IBM® Application Testing Collection for MVS/ESA™
Version 1 Release 5 Modification 0
General Information
Program Number: 5799-GBN
PRPQ: P85579

April 1999

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

This book is also available as an online book that can be viewed with:

- The IBM® BookManager® READ and IBM Library Reader™ licensed programs. The IBM Library Reader is available for downloading from the following Web site:
  http://booksrv2.raleigh.ibm.com/homepage/ilrserv.html
- Adobe Acrobat Reader 2.1 and later, which is available for downloading from the Adobe Web site.

Sixth Edition (April 1999)

This edition applies to Version 1 Release 5 Modification 0 of the IBM Application Testing Collection for MVS/ESA™, program number 5799-GBN. Changes are made periodically to the information herein.

This publication is available in downloadable form from the IBM Year 2000 Technical Support Center Web site:
http://www.software.ibm.com/year2000/
Select the **Testing** link on the main home page and then look for the Application Testing Collection on the page displayed.

Table of Contents

Figures

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (1) the exchange of information between independently created programs and other programs (including this one) and (2) the mutual use of the information which has been exchanged, should contact the IBM Corporation, Department J01, 555 Bailey Avenue, San Jose, CA 95161-9023. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM
BookManager
CICS
DB2
IMS
Library Reader
MVS
MVS/ESA
OS/390
VisualAge

Other company, product, and service names may be trademarks or service marks of others.

About This Book

This book provides an overview and general information about the IBM Application Testing Collection for MVS/ESA. The Application Testing Collection (ATC) is a general purpose application development (AD) test tool suite. However, the test technology it offers also has specific applicability to the Year 2000 problem.

This book provides a brief review of the Year 2000 problem. It also addresses basic testing methodology, specifically issues that make the Year 2000 testing problem unique from more traditional AD testing methodologies.

Finally, the book provides a high-to-medium level view of the ATC tool set. The information and examples presented illustrate how these tools can be used in real-world situations involving Year 2000 testing projects and in ongoing development and maintenance testing projects as well.

Who Should Read This Book

You should read this book if you:

- Are involved in the selection of testing tools to be used in Year 2000 and/or traditional application development testing activities
- Are planning or involved with planning a Year 2000 remediation project that will involve testing of remediated applications
- Will be testing reengineered applications
- Will be testing applications on MVS™.

Conventions and Terminology Used in This Book

The following list shows special ways in which some characters and words are displayed in this manual and describes the meaning associated with each one:

Display Method	Meaning
`Monospaced type`	Shows something that you type (such as a command), an example, or something that is displayed on your monitor (for example, an error message, or the name of a panel or field).
Italic type	Indicates information that you supply (such as a parameter or a variable), or italic type can indicate a new term. For definitions of new terms, see “Glossary” on page 63.
Bold type	Indicates information that you should pay particular attention to.

Related Information

The following publications are available in downloadable form from the IBM Year 2000 Technical Support Center Web site:

- *IBM Application Testing Collection for MVS/ESA Version 1 Release 5 Modification 0 User's Guide,* Program Number 5799-GBN, PRPQ P85579
- *IBM Automated Regression Testing Tool Version 2 Release 2 Modification 0 User's Guide,* Program Number 5799-GBN, PRPQ P85579

To locate these documents on the Internet:

1. Go to http://www.software.ibm.com/year2000/
2. Select the **Testing** link on the main home page, and then look for the Application Testing Collection on the page displayed.

IBM has developed the following Redbook that outlines the Y2K test process and augments it with real-world application examples and tool usage information:

VisualAge 2000 Test Solution: Testing Your Year 2000 Conversion, SG24-2230

To order this Redbook from the Internet:

1. Go to http://www.redbooks.ibm.com/index.html
2. Select the **Redbooks Online!** button at the bottom of the page.
3. In the search field displayed, enter the Redbook's title or document number.
4. Look for ordering instructions on the page displayed.

Summary of Changes (V1R5M0)

| Changes have been made to the following components of the Application Testing
| Collection (ATC) for Version 1 Release 5 Modification 0:

| Coverage Assistant (CA)

| The performance of CA targeted summary has been improved and a batch inter-
| face has been added.

| Coverage Assistant (CA), Distillation Assistant (DA), and Unit Test | Assistant (UTA)

| The SETUP process has been updated to allow load modules to be instrumented
| directly as an alternative to instrumenting object modules, which are then linked into
| load modules.

| Automated Regression Testing Tool (ARTT)

| Support for Capture mode and Virtual Replay mode has been added for CICS,
| DB2, and IMS.

Summary of Changes (V1R4M0)

Changes have been made to the following components of the Application Testing Collection (ATC) for Version 1 Release 4 Modification 0:

Coverage Assistant (CA)

CA now supports IBM High Level Assembler (HLASM) Release 1 Version 3.

Unit Test Assistant (UTA)

File warping is the modification of variables (typically date variables) in program input files to simulate input conditions for testing. For example, dates in input files can be modified to post-year 2000 values to test Y2K remediated programs.

The new UTA file warp feature can copy any QSAM or VSAM file and warp fields in the copied file for testing. Any zoned or packed numeric field can be incremented, decremented, or set. Any zoned, packed, or character field can be set to a common value. For example, file warping could be used to clear fields in test copies of production input files for privacy or security reasons.

Source Audit Assistant (SAA)

SAA now supports IBM High Level Assembler Release 1 Version 3.

Automated Regression Testing Tool (ARTT)

ARTT, an automated capture and verification tool, has been added to the collection of tools.

Summary of Changes (V1R3M4)

Changes have been made to the following components of the Application Testing Collection (ATC) for Version 1 Release 3 Modification 4:

Coverage Assistant (CA), Unit Test Assistant (UTA), and Distillation Assistant (DA)

- Support has been added for DBCS characters in the input compiler and assembler listings, and in identifiers and comments in the control files.
- Support has been added for the COBOL LANGUAGE(JAPANESE) compiler option.
- The start monitor job now detects when another session has already been started with a matching BRKTAB/listing/obj combination (error message CMD5023W).

Source Audit Assistant (SAA)

- Support for DBCS characters in the input source/listings has been added for the Comments and Declares filters for COBOL and assembler.
- Restrictions on the length of input data set names have been removed.
- Support for DBCS characters has been added to the SAA postprocessor.
- The seed list data set is no longer required in the SAA postprocessor; however, if it is not specified, a change validation report is not created.

Summary of Changes (V1R3M0)

Changes have been made to the following components of the Application Testing Collection (ATC) for Version 1 Release 3 Modification 0:

Unit Test Assistant (UTA)

ATC now includes the UTA tool, which allows you to capture and log the values assigned to selected variables in your application programs at selected points during their execution. This is called *unit testing*. Unit testing allows you to confirm the effectiveness of Year 2000 changes that have been made to an application program.

In addition, Unit Test Assistant offers the ability to perform data *warping*. This means that variables can be modified automatically as they are encountered during program execution. UTA will intercept data entering or leaving a program at I/O time (or at other times where application logic dictates) and change the value of that data in a manner that you specify. This feature is especially useful when doing internal *white box* testing of Year 2000 dates.

ATC User Documentation

The following changes pertain to the ATC user documentation. Information that has changed since the last publication of the *User's Guide* and the *General Information* document is marked with revision bars in the left margins of affected pages.

- New information about DBCS support has been added to the *User's Guide* and the *General Information* document.
- Changes to the *User's Guide* and the *General Information* document have been made to reflect functional changes to the product. Also, minor technical and editorial changes have been made throughout both documents.
- A BookManager READ bookshelf, containing the *User's Guide* and the *General Information* document, has been added to the group of documentation offerings shipped with the ATC package.

Overview of the Year 2000 Problem

The Y2K “bug” is found in code that performs comparisons and/or arithmetic computations on data that represents years with two digits. Approaches to ridding code of the Y2K bug have included the following:

Expansion	Expanding all dates to four digits or storing the century value along with the year. At first glance this solution sounds simple, but it would require not only modifying the application code, but also restructuring the entire data environment, including backup data files.
Windowing	Windowing allows the data to remain in a two-digit format, but each location in the application code that carries out a comparison or arithmetic instruction on a date value is updated to add logic that interprets the 00 as a value greater than 99.
Compression	With this approach, the length of the date value remains at two bytes, but the value is stored in a data representation other than character.

Although a small number of information system (IS) shops have found that the expansion approach is effective, most shops today are adopting the windowing approach. Compression has yet to gain wide acceptance.

Primarily three factors distinguish Y2K code fixes from traditional code fixes:

- **The Function of the Code Is Not Changing.** Applications with code that has been fixed/remediated should work the same as they did before they were modified. New function is not being added.
- **The Interpretation of the Data Entering the Application Is Changing.** The year data entering the application from data files, system calls, and user-prompted input (terminal/screen input, for example) is changing from the logic base for which the application was designed.
- **The Time in Which You Have to Complete the Fix and Testing Is Shrinking.** Time is running out as you read this.

A testing approach that takes into account the unique challenges of the Y2K problem is essential. Such an approach generates the following testing requirements:

1. Testing activities must exercise all applied code changes.
2. All application programs that have been fixed/remediated must be able to accept future dates as input.
3. The previous requirements, 1 and 2, must be accomplished in the earliest phases of the testing process to beat the time limit. If, for instance, you schedule the testing of future dates as the last phase of testing, you may run out of time before you get there.

The IBM Application Testing Collection (ATC) is a set of testing tools designed to manage these unique requirements and increase your confidence that fixed/remediated code will perform correctly when handling post-1999 dates.

Introducing the IBM Application Testing Collection

The IBM Application Testing Collection (ATC) is a set of testing tools that can be used independently or in an integrated manner to address the unique nature of the Y2K testing requirements. The tools that make up the collection are:

Coverage Assistant
A code coverage tool that reports the percentage of coverage of a test suite, and the statements and branch conditions in the program that have not been exercised. An application can be measured by listing all source lines executed or by *targeting* the coverage to a select number of source lines or all lines affected by a set of specified variables.

Source Audit Assistant
A code comparison tool that compares source before and after conversion, identifying changes for audit purposes. The identified changed source can also be directed as input to other tools in the collection, such as Coverage Assistant's targeted summary.

Distillation Assistant
A data distillation tool that reduces a data set to the minimum size that provides test coverage that is equivalent to the test coverage provided by the original (larger) file.

Unit Test Assistant
A variable analysis tool that reads and logs variable values from a program during its execution for later examination. Also, values that are entering or exiting an application from or to data files, system calls, and user prompts can be incremented, decremented, or initialized to selected values at application run time. In addition to dynamically warping dates as they are encountered during program execution, UTA has a file warp feature that modifies copies of your input files according to warp definitions specified in the control file.

Automated Regression Testing Tool
A capture and verification tool that records a baseline execution of your program and automatically compares it with a proof execution using real or previously captured input. ARTT permits all levels of testing (unit, function, integration, and system) with or without a production system, and its data conversion capability allows testing to continue when I/O and programs are in incompatible formats (for example, when one has been modified to accept future dates and the other has not). Data conversion can be particularly useful if you are unsure of the status of I/O received from outside sources.

These tools can be used in conjunction with IBM's VisualAge® 2000 Test Solution. IBM's VisualAge 2000 Test Solution offers MVS COBOL, PL/I, and assembler users a process and a tool set to test Y2K changes to applications. Rather than testing to ensure that new enhancements to programs work correctly, Y2K testing focuses on ensuring that applications behave the same way they did before Y2K fixes were applied.

IBM has developed a *Redbook* that outlines the Y2K test process and augments it with real-world application examples and tool usage information. To order this Redbook from the Internet:

1. Go to http://www.redbooks.ibm.com/index.html
2. Select the **Redbooks Online!** button at the bottom of the page.
3. In the search field displayed, enter the following title or document number: *VisualAge 2000 Test Solution: Testing Your Year 2000 Conversion,* document number SG24-2230.
4. Look for ordering instructions on the page displayed.

The remainder of this chapter focuses on the ability of the ATC tools to address the Y2K unique testing requirements.

Year 2000 Testing Requirements

For a variety of reasons, companies are finding that their Y2K testing processes will have to be performed in the short amount of time remaining before applications are forced to handle post-1999 dates. Testing may never be performed on items scheduled to be tested later in the process because of this hard and fast cutoff. Therefore, it is imperative that the Y2K testing process:

- Provide a higher level of confidence than traditional testing practices that applications will perform correctly with post-1999 dates
- Provide that higher level of confidence earlier in the testing process than with more traditional testing practices (which typically have you wait until the later phases to do post-1999 date testing).
- Reduce test cycle times so that the chances of completing a full test plan are maximized
- Permit testing on isolated components
- Permit testing offsite without replicating complex environments.

Testing Methods and the Application Testing Collection

The concepts of *white box* and *black box* testing were first popularized in the early 1970s. These terms represent two very different, but complementary, methods of testing. One method provides limited information about an application's internal execution at run time; the other method offers detailed execution information. By combining these two methods, described in the paragraphs that follow, you can maximize your confidence that your applications will function correctly with post-1999 dates.

Black Box Testing

Black box testing is data driven testing; that is, an application is tested by examining its inputs and outputs. Little internal runtime knowledge of the application behavior is gathered. The application internals are viewed as a “black box” and only the application outputs as compared to its inputs provide information about the success or failure of a test case run. Some refer to this as testing the business functionality of the application.

For example, you could perform black box testing on an MVS batch application by saving the application's input and output files before code modification. These files serve as baseline files. After modification, rerun the saved baseline input through the modified program. Compare the new output to the baseline output. Any changes in the new output reflect either functional changes made to the application or errors that have been introduced into the code with the changes.

ATC Black Box Testing

On the surface black box testing sounds simple, but in practice it can be quite challenging. Take regression testing, for example. When a modified application is found to have introduced new defects and the function being implemented does not perform to specification, the code is corrected by development and later returned to the testing team. The test case that originally identified the failure is then run against the modified application. This method of saving and rerunning a test case suite to ensure that an application still functions according to specification after being changed is a black box method that many shops employ. If you are familiar with the process, you may also be familiar with the difficulties sometimes associated with this testing method:

- Many black box testing tools require you to shut down your system in order to take a snapshot of your data.
- A lot of time is needed to modify program logic, data, and in some cases, database schema.
- Many current tools also require increased DASD to hold multiple copies of data sets.
- Inefficient test cases require too much CPU time and take so long to execute that there is not enough time to complete testing.

ATC black box testing tools, however, simplify the process and decrease the time and resources required by most existing tools.

The following ATC tools supply black box results:

Automated Regression Testing Tool (ARTT)

ARTT gathers black box information by:

- Intercepting application I/O events and data
- Checking that all application I/O requests to file handlers and database managers are in the original sequence
- Building a detailed audit trail for a baseline execution or checking in real time a modified execution against a previously recorded baseline execution.
- Performing data conversion
- Performing data aging or rejuvenation
- Reporting differences between the baseline and proof runs.

Distillation Assistant (DA)

DA assists black box testing by:

Creating an efficient, cost-effective test bed. DA does this by reducing test data to the minimum number of records required to provide test coverage that is equivalent to the test coverage provided by the complete data. An efficient test case suite:

- Directly translates to a higher quality application portfolio for your organization because your AD team will be able to do a much better job of regression testing all changes (whether new function changes or defect fixes) before passing modified applications back into production.
- Provides a return on investment by remaining usable beyond its initial run. Distillation Assistant can help you come out of the Y2K test period with an efficient, compact suite of test cases that serve as an ongoing regression test bed for future development.
- Extensively exercises modified applications and is cost-efficient by using a minimum amount of CPU time.

Black box testing indicates how well your test ran and points out any areas that you need to examine more closely. Once you have collected all of the information obtainable from black box testing, you will want to gather white box testing information. By letting you see inside your application at run time, white box testing lets you ensure that Y2K fixes to your application code are exercised.

White Box Testing

The concept of *white box* testing refers to application testing that measures internal application behavior during run time. For example, your testing could measure what sections of the application were exercised during the test run or what variable values were located at a specific internal point of the application run.

You can obtain white box information by designing and writing application programs with internal code logic that provides it, or by using specific testing tools, such as debuggers or code coverage engines.

Again, using an MVS batch application as an example, suppose you wanted to gather information about which lines of source code were exercised in a test run. The entire application could have been written with logic at each source instruction to log this information, but this is not practical and is rarely done. A more practical approach would be to use a testing tool that can *instrument* the application to provide this information during the test run. All source instructions that were exercised can be logged and evaluated to ensure that the affected sections of the tested application were exercised. If a critical section was not exercised, then an existing test case can be modified or a new test case can be created that will force the execution of the untested section of code.

ATC White Box Testing

Black box testing is a starting point in the Y2K testing process. However, by itself, it is incomplete. Many AD shops are shocked to learn (after they get white box tools in their shops) that what they thought was an adequate test case suite is in actuality a methodology that leaves 50-60% of their application logic totally unexercised. It is no wonder that enhancements and fixes are introducing new errors into production on a continual basis.

Given the magnitude of the code changes required to handle the Y2K problem, the potential for modified code not being tested is considerable if white box testing is not employed. Entire sections of Y2K impacted code may never be exercised, and black box testing, by itself, would not warn the tester of this exposure. White box results are required to ensure that affected areas of the modified application have been exercised.

The following ATC tools supply white box results:

Coverage Assistant (CA)

CA supplies white box results by:

- Reporting on each source level line of code that was executed during one or more test runs
- Providing annotated listings that show which source lines have or have not been executed
- Providing summary level information.

Coverage Assistant Targeted Summary

CA targeted summary supplies white box results by:

Reporting on the source lines executed that were impacted by code changes. These impacted lines are provided as source line numbers or as variable names. From the provided list of line numbers and variable names, targeted summary can identify any affected source statements and provide coverage information on this subset of source statements. You can determine the affected source statement variables using the ATC tool Source Audit Assistant (also described in this list).

Source Audit Assistant (SAA)

SAA supplies white box results by:

Generating the input you give to Coverage Assistant's targeted summary. By comparing the application source or listing before and after Y2K remediation and then building a list of affected variables, Source Audit Assistant generates a file that Coverage Assistant targeted summary can use to report code coverage data for code involving the affected variables.

Distillation Assistant (DA)

DA assists white box testing by:

Reducing input data files to a minimum number of records that will cause equivalent test case coverage as the original files. It is used to reduce testing time and expense. By reducing the size of the input files, the amount of CPU time is cut significantly for repeated runs of the test case suite, thus shortening the test cycle schedules.

Distillation Assistant can be used to reduce the size of QSAM or VSAM input files for COBOL and PL/I (record I/O only) programs.

Unit Test Assistant (UTA)
UTA assists white box testing by:

Providing white box logging and warping of variable values at selected points during the application test run. Also, the UTA file warp feature can copy your input files and then warp specified fields in the copied files for testing.

Automated Regression Testing Tool (ARTT)
ARTT assists white box testing by:

Allowing you to run ATC white box tools offsite or without any complex environment setup, thus greatly reducing the cycle time required to test.

Fundamentally, you want to answer the question “Did I test what I changed?” Since one of the first premises of the Y2K testing problem is that the functionality of the application has not changed, then it is most critical to, at a minimum (and early), test all of the code that has changed through Y2K remediation. Together, ATC black box and white box test tools can provide concrete data to confirm that you have indeed tested what you have changed.

Data Warping

Y2K *readiness* testing is confirming that the converted application will perform correctly when future dates are input via:

- File input
- Application calls to get the system date
- Terminal input/output
- Program invocation parameters

The Y2K conversion/testing process has evolved into the following procedure:

1. Update the application code to be Y2K ready.
2. Test the application with present dates to ensure that it will continue to perform correctly in the present production environment.
3. Move the application back into production.
4. As time and resources allow, test the application for Y2K readiness to ensure that it will work with post-1999 dates.

The optimal plan for readiness testing is to migrate the Y2K converted application to a stand-alone CPU with the system clock set forward and dates in the input files set forward to post-1999 dates. But as the schedule shortens and the cost of readiness testing rises, another solution is needed. Data warping offers part of that solution by providing a means of incorporating Y2K readiness testing into the early testing phases without requiring a dedicated Y2K CPU and aged data resources.

ATC File Warping

A standard data warping process is to age, or *warp,* occurrences of dates in the input data files, setting them to values beyond the year 2000. The UTA file warp feature supports this process. It can copy any flat file or VSAM file and then warp specified fields in the copied file for testing. Numeric fields can be incremented, decremented, or set to a value.

While this process addresses most of the problem by providing a degree of confidence that the application will perform correctly when running post-1999 dates, it does not handle dates introduced through system calls and user prompts, and it increases the effort of maintaining test data.

ATC Dynamic Data Warping

A preferred approach for data warping is providing a means for you to intercept dates as they enter and exit the application and allowing you to modify, or *warp*, the dates at that point. This allows you to not only modify dates entering from files, but from system calls and user prompts as well. This also reduces the cost of maintaining test data because all of your date testing can be driven from one set of data containing current date values.

Unit Test Assistant (UTA) lets you do exactly this type of data warping. UTA's dynamic data warping function allows testers to define, at an application source level, where dates enter the application and also define how the dates should be modified at those points. This is done without modification to the application source code. The input and output dates can be incremented, decremented, or initialized.

When this UTA function is used with Coverage Assistant, testers can track the application statement execution affected by the warped dates. This provides a means of incorporating readiness testing into unit, function, integration, and system testing.

Automated Regression Testing Tool (ARTT) also lets you perform data warping, but only on a file record basis rather than on a variable or application source-level basis.

Data Distillation

As stated in our requirements for the Y2K testing process, anything that can be done to decrease testing cycle times is of great value. Because time is so limited, Y2K test projects are in danger of simply running out of runway, so to speak.

A procedure that helps decrease testing cycle times is distillation. Distillation works by reducing input data sets to the minimum number of records required to provide code coverage that is equivalent to the coverage provided by the larger data sets.

ATC Data Distillation

ATC's answer to distillation is the Distillation Assistant (DA) tool. DA accepts test case input data sets and reduces them to the smallest size that provides source code coverage equivalent to that obtained when using the original (larger) data set.

For example, suppose you have a test case that provides 10,000 input transaction records to the application being tested. The test run takes ten minutes of CPU time and three hours of clock time to execute. The Coverage Assistant (CA) measured code coverage is 70%. DA might reduce the transaction input data set to 1000 records, requiring one minute of CPU time and 20 minutes of clock time to execute, and the CA measured code coverage will remain at or near 70%.

You can gain a significant reduction in test cycle times if DA is applied across applications that can take advantage of this functionality, especially applications that will be tested through many cycles over the life of the Y2K remediation/test project or that will be tested on an ongoing basis due to normal application enhancement/fix activities.

Coverage Assistant

What Problem Does Coverage Assistant Solve?

Coverage Assistant (CA) is a white box test tool that monitors and collects code coverage information at the source level as an application is executed against a suite of test cases. You can get detailed and/or summary level information about what statements and branches have or have not been executed during a run of a suite of test cases intended to test the target program or programs.

There is also the capability of doing targeted summary against a set of source statements and/or statements involving a list of specified variables in which you may be interested.

Questions and Answers

1. **What exactly is "code coverage"?**

   Code coverage, at the most basic level, is the ability to monitor and track which instructions (at the source level) have or have not been executed at least once while executing a defined suite of test cases. The test cases that would be run to do this are completely up to you. In practice, they probably would be the suite of test cases that you would normally run to test the business function of an application (that is, black-box test cases).

   Knowing which instructions have been executed as well as which branch statements have been executed (none, or one or both branches taken), you can begin to get a detailed picture of just how much of your application you have actually tested.

2. **How does this new information allow me to improve my testing?**

   Let us take a simple example. Assume that you have an application, X, and you have 100 test cases set up as a regression bucket that you believe adequately tests the business function that X is supposed to provide for you. How did you make that determination? Whether you did it explicitly or not, you did a black box, or business function, analysis and decided that you had put together enough test cases to test all of the various functions of application X. As long as you can run those test cases successfully, you consider your testing successful/adequate. But is it?

   Now with CA, you can collect code coverage information about all the routines (compile units [CUs]) that make up application X while you are running the 100 test cases that you have assumed comprise an adequate test bed. If your experience is like most projects that have used this tool in the past, you will find that you typically have only covered 40-50% of the statements in the application.

3. **Are you saying that over half of the statements never get executed?**

   That is correct, at least when they are monitored in a testing situation. Keep in mind that this is based on real experience over a period of more than 10 years. Most development groups are amazed at how much of their code never gets tested when they only perform black box testing. With CA, you can get detailed information about what code is or is not getting executed. Once you see the code segments that are not being exercised, you can quickly augment the test bed with test cases that cause the previously unexecuted/untested lines to be executed/tested.

Again, real experience has shown that with relatively little effort in building a targeted set of additional test cases, you can increase the code coverage from 40-50% to 70-80%.

4. **Why would I not want to get to 100% code coverage?**

   That is the ideal of course. However, experience has shown that you reach a point of diminishing returns in terms of effort to raise the level above the 80% level. When you get into that last 20% of the code, you are typically dealing with exception processing, hardware dependencies, and so on. You will expend more and more effort to create the specialized type of test cases that you would need to force execution of the code that handles those exception-type conditions.

   When you get into this territory, you have to apply risk analysis to determine when enough testing is enough. In other words, what is the risk that is taken if a particular section of code is left untested given that it would take a considerable effort to put together the test case to force it to be tested. There is no right or wrong answer here, and we don't make any claims in this area. Each application owner needs to assess the application criticality of the areas that are not yet tested to decide what the goal is in terms of code coverage.

   However, the bottom line is that CA gives you a great deal of information that you did not have previously in assessing your confidence in your present testing. Further, you have the information to raise that confidence to a level that you determine to be satisfactory for your application(s).

5. **Can you be more specific about the information you can get from CA?**

   At the summary level, you get a report that shows at the compile unit level the total number of code statements for each procedure/paragraph and the number that have been executed (as a percentage), as well as the number of branches that are possible and how many have actually been taken (as a percentage).

   You also get a section identifying statements that have not been executed and a section identifying statements that are branches that have not gone both ways.

   At a more detailed level, CA *annotates* the listing of any compile units that make up your application to show which statements and branches have or have not been executed. This is what would be most helpful to a programmer who is trying to identify areas of code that are untested so that additional test cases can be written.

6. **How do I use CA to get this code coverage information?**

   The tool is easy to use. At a high level, the steps to run CA on an application are as follows:

   a. Make sure you have all the correct source, include, or copy members that it takes to compile and link edit your application.

   b. Compile your source with compiler options required by CA. For the most part, you can run with whatever options you normally run with. However, a few options are required for CA to function correctly. Save the object and listing data sets from this set of compiles because they are used by CA to do its analysis.[1]

   c. You need to create a CA control file that describes the structure of your application. This includes what load modules you want to monitor, what object modules make up each load module, and where your object and listing data sets are stored.[1]

   d. Run a setup job that instruments the object code that makes up your application.[1]

   e. Run a link-edit job that links your application using the instrumented object modules.[1]

   f. Start the CA monitor session.

   g. Run whatever test cases that make up your test case suite against your application.

   h. Stop the CA monitor session.

   i. Run a reports job that provides summary and detailed data from the test run.

7. **Sounds like there are a lot of jobs I have to run to make all this happen.**

   There are jobs that must be run and that must be run in sequence. However, your job is made very simple in that ATC has an ISPF interface with options that very easily lead you through the process of creating all the jobs required to run CA (and the other tools). For new users, it makes the process very simple and straightforward.

   As you get more experienced with the tools, you will begin to incorporate the various jobs into your normal development and test processes and procedures, and the ISPF interface will be less important.

8. **At the beginning, a concept called *targeted summary* was mentioned. What is targeted summary?**

   Most simply, targeted summary works this way. First, you run your defined set of test cases that will generate total code coverage over the set of programs you have defined to CA. As we have already discussed, just doing that will allow you to generate code coverage reports across the programs that are being monitored by CA. Then, by running targeted summary, you can target certain statements or variables of interest and see coverage data on just those specified statements or statements that are involved somehow with any specified variables. You do not run additional test cases to do this if you have run

[1] Note that an object module can be instrumented and then linked into a new load module, or a load module can be instrumented directly.

CA against all the test cases you deem necessary for full coverage. Targeted summary is simply a postprocess of the already collected coverage data that is focused on the areas of code in which you are truly interested.

If you request targeted summary, CA produces summary reports that limit the measurements to just the statements that pertain to your targeted set of statements. The format of the reports is identical to what you would see on a regular coverage summary report, but the scope covers only the statements that you have specified.

9. **Explain why targeted summary is more useful to me than regular code coverage.**

   In order to explain the power of targeted summary, we need to step back and talk about general testing philosophy. If you have a stable, functioning application **and** an adequate test bed (as you determine it to be adequate), then what are the most important things to do as you make enhancements or fixes to the application? Typically, you want to:

   a. Run whatever test case(s) that are required to validate that the new enhancement works as designed or that a bug has been fixed so that the program works as designed.

   b. Run the modified application through a regression cycle to ensure that existing functionality has not been broken.

   c. Make sure that all code that has been modified or affected has been exercised.

   Items 9a and 9b are fairly self-explanatory. The regression cycle is simply running the modified application against a set of known test cases to make sure that something that was previously working has not inadvertently been broken.

   The third item is something that too few development/maintenance shops enforce and is needed to ensure that all code that has been either directly modified or affected by a modification (say to a variable declare) has, at a minimum, been executed. In other words, have you truly tested what you have changed? CA targeted summary lets you know how you are doing in this area. Once this technology is available, many development teams are shocked to find that their testing would often miss new code before putting the code into production.

10. **As a programmer, how do I take advantage of this capability?**

    Once you have made the changes to the source code and gone through the steps required to prepare the application for testing, do the following:

    a. Write any test cases you think are needed to test the new/modified functionality of your program.

    b. Run those test cases outside the control of CA just to ensure that they test the new function and provide the expected output. Debug the application and fix any bugs you discover.

    c. Run the regression test suite and the new test cases under control of CA.

    d. Use Source Audit Assistant to analyze the listings for any of the compile units you modified and produce a targeted summary control file.

e. You will have to make a few manual adjustments to the targeted summary control file produced by SAA, such as indicating what data set(s) your listing(s) are stored in.

f. Run a targeted summary report to generate the summary reports that point out areas of the modified and affected lines of code that are not getting exercised.

g. Use the targeted summary report and annotated listings to add/modify your new test cases to ensure that all of the lines are getting exercised.

h. Add your new test cases to your regression test suite.

11. **What environments does CA support?**

    CA can monitor applications running in many environments in the MVS world: batch, TSO, CICS®, IMS™ TM, and ISPF. Also, you don't have to do anything differently in terms of setting up for CA because of the target environment. CA operates independently from all these environments, which makes this tool easy to use even when testing spans multiple environments.

    The monitor used by Coverage Assistant, Distillation Assistant, and Unit Test Assistant uses SVCs as breakpoints, gaining control when the SVCs are invoked. This technique minimizes runtime overhead and allows the programs being tested to run in any environment (batch, online, under CICS, and so on).

12. **What compilers and assemblers are supported?**

    - IBM COBOL for OS/390® & VM 2.1 (plus Millennium Language Extensions [MLE])
    - IBM COBOL for MVS & VM 1.2 (plus Millennium Language Extensions [MLE])
    - IBM VS COBOL II Release 4.0
    - IBM OS/VS COBOL Release 2.4
    - IBM PL/I for MVS & VM 1.1.1 (plus Millennium Language Extensions [MLE])
    - IBM OS PL/I Optimizing Compiler 2.3.0
    - IBM PL/I Optimizing Compiler 1.5.1
    - IBM High Level Assembler Version 1 Releases 2 and 3
    - IBM Assembler H Version 2

13. **Is there much overhead added by the CA monitor?**

    For a test case coverage run, CA typically adds very little execution time to the program. CA inserts SVCs (supervisor calls) into the application object modules as breakpoints and then intercepts the breakpoints. Most breakpoints are removed after their first execution. By using this technique, the increase in test program execution time is minimal.

Overview of How to Use Coverage Assistant

A typical coverage run would look like this:

1. Compile or assemble your routines (compile units) using the options required by CA. Save the listings and object modules into data sets.[2]
2. Determine how you want to group your compile units together for testing. CA can generate reports for either a single compile unit or any arbitrary collection of compile units. Each grouping or collection of compile units is defined by a CA control file.
3. Create a CA control file that defines your collection of compile units. This control file will consist of one control statement per compile unit that defines the name of the compiler listing, the object module data set produced by the compiler, and the output object module data set produced by CA.[2]
4. Create the JCL for the CA SETUP program and run it. This program will analyze your compiler listings and create a new set of object modules that contain CA breakpoints.[2] In addition, SETUP creates a data set called a BRKTAB (breakpoint table).

   Experienced users can integrate this SETUP program into their normal compile procedures.
5. Link edit your program(s) using the new object modules created by CA SETUP.[2]
6. Start a CA monitor session with the BRKTAB created by the CA SETUP program.
7. Run your test cases. The CA monitor intercepts the breakpoints inserted into your program and records the coverage data. The handling of the breakpoints is transparent to your program (and any run time that you are using).
8. Stop the CA monitor session.
9. Run the CA Summary and Annotated Report jobs.

   For each COBOL paragraph, PL/I procedure, ON-unit, Begin-block, or assembler listing, CA Summary provides you with:

   - The percentage of statements executed and a list of unexecuted statements
   - The percentage of conditional branches executed and a list of conditional branches that have not executed in both directions

   The Annotated listing will contain a copy of your compiler listings where each executable statement has been annotated with a symbol showing its execution status.
10. Create a targeted summary control file that defines what COBOL or PL/I statements or variables that you would like to target. This control file can be created using either Source Audit Assistant or an editor, if you prefer to create the file manually.
11. Run the CA targeted summary program to produce a targeted summary report.

[2] Note that an object module can be instrumented and then linked into a new load module, or a load module can be instrumented directly.

Inputs

Figure 1 is an example of a CA control file for a COBOL load module consisting of three COBOL object modules. The control file is how you let CA know what routines (compile units) make up the application to be monitored and where to find all the data sets associated with those routines.

```
*
* Cobol Example
*
* Statements required for coverage
*
               Defaults ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(*),
                        LoadMod=COB01M,
                        FromObjDsn=ATC.V1R5M0.SAMPLE.OBJ,
                        ToObjDsn=ATC.V1R5M0.SAMPLE.ZAPOBJ

COB01AM:       COBOL ListMember=COB01AM
COB01CM:       COBOL ListMember=COB01CM
COB01DM:       COBOL ListMember=COB01DM
```

Figure 1. Control File for COB01M

All ATC control files are keyword oriented and are easy to navigate. In the case of Figure 1, the tester/programmer has specified a control file that, when processed by CA, will be interpreted as follows:

- The data set containing the listings is ATC.V1R5M0.SAMPLE.COBOLST.
- The load module is named COB01M.
- The data set containing the object code from the compiler is ATC.V1R5M0.SAMPLE.OBJ.
- The data set containing the CA modified object code (used to create an instrumented load module that is used only when CA is monitoring the application) is in ATC.V1R5M0.SAMPLE.ZAPOBJ.
- The tester/programmer wants to monitor coverage on three compile units (COB01AM, COB01CM, and COB01DM).

This is a simple case and of course more complex control files can be built as needed, but the point of this illustration is to show that understanding the control file is a fairly easy task.

Figure 2 is an example of a CA targeted summary control file for a COBOL routine. You use the targeted summary control file to let CA know what routines (compile units), and within those routines what source statements or variables, you are interested in getting code coverage data on. This file can be created manually or by using the Source Audit Assistant postprocessor (which compares your pre- and post-remediated code) to generate a targeted summary control file containing all the variables that have been part of a change as a result of the remediation.

```
Cobol ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)
  Scope ExtProgram-Id=COB01AM
    TargetVar Name=(STATE In LOC-ID In TASTRUCT)
  Scope ExtProgram-Id=COB01AM,NestedProgram-Id=COB01BM
    TargetVar Name=(TBPARM2)
  TargetStmt Stmts=(46,62,67)

Cobol ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)
  Scope ExtProgram-Id=COB01CM
    TargetVar Name=(TCPARM1)
```

Figure 2. Targeted Summary Control File for COB01M

In this case, the tester/programmer is asking to see targeted code coverage data for the following:

- In the COB01AM compile unit, within load module COB01M, any lines that reference or modify the variable STATE In LOC-ID In TASTRUCT.
- In the nested program COB01BM, within the COB01AM compile unit, any line that references or modifies the variable TBPARM2 and statements 46, 62, and 67 (from the listing).
- In the COB01CM compile unit, within load module COB01M, any lines that reference or modify the variable TCPARM1.

There is one other feature of CA targeted summary that is worth pointing out here. If a variable is specified in the target control file, CA is able to detect what lines/statements reference that variable both explicitly and implicitly. In the previous example, suppose TCPARM1 is part of the following COBOL record definition:

```
01 BUFFER.
   03 FIELD-1        PIC X(2).
   03 TCPARM1        PIC 9(4).
   03 FILLER         PIC X(74).
```

and you had in the Procedure Division the following statement:

```
   MOVE SPACES TO BUFFER.
```

If TCPARM1 had been a field that represented a 2-digit year that had been modified to 4 digits, you would want to know if any code referencing that variable had been exercised. The MOVE statement shown actually changes the value of TCPARM1 even though it is not directly referenced. CA targeted summary is able to pick up this type of implicit usage and report on lines such as this.

Outputs

Coverage Summary Report

Figure 3 is an example of a CA summary report. The CA summary report gives statistics on the coverage of all program areas during the test run. The summary report is divided into the following sections:

Program Area Data
Lists the following summary data for each program area:

- The total number of code statements
- The number of executed code statements
- The total number of branches
- The number of executed branches

Unexecuted Code
Lists the unexecuted code statements in each program area.

Branches That Have Not Gone Both Ways
Lists the conditional branches that have not executed in both directions for each program area.

```
1 *********  CA SUMMARY:                     PROGRAM AREA DATA                     ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                      -->|
  |                             |                                         |  STATEMENTS:          | BRANCHES:           |
  | PA LOAD MOD PROCEDURE       | LISTING NAME                            |  TOTAL     EXEC   %   | CPATH   TAKEN    %  |
  -----------------------------------------------------------------------------------------------------------------------
    1 COB01M                    ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)             6        6 100.0       0        0 100.0
    2          PROGA                                                           8        7  87.5       6        5  83.3
    3          PROCA                                                           1        0   0.0       0        0 100.0
    4          PROGB                                                           7        5  71.4       6        3  50.0
    5          PROCB                                                           2        2 100.0       0        0 100.0
    6 COB01M   PROGC            ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)             7        5  71.4       6        5  83.3
    7          PROCC                                                           3        2  66.7       2        1  50.0
    8 COB01M   PROGD            ATC.V1R5M0.SAMPLE.COBOLST(COB01DM)             6        0   0.0       6        0   0.0
    9          PROCD                                                           1        0   0.0       0        0 100.0
  -----------------------------------------------------------------------------------------------------------------------
  Summary for all PAs:                                                        41       27  65.9      26       14  53.8
```

```
1 *********  CA SUMMARY:                     UNEXECUTED CODE                       ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                     -->|
  |                            |                                         |
  | PA LOAD MOD PROCEDURE      | LISTING NAME                            | start    end     start    end     start   end
  -----------------------------------------------------------------------------------------------------------------------
    2 COB01M   PROGA            ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)          67     67
    3          PROCA                                                        79     79
    4          PROGB                                                       118    119
    6 COB01M   PROGC            ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)          39     40
    7          PROCC                                                        58     58
    8 COB01M   PROGD            ATC.V1R5M0.SAMPLE.COBOLST(COB01DM)          37     46
    9          PROCD                                                        51     51
  -----------------------------------------------------------------------------------------------------------------------
```

Figure 3 (Part 1 of 2). Summary Report for COB01M

```
1 *********  CA SUMMARY:                    BRANCHES THAT HAVE NOT GONE BOTH WAYS ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                   PROGRAM IDENTIFICATION                    -->|
  |                            |                                      |
  | PA LOAD MOD PROCEDURE      | LISTING NAME                         |  stmt      stmt      stmt      stmt      stmt
  --------------------------------------------------------------------------------------------------------------------
     2 COB01M   PROGA           ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)         66
     4          PROGB                                                     113       118
     6 COB01M   PROGC           ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)         37
     7          PROCC                                                      57
     8 COB01M   PROGD           ATC.V1R5M0.SAMPLE.COBOLST(COB01DM)         37        41        45
  --------------------------------------------------------------------------------------------------------------------
```

Figure 3 (Part 2 of 2). Summary Report for COB01M

Annotated Listing Report

If you want more information on some or all of the modules that have been tested, you can create an annotated listing. This listing contains information about each breakpoint. To the right of each statement number, one of the following characters is shown to indicate the results of the execution of that statement:

&	Conditional branch instruction has executed both ways.
>	Conditional branch instruction has branched, but not fallen through.
V	Conditional branch instruction has fallen through, but not branched.
:	Non-branch instruction has executed.
¬	Instruction has not executed.

Figure 4 on page 21 shows a sample annotated listing.

```
000001                IDENTIFICATION DIVISION.
000002                PROGRAM-ID. COB01AM.
000003               ******************************************************************
000004               *                                                                *
000005               * LICENSED MATERIALS - PROPERTY OF IBM                           *
000006               *                                                                *
000007               * 5799-GBN                                                       *
000008               *                                                                *
000009               * (C) COPYRIGHT IBM CORP. 1997 ALL RIGHTS RESERVED               *
000010               *                                                                *
000011               * US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR    *
000012               * DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM    *
000013               * CORP.                                                          *
000014               *                                                                *
000015               *                                                                *
000016               ******************************************************************
000017               ******************************************************************
000018               *                                                                *
000019               * COBOL FOR MVS & VM TEST.                                       *
000020               *                                                                *
000021               * MEMBER COB01AM HAS ENTRY POINT COB01AM.                        *
000022               * CALLS COB01BM, WHICH CALLS COB01CM, WHICH CALLS COB01DM.       *
000023               ******************************************************************
000024
000025                ENVIRONMENT DIVISION.
000026
000027                DATA DIVISION.
000028
000029                WORKING-STORAGE SECTION.
000030                01 TAPARM1      PIC 99 VALUE 5.
000031                01 TAPARM2      PIC 99 VALUE 2.
000032                01 COB01BM      PIC X(7) VALUE 'COB01BM'.
000033                01 P1PARM1      PIC 99 VALUE 0.
000034
000035                01 TASTRUCT.
000036                  05 LOC-ID.
000037                    10 STATE    PIC X(2).
000038                    10 CITY     PIC X(3).
000039                  05 OP-SYS     PIC X(3).
000040
000041                PROCEDURE DIVISION.
000042
000043               * THE FOLLOWING ALWAYS PERFORMED
000044
000045               * ACCESS BY TOP LEVEL QUALIFIER
000046 :                  MOVE 'ILCHIMVS' TO TASTRUCT.
000047
000048               * ACCESS BY MID LEVEL QUALIFIERS
000049 :                  MOVE 'ILSPR' TO LOC-ID.
000050 :                  MOVE 'AIX' TO OP-SYS.
000051
000052               * ACCESS BY LOW LEVEL QUALIFIERS
000053 :                  MOVE 'KY' TO STATE.
000054 :                  MOVE 'LEX' TO CITY.
000055 :                  MOVE 'VM ' TO OP-SYS.
000056
000057                PROGA.
000058
000059               * THIS PERFORM EXECUTED
000060 &                  PERFORM WITH TEST BEFORE UNTIL TAPARM1 = 0
000061 :    1               SUBTRACT 1 FROM TAPARM1
000062 :    1               CALL 'COB01BM'
000063                     END-PERFORM
```

Figure 4 (Part 1 of 2). Annotated COBOL Listing

```
000064
000065               * THIS IF ALWAYS FALSE
000066 >                  IF TAPARM2 = 0
000067 ¬    1               PERFORM PROCA
000068                    END-IF
000069
000070               * THIS PERFORM EXECUTED
000071 &                  PERFORM WITH TEST BEFORE UNTIL TAPARM2 = 0
000072 :    1               SUBTRACT 1 FROM TAPARM2
000073                    END-PERFORM
000074 :                  STOP RUN
000075                    .
000076
000077                PROCA.
000078               * PROCA NEVER CALLED
000079 ¬                  MOVE 10 TO P1PARM1
000080                    .
000081
000082               * START OF COB01BM NESTED IN COB01AM
000083
000084   1            IDENTIFICATION DIVISION.
000085   1            PROGRAM-ID. COB01BM.
000086   1           *******************************************************************
000087   1           *                                                                 *
000088   1           * COBOL FOR MVS & VM TEST.                                        *
000089   1           *                                                                 *
000090   1           * COB01BM, CALLED BY COB01AM.                                     *
000091   1           *******************************************************************
000092   1
000093   1            ENVIRONMENT DIVISION.
000094   1
000095   1            DATA DIVISION.
000096   1
000097   1            WORKING-STORAGE SECTION.
000098   1            01 TBPARM1      PIC 99 VALUE 5.
000099   1            01 TBPARM2      PIC 99 VALUE 0.
000100   1            01 COB01CM      PIC X(7) VALUE 'COB01CM'.
000101   1            01 P1PARM1      PIC 99 VALUE 0.
000102   1
000103   1            PROCEDURE DIVISION.
000104   1
000105   1            PROGB.
000106   1           * THIS PERFORM EXECUTED
000107 & 1                PERFORM WITH TEST BEFORE UNTIL TBPARM1 = 0
000108 : 1  1               SUBTRACT 1 FROM TBPARM1
000109 : 1  1               CALL 'COB01CM'
000110   1                END-PERFORM
000111   1
000112   1           * THIS IF EXECUTED
000113 V 1                IF TBPARM2 = 0
000114 : 1  1               PERFORM PROCB
000115   1                END-IF
000116   1
000117   1           * THIS PERFORM NOT EXECUTED
000118 ¬ 1                PERFORM WITH TEST BEFORE UNTIL TBPARM2 = 0
000119 ¬ 1  1               SUBTRACT 1 FROM TBPARM2
000120   1                END-PERFORM
000121   1                .
000122   1
000123   1            PROCB.
000124   1           * PROCB EXECUTED
000125 : 1                MOVE 10 TO P1PARM1
000126   1                .
000127   1
000128 : 1                EXIT PROGRAM.
000129   1
000130   1            END PROGRAM COB01BM.
000131                END PROGRAM COB01AM.
```

Figure 4 (Part 2 of 2). Annotated COBOL Listing

Targeted Summary Report

Figure 5 is an example of a CA targeted summary report. As mentioned earlier, it gives you a report identical in format to the CA coverage summary report, but in this case only, is reporting on source lines that were the target based on the control file.

For convenience, the target control cards of interest are included at the top of the report. Similarly, all statistics calculated are based upon the scope of what lines are targeted so that you can tell how well you have tested what you have changed.

```
1 *********  CA TARGETED SUMMARY:            CONTROL RECORDS                      ********
0         CONTROL DSN: 'ATC.V1R5M0.SAMPLE.TARGCTL(COB01M)'
0                TARGETED SUMMARY DATE: 02/02/1999
                 TARGETED SUMMARY TIME: 19:57.15
0 |<--                     TARGET PSEUDO CONTROL STATEMENTS                            -->

------------------------------------------------------------------------------------------------------------------------
  Variable  ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01AM),ExtProgram-Id=COB01AM,Name=(STATE In LOC-ID In TASTRUCT)
  Variable  ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01AM),ExtProgram-Id=COB01AM,NestedProgram-Id=COB01BM,Name=TBPARM2
  Statement ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01AM),Stmts=(46,62,67)
  Variable  ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB01CM),ExtProgram-Id=COB01CM,Name=TCPARM1

1 *********  CA TARGETED SUMMARY:            PROGRAM AREA DATA                     ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                     -->|
  |                            |                                         |  STATEMENTS:          | BRANCHES:            |
  | PA LOAD MOD PROCEDURE      | LISTING NAME                            |  TOTAL      EXEC   %  | CPATH    TAKEN    %  |
  ----------------------------------------------------------------------------------------------------------------------
     1 COB01M                ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)              3       3 100.0       0       0 100.0
     2          PROGA                                                        2       1  50.0       0       0 100.0
     4          PROGB                                                        3       1  33.3       4       1  25.0
     6 COB01M   PROGC        ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)              2       1  50.0       2       1  50.0
  ----------------------------------------------------------------------------------------------------------------------
  Summary for all PAs:                                                      10       6  60.0       6       2  33.3

1 *********  CA TARGETED SUMMARY:            UNEXECUTED CODE                       ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                     -->|
  |                            |                                         |
  | PA LOAD MOD PROCEDURE      | LISTING NAME                            |  stmt      stmt      stmt      stmt      stmt
  ----------------------------------------------------------------------------------------------------------------------
     2 COB01M   PROGA        ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)           67
     4          PROGB                                                    118       119
     6 COB01M   PROGC        ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)           39
  ----------------------------------------------------------------------------------------------------------------------

1 *********  CA TARGETED SUMMARY:            BRANCHES THAT HAVE NOT GONE BOTH WAYS ********
0                DATE: 12/11/1998
                 TIME: 11:37.10
         TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                     -->|
  |                            |                                         |
  | PA LOAD MOD PROCEDURE      | LISTING NAME                            |  stmt      stmt      stmt      stmt      stmt
  ----------------------------------------------------------------------------------------------------------------------
     4 COB01M   PROGB        ATC.V1R5M0.SAMPLE.COBOLST(COB01AM)           113       118
     6 COB01M   PROGC        ATC.V1R5M0.SAMPLE.COBOLST(COB01CM)            37
  ----------------------------------------------------------------------------------------------------------------------
```

Figure 5. Targeted Summary Report for COB01M

Source Audit Assistant

What Problem Does Source Audit Assistant Solve?

Source Audit Assistant (SAA) is a tool that helps you easily view changes that you have made to your source code. It allows you to do a comparison of your code base and provides several views of that comparison for your use.

Questions and Answers

1. **What do you mean by a comparison of my code base?**

   Think about a simple program made up of several source files that get compiled and linked together to form your application. Suppose you have your production copy of those source files in a partitioned data set (PDS). Further, suppose you have to change one or more of those files in order to either add a new function or fix a bug. You make your changes and store your changed files into a new PDS that only contains the files that have been modified. Now we are ready to delve into exactly what SAA can do for you.

   In the simplest of terms, SAA performs a record level file comparison of the old PDS and the new PDS and generates a report containing all common members, with only the differences between the old and new files appearing in the report. In other words, all you will see in the report are the differences between your original code and your modified code.

2. **Isn't this simply like any standard file compare utility?**

   No. While SAA at its base does a file comparison, it does much more for you that differentiates it from standard file compare utilities.

3. **Like what?**

   The name of the tool—Source Audit Assistant—implies that it assists you in *auditing* changes to your code base. Many information systems (IS) shops have a very structured process for reviewing, or *auditing,* all changes made to code before allowing it back into production.

   SAA provides that audit function so that the programmer, or an independent review team, can look at all changes to validate that they are needed changes.

4. **Sounds good but I still don't see why a simple file comparison utility won't do the same thing for me.**

   For a number of reasons. One is that SAA is set up to help you audit massive numbers of changes over a large number of files. A simple file utility could bury you in such a situation with a comparison file that is so large that it would literally take days to go through and review all of the changes that were made.

   SAA provides a series of filters which help mask out or eliminate certain changes that may be uninteresting to you. The filter simply ignores certain types of changes so that they do not appear in the resulting report file, thereby not cluttering the file with changes that you do not care to see.

5. **What types of filters are provided?**

   Three filters are currently provided:

   a. Comments filter. The comments filter, if specified, will mask out (ignore) changes made to comments. That brings up another feature of SAA that you will not find in most standard file comparison utilities. It “understands” the languages it supports to a large extent. It is not as sophisticated as a compiler, but it understands the basic syntax of its supported languages. Having this understanding of the languages, SAA can detect when it finds a difference in a record that is part of a comment. If you specify this filter, then all comment changes will be filtered out, and you won't have to deal with them in your output report.

   b. Variable declaration filter. Again, using its language understanding, SAA can filter out any changes that are made to statements that declare variables in the source code. So if you decide you don't need to see changes that fall into this category, simply specify that filter when you use SAA and those changes will be ignored.

   c. Reformatted lines filter. This filter is intended to ignore changes on lines that remain logically the same, but may have moved left or right because you have changed the indentation of the line. For example, if you had a series of statements that were always executed, you may have lined them up to start in a certain column, just for readability and maintainability. Now, to implement your change, you had to bracket those statements within an IF statement.

      You will probably insert your IF and ELSE at the same columns the already existing statements are on and then indent the existing statements a few columns, again simply for readability and maintainability. For example, suppose you had a PL/I program that originally had the following code segment:

      ```
      WEEKS_PAY = 0;
      ```

      and you changed that to:

      ```
      IF NEW_WEEK = TRUE THEN
        WEEKS_PAY = 0;
      ELSE;
      ```

      Without the reformatted lines filter, SAA would show all three statements as having changed. However, logically, the middle statement did not change, it was simply moved over two columns. If the reformatted lines filter is turned on, then only the IF and ELSE lines would show in the output report as having changed because those two lines are the ones that are truly different between the old and new files.

   So as you can see, you have the ability to filter out certain types of changes to help you focus on changes that you really care about and need to check.

6. **OK. I see how the filters can help cut down the amount of data that I have to look at. What other features are there?**

   Several. SAA can handle comparisons of not only source files, but compiler listings as well. SAA strips away all of the non-source portions of the listing, such as cross-reference and attribute sections, and then does a normal source type comparison with what remains, which is your source code as it had been processed by the compiler when the listing was created.

7. **Why would I want to compare listing files when it's my source that has been modified?**

   When you compare source files, you are viewing all of the pieces of your program separately, which may be what you want. However, most programs you are interested in are made up of files that are combined together at compile time to make a functioning program. The most obvious and common instance of this would be the practice of isolating record and structure definitions in separate copy or include members that are introduced at compile time and are processed by the compiler as one “logical” file. By having SAA compare your listing files, you get to view all of the files that were included in the compile as a single logical unit.

   It depends on what you are trying to accomplish, but just know that you have this option of comparing either source or listing input.

8. **What other reason might I want to compare listings rather than source files?**

   Two reasons primarily. When you compare source files, you usually don't have a view of the data and executable part of your code at the same time. Typically, programs are broken up into pieces. Data structures are kept in separate data sets so they can be shared among many programs that need to utilize them. However, if you take a listing and use that as your base of comparison, you have all of the copy books/include members brought in and in context with the executable part of the program. Additionally, they're all visible in the listing.

   This way, viewing and analyzing code changes is easier in most cases because the data and executable portions of the program are all in one comparison file.

   The second reason is that you can have SAA generate a CA targeted summary control file that can be passed to CA for processing. It is much more straightforward generating this control file that SAA creates by pulling information from the listing (where all the data structures are now present as was mentioned previously) rather than having to generate it from a comparison of the source files (in which the data structures may be spread out over several data sets).

   In order to learn more about how these two tools work together, see “Coverage Assistant” on page 11.

9. **Are there other features of SAA that can help me audit my changes?**

   Certainly. Another key feature is the ability to generate a change validation report, which will help you further analyze the changes that have been made to your source code.

   This part of SAA requires that you provide a data set that contains a list of seed variables. These variables are ones that you are expecting to be used in the changes between the production, or old files, and the changed, or new files. In other words, you can provide a list of variables that you would expect to find in the changes to your programs. With that input, SAA will provide you with a change validation report that contains three parts.

   The first part of the report simply identifies what data sets were used in the comparison, the date and time of the run, and so on.

   The second part of the report shows you a list of those variables that were not found in any of the changed lines for the two compared files. It is up to you to decide if there is a problem or not, but again this allows you to identify variables that you were expecting to be involved in the changes, but for one reason or another, did not appear in any of the changed lines.

The third part is a report of any changed lines that did not contain at least one of the variables in the provided list of variables. In this case, you will be shown all lines that were changed, but did not contain any of the provided variables, which may mean you are looking at possible unauthorized changes. You can review this part of the report to look for just such instances.

10. **Where does the list of expected variables that I need to provide to SAA come from?**

    This list comes in the form of a sequential data set or a PDS member containing the variable names. Where it comes from will depend on your particular environment. There are many analysis tools available that analyze source code, looking for variables of a certain type or description based on the name of the variable. If your shop is using tools of this nature, you should be able to create these files fairly easily using the output of those tools.

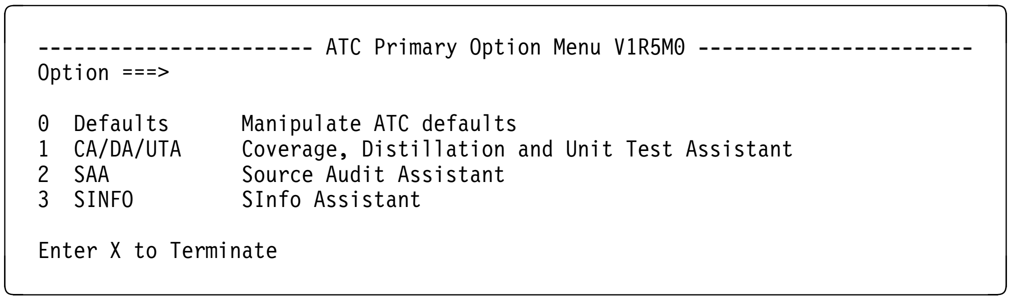
    Again, keep in mind that this feature is an option for you. It is not required in order to use the SAA base function. If you have other tools that do analysis of your code, then you can take advantage of those tools and allow SAA to help audit your changes even further.

11. **What languages are supported by SAA?**

    Currently, SAA supports 370/390 Assembler, C, C++, COBOL, and PL/I source files, as well as assembler, COBOL, and PL/I listing files.

Overview of How to Use Source Audit Assistant

Source Audit Assistant is very easy to use. You start it by selecting option 2 from the `ATC Primary Option Menu`. Figure 6 shows you what that ISPF panel looks like.



```
----------------------- ATC Primary Option Menu V1R5M0 -----------------------
Option ===>

0  Defaults      Manipulate ATC defaults
1  CA/DA/UTA     Coverage, Distillation and Unit Test Assistant
2  SAA           Source Audit Assistant
3  SINFO         SInfo Assistant

Enter X to Terminate
```

Figure 6. ATC Primary Option Menu

The `Execute Source Audit Assistant` panel, shown in Figure 7 on page 29, is then displayed.

```
------------------------ Execute Source Audit Assistant ----------------------
Option ===>

1  Background    Execute as Batch Job(s) via generated JCL
2  Foreground    Execute in the Foreground under ISPF
3  Compare       View Compare Dsn
4  Log           View Log Dsn
5  Postprocessor Validate Changes & Create Prototype Target Control Dsn

Enter END to Terminate
```

Figure 7. Execute Source Audit Assistant Panel

From this panel, you can choose to execute the code comparison in the foreground or in the background (batch) so that you don't tie up your online ISPF session.

Executing SAA in the Background

If you want to execute the code comparison in the background, select the `Background` option on the `Execute Source Audit Assistant` panel. The panel shown in Figure 8 is displayed.

```
--------------------- SAA Background Execution Parameters ---------------------
Option ===>

New Source Dsn: . . .  'project.newgroup.cobol(*)'

Old Source Dsn: . . .  'project.oldgroup.cobol(*)'

Output Compare Dsn: .  'yourid.SAA.YYY.CMP'

Output Log Dsn: . . .  'yourid.SAA.YYY.LOG'

Programming Language:  COBOL   (ASM,C,C++,COBOL,PL/I,LASM,LCOB,or LPL/I)

Select Line Audit Filters (Y = apply filter, N = do not apply filter):
  Comments  Y (Y or N)   Declares  Y (Y or N)   Reformatted  Y (Y or N)

Select Comparison Columns:                     Select Execution Option:
  Start Col 1   (1-176)   End Col 72  (1-176)    Edit JCL Y (Y or N)

DBCS support:
  Enable  N (Y or N)

Press END to Terminate
```

Figure 8. SAA Background Execution Parameters Panel

You simply fill in the fields to indicate where your old and new code files are stored, as well as the language of those files, what filters you want to apply, and the data set to which you want the comparison report written.

In addition, you can choose to edit the JCL created for you. While in edit, you can save the file for future use or make changes and submit it yourself. The other choice is to do the job submission now as the JCL is being created.

Lastly, you need to indicate whether the input files contain DBCS characters.

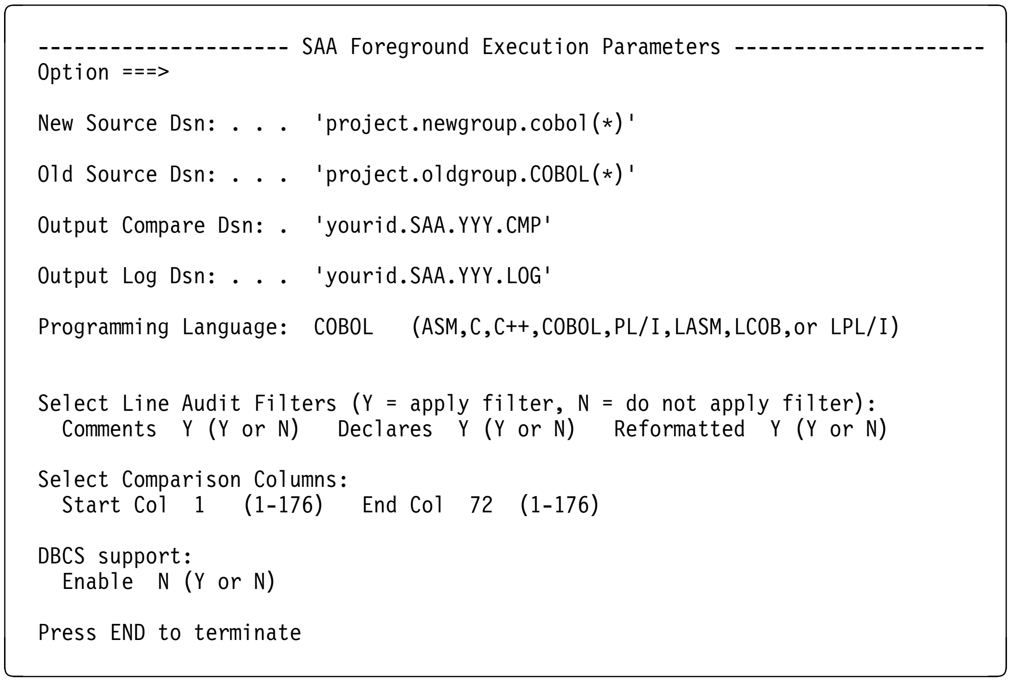
Depending on how many members you have in your old and new files, you will get one or more jobs generated (and executed) in order to compare all of the files in your data sets.

Once your job(s) have completed, you can view the results in the output data set you specified.

Executing SAA in the Foreground

If you want to execute the code comparison in the foreground, you select the `Foreground` option on the `Execute Source Audit Assistant` panel. The `SAA Foreground Execution Parameters` panel is displayed. Simply fill in the requested fields and press Enter to run the comparison.

Figure 9 shows the `SAA Foreground Execution Parameters` panel.



```
--------------------- SAA Foreground Execution Parameters ---------------------
Option ===>

New Source Dsn: . . .  'project.newgroup.cobol(*)'

Old Source Dsn: . . .  'project.oldgroup.COBOL(*)'

Output Compare Dsn: .  'yourid.SAA.YYY.CMP'

Output Log Dsn: . . .  'yourid.SAA.YYY.LOG'

Programming Language:  COBOL   (ASM,C,C++,COBOL,PL/I,LASM,LCOB,or LPL/I)


Select Line Audit Filters (Y = apply filter, N = do not apply filter):
  Comments  Y (Y or N)   Declares  Y (Y or N)   Reformatted  Y (Y or N)

Select Comparison Columns:
  Start Col  1   (1-176)   End Col  72  (1-176)

DBCS support:
  Enable  N (Y or N)

Press END to terminate
```

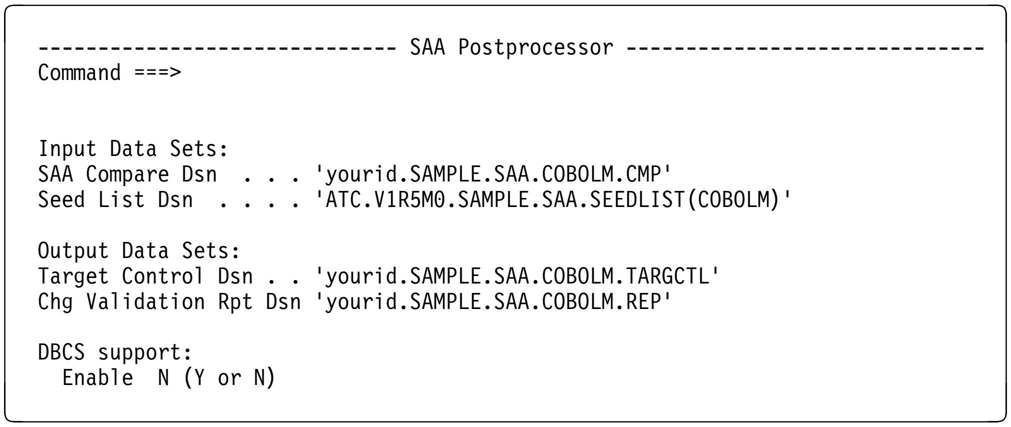
Figure 9. SAA Foreground Execution Parameters Panel

Creating an SAA Change Validation Report and Targeted Summary Control File

After you run a comparison of your code base, you can then run the SAA postprocessor, which will create an SAA change validation report and a prototype targeted summary control file that can be used as input to the Coverage Assistant.

In order to create an SAA change validation report, select the `Postprocessor` option on the `Execute Source Audit Assistant` panel. The `SAA Postprocessor` panel is displayed. Simply fill in the requested fields and press Enter to run the SAA postprocessor.

Figure 10 on page 31 shows the `SAA Postprocessor` panel.



```
------------------------------ SAA Postprocessor ------------------------------
Command ===>


Input Data Sets:
SAA Compare Dsn  . . . 'yourid.SAMPLE.SAA.COBOLM.CMP'
Seed List Dsn  . . . . 'ATC.V1R5M0.SAMPLE.SAA.SEEDLIST(COBOLM)'

Output Data Sets:
Target Control Dsn . . 'yourid.SAMPLE.SAA.COBOLM.TARGCTL'
Chg Validation Rpt Dsn 'yourid.SAMPLE.SAA.COBOLM.REP'

DBCS support:
  Enable  N (Y or N)
```

Figure 10. SAA Postprocessor Panel

Inputs

The most obvious inputs to SAA are the two data sets that contain the old and new source/listings that you want to compare. You can compare two sequential data sets or you can compare two PDS files. If you have PDS files that have different sets of member names, you will only get comparisons for the members that are common to both data sets.

Other key inputs to SAA are as follows:

- Programming language of the file being compared.
- Setting of the three available filters to be used during the comparison.
- Specification of the columns to use during the comparison.

The SAA postprocessor has two main inputs. The first is an SAA compare data set which is the output of the main SAA compare process. It can be a sequential data set or a member of a PDS.

The second input is the seed list data set name, which contains a list of variables that you are expecting to be found in lines of code that have been modified. It can be a sequential data set or a member of a PDS. It contains free-form records, with one or more variables per record. If multiple variables are specified per line, they must be separated by one or more blanks.

Outputs

The SAA has two main outputs. The first is the comparison file, which contains the actual comparison report done by SAA. Figure 11 on page 32 shows a sample comparison report.

```
Source Audit Assistant V01.2            Date 12/02/1997  Time 11.07
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         New File Name ->  ATC.V1R5M0.SAMPLE.SAA.NEW.COBOL(COBOLM)
         Old File Name ->  ATC.V1R5M0.SAMPLE.SAA.OLD.COBOL(COBOLM)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                        COBOL comments have been included
                        COBOL declares have been included
                        Reformatted lines have been included
                        The compare was from column 1 to column 176
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Line #  File  Contents
------  ----
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----10---+----11---+----

THE FOLLOWING LINE(S) HAVE BEEN DELETED

   312   Old              WHEN NUM-MINUTES > 0 AND <= 20
   313   Old                 COMPUTE INIT-COST = INIT-COST + (NUM-MINUTES * 8)

THE FOLLOWING LINE PAIR(S) HAVE BEEN REFORMATTED

   313   New                 COMPUTE INIT-COST = INIT-COST + ( NUM-MINUTES * 7 )
   315   Old                 COMPUTE INIT-COST = INIT-COST + (NUM-MINUTES * 7)


THE FOLLOWING LINE(S) HAVE BEEN INSERTED

   369   New            MOVE 3        TO COST.

THE FOLLOWING LINE PAIR(S) HAVE BEEN REPLACED

   401   New            VARYING SUB1 FROM 2 BY 1 UNTIL SUB1 = CALLS-MADE
   402   Old            VARYING SUB1 FROM 1 BY 1 UNTIL SUB1 = CALLS-MADE

   431   New            COMPUTE INIT-COST = INIT-COST + TOTAL-COST + 6.
   432   Old            COMPUTE INIT-COST = INIT-COST + TOTAL-COST + 5.

   460   New            COMPUTE INIT-COST = INIT-COST + TOTAL-COST + 20.
   461   Old            COMPUTE INIT-COST = INIT-COST + TOTAL-COST + 10.
```

Figure 11. Output Report Created by Comparing COBOL Source Files

For your output file, if you run in the background, you have the option of specifying a PDS or a sequential data set to contain the comparison report. If your input files are PDS files (which is typical) and you specify a PDS for the output, you will get comparison reports individually in the output PDS. The member names of the input PDS files will be used as the member names of the output PDS files. If you specify a sequential file for the output file in this same case, all the various member compare reports will be combined into the single sequential data set.

If you are running in foreground, you only have the sequential file option for the output file. In either case, if you intend to use the SAA postprocessor to generate a prototype targeted summary control file, it is highly recommended that your SAA compare output file always be a sequential file.

The second output is the SAA log file. Primarily, this file simply logs any errors that SAA encounters while it is doing the comparison. This file will be uninteresting to you 99.9% of the time, but if SAA fails for some reason and the problem is reported to IBM, this file may be requested since it may contain information to help the development team determine what the problem is.

The SAA postprocessor has two main outputs. The first is the SAA change validation report, which, as described previously, contains information to help you audit your changes against a set of variables that you were expecting to see in the changes to the code.

The second output file is a prototype targeted summary control file. This targeted summary control file, when completed, can be used as input to the Coverage Assistant to generate a targeted summary report.

Distillation Assistant

What Problem Does Distillation Assistant Solve?

Distillation Assistant (DA) is a tool provided to help you reduce the time and resources required in your testing process by distilling the input data sets processed by your application.

Questions and Answers

1. **What do you mean by "distillation"?**

   Distillation is the reduction of a data set used as an input by your application to the minimum number of records needed to provide code coverage equivalent to that provided by the original data set. This distilled data set can then be used in future tests in place of the larger original data set.

2. **Equivalent code coverage?**

   Code coverage measures which executable statements in an application were actually executed during a test run. For a more complete description of coverage, please see "Coverage Assistant" on page 11. Distillation Assistant can usually produce a much smaller data set than the original, which can provide nearly equivalent code coverage.

3. **Why only nearly equivalent?**

   No distilled data set can be guaranteed to provide equivalent coverage in all cases. There are certain cases where equal coverage cannot be attained at a reasonable cost. One example is when a branch depends on the summation of a field from multiple records. Only the first record and the one which put the summation over the limit would be saved. However, in many cases the loss of coverage is minimal.

4. **How is the distillation done?**

   | In the setup step, DA adds breakpoints to a copy of your application's object or
   | load modules. As your application runs, the breakpoints are removed as the statements are executed, and the key of any record that caused new coverage is saved.

   The actual distillation is done in two steps:

   a. Logical distillation

      Collecting keys while running your application test suite under the control of the monitor.

   b. Physical distillation

      Creating a new smaller file from your master file, including only the records with the keys that were gathered in the logical distillation step.

5. **What are these keys?**

   The keys are any contiguous area in a record that uniquely identifies that record. The keys can contain any data type and can be up to 126 bytes long.

The keys must also be the same length and in the same position in each record.

6. **Do the keys have to be unique?**

   No. Nonunique keys can be used, but the distillation will not be as effective. All records with the same key as the one that caused new coverage will be included in the distilled file.

7. **What kinds of data sets can I distill?**

   The input master data set can be any sequential or VSAM data set that contains logical keys. However, note the following input master data set restrictions:

   - If the data set is sequential, the RECFM may be any valid MVS RECFM except VS and VBS. Spanned records are not supported.
   - VSAM data sets with either KSDS or ESDS organizations can be distilled. However, VSAM data sets that have alternate indexes will not have the corresponding alternate indexes built into the new master data set.
   - Any type of VSAM data set that cannot be distilled directly can be copied using the IDCAMS REPRO function to a sequential data set, which can be distilled and copied back to a VSAM data set.

8. **What languages does Distillation Assistant support?**

   - IBM COBOL for OS/390 & VM 2.1 (plus Millennium Language Extensions [MLE])
   - IBM COBOL for MVS & VM 1.2 (plus Millennium Language Extensions [MLE])
   - IBM VS COBOL II Release 4.0
   - IBM OS/VS COBOL Release 2.4
   - IBM PL/I for MVS & VM 1.1.1 (plus Millennium Language Extensions [MLE])
   - IBM OS PL/I Optimizing Compiler 2.3.0
   - IBM PL/I Optimizing Compiler 1.5.1

   Note: For PL/I, support is for record I/O only.

Overview of How to Use Distillation Assistant

A typical distillation run would look like this:

1. Compile or assemble your routines (compile units) using the options required by DA. Save the listings and object modules into data sets.[3]
2. Create a DA control file that defines your collection of compile units.

   This control file consists of one control statement per compile unit that defines the name of the compiler listing, the object module data set produced by the compiler, and the output object module data set produced by DA. It also contains control statements defining the program or paragraph containing the file reads, the file name used in the application, and the length and location of the key in the records.[3]

[3] Note that an object module can be instrumented and then linked into a new load module, or a load module can be instrumented directly.

3. Start the ISPF panel interface, create the JCL for the SETUP program with DA enabled, and run it.

   This program analyzes your compiler listings and creates a new set of object modules that contain DA breakpoints. In addition, SETUP creates data sets called BRKTAB (breakpoint table) and DBGTAB (debug table).[3]

   Experienced users can integrate this SETUP program into their normal compile procedures rather than use the ISPF interface every time.

4. Link edit your program(s) using the new object modules created by DA SETUP.[3]

5. Start a DA monitor session with the BRKTAB and DBGTAB data sets created by the SETUP program.

6. Run your test cases.

   The DA monitor intercepts the breakpoints inserted into your program and records the keys of all records which cause new coverage. The breakpoints are removed as they are encountered, so the performance impact is minimal. The handling of the breakpoints is transparent to your program (and any run time that you are using).

7. Stop the DA monitor session.

8. If you choose, generate the JCL to create the DA key list, an editable file in which you can add or delete keys.

9. Generate the JCL to distill the master data set using the key list.

Inputs

Distillation Assistant requires the following inputs:

- A compiler listing file generated with the required options.
- The object or load modules which make up your application.
- A control file containing information on the data sets to be used, and the file reads to be monitored. You can see an example of this control file in Figure 12.

```
COBOL ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(COB11M),
      LoadMod=COB11M,
      FromObjDsn=ATC.V1R5M0.SAMPLE.OBJ,
      ToObjDsn=ATC.V1R5M0.SAMPLE.ZAPOBJ

Scope ExtProgram-Id=COB11M
File  File=QSAMIN,KeyPosition=15,KeyLen=20
```

Figure 12. Distillation Control File for COB11M

Outputs

Distillation Assistant provides the following outputs:

- An intermediate file containing the list of keys. This file can be edited to add or remove keys as desired.
- A data set distilled from your master data set containing only those records with keys in the key list.

Unit Test Assistant

What Problem Does Unit Test Assistant Solve?

The Unit Test Assistant (UTA) is a tool that allows you to log and change (warp) the values of selected variables in your application programs by intercepting them at user-defined points during program execution. UTA offers these two important functions while avoiding the need for modifications to source code or input files. You execute your programs in your normal runtime environment, not under the control of a debugger.

A standard data warping process is to age, or warp, occurrences of dates in the input data files. The UTA file warp feature can copy your input files and then warp specified fields in the copied files for testing.

Questions and Answers

1. **When would UTA be used?**

   UTA logs variable values during the application test run. The log can then be examined for trace back debugging. For instance, if during a test run of a remediated application a date variable becomes contaminated with invalid data, UTA can be used to locate the I/O record and source statement in which the contamination originated.

   Furthermore, UTA's data warping capability enables you to statically warp dates in input files or dynamically warp dates during runtime. Dynamic date warping allows you to perform runtime Year 2000 testing of date variables without the need to maintain aged input files or a system whose clock has been set to a future date. When combined with CA, you can easily determine the effects of post-1999 dates on program execution.

2. **What exactly is data warping?**

   Data warping is the modification of the data used by an application program. In the context of the Year 2000 problem, it usually refers to the aging of all date variables to dates on or around the year 2000. One way to do this is to physically change the data in your input files so that the year portion of every date is post-1999. UTA's warping capabilities can help you do this. The file warp feature modifies dates in input files and the dynamic data warping feature modifies dates during program execution. With dynamic data warping, the values of dates are advanced only in program storage.

3. **What does UTA provide that is not provided by an interactive debugger, such as IBM Debug Tool?**

   UTA, like CA and DA, does not require an interactive terminal session during test case execution and therefore operates independently of the environment under which the application is running. For example, the programs to be tested can execute in batch, CICS, IMS TM, or ISPF.

   In the case of a batch application reading an input file of 1000 records, each record containing multiple fields, at least one of which is numeric, UTA's value is evident. If you are attempting to identify which record is causing the program logic to contaminate a data field, it would be impractical to interactively step through all of the records and examine the data value at each application instruction. Similarly, large amounts of time could be wasted if you had to alter the value of a numeric field every time a new record was read. Using the proper control file, UTA will not only log the data values for analysis after the application test run has completed, but it will also “warp” the numeric field of each record as it enters program storage.

4. **How does Unit Test Assistant log or dynamically warp variables?**

   A user-edited control file determines which variables are logged, dynamically warped, or both, and where in the program these actions will occur. A setup step analyzes your listings that contain the variables you want to log or warp. The setup step inserts user SVC breakpoints at appropriate places in your object or load modules. You start a monitor session to wait for these breakpoints to be executed. You then run your programs normally. When a breakpoint is executed, UTA gets control and either places the value of a selected variable into a log file or warps the value of a selected numeric variable. For COBOL, UTA can log or warp most program variables. It has the ability to determine if the requested variable is being evaluated from within a data structure or by itself. For PL/I, UTA can warp file input buffers. After you are finished testing your programs, you can generate a report of the logged variables.

5. **What compilers are supported?**

   - IBM COBOL for OS/390 & VM 2.1 (plus Millennium Language Extensions [MLE])
   - IBM COBOL for MVS & VM 1.2 (plus Millennium Language Extensions [MLE])
   - IBM VS COBOL II Release 4.0
   - IBM OS/VS COBOL Release 2.4
   - The following PL/I compilers are supported for data warping of file input buffers **only**:
     - IBM PL/I for MVS & VM 1.1.1 (plus Millennium Language Extensions [MLE])
     - IBM OS PL/I Optimizing Compiler 2.3.0
     - IBM PL/I Optimizing Compiler 1.5.1

6. **How does Unit Test Assistant file warp work?**

   The UTA file warp feature changes date fields in flat files or VSAM files. You supply a warp control file to control which fields are incremented, decremented, or set to a value. Each field that is warped can be controlled by the contents of another field in the file by assigning a label to the control field. You can then modify other fields based upon the value of the labeled field by referencing the label.

   For instance, you might specify that records referencing a field labeled 1 be incremented by 1, incremented by 2, or decremented by 1, depending on the value of the labeled control field when the warp occurs. Records referencing a field labeled 2 might be incremented by 4 or incremented by five, depending on the value of the labeled field when the warp occurs. The warp control file is similar in structure to a copy book defining your input file.

Some major advantages of using Unit Test Assistant are:

- Low overhead. UTA typically adds very little execution time to a program.
- Panel-driven user interface. You can use an ISPF panel-driven interface to create JCL for executing UTA programs.
- Simple, flexible control. The control file used to define monitored/warped variables provides a simple method of controlling operations.

Overview of How to Use Unit Test Assistant to Log or Warp Variables

A typical run would be as follows:

1. Compile your routines (compile units) that contain variables to log/warp using the options required by UTA. Save the listings and object modules into data sets.[4]
2. Create a UTA control file that defines the variables which you want to log or dynamically warp. This control file consists of one control statement per compile unit describing the data sets to be used in this test, followed by statements describing the variables of interest and the action(s) to be performed for each variable.
3. Create the JCL for the SETUP program and run it. This can be generated from the ISPF panel interface with the `Enable UTA` option set to `Yes`. The SETUP program analyzes the listings and creates a copy of the object modules with breakpoints inserted.[4] Experienced users can integrate this SETUP program into their normal compile procedures rather than use the ISPF interface every time.
4. Link edit your program(s) using the copies of the object modules with the breakpoints inserted by the SETUP program.[4]
5. Create the JCL to start a monitor session with UTA enabled (text deleted) and run it.
6. Run your test cases as normal. UTA will intercept the breakpoints and perform log/warp actions as specified in the control file.
7. Stop the monitor session.
8. Create and run the UTA report JCL. This creates the report of logged variables.

Inputs

Unit Test Assistant requires the following inputs:

- Compiler listings containing the variables to be logged/warped.
- The object or load modules containing the variables to be logged/warped.
- A control file which defines the data sets to be used, the variables of interest, and the action to be taken for each variable.

[4] Note that an object module can be instrumented and then linked into a new load module, or a load module can be instrumented directly.

Figure 13 on page 42 is an example of a UTA control file.

```
     Defaults ListDsn=ATC.V1R5M0.SAMPLE.COBOLST(*),
              LoadMod=COB02M,
              FromObjDsn=ATC.V1R5M0.SAMPLE.OBJ,
              ToObjDsn=ATC.V1R5M0.SAMPLE.ZAPOBJ

     COBOL ListMember=COB02M

     Scope ExtProgram-Id=COB02M

     Variable Name=JULIAN-DATE
*                  set JULIAN-DATE using Data Warping
         Warp Action=Set, Value=0099365,
              Datatype=Zoned,Unsigned,Stmts=(87)

         Coverage Stmts=(94) // read JULIAN-DATE after it is warped and modified

     Variable Name=CURR-DATE
*                  set CURR-DATE using Data Warping
         Warp Action=Set, Value=00991231,
              Datatype=Zoned,Unsigned,Stmts=(97)

         Coverage Stmts=(103) // read CURR-DATE after it is warped

     Variable Name=YEAR4
*                  increment YEAR4 by 3 using Data Warping
         Warp Action=Increment,Value=3,
              Datatype=Zoned,Unsigned,Stmts=(106)

         Coverage Length=4,
              Stmts=(114) // read YEAR4 after it is warped and modified

     Variable Name=YEAR IN YEAR-BY-FIELD IN DATE-BY-FIELD
*                  decrement YEAR by 1 using Data Warping
         Warp Action=Decrement, Value=1,
              Datatype=Zoned,Unsigned,Stmts=(106)

     Variable Name=YEAR2
         Coverage Length=2,
              Stmts=(117) // read YEAR2 after it gets the 2 digit
*                            year from CURR-DATE decremented by 1

     Variable Name=(BEGIN-DATE In LOAN)
         Coverage Length=8,
              NAME               // read BEGIN-DATE of LOAN
*                         structure by NAME

         Coverage Length=8,
              FULL               // read BEGIN-DATE of LOAN structure
*                         by FULL (when LOAN referenced)

     Variable Name=JULIAN-DATE
         Coverage Length=7,
               Stmts=(143) // read initialization of INC-DATE

     Variable Name=J-DAY IN J-DATE
         Coverage Length=3,ReadEvery=100,
              MaxSave=5,Stmts=(153) // read J-DAY in loop
*                         every 100 times maximum of 5 times
```

Figure 13. Control File for COB02M

Outputs

The UTA report contains the logged variable data. To reduce the amount of logging, you can specify options in the control file that instruct UTA to log only on the first execution of the statement, or only on some specified interval, for example every fifth time. A sample report follows:

```
* DATE: 12/15/1998
* TIME: 09:47:48
*
*
* var-ID  CU-Name    prog-ID          var-name                         stmt-num    data
*---------------------------------------------------------------------------------------
      1  COB02M     COB02M           JULIAN-DATE                            94    1999365
     10  COB02M     COB02M           BEGIN-DATE of LOAN                     97
      9  COB02M     COB02M           BEGIN-DATE of LOAN                     97
      3  COB02M     COB02M           CURR-DATE                             103    00991231
      5  COB02M     COB02M           YEAR4                                 114    2002
     10  COB02M     COB02M           BEGIN-DATE of LOAN                    117    00991231
      9  COB02M     COB02M           BEGIN-DATE of LOAN                    117    00991231
      8  COB02M     COB02M           YEAR2                                 117    98
     10  COB02M     COB02M           BEGIN-DATE of LOAN                    124    19981231
     10  COB02M     COB02M           BEGIN-DATE of LOAN                    127    19981231
     10  COB02M     COB02M           BEGIN-DATE of LOAN                    130    19981231
     10  COB02M     COB02M           BEGIN-DATE of LOAN                    134    19981231
     12  COB02M     COB02M           J-DAY of J-DATE                       153    098
     12  COB02M     COB02M           J-DAY of J-DATE                       153    198
     12  COB02M     COB02M           J-DAY of J-DATE                       153    298
     12  COB02M     COB02M           J-DAY of J-DATE                       153    032
     12  COB02M     COB02M           J-DAY of J-DATE                       153    132
     11  COB02M     COB02M           JULIAN-DATE                           143    2002267
```

Figure 14. Report for COB02M

Overview of How to Use UTA's File Warp Feature

For a file that has the following structure:

```
 01  EMPLOYEE-RECORD.
     03  EMPLOYEE_NAME                  PIC (X)20.
     03  SOCIAL_SECURITY_NUMBER         PIC 9(9).
     03  SPACE1                         PIC (X)1.

     03  HIRE_DATE

                   05   YEAR            PIC 9(2).
                   05   MONTH           PIC 9(2).
                   05   DAY             PIC 9(2).
     03  SPACE2                         PIC (X)1.

     03  LAST_PROMOTION_DATE
                   05   YEAR            PIC 9(2) PACKED-DECIMAL.
                   05   MONTH           PIC 9(2) PACKED-DECIMAL.
                   05   DAY             PIC 9(2) PACKED-DECIMAL.
     03  SPACE3                         PIC (X)1.
     03  CURRENT_LEVEL                  PIC 9(1).
     03  SPACE4                         PIC (X)1.
     03  CURRENT_SALARY                 PIC 9(7).
```

Figure 15. Sample Record Structure for File Warp

The following warp control file could be used:

```
01  EMPLOYEE-RECORD.
          03  EMPLOYEE_NAME              20 CHARACTER
          03  SOCIAL_SECURITY_NUMBER      9 ZONED  = 0
          03  SPACE1                      1 CHARACTER
          03  HIRE_DATE
                   05   YEAR              2 ZONED = 01
                   05   MONTH             2 ZONED
                   05   DAY               2 ZONED
          03  SPACE2                      1 CHARACTER
          03  LAST_PROMOTION_DATE
                   05 YEAR                2 PACKED SIGNED
R1:            1                          2 PACKED SIGNED + 1
R1:            2                          2 PACKED SIGNED + 2
R1:           DEFAULT                     2 PACKED SIGNED - 1
                   05 MONTH               2 PACKED SIGNED
                   05 DAY                 2 PACKED SIGNED
           03  SPACE3                     1 CHARACTER
1:         03  CURRENT_LEVEL              1 ZONED
           03  SPACE4                     1 CHARACTER
           03  CURRENT_SALARY             7 ZONED = 0
```

Figure 16. Warp Control File Example

After creating the warp control file using your control book as a model, you execute the ATC file warp program. ATC file warp copies your input file and warps the fields defined in the warp control file. In the previous example, the following fields would be warped as follows:

- For all records:
  - The SOCIAL_SECURITY_NUMBER and CURRENT_SALARY fields are set to 0.
  - The 2-digit zoned field YEAR in HIRE_DATE is set to 01.
- For records with CURRENT_LEVEL = 1, 1 is added to the packed field YEAR in LAST_PROMOTION_DATE.
- For records with CURRENT_LEVEL = 2, 2 is added to the packed field YEAR in LAST_PROMOTION_DATE.
- For records with any other CURRENT_LEVEL, 1 is subtracted to the packed field YEAR in LAST_PROMOTION_DATE.

Automated Regression Testing Tool

What Problem Does Automated Regression Testing Tool Solve?

Automated Regression Testing Tool (ARTT) is a black box test tool that captures I/O events and data during an application's baseline execution and compares it with output from the application's proof execution. ARTT can also modify data automatically as it is encountered during execution to make it compatible with the format expected by the application, a feature especially useful when date testing year 2000 remediated code.

Because Automated Regression Testing Tool automates much of the testing process, your regression testing is more efficient. With ARTT you can:

- Complete testing more quickly, thereby saving cycle time and reducing costs.
- Complete more testing.
- Perform all levels of testing (unit, function, integration, and system) with or without a production system.
- Improve the quality of your tests. Because Automated Regression Testing Tool always uses the same input and produces the same type of output, you can have more confidence in the validity of your measurements.
- Test earlier and later in the testing cycle.
- Test when programs and inputs are at different levels of remediation.

Questions and Answers

1. **What is regression testing?**

   Regression testing is black box function testing that attempts to ensure that no errors were introduced and no loss of function occurred when changes were made to an existing application. It does this by comparing the output from a baseline execution of the application before modification with the output from a proof run of the application after modification. Typically, this should be done at each stage of testing (unit, function, integration, and system) as the changed application is promoted into production.

2. **Can't I just capture my own test cases and then run them over again myself?**

   Yes you can, but without tools you may have to have highly skilled people to set up the environment, run the test cases, and then interpret whether the test cases were successful, AND you will need the same highly skilled people to run the test cases each and every time. A more effective and economical way is to use a good set of tools.

3. **OK, so how does ARTT help me to do a better job of regression testing?**

   ARTT assists regression testing in several ways. For instance, ARTT:

   a. Allows testing to occur separately from the I/O environment.
   b. Automatically compares outputs and alerts you to differences that indicate changed application behavior.
   c. Allows programs with both transformed and untransformed data to be tested in integration with other systems or files.

4. **What do you mean by allowing me to run my testing separately from the I/O environment?**

   You can capture the initial baseline execution with ARTT, copy the log file offsite, and then replay modified versions of the program again and again without having to copy real data or set up complex environments. ARTT can test applications running in batch, CICS, DB2®, and IMS.

5. **How does the tool tell me if there is a problem with a test case that has previously run successfully?**

   ARTT automatically generates both a summary report of all I/O activity, including the number of differences found between the Capture and Replay runs, and a detailed listing, in both hex and EBCDIC, of every difference found.

6. **Since ARTT is logging data as it comes into and goes out of the program, can it operate on or make changes to the data as it does that?**

   Yes. Data can be transformed to and from one format to another. For example, ARTT could change a YYMMDD (981109) format to a DDMMMYYYY (09Nov1998) format, as well as rejuvenate or age all dates automatically as they enter and exit the program.

7. **What about Year 2000 testing? Does ARTT help me in this special testing area?**

   Yes. It helps with Year 2000 testing in a number of ways. ARTT supports over 100 different date formats that can be used to transform and compare data on the fly. For a date expansion solution, this capability allows ARTT to work with any combination of transformed or untransformed programs and transformed or untransformed files.

   For example, suppose that your input files contain dates in the format YYMMDD (981109), your program expects dates in the format DDMMMYYYY (09NOV1998), and your output files must contain dates in the same format as your input files. In this scenario, your testing could come to an abrupt halt without a method of converting data.

   ARTT's data conversion capability solves compatibility problems by dynamically transforming data to required formats, as well as automatically rejuvenating or aging all dates entering and exiting the program. In this way, ARTT allows testing to continue uninterrupted.

8. **So can I do time or future date testing with ARTT?**

   ARTT can roll input and output dates either forward or backward by any number of days or years, which allows time testing without statically aging data files and databases. Therefore, with ARTT's date rolling capability, you could test any future dates by instructing ARTT to dynamically roll dates forward (age) as they enter or exit the program, and you could ensure proper functioning of past dates by having ARTT dynamically roll dates backward (rejuvenate) as they enter or exit the program.

9. **Are there any other Year 2000 testing features?**

   When Y2K date expansion is required, ARTT could be used as a bridge in a production environment to allow applications that have been remediated to run with data files or databases that have not been remediated. This capability eliminates the cost of having to write throwaway “bridge” programs.

10. **What environments does ARTT support?**

    ARTT can be used to test applications running in the following MVS environments. You can even capture the initial baseline execution with ARTT, copy the log file offsite, and then replay modified versions of the program again and again without the actual I/O data or the I/O environment.

    Batch ARTT supports QSAM and VSAM.

    CICS ARTT supports applications invoked by a terminal as well as the following transaction types:
    - Terminal control
    - File control
    - Basic mapping support
    - IMS attach facility
    - DB2 attach facility

    DB2 ARTT supports applications invoked by the DSN command processor, and only static SQL statements within those applications.

    IMS ARTT supports BMP, DL/I, and MPP applications. You should run all MPP applications that are within a region as a unit in Capture mode and in Virtual Replay mode.

Overview of How to Use Automated Regression Testing Tool

The following figures illustrate running your programs in each of ARTT's execution modes. For detailed instructions on using ARTT, see the ARTT user documentation, which was shipped with the Application Testing Collection.

A typical Capture run under ARTT control would look like this:

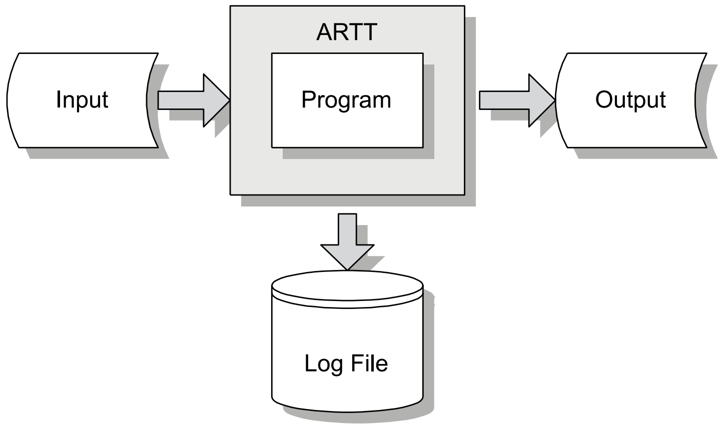


Figure 17. Batch Program Running in ARTT Capture Mode

A typical Real Replay run under ARTT control would look like this:

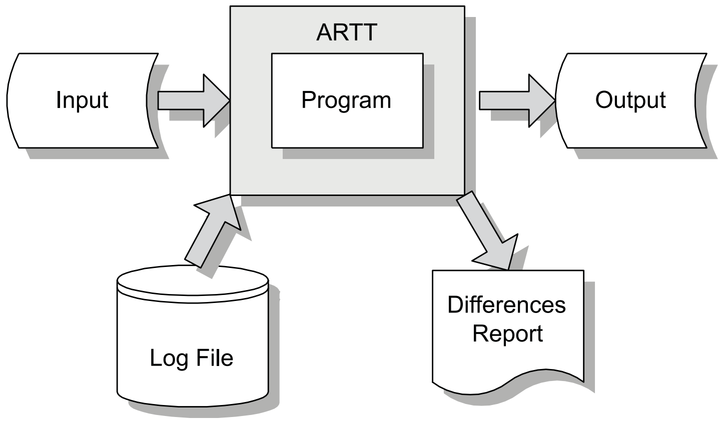


Figure 18. Batch Program Running in ARTT Real Replay Mode

A typical Virtual Replay run under ARTT control would look like this:

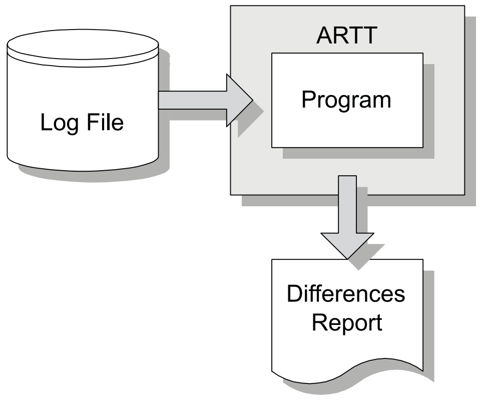


Figure 19. Batch Program Running in ARTT Virtual Replay Mode

Inputs

Capture Mode To run in Capture mode, you need to supply the logical JOB and STEP specifications of the program you want to capture.

Replay Modes To run in either of the Replay modes, you need to supply the logical JOB and STEP specifications of the program you want to test. In addition, if you want to do date rolling or data conversion, you must provide the logical file record formats for the data in question.

Outputs

Capture Report

ARTT automatically generates a Capture report.

```
1ARTT V2.R2.M0 -- Execution Report                                                   Date 03/04/1999    Time 21:24:44    Page 0001
 Copyright (C) 1996-1999 National Westminster Bank, Plc., All Rights Reserved.
-
 ATGA0001I Program Control File = ATC.ART.V2R2M0.PROG.VSAM
 ATGA0002I File Control File. . = ATC.ART.V2R2M0.FILE.VSAM
 ATGA0003I ARMS Control File. . = ATC.ART.V2R2M0.ARMS.VSAM
 ATGA0004I User Library . . . . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0005I Conversion Library . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0006I Compare Library. . . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0008I Test Control File. . = ATC.ART.V2R2M0.TEST.VSAM
 ATGA0704I Log File . . . . . . = ATC.ART.V2R2M0.ARTP020C.BASE.LOG
 ATGA0705I Program ARTP020C executed with:
 ATGA0706I    Run Mode = CAPTURE
 ATGA0707I    Status = N
 ATGA0708I    Run Id = BASE
 ATGA0030I Task ARTP020C attached with:
 ATGA0031I    Parameters = "" (0 BYTES).
 ATGA0032I Initialisation complete. Passing control to application.
 ATGA0040I Program ARTP020C completed with Return Code = 4.
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0713I I/O Activity from Current Run:
 ATGA0714I Name     s  Opens Closes   Inputs  Outputs    Diffs    Shown ConvMod  CompMod  Dataset Name
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0715I SYSOUT   N      1      1        0        1        0        0                   NICH.NICHA.JOB21974.D0000109.?
 ATGA0715I SYSPRINT N      1      1        0       23        0        0                   NICH.NICHA.JOB21974.D0000108.?
 ATGA0715I ARTPH20  Y      1      1        3        0        0        0 FAMP030
 ATGA0715I ARTFKSDI N      1      1        6        0        0        0                   NICH.ARTFKSIN.VSAM
 ATGA0715I ARTFKSDO N      2      2        1        4        0        0 FAMP010           NICH.ARTFKSON.VSAM
 ATGA0715I ARTFESDI N      1      1        4        0        0        0                   NICH.ARTFESIN.VSAM
 ATGA0715I ARTFESDO N      1      1        0        3        0        0                   NICH.ARTFESON.VSAM
 ATGA0715I ARTFRRDI N      1      1        4        0        0        0 FAMP040           NICH.ARTFRRIN.VSAM
 ATGA0715I ARTFRRDO N      1      1        0        3        0        0 FAMP040           NICH.ARTFRRON.VSAM
 ATGA0715I ARTFQSMI N      1      1        4        0        0        0 FAMP020           NICH.ARTFQSIN.QSAM
 ATGA0715I ARTFQSMO N      1      1        0        4        0        0 FAMP020           NICH.ARTFQSON.QSAM
 ATGA0719I Log File Closed.
 ATGA0041I Main task cleanup complete.
```

Figure 20. Sample ARTT Output Report from Capture Mode

The report provides the following types of information:

Header The header information provided in the ARTT Capture report includes general facts about the Capture run, such as the name of the program executed in Capture mode, the run ID, the program's transformed status, and so on.

Summary The ARTT Capture report provides a summary of all I/O activity.

Replay Report

```
1ARTT V2.R2.M0 -- Execution Report                                                   Date 03/04/1999    Time 21:25:00    Page 0001
 Copyright (C) 1996-1999 National Westminster Bank, Plc., All Rights Reserved.
-
 ATGA0001I Program Control File = ATC.ART.V2R2M0.PROG.VSAM
 ATGA0002I File Control File. . = ATC.ART.V2R2M0.FILE.VSAM
 ATGA0003I ARMS Control File. . = ATC.ART.V2R2M0.ARMS.VSAM
 ATGA0004I User Library . . . . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0005I Conversion Library . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0006I Compare Library. . . = ATC.ART.V2R2M0.SATGSLOD
 ATGA0008I Test Control File. . = ATC.ART.V2R2M0.TEST.VSAM
 ATGA0704I Log File . . . . . . = ATC.ART.V2R2M0.ARTP021C.BASE.LOG
 ATGA0705I Program ARTP021C executed with:
 ATGA0706I    Run Mode = REAL
 ATGA0707I    Status = Y
 ATGA0708I    Run Id = BASE
 ATGA0030I Task ARTP021C attached with:
 ATGA0031I    Parameters = "" (0 BYTES).
 ATGA0032I Initialisation complete. Passing control to application.
1ARTT V2.R2.M0 -- Differences Report                                                 Date 03/04/1999    Time 21:25:05    Page 0002
 Copyright (C) 1996-1999 National Westminster Bank, Plc., All Rights Reserved.
-
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Current Resource Id: SYSPRINT                     Logged Resource Id: SYSPRINT
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Cur: ReqNo=00000016 FileReqNo=00000004 Length=00133 RetCode=00000000                   Service=QSAM     Function=PUT
 Log: ReqNo=00000016 FileReqNo=00000004 Length=00133 RetCode=00000000                   Service=QSAM     Function=PUT
                                                                                        |                                |
 Cur: 0000   40C1D9E3 D7F0F2F1 C3406040 C1A4A396   9481A385 8440D985 879985A2 A2899695  | ARTP021C - Automated Regression|
 Log: 0000   40C1D9E3 D7F0F2F0 C3406040 C1A4A396   9481A385 8440D985 879985A2 A2899695  | ARTP020C - Automated Regression|
                             *                                                          |       *                        |
 Cur: 0020   40E385A2 A3899587 40C48594 9695A2A3   9981A389 969540D7 99968799 81944040  | Testing Demonstration Program  |
 Log: 0020   40E385A2 A3899587 40C48594 9695A2A3   9981A389 969540D7 99968799 81944040  | Testing Demonstration Program  |
                                                                                        |                                |
 Cur: 0040   40404040 40404040 40404040 C481A385   40F0F361 F0F461F1 F9F9F940 404040E3  |            Date 03/04/1999    T|
 Log: 0040   40404040 40404040 40404040 4040C481   A38540F0 F361F0F4 61F9F940 404040E3  |              Date 03/04/99    T|
                                        ****** *   ********  ***** * **                 |            *************       |
 Cur: 0060   89948540 F2F17AF2 F57AF0F3 40404040   D7818785 40404040 F1404040 40404040  |ime 21:25:03    Page    1       |
 Log: 0060   89948540 F2F17AF2 F47AF4F6 40404040   D7818785 40404040 F1404040 40404040  |ime 21:24:46    Page    1       |
                                *   * *                                                 |        * **                    |
 Cur: 0080   40404040 40                                                                |                                |
 Log: 0080   40404040 40                                                                |                                |
                                                                                        |                                |
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Current Resource Id: SYSPRINT                     Logged Resource Id: SYSPRINT
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Cur: ReqNo=00000019 FileReqNo=00000007 Length=00133 RetCode=00000000                   Service=QSAM     Function=PUT
 Log: ReqNo=00000019 FileReqNo=00000007 Length=00133 RetCode=00000000                   Service=QSAM     Function=PUT
                                                                                        |                                |
 Cur: 0000   40F0F040 40404040 F0F04040 F0404040   40404040 40F0F0F0 40404040 4040C885  | 00     00  0        000      He|
 Log: 0000   40F0F040 40404040 F0F04040 F0404040   40404040 40F0F0F0 40404040 4040C885  | 00     00  0        000      He|
                                                                                        |                                |
 Cur: 0020   81848599 40998583 96998440 8481A385   4089A240 F0F361F0 F461F1F9 F9F94040  |ader record date is 03/04/1999  |
 Log: 0020   81848599 40998583 96998440 8481A385   4089A240 F0F361F0 F461F9F9 40404040  |ader record date is 03/04/99    |
                                                                          *   ****      |                          * **  |
 Cur: 0040   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040  |                                |
 Log: 0040   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040  |                                |
                                                                                        |                                |
 Cur: 0060   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040  |                                |
 Log: 0060   40404040 40404040 40404040 40404040   40404040 40404040 40404040 40404040  |                                |
                                                                                        |                                |
 Cur: 0080   40404040 40                                                                |                                |
 Log: 0080   40404040 40                                                                |                                |
                                                                                        |                                |
```

Figure 21 (Part 1 of 2). Sample ARTT Output Report from Real Replay Mode

```
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Current Resource Id: ARTFKSDI                    Logged Resource Id: ARTFKSDI
 -----------------------------------------------------------------------------------------------------------------------------------
-
 Cur: ReqNo=00000021 FileReqNo=00000005 Length=00082 RetCode=00000000 FeedBck=00000000  Service=VSAM     Function=GET
 Log: ReqNo=00000021 FileReqNo=00000005 Length=00082 RetCode=00000000 FeedBck=00000000  Service=VSAM     Function=GET
                                                                                        |                                |
 Cur: 0000   F1F0F0F0 F0F0F0F0 F1404040 40404040   40404040 40404040 40404040 4040F1F9  |100000001                     19|
 Log: 0000   F1F0F0F0 F0F0F0F0 F1404040 40404040   40404040 40404040 40404040 4040F2F0  |100000001                     20|
                                                                                   * *  |                              **|
 Cur: 0020   F4F4F0F4 F0F6D2A2 84A240D9 85839699   8440F0F0 F0F0F0F0 F0F14040 40404040  |440406Ksds Record 00000001      |
 Log: 0020   F4F4F0F4 F0F6D2A2 84A240D9 85839699   8440F0F0 F0F0F0F0 F0F14040 40404040  |440406Ksds Record 00000001      |
1ARTT V2.R2.M0 -- Differences Report                                                 Date 03/04/1999    Time 21:25:05    Page 0003
 Copyright (C) 1996-1999 National Westminster Bank, Plc., All Rights Reserved.
-
                                                                                        |                                |
 Cur: 0040   40404040 40404040 40404040 40404040   4040                                 |                                |
 Log: 0040   40404040 40404040 40404040 40404040   4040                                 |                                |

 ...

 ATGA0040I Program ARTP021C completed with Return Code = 0.
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0713I I/O Activity from Current Run:
 ATGA0714I Name     s  Opens Closes   Inputs  Outputs    Diffs    Shown ConvMod  CompMod  Dataset Name
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0715I SYSOUT   Y      1      1        0        1        0        0                   NICH.NICHA.JOB21975.D0000106.?
 ATGA0715I SYSPRINT Y      1      1        0       23       11       11                   NICH.NICHA.JOB21975.D0000105.?
 ATGA0715I ARTPH20  Y      1      1        3        0        1        1 FAMP030
 ATGA0715I ARTFKSDI Y      1      1        6        0        2        2 FAMP010           NICH.ARTFKSIT.VSAM
 ATGA0715I ARTFKSDO Y      2      2        1        4        2        2 FAMP010           NICH.ARTFKSOT.VSAM
 ATGA0715I ARTFESDI Y      1      1        4        0        0        0                   NICH.ARTFESIT.VSAM
 ATGA0715I ARTFESDO Y      1      1        0        3        0        0                   NICH.ARTFESOT.VSAM
 ATGA0715I ARTFRRDI Y      1      1        4        0        1        1 FAMP040           NICH.ARTFRRIT.VSAM
 ATGA0715I ARTFRRDO Y      1      1        0        3        1        1 FAMP040           NICH.ARTFRROT.VSAM
 ATGA0715I ARTFQSMI Y      1      1        4        0        2        2 FAMP020           NICH.ARTFQSIT.QSAM
 ATGA0715I ARTFQSMO Y      1      1        0        4        2        2 FAMP020           NICH.ARTFQSOT.QSAM
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0718I I/O Activity from Capture Run:
 ATGA0714I Name     s  Opens Closes   Inputs  Outputs    Diffs    Shown ConvMod  CompMod  Dataset Name
 ATGA0712I
--------------------------------------------------------------------------------------------------------------------------
 ATGA0715I SYSOUT   N      1      1        0        1        0        0                   NICH.NICHA.JOB21974.D0000109.?
 ATGA0715I SYSPRINT N      1      1        0       23        0        0                   NICH.NICHA.JOB21974.D0000108.?
 ATGA0715I ARTPH20  Y      1      1        3        0        0        0 FAMP030
 ATGA0715I ARTFKSDI N      1      1        6        0        0        0                   NICH.ARTFKSIN.VSAM
 ATGA0715I ARTFKSDO N      2      2        1        4        0        0 FAMP010           NICH.ARTFKSON.VSAM
 ATGA0715I ARTFESDI N      1      1        4        0        0        0                   NICH.ARTFESIN.VSAM
 ATGA0715I ARTFESDO N      1      1        0        3        0        0                   NICH.ARTFESON.VSAM
 ATGA0715I ARTFRRDI N      1      1        4        0        0        0 FAMP040           NICH.ARTFRRIN.VSAM
 ATGA0715I ARTFRRDO N      1      1        0        3        0        0 FAMP040           NICH.ARTFRRON.VSAM
 ATGA0715I ARTFQSMI N      1      1        4        0        0        0 FAMP020           NICH.ARTFQSIN.QSAM
 ATGA0715I ARTFQSMO N      1      1        0        4        0        0 FAMP020           NICH.ARTFQSON.QSAM
 ATGA0719I Log File Closed.
 ATGA0041I Main task cleanup complete.
```

Figure 21 (Part 2 of 2). Sample ARTT Output Report from Real Replay Mode

The report provides the following types of information:

Header The header information provided in the ARTT Replay report includes general facts about the Replay run, such as the name of the program executed in Replay mode, the run ID, the program's transformed status, and so on.

Detail The ARTT Replay report contains a detailed listing of every difference found. The differences are provided in both hex and EBCDIC.

Summary The ARTT Replay report contains a summary of all I/O activity, including the number of differences found between the Capture and Replay runs.

Integrating ATC and Using It in Testing Processes

The tools that comprise ATC are:

- Designed for ease of use
- Engineered to compliment existing testing processes
- Architected to work in an integrated manner to address the Y2K testing problem.

This chapter describes various strategies for integrating ATC into your testing process. The following terms are used throughout:

Baseline	A compile unit before making changes needed to correct a defect or implement a feature
Compile listing	An output file showing a compile unit's source statements produced by a translator
Compile unit	A file containing source statements suitable for submission to a translator
Object file	An output file produced by a translator, suitable for submission to the linker
Run unit	An executable module resulting from the linking of one or more object files
Translator	An assembler or a compiler
White space	One or more space characters in your source code

Most high-level views of a testing process include the following tasks. ATC is designed to compliment these tasks, thereby reducing the impact to an existing test process. Each ATC component can be integrated into one (or more) of these tasks, resulting in an overall strengthening of the testing process.

1. Assemble/compile the baseline compile units.
2. Link object files generated from the baseline compile units to create baseline run units.
3. Execute the baseline run units with test data, capturing baseline input data sets and output results.
4. Change baseline compile units (implementing a feature or correcting a defect) to create new compile units.
5. Compare the baseline compile units to the new compile units, identifying the changed code and accommodating the code review process of the changes.
6. Assemble/compile the new compile units.
7. Link object files generated from the new compile units to create new run units.
8. Execute the new run units with baseline inputs and capture the new output results.
9. Compare the baseline output results to the new output results to verify the accuracy of the changes.

The listed steps should be the base of any Y2K testing process. More detail and implementation can be added, but the core process steps should be the same for all Y2K testing processes.

The remainder of this chapter describes how the each ATC tool could be used to implement the defined Y2K testing process.

Source Audit Assistant Integration

Source Audit Assistant (SAA) assists the testing process by:

1. Providing an efficient way to compare compile units and/or their listing files
2. Indicating records in which changes were made, but were not intended
3. Indicating variables that should have been found in changed lines, but were not
4. Generating a control file that Coverage Assistant can use to produce its targeted summary report.

SAA is first used as a compare tool because it provides the means to determine how a compile unit was changed. By identifying the source statement(s) that differ in the compile units before and after the remediation, the SAA reports can be used to:

- Hold design reviews
- Audit the changes for security or other quality assurance reasons

It should be clear that without actually executing the resulting run units, the differences cannot really be proven satisfactorily. However, SAA provides an efficient and objective method to assist code reviews on the changed compile units and/or compile listings.

SAA has three filters, which may be used to further reduce the volume of differences reported. The most efficient SAA comparison report, in terms of the volume of differences to look at when assisting in code reviews, is the SAA comparison report that is generated when the comments filter and the reformatted filter are set to Yes. Changes to program comments and reformatted source lines (keeping the source line logic the same but changing the source line breaks and/or the source line indentation) will not affect the program function or execution. These filtering features distinguish SAA from other comparison tools.

The second and third uses of SAA are reflected in its name: *auditing* provides a means to target those changes which may not have been intended. This is accomplished by providing SAA with a file containing a list of the variables for which changes were intended (that is, a list of data elements of interest). For example, in the case of Y2K testing, a list of date variables. After producing a comparison file of the differences between a baseline and its related changed compile unit, SAA can apply postprocessing to the comparison file. It then generates a change validation report containing those records which did not contain a data element of interest. It also identifies any of the variables of interest that do not appear in any of the changed source lines. This audit function can identify modified code that may be suspect, because they do not have any relationship to the data elements of interest.

The third use of SAA is to provide data for integration with Coverage Assistant (CA) and its targeted summary function. By generating a targeted summary control file containing variables that were found within changed lines, Coverage Assistant can provide targeted summary reports that focus on code that has been impacted by the changes.

Coverage Assistant Integration

Coverage Assistant (CA) assists the testing process by:

1. Providing an efficient way to see what program logic was executed by test cases
2. Reducing the amount of information to review after a test case has been run by targeting only those compile unit source lines of interest

The tester must ensure that the source lines changed and/or affected by a change are exercised by the test case. CA assists in this verification by indicating all logic that was executed. It can, optionally, target only those lines of code that were affected by changes. The best way to take advantage of CA is to fully integrate it into the normal development/testing process. While it can be invoked through its ISPF interface in an on-demand fashion, this usually results in a time consuming effort for each tester assigned to do testing. When fully integrated, testers can focus on testing, not on coverage monitoring preparation. The integration process is straightforward, consisting of just three steps:

1. Activate CA setup steps in the normal compile and link process
2. Invoke the CA monitor around test cases as they are run
3. Produce and review CA reports

When the first step is complete, CA becomes very scalable. That is, the testers can quickly monitor coverage for 100 (or more) test cases as easily as they can monitor a single compile unit.

This expansion is achieved by modifying the shop's normal compile and link processes/JCL for CA enablement so that the coverage gathering is available to the entire development/test team. Doing this enablement does not mean CA must be used. It simply means that the data sets required for CA to properly function will be created and saved through the normal course of program development and maintenance. Then when CA is required, the proper data sets will have already been created and coverage information can be gathered and reported at will.

By modifying the compile and link procedures slightly, the invocation of CA can be controlled by the test scripts that are submitted. The modifications to the compile and link procedures include:

1. Specifying the CA required translator options
2. Saving the translator listings
3. Building CA control files for each run unit
4. Running the CA setup process against the object files[5]

| [5] Note that an object can be instrumented and then linked into a new load module, or a load module can be instrumented directly.

5. Saving the original and instrumented (created in the previous step) object files[5]
6. Linking both the noninstrumented and instrumented object files into different load libraries[5]

After choosing naming conventions for the listing, object, and load data sets, modify the compile and link JCL to store the CA enabled and non-CA enabled files. Following is a simple naming convention that could be used:

	Before ATC	**Noninstrumented Data Sets**	**Instrumented Data Sets**
Source	SOURCE	Same	Same
Include	MACLIB	Same	Same
Listing	SYSOUT=*	LISTING	Same
Object	&&TEMP	OBJ	ZAPOBJ
Run unit	LOAD	Same	ZAPLOAD

The organization and structure of a shop's data sets will vary so this naming convention will have to be adapted to each situation. The main point to remember is that CA will generate an instrumented object code or load data set that will need to be kept as well as the translator listing, in order to perform its task.

After the compile and link processes have been modified, and all CA control files have been built, CA should be used to measure coverage while executing the baseline run units. Do not be surprised the first time if measurements of 40-50% are produced, since this is what past experience with code coverage has shown. The test team can use this data to augment the existing test bed to raise that percentage to an acceptable level of code coverage. It is up to the test team and/or management to determine what is considered to be an acceptable level of code coverage.

After completing the comparison of the baseline compile units to the new compile units, the test team should use SAA to create CA targeted summary control files. These files, along with the test data from the baseline test, will produce reports that can ensure that all changes made were, in fact, exercised by the test case suite.

This is accomplished even when full run unit coverage is not 100% because it is very possible, even desirable, for the targeted summary to be 100%. That is, those lines that were changed (as indicated by SAA) were actually exercised by the test data during their execution. If code coverage results are not verified as a part of the testing process, latent errors may surface after the application is moved back into production because production data causes the code logic that was left untested during the test cycle to be exercised.

Upon satisfactory results in the coverage reports, the final step of testing should be completed. This verifies that the defect correction or feature implementation has been done correctly from an external point of view.

Finally, with both external test validation and code coverage data showing that all code that has been modified or affected by a modification has been exercised, the application can be moved back into production with a high level of confidence that the code will operate as required.

Distillation Assistant Integration

Distillation Assistant (DA) assists the testing process by providing an efficient way to reduce the volume of a test case's input data, thereby reducing the CPU cycle time a test case takes to process.

To say that Distillation Assistant should be integrated into the testing process may be too strong of a statement. The value of DA comes when it is applied to test cases that test high impact applications. If a test case is not built to be run more than one time, DA may not apply. For those that have an input transaction file which drives the test, DA can be used to optimize resources (save DASD and CPU cycle time) compared to a full-file test run.

The usefulness of DA is directly related to the size of the input data being processed by any given test case and the number of times the test case will be run. Usually there are a few data sets within a test environment that have a large impact on the setup, processing, and restore time used to drive each run of a test case. With DA, the input data can be reduced while maintaining equivalent code coverage of the tested application. In addition, Coverage Assistant information can be derived at the same time the DA process is running. You can, with the CA information as a guide, augment the distilled file to drive up the coverage percentage.

The setup process for DA is very similar to the process for Coverage Assistant. The main difference is in the user provided input control file information and the distillation step itself.

The tester should identify, prior to beginning a test case if the data set or data sets which drive the test case meet the criteria for distillation. If they do, the tester should run the test case with Distillation Assistant enabled to produce a distilled file. This distilled file can then be used on all subsequent runs and will produce equivalent statement coverage of the application.

Note: DA always requires an initial run of the application using the original (large) input file. This run will create the code coverage and key data required to generate the distilled (smaller) file that will be used in subsequent test runs.

Unit Test Assistant Integration

Unit Test Assistant (UTA) assists the testing process by:

1. Providing a method of logging data variable values at selected source locations during test case execution
2. Allowing data items to be “warped” at selected source locations during test case execution.
3. Making copies of input files with date fields warped for testing.

These functions do not require modification to your test suite program logic, and the first two functions do not require modification to your data.

How a tester would use data logging is fairly straightforward:

1. A selected data variable from an application to be tested is defined.
2. Each time the variable is affected by the application logic, the values are logged after the statement executes.
3. At the completion of the test case run, the logged values are reviewed.

If a data value becomes contaminated during a test case run, data logging provides a post execution log to aid in debugging. This capability is unique from that of a standard foreground debugger, in that it does not require either a dedicated foreground terminal session or the intervention of a tester at each statement affecting the data variable.

How a tester would use data warping may be initially less apparent, but this function is equally as powerful as data logging, and more powerful in some types of testing. Data warping allows the tester to intercept data values at selected statements and increment, decrement, or set the values after the statement is executed. This allows the tester to change program data values without modifying the application logic, the test data, or both. For example, a tester could increment an input record value at the record read statement and decrement the value at the statement prior to the record write.

Data values enter and exit an application by data files, system calls, screen input, and program load parameters. UTA data warping provides a way to intercept and modify values entering or exiting through any of these means. Without the UTA data warping function, the tester would be required to modify and maintain separate copies of input and output data files. The tester could also be required to modify program logic to control and test various data values from system and program calls. Using UTA data warping, the tester can modify (and log) data values without modifying the application or test data. However, the UTA data warping function **does** require the tester to understand the program logic of the application being tested.

Also worth noting are the UTA functions that distinguish it from interactive debuggers. An interactive debugger would require a screen session and intervention by the tester at various points in the program logic to change and record data values. This could only be done with languages and subsystems supported by a selected debugger.

UTA functions run in the same fashion as CA and DA functions—unobtrusively, under the control of the monitor. The capability to run in any environment and work with other programs gives UTA logging and warping power that is unattainable with interactive debugger tools. The UTA advantage is evident when you consider the task of testing 100 applications, each processing test files of at least 500 records. In this scenario, if an interactive debugger were used, the tester would be required to intervene as many as 50,000 times through the debugger interface.

The setup process for UTA is similar to CA and even more similar to DA. The program variables to be logged and warped are defined in the ATC control file. For the UTA logging function, the tester defines the variable, and UTA determines all affected program source statements and logs the values when those statements are executed. For UTA data warping, the tester must provide additional information, specifically, the data variable, the warp value, and the program statement at which to warp. To use the UTA warping function, the tester must have an understanding of the program source logic to determine the source statements at which the warp action must take place to achieve the desired results.

Automated Regression Testing Tool Integration

Automated Regression Testing Tool (ARTT) assists:

- The testing process by allowing you to perform regression analysis
- The efficient execution of ATC white box tools by allowing offsite testing with no complex setup.

Summary

This section is not intended to define all applications of the ATC testing tools to a shop's testing process. As stated previously, each shop will have their own implementation of the base testing process, which will have been determined by the unique needs of that organization. What we have attempted to do is *prime* the process of understanding and implementing ATC into each testing shop's process.

Once the base functions and benefits of the ATC tools are understood, they can be applied and integrated into the testing process that is already in place for either normal application support and maintenance or Y2K remediation and testing.

Appendix A. DBCS Support

DBCS[6] support for the Application Testing Collection (ATC) varies among the tools as follows:

Coverage Assistant (CA), Distillation Assistant (DA), and Unit Test Assistant (UTA)

DBCS characters are permitted in:

- Input compiler and assembler listings
- Control file identifiers and comments.

Source Audit Assistant (SAA)

If the input to SAA contains DBCS characters, then the:

- General compare function (with no filters turned on) is supported.
- Reformatted lines filter is supported.
- Comments and Declares filters are supported for COBOL and assembler.
- Postprocessor is supported.

For more details, see the DBCS information in the *Application Testing Collection for MVS/ESA Version 1 Release 5 Modification 0 User's Guide*.

[6] Double-byte character set. A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS.

Glossary

This glossary defines terminology and acronyms unique to this document or not commonly known.

A

annotated listing. Compiler or assembler listing that contains Coverage Assistant information about the execution.

B

background execution. The execution of lower-priority computer programs when higher-priority programs are not using the system resources. Contrast with *foreground execution*.

baseline execution. An execution in which Automated Regression Testing Tool records a program's I/O activity, including program reads or writes, in a log file, which can then be used as a base for comparison with subsequent proof runs in Real Replay mode or Virtual Replay mode.

black box testing. A testing method that provides information about a test run by comparing an application's inputs and outputs. Black box testing provides limited information about an application's internal execution at run time. Contrast with *white box testing*.

BP. Breakpoint.

breakpoint (BP). The practice of replacing an instruction op code with a user SVC instruction so that the ATC monitor gets control from the operating system.

BRKTAB. The DDNAME of the file of breakpoint data (breakpoint table) created during Coverage Assistant Setup and used during Coverage Assistant Execution.

C

Capture mode. A feature of Automated Regression Testing Tool, which records your program's I/O activity, including program reads or writes, in a log file that can be used as a base for comparison with subsequent proof runs in Real Replay or Virtual Replay mode. See also *Real Replay mode* and *Virtual Replay mode*.

change validation report. A report that allows you to verify changes found in the Source Audit Assistant comparison report against your seed list in order to make sure that only planned changes were made and that all seed variables were changed.

code coverage. A measurement of the number of code statements that have been executed.

comparison report. A Source Audit Assistant report that allows you to see differences between original source data and changed source data and helps you to identify items that need closer examination.

compile unit (CU). The programs contained within one compiler listing.

compression. Any encoding to reduce the number of bits used to represent a given message or record.

control file. A file that contains information describing the compile units to be analyzed, the file that is to be monitored, and the values of the variables to be recorded. Coverage Assistant, Distillation Assistant, and Unit Test Assistant share the same control file.

CU. Compile unit.

D

data aging. See *date rolling*.

data conversion. The process of transforming data input to, or output from, a Real Replay or Virtual Replay run by changing date formats (for example, from mm/dd/yy to mm/dd/yyyy) or by rolling dates forward (aging them) or backward (rejuvenating them).

data rejuvenation. See *date rolling*.

date rolling. A feature of ARTT, which automatically ages dates (by rolling them forward) or rejuvenates dates (by rolling them backward) in input and output files according to values that you specify. Date rolling is useful for verifying that your program will work correctly with past or future dates.

DBCS. Double-byte character set.

DBGTAB. Debug table. The DDNAME of a file generated by the Setup step when Distillation Assistant or Unit Test Assistant is enabled. The DBGTAB is used during the Distillation Assistant Logical Distillation step and the Unit Test Assistant Report step. For a standard coverage run, this file contains no useful data and its DD card is coded DD DUMMY.

distillation. The reduction of a data set to the minimum size that provides the same test coverage as the complete data set.

Double-byte character set. A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. Contrast with *single-byte character set*.

dsname. Data set name.

dynamic data warping. The process of changing predefined variable values during runtime. Dynamic data warping might be used to age, or warp, dates in input data files, for example. Contrast with *file warping*.

E

EBCDIC. Extended binary-coded decimal interchange code. A coded character set of 256 8-bit characters.

Execute. The Coverage Assistant step that monitors your program while it is being executed to collect test case coverage statistics.

Expansion. A method of coding that increases the number of bits used to represent a given message or record.

F

file warping. The process of statically modifying predefined variables in copies of VSAM or QSAM input files to simulate input conditions for testing. File warping might be used to clear fields in test copies of production input files for privacy or security reasons. Contrast with *dynamic data warping*.

filter. In Source Audit Assistant, a capability keeps changes that are not of interest to you from appearing in the report file. You can choose to filter one or more of the following: comments, variable declarations, and reformatted lines.

foreground execution. The execution of a computer program that preempts the use of computer facilities.

I

input master data set. A data set to be distilled into an output master data set by physical distillation.

instrument. To insert instructions into object modules that tell the application to turn control over to the monitor.

K

key. One or more characters used to identify the record and establish the order of the record within an indexed file.

key list. A list of characters used to identify the record and establish the order of the record within an indexed file.

L

logical distillation. Instrumenting your object code and executing the instrumented code under the Distillation Assistant monitor. As the instrumented code reads records from the specified input master data set, the monitor determines which keys in the input master data set caused new code coverage in the instrumented code. The list of these keys is then saved for the physical distillation step.

logical key. As used in reference to distillation, a logical key is simply a field within a data record that can be used to identify the record. This field may, or may not, be identified as a physical key to the file system or database manager involved in actually reading the record.

Multiple records can have the same logical key. In this case, it is assumed that all records with this key are required to obtain the necessary code coverage.

M

Millennium Language Extensions (MLE). A compiler-assisted solution for the Year 2000 problem. Available for IBM's COBOL and PL/I compilers.

MLE. Millennium Language Extensions.

monitor. The program (MONSVC) that measures test case coverage during execution of your programs.

monitor session. A distinct invocation of the monitor program.

N

new source file. A data set you want Source Audit Assistant to compare. Typically, you want to compare a modified data set (new source file) with an original data set (old source file).

O

old source file. A data set you want Source Audit Assistant to compare. Typically, you want to compare an original data set (old source file) with a modified data set (new source file).

P

PA. Program area.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

PDS. See *partitioned data set*.

physical distillation. This step consists of creating a new master data set by reading the list of keys produced in the first step (logical distillation) and the input master data set. The new master data set consists of only those records in the input master data set whose logical key appears in the list of keys.

Program area (PA). Each specific PA contains all of the breakpoints for one COBOL paragraph, PL/I block, or assembler listing.

proof execution. An execution in which Automated Regression Testing Tool monitors a program's I/O events and data and compares them with the events and data recorded in the log file during the program's base run in Capture mode. ARTT records any differences between the two runs in its output report. Perform a proof execution using either ARTT's Real Replay mode or Virtual Replay mode. See also *Real Replay mode* and *Virtual Replay mode*.

Q

QSAM. Queued sequential access method.

Queued sequential access method (QSAM). An extended version of the basic sequential access method (BSAM). When this method is used, a queue is formed of input data blocks that are awaiting processing or of output data blocks that have been processed and are awaiting transfer to auxiliary storage or to an output device.

R

Real Replay mode. A feature of Automated Regression Testing Tool, which monitors the I/O events and data for the proof run and compares them with the events and data recorded in the log file for the program's base run in Capture mode. ARTT records any differences between the two runs in its output report. See also *Virtual Replay mode* and *Capture mode*.

regression testing. The process of verifying that an application functions in the same way as it did before changes were introduced into the application itself, the data on which the application operates, or some related software or hardware on which the application depends.

Report. The Coverage Assistant step that produces the summary and annotated listing reports after a test case run.

S

SAA postprocessor. Generates a change validation report or a prototype Coverage Assistant target control file using a compare file generated by a previous Source Audit Assistant run and a list of seed variables that you want to monitor.

seed variable. A variable name used as input to the Source Audit Assistant postprocessor. This is generally a variable name that was identified as one needing to be changed.

session. A distinct invocation of the monitor program.

SETUP. The Coverage Assistant program that analyzes your assembler listings in order to produce a table of breakpoint data and insert breakpoints into disk resident programs.

summary report. A Coverage Assistant report that provides the summary statistics for PAs.

supervisor call (SVC). A request that serves as the interface into operating system functions, such as allocating storage. The SVC protects the operating system from inappropriate user entry. All operating system requests must be handled by SVCs.

SVC. Supervisor call.

T

targeted summary. A measurement of the number of specific code statements and/or variables that have been executed.

targeted summary report. A report that allows you to target certain statements and/or COBOL or PL/I variables. The format of a targeted summary report is identical to the format of a summary report, except that the content is restricted to statements that you specify. (You can specify a statement number, or you can specify all statements that reference specific COBOL or PL/I variables.)

test bed. A collection of test data.

U

unit testing. A method of capturing and logging values assigned to selected variables in your application program at selected points during their execution.

V

Virtual Replay mode. A feature of Automated Regression Testing Tool, which uses input from the log file to monitor the I/O events and data for the proof run and then compares them with the events and data recorded in the log file for the program's base run in Capture mode. ARTT records any differences between the two runs in its output report. See also *Real Replay mode* and *Capture mode*.

VSAM. Virtual storage access method.

Virtual storage access method (VSAM). An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

W

warping. The process of statically (file warping) or dynamically (dynamic data warping) modifying predefined variables to simulate input conditions for testing. See *dynamic data warping* and *file warping*.

white box testing. A testing method that measures internal application behavior during run time. White box testing provides detailed information about an application's internal execution at run time. Contrast with *black box testing*.

Windowing. Coding that adds logic to arithmetic instructions, which enables values to be interpreted differently.

Index