement at a time; you can view the code and check the current value of variables as it runs. SAS has two built-in debuggers, one for data step code and one for SAS Component Language (SCL) programs. In either case, if the SAS log is being redirected away from the log window, some of the debugger output may not appear in the window as expected.

The data step debugger (Riba, 2000) is invoked by simply adding the DEBUG option to your DATA statement.

```
      data dataset-name / debug ;
```

When the data step runs, SAS opens two additional windows: the debugger source window shows the code for the data step with the current line highlighted; the debugger log window shows the output generated by the debugger and provides a line on which you can enter commands to control execution or display values.



```
DATA STEP Source Level Debugger

Stopped at line 117 column 4
>
Stepped to line 118 column 4
>
Stepped to line 119 column 3
>
Stepped to line 117 column 4
>
Stepped to line 118 column 4
```

```
114 data work.test2 / debug ;
115    length Z $8 ;
116    retain Z '' ;
117    set work.test ;
118    Z=catS(y, put(X, 2.)) ;
119   run ;
```

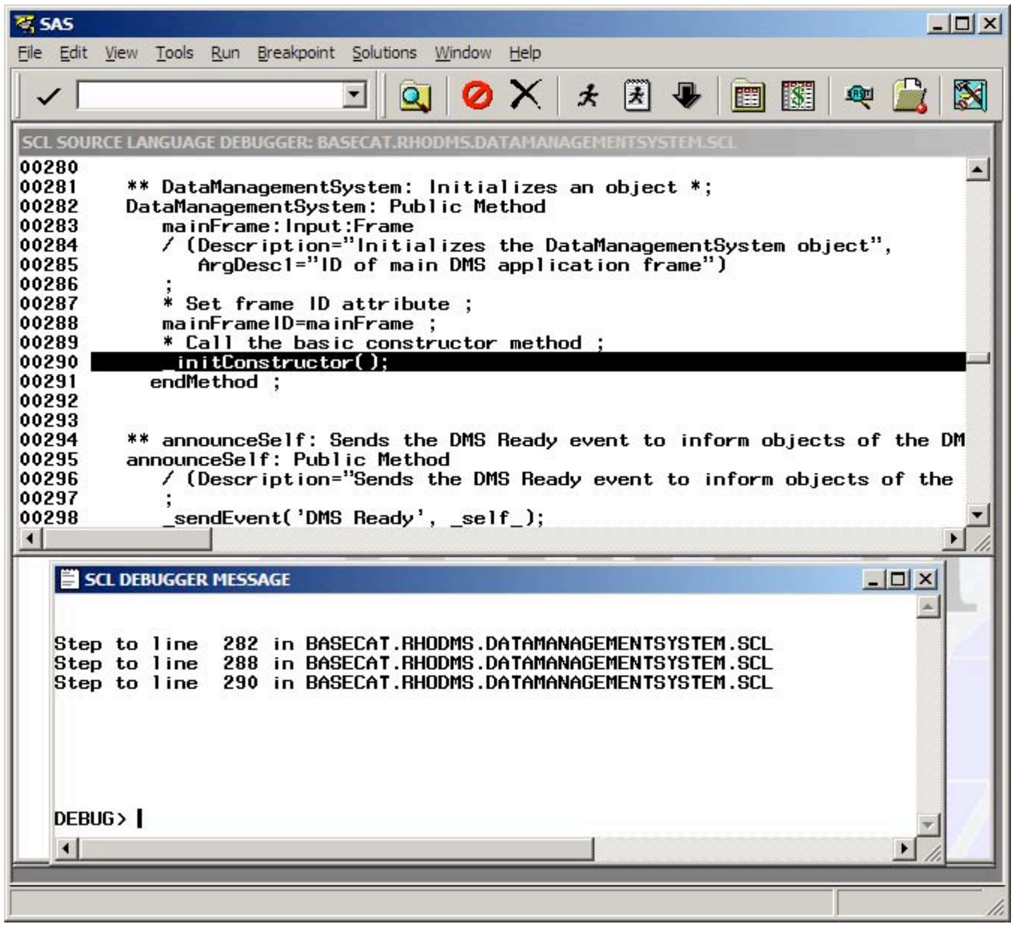
**Figure 1: The DATA Step Debugger**

Note that the DATA step debugger will walk through your code once for each iteration of the data step loop. If you are working with a large dataset, you may want to create a subset of the data to examine more closely using the debugger.

Applications written in SAS Component Language have access to the SCL debugger, which is enabled with the DEBUG command or with a toolbar button:

**Figure 2: The DEBUG Toolbar Button**

To step through a piece of SCL code, the debugger must be enabled both when you compile the code and when you run the application. By limiting the entries you compile with the debugger enabled, you can step through just the portion of the code you need to examine most closely.


SAS
File Edit View Tools Run Breakpoint Solutions Window Help
SCL SOURCE LANGUAGE DEBUGGER: BASECAT.RHODMS.DATAMANAGEMENTSYSTEM.SCL
SCL DEBUGGER MESSAGE

```
00280
00281    ** DataManagementSystem: Initializes an object *;
00282    DataManagementSystem: Public Method
00283       mainFrame:Input:Frame
00284       / (Description="Initializes the DataManagementSystem object",
00285          ArgDescl="ID of main DMS application frame")
00286       ;
00287       * Set frame ID attribute ;
00288       mainFrameID=mainFrame ;
00289       * Call the basic constructor method ;
00290       _initConstructor();
00291      endMethod ;
00292
00293
00294    ** announceSelf: Sends the DMS Ready event to inform objects of the DM
00295    announceSelf: Public Method
00296       / (Description="Sends the DMS Ready event to inform objects of the
00297       ;
00298       _sendEvent('DMS Ready', _self_);
```

```
Step to line  282  in BASECAT.RHODMS.DATAMANAGEMENTSYSTEM.SCL
Step to line  288  in BASECAT.RHODMS.DATAMANAGEMENTSYSTEM.SCL
Step to line  290  in BASECAT.RHODMS.DATAMANAGEMENTSYSTEM.SCL


DEBUG>
```

**Figure 3: The SCL Debugger**

## SCL PERFORMANCE ANALYZER

The SCL Performance Analyzer is a built-in tool for looking at your SAS/AF application and understanding its performance: what entries and programs it spends the most time running. This is useful to understand how your application is running, and can tell you which lines of code are taking the most time to run. More information is available in the SAS documentation and in a paper by Jeff Lessenberry (2000).

While the SCL Performance Analyzer is useful for finding some performance issues, it also adds overhead at execution time: for particularly long-running or complex processes, or when in a complex application, simple timing measures with PUT statements may be easier than using the entire SCL Performance Analyzer framework.

## TESTING AND DEBUGGING FRAMEWORKS

Our application testing framework is also very useful for application debugging. For each test case, we have a folder containing an initial set of study data and a batch script that copies the data and invokes the application pointing to the fresh copy. This has several uses in the debugging process as well.

First, at any time, we can easily build a “test case” around a snapshot of the current production data from a study. This allows troubleshooting from a copy of the same data on which the users have experienced the problem.

Then, we can use the test framework to perform any action, even those that modify data, and restart the test framework to restore the data to the initially copied state.

We can modify the copy of the application run by the test framework and then easily run the modified application against a fresh copy of the snapshot data.

If the test data needs to be modified, we can use either SAS programs or the application itself to make changes and then copy the modified data back to our test case.

## MACHINE VIRTUALIZATION

Sometimes you will find yourself unable to replicate for yourself a problem that is experienced by multiple users and on multiple workstations. The answer “it works on MY machine” is unfortunately not a resolution unless either you plan to give the user your machine or you can identify and resolve the relevant differences between your machine and the user’s machine.

Your developer’s workstation is likely configured in ways your users’ machines are not. Virtualization software such as VMWare lets you set up a “virtual machine” that can exactly replicate the environment of a user workstation and user account—ideally, it can use the same disk image your IT department sets up for new user workstations.

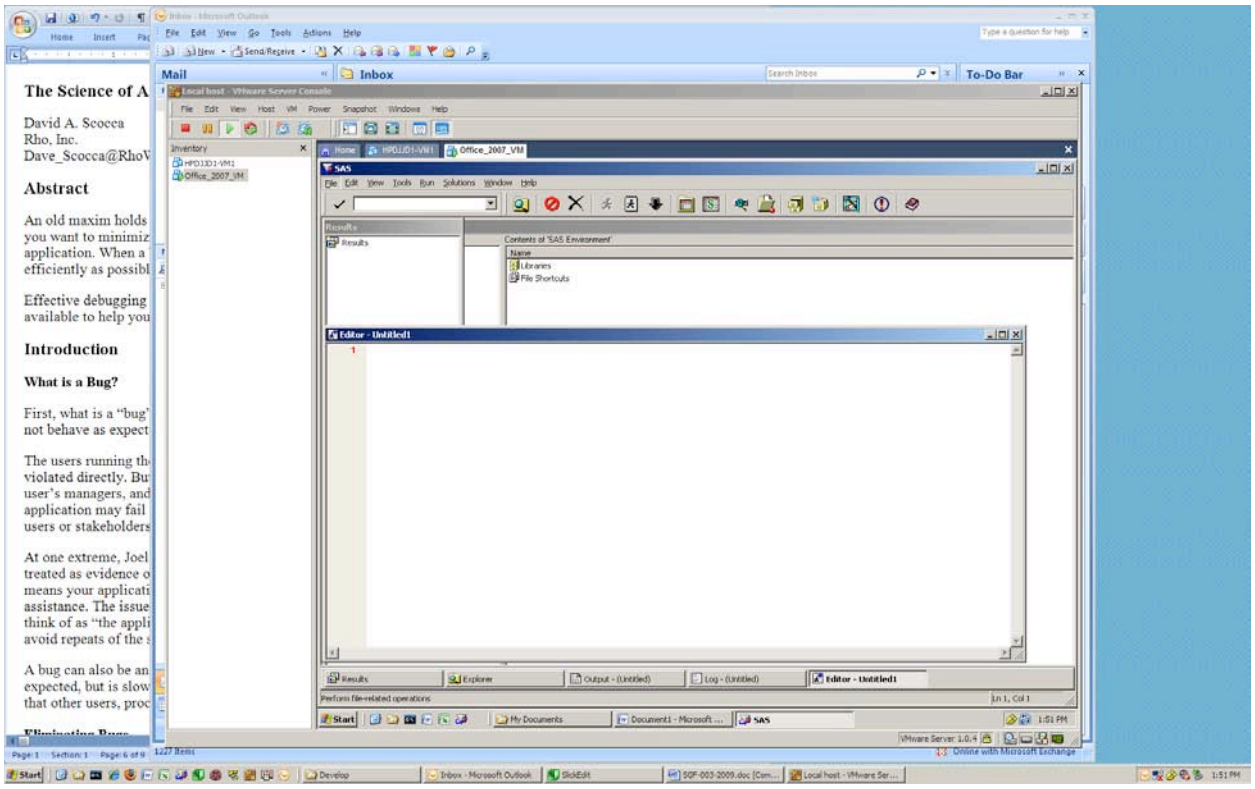


**Figure 4: A SAS Session Running in a Virtual Machine**

A virtual environment can allow you to access your usual development tools and settings while still being able to run tests in an environment identical to that of a regular user.

## DEBUGGING MODE

Since more information makes debugging easier, many applications are built on a framework that allows a “debugging mode” to be turned off or on at run-time. The advantage of such a mode is that you can turn it on to generate more information about the status and operation of the application without having to actually modify the application code and run a new version.

Some things you might want to do when a debugging mode is enabled include:

- Write additional application state to the SAS log.
- Avoid deleting temporary datasets after using them, so you can examine them after a routine has finished running.
- Prevent normal log redirection, so you can use a stepwise debugger.

- Disable automatic printing of output files to save time and printing costs during your testing.
- Either disable email notifications or re-direct them to a test account to avoid sending test messages to real users.

The danger of having a debugging mode is the possibility of setting up a "heisenbug"—a bug that behaves differently or goes away when you try to study it. For example, if you skip the routine that deletes a temporary dataset after you have used it, the dataset may then be present the next time you attempt the operation and the application might behave differently because of the existing dataset. This can be avoided by having your application—whether in debugging mode or not—delete any pre-existing copies of temporary tables before starting a process that will create the tables.

## CONCLUSION

Debugging can be difficult. No approach to debugging will either eliminate all of your bugs or will guarantee that the issues you are troubleshooting will be easy to find or fix. While not a miracle cure, a scientific approach to debugging will let you make the most efficient use of your debugging time and effort. A well-thought-out debugging strategy and a skilled use of the available tools will help you focus your investigation more quickly on the appropriate components of your application.

## REFERENCES

- DiIorio, Frank. 2001. "The SAS Debugging Primer". *Proceedings of the 26th Annual SAS Users Group International Conference*, Long Beach, 54-26.
- Knapp, Peter. 2004. "Debugging 101". *Proceedings of the 29th Annual SAS Users Group International Conference*, Montréal, 257-29.
- Lessenberry, Jeff. 2000. "Uncorking SAS/AF® Bottlenecks with the SCL Dynamic Performance Analyzer". *Proceedings of the 25th Annual SAS Users Group International Conference*, Indianapolis, 11-25.
- Riba, S. David. 2000. "How to Use the Data Step Debugger". *Proceedings of the 25th Annual SAS Users Group International Conference*, Indianapolis, 52-25.
- Scocca, David A. 2004. "Building a More Robust SAS® Application". *Proceedings of the 29th Annual SAS Users Group International Conference*, Montréal, 038-29.
- Scocca, John J. 2008. Personal communication.
- Spolsky, Joel. 2001. "Hard Assed Bug Fixin'". *Joel on Software*. <http://www.joelonsoftware.com/articles/fog0000000014.html> (February 9, 2009)

## ACKNOWLEDGMENTS

Many thanks are due to all who have helped me track down bugs over the years, including the past and present members of the Software Development and Information Technology departments at Rho, Inc., and the extremely helpful folks at SAS who support SAS/AF and SCL. Any bugs that you may find in this paper are entirely my own.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David A. Scocca
Rho, Inc.
6330 Quadrangle Drive, Suite 500
Chapel Hill, NC 27517
E-mail: Dave_Scocca@RhoWorld.com