Application Testing Collection
for MVS/ESA & OS/390

General Information

Version 2 Release 1

GC26-9870-02

Application Testing Collection
for MVS/ESA & OS/390

General Information

Version 2 Release 1

GC26-9870-02

Note!

Before using this information and the product it supports, read the information in “Notices” on page 57.

| **Third Edition (September 2000)**

| This edition applies to version 2 release 1 modification 2 of the IBM® Application Testing Collection for MVS/ESA™ & OS/390®
| (product number 5655-B97) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition
| reflects documentation changes required to support Service APAR's PQ40563, PQ40565, PQ40567, and PQ40568.

Table of contents

About this book . . . vii
Who should read this book . . . vii
Conventions and terminology used in this book . . . vii
Related information . . . vii
Sending your comments . . . viii

Summary of changes . . . ix
Summary of changes (V2R1M2) . . . ix

Introducing the IBM Application Testing Collection . . . 1
Basic development and maintenance testing requirements . . . 2
Testing methods and the Application Testing Collection . . . 2
Black box testing . . . 2
ATC black box testing . . . 2
White box testing . . . 3
ATC white box testing . . . 4

Coverage Assistant . . . 7
What problem does Coverage Assistant solve? . . . 7
Questions and answers . . . 7
Overview of how to use Coverage Assistant . . . 12
Inputs . . . 13
Outputs . . . 15
Summary report . . . 15
Annotated Listing report . . . 17
Targeted Summary report . . . 19

Source Audit Assistant . . . 21
What problem does Source Audit Assistant solve? . . . 21
Questions and answers . . . 21
Overview of how to use Source Audit Assistant . . . 24
Running the SAA Comparison Analysis tool . . . 24
Inputs . . . 25
Outputs . . . 25

Distillation Assistant . . . 27
What problem does Distillation Assistant solve? . . . 27
Questions and answers . . . 27
Overview of how to use Distillation Assistant . . . 29
Inputs . . . 31
Outputs . . . 31

Unit Test Assistant . . . 33
What problem does Unit Test Assistant solve? . . . 33
Questions and answers . . . 33
Overview of how to use Unit Test Assistant to log or warp variables . . . 35
Inputs . . . 36
Outputs . . . 38
Overview of how to use UTA's File Warp feature . . . 38

Automated Regression Testing Tool 41
What problem does Automated Regression Testing Tool solve? 41
Questions and answers 41
Overview of how to use Automated Regression Testing Tool 43
Inputs 45
Outputs 45
Capture report 45
Replay report 46

Integrating ATC and using it in testing processes 49
Source Audit Assistant integration 50
Coverage Assistant integration 50
Distillation Assistant integration 52
Unit Test Assistant integration 53
Automated Regression Testing Tool integration 54
Summary 54

Appendix A. DBCS support 55

Notices 57
Trademarks 58

Glossary 59

Index 63

Figures

1. CA—flow diagram . . . 13
2. Control file for COB01 . . . 14
3. Targeted Summary control file for COB01 . . . 14
4. Summary report for COB01 . . . 16
5. Annotated COBOL listing . . . 17
6. Targeted Summary report for COB01 . . . 19
7. ATC Primary Option Menu . . . 24
8. Source Audit Assistant panel . . . 24
9. Create Prototype Target Control Dataset & Validate Changes panel . . . 25
10. DA—flow diagram . . . 30
11. Distillation Assistant control file for COB03 . . . 31
12. UTA—flow diagram . . . 36
13. Control file for COB02 . . . 37
14. Report for COB02 . . . 38
15. Sample record structure for file warp . . . 38
16. Warp control file example . . . 39
17. Batch program running in ARTT Capture mode . . . 44
18. Batch program running in ARTT Real Replay mode . . . 44
19. Batch program running in ARTT Virtual Replay mode . . . 44
20. Sample ARTT output report from Capture mode . . . 45
21. Sample ARTT output report from Real Replay mode . . . 46
22. Sample naming convention . . . 51

About this book

This book provides an overview and general information about the IBM® Application Testing Collection for MVS/ESA™ & OS/390®. The Application Testing Collection (ATC) is a general purpose application development (AD) test tool suite.

This book also 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 ongoing development and maintenance testing projects.

Who should read this book

You should read this book if you:

- Are involved in the selection of testing tools to be used in application development testing activities
- Will be testing reengineered applications
- Will be testing applications on MVS/ESA or OS/390

Conventions and terminology used in this book

The following list describes special ways in which some characters and words are displayed in this book 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).
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 59.
Bold type	Indicates information that you should pay particular attention to.

Related information

The following publications are available in hardcopy form. They are also available in downloadable form from the the ATC Web site:

Application Testing Collection for MVS/ESA & OS/390 Version 2 Release 1, Product Number 5655-B97:

Automated Regression Testing Tool User's Guide, SC26-9872

Fact Sheet, GC26-9883

Licensed Program Specifications, GC26-9884

Program Directory, GI10-8207

User's Guide, SC26-9871

To locate these publications go to:

`http://www.ibm.com/s390/atc`

Each publication is available in Postscript and the following online formats:

- Book, which can be viewed with the IBM® BookManager® READ and IBM Library Reader™ licensed programs. The IBM Library Reader can be downloaded from the following Web Site: http://booksrv2.raleigh.ibm.com/homepage/ilrserv.html
- PDF, which can be viewed with the Adobe Acrobat Reader 3.0 and later. The Adobe Acrobat Reader can be downloaded from the Adobe Web site.

Sending your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other Application Testing Collection documentation. You can use any of the following methods to provide comments:

- Send your comments by e-mail to atchelp@us.ibm.com. Be sure to include the name of the product, the version number of the product, and the name and part number of the book (if applicable). If you are commenting on specific text, please include the location of the text (for example, a chapter and section title, a table number, a page number, or a help topic title).
- Complete the readers' comment form at the back of the book and return it by mail, by fax (859-243-4345 for the United States and Canada), or by giving it to an IBM representative.

Summary of changes

Summary of changes (V2R1M2)

Changes have been made to the following components of the Application Testing Collection (ATC) for Version 2 Release 1 Modification 2:

- VisualAge PL/I Version 2 Release 2 is now supported by Coverage Assistant and Source Audit Assistant.
- DB2 UDB for OS/390 V6 is now supported by ARTT.
- Minor corrections have been made throughout the book.

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 both basic development and mainframe 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 analysis tool that analyzes a SuperC Compare Utility comparison of your 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 modified or set to selected values at application run time. In addition to dynamically warping values 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 base run of your program and automatically compares it with a proof run 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 transformation capability allows testing to continue when I/O and programs are in incompatible formats (for example, when one has been modified to accept a new file format and the other has not). Data transformation can be particularly useful if you are unsure of the status of I/O received from outside sources.

Basic development and maintenance testing requirements

The two basic requirements of testing are to ensure that programs function correctly or as expected and to ensure that what was added or changed is indeed tested.

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 after modification.

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 scheme.
- 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 base run or checking in real time a modified execution against a previously recorded base run
  - Performing data transformation
  - 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 emerge from one testing effort 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 changes 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 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.

When significant code changes are required for a fix or enhancement, the potential for modified code not being tested is considerable if white box testing is not employed. Entire sections of modified 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 listing reports 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 analyzing a SuperC comparison of the application source or listing before and after modifications are made 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 all fixes (and even some enhancements) is that the basic functionality of the application has not changed, then it is most critical to, at a minimum, test all of the code that has changed. Together, ATC black box and white box test tools can provide concrete data to confirm that you have indeed tested what you have changed.

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 that correspond to end-of-quarter or end-of-year. 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 modified or set to a value. This application of data warping can be used to test numeric conversions such as Euro currency testing.

ATC dynamic data warping

A preferred approach for data warping is providing a means for you to intercept values as they enter and exit the application and allowing you to modify, or *warp*, the values at that point. This allows you to not only modify values 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 testing can be driven from one set of data containing current 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 values enter the application and also define how the values should be modified at those points. This is done without modification to the application source code. The input and output values can be modified or set to a value.

When this UTA function is used with Coverage Assistant, testers can track the application statement execution affected by the warped values. 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 testing process, anything that can be done to decrease testing cycle times is of great value.

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 reduces test case input data sets 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 on an ongoing basis.

Coverage Assistant

What problem does Coverage Assistant solve?

Coverage Assistant (CA) is a test case code coverage measurement tool that tests all lines of code affected by changes. For COBOL, PL/I, or assembler programs that execute on MVS, it measures coverage while the programs are executing, and supplies reports that show which high-level statements and conditions were executed. Coverage Assistant does this without requiring any modifications to the source code or runtime environment.

You can generate targeted reports, which focus coverage on statements impacted by changes in the source code (for example, code impacted by Euro currency changes). These reports provide objective and auditable data on how well changes to the code are being tested.

Questions and answers

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

   Code coverage, at the most basic level, is the measurement of 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 your application executables.[1]

   e. Start the CA monitor session.

   f. Run your test cases. Make up your test case suite against your application.

   g. Stop the CA monitor session.

   h. 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 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.

---

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

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 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 SuperC Compare Utility and Source Audit Assistant to analyze the listings for any of the compile units you modified and produce a Targeted Summary control file.

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

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 Listing report 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, DB2, 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?

Coverage Assistant supports the following compilers and assemblers:

- IBM COBOL for OS/390® & VM 2.1 (5648-A25)
- VisualAge COBOL Millennium Language Extensions for OS/390 & VM 1.0 (5648-MLE)
- IBM COBOL for MVS & VM 1.2 (5688-197)
- VisualAge COBOL Millennium Language Extensions for MVS & VM 1.0 (5654-MLE)
- VS COBOL II Version 1 Release 4.0 (5688-022, 5688-023)
- OS/VS COBOL Version 1 Release 2.4 (5740-CB1, 5740-LM1)
- VisualAge PL/I Compiler Version 2 Release 2 (5655-B22)
- IBM PL/I for MVS & VM 1.1.1 (5688-235)
- VisualAge PL/I Millennium Language Extensions for MVS & VM 1.0 (5648-MLX)
- OS PL/I Optimizing Compiler 2.3.0 (5668-909, 5668-910, 5668-911)
- OS PL/I Optimizing Compiler 1.5.1 (5734-PL1, 5734-PL3, 5734-LM4, 5734-LM5)
- High Level Assembler (HLASM) Version 1 Release 2 and 3 (5696-234)
- High Assembler H (HASM) Version 2 (5668-962)

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]

   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 for the programs to be tested.
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 Listing 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 report 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.

Figure 1 on page 13 illustrates this process.

[2] Note that object modules can be instrumented and then linked into a new load module, or a load module can be instrumented directly. (Direct instrumentation of load modules is not supported for modules generated by the VisualAge PL/I Version 2 Release 2 compiler.) In addition, special support is available for COBOL routines compiled with the TEST compiler option so that compiler listings are not required.

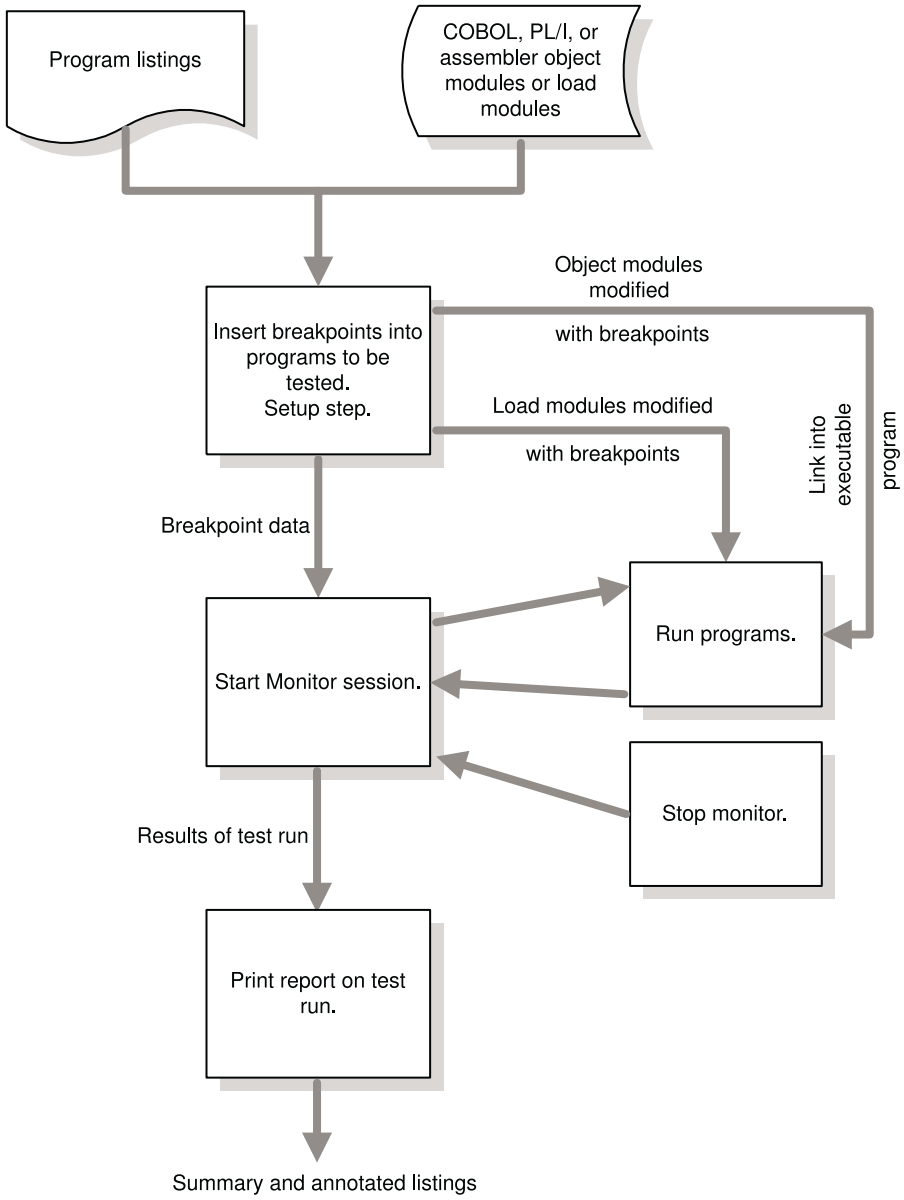

Figure 1. CA—flow diagram

Inputs

Figure 2 on page 14 is an example of a CA control file for a COBOL load module consisting of four 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=YOUNG.SAMPLE.COBOLST(*),
                 LoadMod=COB01,
                 FromObjDsn=YOUNG.SAMPLE.OBJ,
                 ToObjDsn=YOUNG.SAMPLE.ZAPOBJ

        COBOL ListMember=COB01A
        COBOL ListMember=COB01B
        COBOL ListMember=COB01C
        COBOL ListMember=COB01D
```

Figure 2. Control file for COB01

All ATC control files are keyword oriented and are easy to navigate. In the case of Figure 2, 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 YOUNG.SAMPLE.COBOLST.
- The load module is named COB01.
- The data set containing the object code from the compiler is YOUNG.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 YOUNG.SAMPLE.ZAPOBJ.
- The tester/programmer wants to monitor coverage on four compile units (COB01A, COB01B, COB01C, and COB01D).

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 3 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 Comparison Analysis to generate a Targeted Summary control file containing all the variables that have been part of a change as a result of the conversion.

```
 Cobol ListDsn=YOUNG.SAMPLE.COBOLST(COB01A)
   Scope ExtProgram-Id=COB01A
     TargetVar  Name=(STATE In LOC-ID In TASTRUCT)
     TargetStmt Stmts=(46,62,64)

 Cobol ListDsn=YOUNG.SAMPLE.COBOLST(COB01C)
   Scope ExtProgram-Id=COB01C
     TargetVar  Name=(TCPARM1)
```

Figure 3. Targeted Summary control file for COB01

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

- In the COB01A compile unit, any lines that reference or modify the variable STATE In LOC-ID In TASTRUCT and statements 46, 62, and 64.
- In the COB01C compile unit, 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 Summary 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

Summary report

Figure 4 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: 07/06/1999
                TIME: 12:23:35
        TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                    -->|
  |                           |                                          |  STATEMENTS:          | BRANCHES:           |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                             |   TOTAL     EXEC   %  |  CPATH   TAKEN   %  |
  ---------------------------------------------------------------------------------------------------------------------
     1 COB01    PROG           YOUNG.SAMPLE.COBOLST(COB01A)                   6        6 100.0       0       0 100.0
     2          PROGA                                                         5        4  80.0       6       5  83.3
     3          PROCA                                                         1        0   0.0       0       0 100.0
     4          LOOP1                                                         3        3 100.0       2       1  50.0
     5          LOOP2                                                         2        2 100.0       2       1  50.0
     6 COB01    PROGB          YOUNG.SAMPLE.COBOLST(COB01B)                   6        5  83.3       6       4  66.7
     7          PROCB                                                         1        1 100.0       0       0 100.0
     8          LOOP1                                                         3        3 100.0       2       1  50.0
     9 COB01    PROGC          YOUNG.SAMPLE.COBOLST(COB01C)                   5        5 100.0       6       6 100.0
    10          PROCC                                                         3        2  66.7       2       1  50.0
    11          LOOP1                                                         4        3  75.0       4       2  50.0
    12          LOOP2                                                         2        2 100.0       2       1  50.0
    13 COB01    PROGD          YOUNG.SAMPLE.COBOLST(COB01D)                   4        0   0.0       4       0   0.0
    14          PROCD                                                         1        0   0.0       0       0 100.0
    15          LOOP1                                                         1        0   0.0       0       0 100.0
  ---------------------------------------------------------------------------------------------------------------------
  Summary for all PAs:                                                      47       36  76.6      36      22  61.1

1 *********  CA SUMMARY:                    UNEXECUTED CODE                      ********
0               DATE: 07/06/1999
                TIME: 12:23:35
        TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                    -->|
  |                           |                                          |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                             | start    end     start    end     start   end
  ---------------------------------------------------------------------------------------------------------------------
     2 COB01    PROGA          YOUNG.SAMPLE.COBOLST(COB01A)                  59     59
     3          PROCA                                                        69     69
     6 COB01    PROGB          YOUNG.SAMPLE.COBOLST(COB01B)                  41     41
    10 COB01    PROCC          YOUNG.SAMPLE.COBOLST(COB01C)                  47     47
    11          LOOP1                                                        56     56
    13 COB01    PROGD          YOUNG.SAMPLE.COBOLST(COB01D)                  33     38
    14          PROCD                                                        42     42
    15          LOOP1                                                        46     46
  ---------------------------------------------------------------------------------------------------------------------

1 *********  CA SUMMARY:                    BRANCHES THAT HAVE NOT GONE BOTH WAYS ********
0               DATE: 07/06/1999
                TIME: 12:23:35
        TEST CASE ID:
0 |<--                     PROGRAM IDENTIFICATION                    -->|
  |                           |                                          |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                             |  stmt      stmt      stmt      stmt      stmt
  ---------------------------------------------------------------------------------------------------------------------
     2 COB01    PROGA          YOUNG.SAMPLE.COBOLST(COB01A)                  57
     4          LOOP1                                                        72
     5          LOOP2                                                        77
     6 COB01    PROGB          YOUNG.SAMPLE.COBOLST(COB01B)                  35        39
     8          LOOP1                                                        49
    10 COB01    PROCC          YOUNG.SAMPLE.COBOLST(COB01C)                  45
    11          LOOP1                                                        52        54
    12          LOOP2                                                        60
    13 COB01    PROGD          YOUNG.SAMPLE.COBOLST(COB01D)                  33        35
  ---------------------------------------------------------------------------------------------------------------------
```

Figure 4. Summary report for COB01

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 report. 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 5 shows a sample Annotated Listing report.

```
000001   * COB01A - COBOL EXAMPLE FOR CA
000002
000003   IDENTIFICATION DIVISION.
000004   PROGRAM-ID. COB01A.
000005   ****************************************************************
000006   *                                                              *
000007   * LICENSED MATERIALS - PROPERTY OF IBM                         *
000008   *                                                              *
000009   * 5655-B97                                                     *
000010   *                                                              *
000011   * (C) COPYRIGHT IBM CORP. 1997, 1999 ALL RIGHTS RESERVED       *
000012   *                                                              *
000013   * US GOVERNMENT USERS RESTRICTED RIGHTS - USE, DUPLICATION OR   *
000014   * DISCLOSURE RESTRICTED BY GSA ADP SCHEDULE CONTRACT WITH IBM   *
000015   * CORP.                                                        *
000016   *                                                              *
000017   *                                                              *
000018   ****************************************************************
000019
000020                ENVIRONMENT DIVISION.
000021
000022                DATA DIVISION.
000023
000024                WORKING-STORAGE SECTION.
000025                01  TAPARM1      PIC 99 VALUE 5.
000026                01  TAPARM2      PIC 99 VALUE 2.
000027                01  COB01B       PIC X(6) VALUE 'COB01B'.
000028                01  P1PARM1      PIC 99 VALUE 0.
000029
000030                01  TASTRUCT.
000031                  05 LOC-ID.
000032                    10 STATE    PIC X(2).
000033                    10 CITY     PIC X(3).
000034                  05 OP-SYS     PIC X(3).
000035
000036                PROCEDURE DIVISION.
000037
000038               * THE FOLLOWING ALWAYS PERFORMED
000039
000040                PROG.
000041               * ACCESS BY TOP LEVEL QUALIFIER
000042 :                   MOVE 'ILCHIMVS' TO TASTRUCT
000043
000044               * ACCESS BY MID LEVEL QUALIFIERS
000045 :                   MOVE 'ILSPR' TO LOC-ID
000046 :                   MOVE 'AIX' TO OP-SYS
000047
000048               * ACCESS BY LOW LEVEL QUALIFIERS
000049 :                   MOVE 'KY' TO STATE
000050 :                   MOVE 'LEX' TO CITY
000051 :                   MOVE 'VM ' TO OP-SYS
```

Figure 5 (Part 1 of 2). Annotated COBOL listing

```
000052                    .
000053
000054                PROGA.
000055 &                  PERFORM  LOOP1 UNTIL TAPARM1 = 0
000056
000057 >                  IF TAPARM2 = 0   THEN
000058               *   PROCA NOT EXECUTED
000059 ¬    1                  PERFORM PROCA.
000060
000061
000062 &                  PERFORM LOOP2 UNTIL TAPARM2 = 0
000063                    .
000064 :                  STOP RUN
000065                    .
000066
000067                PROCA.
000068               *   PROCA NOT EXECUTED
000069 ¬                MOVE 10 TO P1PARM1
000070                    .
000071                LOOP1.
000072 V                    IF TAPARM1 > 0 THEN
000073 :    1                 SUBTRACT 1 FROM TAPARM1.
000074 :                    CALL 'COB01B'
000075                    .
000076                LOOP2.
000077 V                    IF TAPARM2 > 0 THEN
000078 :    1                 SUBTRACT 1 FROM TAPARM2.
000079
```

Figure 5 (Part 2 of 2). Annotated COBOL listing

Targeted Summary report

Figure 6 is an example of a CA Targeted Summary report. As mentioned earlier, it gives you a report identical in format to the CA Summary report, but in this case, it is only reporting on source lines that were targeted 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: 'YOUNG.SAMPLE.TARGCTL(COB01)'
0                TARGETED SUMMARY DATE: 07/06/1999
                 TARGETED SUMMARY TIME: 12:25:56
0 |<--                    TARGET PSEUDO CONTROL STATEMENTS                          -->

-------------------------------------------------------------------------------------------------------------------------------
  Variable  ListDsn=YOUNG.SAMPLE.COBOLST(COB01A),ExtProgram-Id=COB01A,Name=(STATE In LOC-ID In TASTRUCT)
  Statement ListDsn=YOUNG.SAMPLE.COBOLST(COB01A),Stmts=(46,62,64)
  Variable  ListDsn=YOUNG.SAMPLE.COBOLST(COB01C),ExtProgram-Id=COB01C,Name=TCPARM1

1 *********  CA TARGETED SUMMARY:            PROGRAM AREA DATA                    ********
0                DATE: 07/06/1999
                 TIME: 12:23:35
         TEST CASE ID:
0 |<--                    PROGRAM IDENTIFICATION                    -->|
  |                           |                                        |  STATEMENTS:          | BRANCHES:           |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                           |   TOTAL     EXEC   %  |  CPATH   TAKEN   %  |
  -------------------------------------------------------------------------------------------------------------------------
     1 COB01    PROG            YOUNG.SAMPLE.COBOLST(COB01A)               4        4 100.0       0       0 100.0
     2          PROGA                                                      2        2 100.0       2       2 100.0
     9 COB01    PROGC           YOUNG.SAMPLE.COBOLST(COB01C)               1        1 100.0       2       2 100.0
    11          LOOP1                                                      3        3 100.0       4       2  50.0
  -------------------------------------------------------------------------------------------------------------------------
  Summary for all PAs:                                                   10       10 100.0       8       6  75.0

1 *********  CA TARGETED SUMMARY:            UNEXECUTED CODE                      ********
0                DATE: 07/06/1999
                 TIME: 12:23:35
         TEST CASE ID:
0 |<--                    PROGRAM IDENTIFICATION                    -->|
  |                           |                                        |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                           |  stmt      stmt      stmt      stmt      stmt
  -------------------------------------------------------------------------------------------------------------------------
   N O  U N E X E C U T E D  C O D E
  -------------------------------------------------------------------------------------------------------------------------

1 *********  CA TARGETED SUMMARY:            BRANCHES THAT HAVE NOT GONE BOTH WAYS ********
0                DATE: 07/06/1999
                 TIME: 12:23:35
         TEST CASE ID:
0 |<--                    PROGRAM IDENTIFICATION                    -->|
  |                           |                                        |
  | PA LOAD MOD PROCEDURE     | LISTING NAME                           |  stmt      stmt      stmt      stmt      stmt
  -------------------------------------------------------------------------------------------------------------------------
    11 COB01    LOOP1           YOUNG.SAMPLE.COBOLST(COB01C)              52        54
  -------------------------------------------------------------------------------------------------------------------------
```

Figure 6. Targeted Summary report for COB01

Source Audit Assistant

What problem does Source Audit Assistant solve?

Source Audit Assistant (SAA) is a tool that auditors, programmers, and testers can use. Source Audit Assistant processes an IBM SuperC code comparison and provides auditor reports on unexpected differences and tailored input to Coverage Assistant Targeted Summary reports. In addition, a Source Audit Assistant preprocessor allows SuperC to compare either source code or listings.

Auditors have a tool that will assist them in reviewing source-level changes for security or other system assurance reasons.

Developers and testers can use the tool to help focus on the changed lines of code. Source Audit Assistant analyzes changes and generates targeted control files, which Coverage Assistant can use to produce Targeted Summary coverage reports.

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.

2. **What are the major components of SAA?**

   The major components of SAA are the Source Extraction tool and the Comparison Analysis tool.

   The Source Extraction tool extracts source code from compiler listings. This allows you to run your SuperC comparisons on the source as seen by the compiler after any preprocessor has run, complete with copybooks or includes in place.

   The Comparison Analysis tool analyzes a SuperC comparison report and generates a prototype CA Target Control data set for use in Targeted Summary based on variables found in mismatched lines, and generates a Change Validation report, which lists seed variables not found in any changed lines, and changed lines, which did not contain any seed variables.

3. **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 defi-

nitions 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.

4. **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 7.

5. **Tell me more about the SuperC comparison report analysis.**

   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 seed 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.

6. **From where does the list of expected variables that I need to provide to SAA come?**

   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.

7. **What languages and compilers are supported by SAA?**

   Source Audit Assistant supports the following languages for seed analysis and template control file generation:

   - COBOL
   - PL/I

   Source Audit Assistant supports the following languages in listing form for the listing extractor:

   - IBM COBOL for OS/390 & VM 2.1 (5648-A25)
   - VisualAge COBOL Millennium Language Extensions for OS/390 & VM 1.0 (5648-MLE)
   - IBM COBOL for MVS & VM 1.2 (5688-197)
   - VisualAge COBOL Millennium Language Extensions for MVS & VM 1.0 (5654-MLE)
   - VS COBOL II Version 1 Release 4.0 (5688-022, 5688-023)
   - OS/VS COBOL Version 1 Release 2.4 (5740-CB1, 5740-LM1)
   - VisualAge PL/I Compiler Version 2 Release 2 (5655-B22)
   - IBM PL/I for MVS & VM 1.1.1 (5688-235)
   - VisualAge PL/I Millennium Language Extensions for MVS & VM 1.0 (5648-MLX)
   - OS PL/I Optimizing Compiler 2.3.0 (5668-909, 5668-910, 5668-911)
   - OS PL/I Optimizing Compiler 1.5.1 (5734-PL1, 5734-PL3, 5734-LM4, 5734-LM5)
   - High Level Assembler (HLASM) Version 1 Releases 1, 2, and 3 (5696-234)
   - High Assembler H (HASM) Version 2 (5668-962)

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 7 shows you what that ISPF panel looks like.

```
------------------------ ATC Primary Option Menu V2R1M0 -----------------------
Option ===>

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

Enter X to Terminate
```

Figure 7. ATC Primary Option Menu

The Source Audit Assistant panel, shown in Figure 8, is then displayed.

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

1  Extraction    Extract Source Code from Listings
2  Analysis      Create Prototype Target Control Dataset & Validate Changes

Enter END to Terminate
```

Figure 8. Source Audit Assistant panel

From this panel, you can choose to execute the listing extractor or the Comparison Analysis tool.

Running the SAA Comparison Analysis tool

After you run a comparison of your code base, you can then run the SAA comparison analyzer, 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 Analysis option on the Source Audit Assistant panel. The Create prototype Target Control Dataset & Validate Changes panel is displayed. Simply fill in the requested fields and press Enter to run the SAA comparison analyzer.

Figure 9 on page 25 shows the SAA Comparison Analysis tool panel.

```
---------- Create Prototype Target Control Dataset & Validate Changes --------
Option ===>

1  Generate      Generate JCL from parameters
2  Edit          Edit JCL
3  Submit        Submit JCL
4  Foreground    Run the SAA Comparison Analysis in the foreground

Enter END to Terminate

Input Data Sets:
Compare Dataset Dsn. . 'YOUNG.SUPERC.LIST'
Seed List Dsn  . . . . 'YOUNG.SAMPLE.SEEDLIST(COB01)'

Programming Language:  COBOL   (COBOL or PL/I)

DBCS Support Enable:   N (Y or N)

Target Listing Dsn . . 'YOUNG.SAMPLE.COBOLST'

Use Program Name for File Name  YES (Yes|No) Program Name  COB01

Output Data Sets:
Target Control Dsn . . 'YOUNG.SAMPLE.TARGCTL(COB01)'
Chg Validation Rpt Dsn 'YOUNG.SAMPLE.COB01.CHNGREP'

JCL Library and Member:
  JCL Dsn. . . . . 'YOUNG.SAMPLE.JCL(ACOB01)'
```

Figure 9. Create Prototype Target Control Dataset & Validate Changes panel

Inputs

The SAA Comparison Analyzer has two main inputs. The first is a SuperC comparison report. 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 comparison analyzer 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) minimizes the amount of system CPU resource required to execute a test case. This is done by reducing the test case input files to the minimum number of records that cause test case coverage that is equivalent to the coverage provided by the complete files. Distillation Assistant can reduce the size of QSAM or VSAM input files for COBOL and PL/I (record I/O only) programs. By reducing the size of the input files, the amount of CPU time may be cut significantly for repeated runs of the test case suite, which could result in shortened schedules and costs over the course of a project.

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 7. 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 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 compilers does Distillation Assistant support?**

   Distillation Assistant supports the following compilers:

   - IBM COBOL for OS/390 & VM 2.1 (5648-A25)
   - VisualAge COBOL Millennium Language Extensions for OS/390 & VM 1.0 (5648-MLE)
   - IBM COBOL for MVS & VM 1.2 (5688-197)
   - VisualAge COBOL Millennium Language Extensions for MVS & VM 1.0 (5654-MLE)
   - VS COBOL II Version 1 Release 4.0 (5688-022, 5688-023)
   - OS/VS COBOL Version 1 Release 2.4 (5740-CB1, 5740-LM1)
   - IBM PL/I for MVS & VM 1.1.1 (5688-235)
   - VisualAge PL/I Millennium Language Extensions for MVS & VM 1.0 (5648-MLX)
   - OS PL/I Optimizing Compiler 2.3.0 (5668-909, 5668-910, 5668-911)
   - OS PL/I Optimizing Compiler 1.5.1 (5734-PL1, 5734-PL3, 5734-LM4, 5734-LM5)

   Note that although the VisualAge PL/I Compiler Version 2 Release 2 (5655-B22) is supported by Coverage Assistant and Source Audit Assistant, it is *not currently supported by Distillation Assistant*.

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. 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 for the program you are testing.
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 and run the JCL to create the DA key list, an editable file in which you can add or delete keys.
9. Generate and run the JCL to distill the master data set using the key list.

Figure 10 on page 30 illustrates this process.

[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.

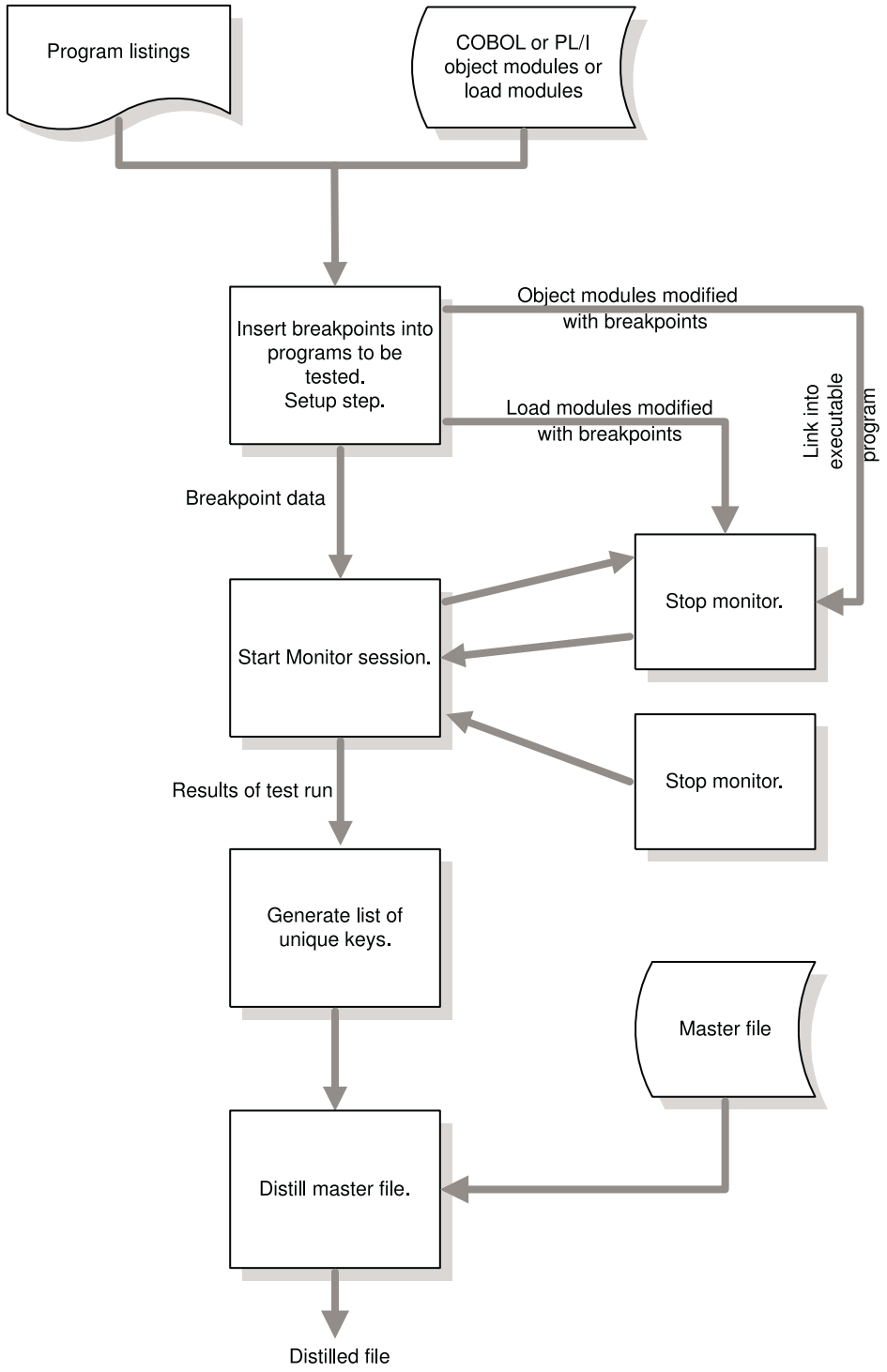

Figure 10. DA—flow diagram

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 11.

```
COBOL ListDsn=YOUNG.SAMPLE.COBOLST(COB03),
      LoadMod=COB03,
      FromObjDsn=YOUNG.SAMPLE.OBJ,
      ToObjDsn=YOUNG.SAMPLE.ZAPOBJ

Scope ExtProgram-Id=COB03

File  File=QSAMIN,KeyPosition=15,KeyLen=20
```

Figure 11. Distillation Assistant control file for COB03

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?

Unit Test Assistant (UTA) is a tool that allows you to dynamically warp data (modify and set data values) at defined points in the program logic. This capability allows data files, databases, and other data sources to be modified on the fly to test conditions without having to modify the data sources.

Some examples of the use of data warping are changing dates (for end-of-quarter testing, end-of-year testing, etc.) or currency values (Euro conversion testing, etc.). Files, databases, and other data sources containing current dates, currency values, etc. can be used as-is to do testing rather than creating static copies of converted data. You specify what data is to be warped, how data is to be warped, and where in the program logic data is located. If you prefer static copies of data files for testing, the tool can make copies of QSAM or VSAM files with modified data fields (called *file warping*).

The other major function of Unit Test Assistant is logging variables at specified points in the program logic. You tell UTA to log data variables at key points in the logic, and then use the log created during program execution to do problem diagnosis and verification.

Unit Test Assistant provides extensive support for COBOL and limited support for PL/I.

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 converted 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 values in input files or dynamically warp values during runtime. Dynamic data warping allows you to perform runtime testing of modified variables without the need to maintain test copies of input files.

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

   Data warping is the modification of the data used by an application program. One way to do this is to physically change the data in your input files. UTA's warping capabilities can help you do this. The File Warp feature modifies values in input files and the dynamic data warping feature modifies valuee during program execution. With dynamic data warping, the values 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?**

   Unit Test Assistant supports the following compilers:

   - IBM COBOL for OS/390 & VM 2.1 (5648-A25)
   - VisualAge COBOL Millennium Language Extensions for OS/390 & VM 1.0 (5648-MLE)
   - IBM COBOL for MVS & VM 1.2 (5688-197)
   - VisualAge COBOL Millennium Language Extensions for MVS & VM 1.0 (5654-MLE)
   - VS COBOL II Version 1 Release 4.0 (5688-022, 5688-023)
   - OS/VS COBOL Version 1 Release 2.4 (5740-CB1, 5740-LM1)
   - IBM PL/I for MVS & VM 1.1.1 (5688-235)
   - VisualAge PL/I Millennium Language Extensions for MVS & VM 1.0 (5648-MLX)
   - OS PL/I Optimizing Compiler 2.3.0 (5668-909, 5668-910, 5668-911)
   - OS PL/I Optimizing Compiler 1.5.1 (5734-PL1, 5734-PL3, 5734-LM4, 5734-LM5)

   Note that although the VisualAge PL/I Compiler Version 2 Release 2 (5655-B22) is supported by Coverage Assistant and Source Audit Assistant, it is *not currently supported by Unit Test Assistant*.

6. **How does Unit Test Assistant File Warp work?**

   The UTA File Warp feature changes 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.

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 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.

[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 12 illustrates this process.

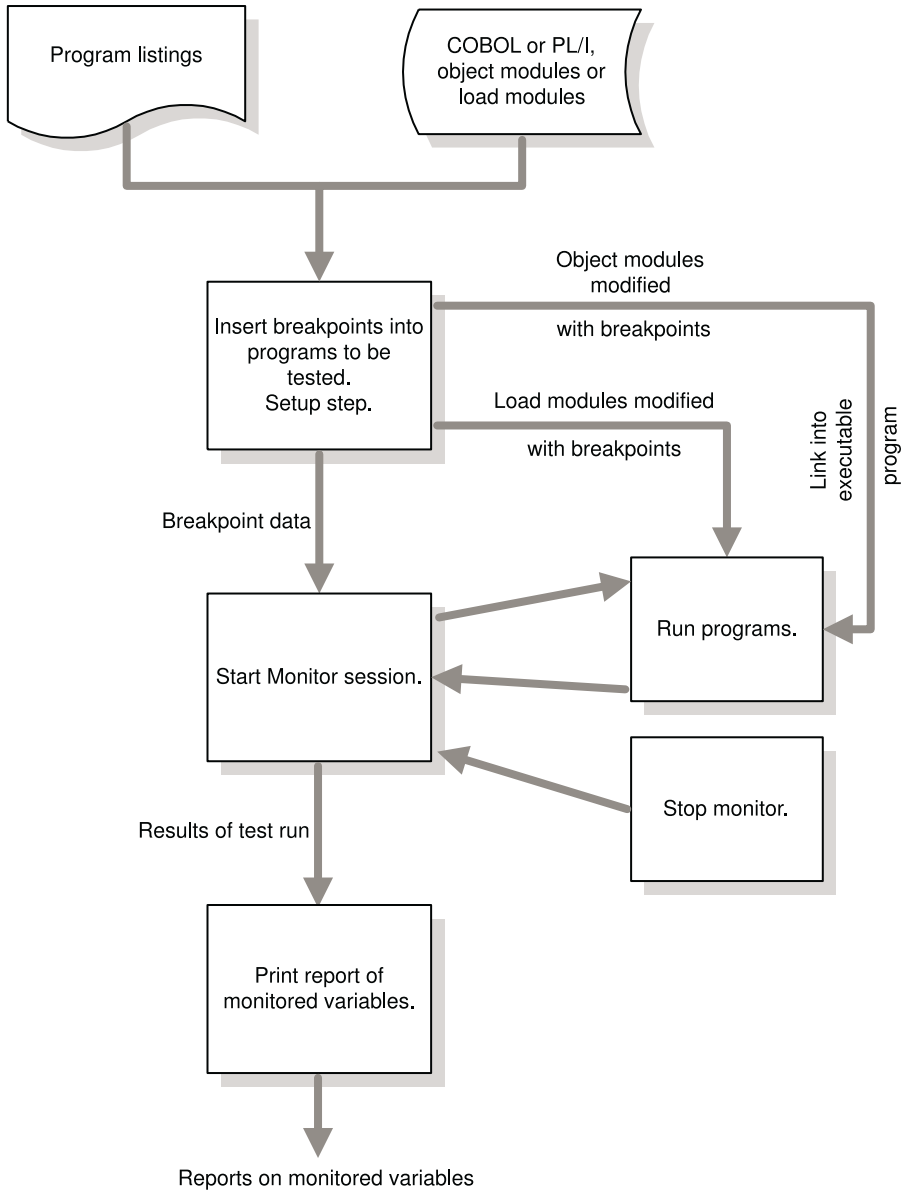

Figure 12. UTA—flow diagram

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.

Figure 13 is an example of a UTA control file.

```
     Defaults ListDsn=YOUNG.SAMPLE.COBOLST(*),
              LoadMod=COB02,
              FromObjDsn=YOUNG.SAMPLE.OBJ,
              ToObjDsn=YOUNG.SAMPLE.ZAPOBJ

     COBOL ListMember=COB02
     Scope ExtProgram-Id=COB02

     Variable Name=JULIAN-DATE
*                 set JULIAN-DATE using Data Warping
         Warp Action=Set, Value=2000001,
              Datatype=Zoned,Unsigned,Stmts=(59)

*        read JULIAN-DATE after it is set
         Coverage Stmts=(62)

*                 add 5 to  JULIAN-DATE
         Warp Action=Increment, Value=5,
              Datatype=Zoned,Unsigned,Stmts=(64)

*        read JULIAN-DATE after 5 has been addeed to it
         Coverage Stmts=(67)
     Variable Name=LOAN
*        read LOAN before it is multiplied
         Coverage Stmts=(78)
     Variable Name=INTEREST-DUE
*        read INTEREST-DUE before LOAN is multiplied
         Coverage Stmts=(78)

     Variable Name=LOAN
*        multiply LOAN by 2.5
         Warp Action=multiply, Value=2.5,
              Datatype=Zoned,Unsigned,Stmts=(81)
*        read LOAN after multiply by 2.5
         Coverage Stmts=(86)
     Variable Name=INTEREST-DUE
*        read INTEREST-DUE after LOAN is multiplied
         Coverage Stmts=(86)

     Variable Name=LOAN
*        divide LOAN by 2.5
         Warp Action=divide, Value=2.5,
              Datatype=Zoned,Unsigned,Stmts=(91)
*        read LOAN after divide by 2.5
         Coverage Stmts=(94)
     Variable Name=INTEREST-DUE
*        read INTEREST-DUE after LOAN is divided
         Coverage Stmts=(94)

*    read J-DAY in loop every 100 times, maximum of 5 times
     Variable Name=J-DAY IN J-DATE
         Coverage Length=3,ReadEvery=100,
              MaxSave=5,Stmts=(120)
```

Figure 13. Control file for COB02

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: 03/28/2000
* TIME: 05:24:46
*
*
* var-ID  CU-Name    prog-ID          var-name                        stmt-num    data
*-------------------------------------------------------------------------------------
       1  COB02      COB02            JULIAN-DATE                       62    2000001
       2  COB02      COB02            JULIAN-DATE                       67    2000006
       6  COB02      COB02            INTEREST-DUE                      78    000500
       5  COB02      COB02            LOAN                              78    010000
       9  COB02      COB02            INTEREST-DUE                      86    001250
       7  COB02      COB02            LOAN                              86    025000
      12  COB02      COB02            INTEREST-DUE                      94    000500
      10  COB02      COB02            LOAN                              94    010000
      13  COB02      COB02            J-DAY of J-DATE                  120    106
      13  COB02      COB02            J-DAY of J-DATE                  120    206
      13  COB02      COB02            J-DAY of J-DATE                  120    306
      13  COB02      COB02            J-DAY of J-DATE                  120    040
      13  COB02      COB02            J-DAY of J-DATE                  120    140
```

Figure 14. Report for COB02

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) automates the creation of batch and on-line test data, reports the program I/O deviations from a baseline and post baseline run, and encapsulates the test data environment. ARTT allows dynamic conversion of the captured/created test data, supporting data aging and data bridging when the data and programs are incompatible. ARTT also allows this to be done either in the original data environment or in a virtual data environment without the actual database engines being present.

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 base run 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 base run 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. **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.

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

   ARTT can be used to test applications running in the following MVS environments. You can even capture the initial base run 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 environment. Automated Regression Testing Tool supports the following data environments:

   - QSAM and VSAM access methods invoked from batch programs in Capture, Real Replay, and Virtual Replay modes with Data Transformation and Date Rolling for all system levels listed as prerequisites
   - Static SQL Select, Fetch, Insert, Update, and Delete statements invoked from DSN programs in Capture and Virtual Replay modes with Data Transformation and Date Rolling for DB2 for MVS/ESA V4, DB2 for OS/390 V5, and DB2 UDB for OS/390 V6.
   - Dynamic SQL Select, Fetch, Insert, Update, and Delete statements invoked from DSN programs in Capture and Virtual Replay modes without Data Transformation or Date Rolling for DB2 for MVS/ESA V4, DB2 for OS/390 V5, and DB2 UDB for OS/390 V6.
   - DLI, BMP, and MPP programs in Capture and Virtual Replay modes with Data Transformation and Date Rolling for IMS IMS/ESA V5R1M0 and V6R1M0.
   - BMS, FC, TC, TS, TD, IC, DB2, and IMS CICS EXEC statements invoked from online transactions in Capture and Virtual Replay modes, as limited in previous list items, for CICS/ESA V4R1M0.

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:

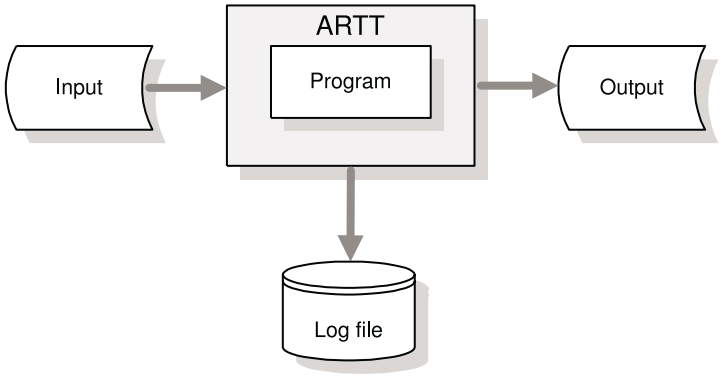

Figure 17. Batch program running in ARTT Capture mode

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

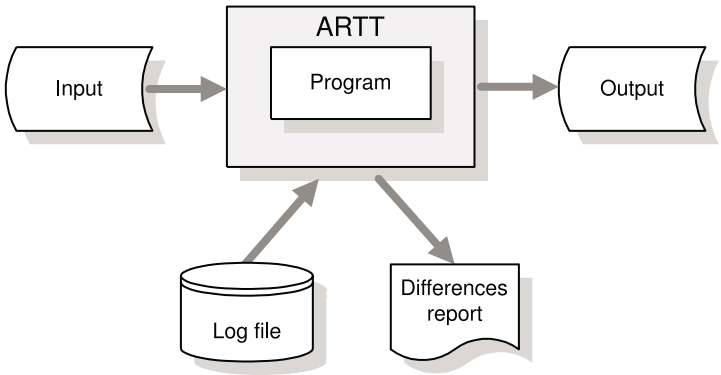

Figure 18. Batch program running in ARTT Real Replay mode

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

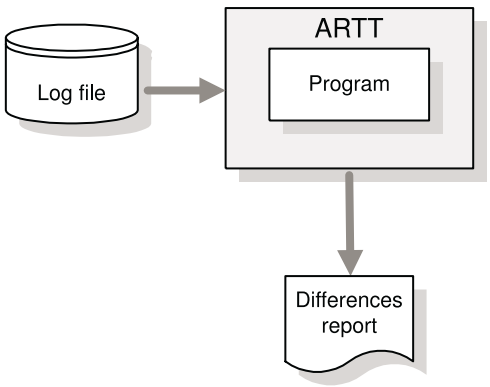

Figure 19. Batch program running in ARTT Virtual Replay mode

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 transformation, you must provide the logical file record formats for the data in question.

Outputs

Capture report

ARTT automatically generates a Capture report.

```
ARTT V2.R1.M2 -- Execution Report                                          Date 08/21/2000    Time 06:45:42    Page 0001
5655-B97 Copyright (C) 2000-2000 IBM, All Rights Reserved.
5655-B97 Copyright (C) 1996-2000 National Westminster Bank, Plc., All Rights Reserved.

ATGA0001I Program Control File = ATG.SMP212.PROG.VSAM
ATGA0002I File Control File. . = ATG.SMP212.FILE.VSAM
ATGA0003I ARMS Control File. . = ATG.SMP212.ARMS.VSAM
ATGA0004I User Library . . . . = ATG.SMP212.SATGSLOD
ATGA0005I Conversion Library . = ATG.SMP212.SATGSLOD
ATGA0006I Compare Library. . . = ATG.SMP212.SATGSLOD
ATGA0008I Test Control File. . = ATG.SMP212.TEST.VSAM
ATGA0704I Log File . . . . . = ATG.SMP212.ATGBCP2N.BASE.LOG (dynamic allocation)
ATGA0705I Program ATGBCP2N executed with:
ATGA0706I    Run Mode = CAPTURE
ATGA0707I    Status = N
ATGA0708I    Run Id = BASE
ATGA0030I Task ATGBCP2N attached with:
ATGA0031I    Parameters = "" (0 bytes).
ATGA0032I Initialisation complete. Passing control to application.
ATGA0040I Program ATGBCP2N 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                  SYSADM.USRT037A.JOB00135.D0000109.?
ATGA0715I SYSPRINT N      1      1        0       27        0        0                  SYSADM.USRT037A.JOB00135.D0000108.?
ATGA0715I ATGBAPHN Y      1      1        3        0        0        0 ATGBFPC3
ATGA0715I ATGFKSDI N      1      1        6        0        0        0                  ATG.SMP212.ATGFKSIN.VSAM
ATGA0715I ATGFKSDO N      2      2        1        4        0        0 ATGBFPC1         ATG.SMP212.ATGFKSON.VSAM
ATGA0715I ATGFESDI N      1      1        4        0        0        0                  ATG.SMP212.ATGFESIN.VSAM
ATGA0715I ATGFESDO N      1      1        0        3        0        0                  ATG.SMP212.ATGFESON.VSAM
ATGA0715I ATGFRRDI N      1      1        4        0        0        0 ATGBFPC4         ATG.SMP212.ATGFRRIN.VSAM
ATGA0715I ATGFRRDO N      1      1        0        3        0        0 ATGBFPC4         ATG.SMP212.ATGFRRON.VSAM
ATGA0715I ATGFQSMI N      1      1        4        0        0        0 ATGBFPC2         ATG.SMP212.ATGFQSIN.QSAM
ATGA0715I ATGFQSMO N      1      1        0        4        0        0 ATGBFPC2         ATG.SMP212.ATGFQSON.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

```
ARTT V2.R1.M2 -- Execution Report                                        Date 08/21/2000    Time 06:45:48    Page 0001
5655-B97 Copyright (C) 2000-2000 IBM, All Rights Reserved.
5655-B97 Copyright (C) 1996-2000 National Westminster Bank, Plc., All Rights Reserved.

ATGA0001I Program Control File = ATG.SMP212.PROG.VSAM
ATGA0002I File Control File. . = ATG.SMP212.FILE.VSAM
ATGA0003I ARMS Control File. . = ATG.SMP212.ARMS.VSAM
ATGA0004I User Library . . . . = ATG.SMP212.SATGSLOD
ATGA0005I Conversion Library . = ATG.SMP212.SATGSLOD
ATGA0006I Compare Library. . . = ATG.SMP212.SATGSLOD
ATGA0008I Test Control File. . = ATG.SMP212.TEST.VSAM
ATGA0704I Log File . . . . . = ATG.SMP212.ATGBCP2T.BASE.LOG (dynamic allocation)
ATGA0705I Program ATGBCP2T executed with:
ATGA0706I    Run Mode = REAL
ATGA0707I    Status = Y
ATGA0708I    Run Id = BASE
ATGA0030I Task ATGBCP2T attached with:
ATGA0031I    Parameters = "" (0 bytes).
ATGA0032I Initialisation complete. Passing control to application.
ARTT V2.R1.M2 -- Differences Report                                      Date 08/21/2000    Time 06:45:50    Page 0002
5655-B97 Copyright (C) 2000-2000 IBM, All Rights Reserved.
5655-B97 Copyright (C) 1996-2000 National Westminster Bank, Plc., All Rights Reserved.

-------------------------------------------------------------------------------------------------------------------------
Current: Service=QSAM     Resource=SYSPRINT                  Logged: Service=QSAM     Resource=SYSPRINT
-------------------------------------------------------------------------------------------------------------------------
Cur: ReqNo=00000008 FileReqNo=00000004 Length=00133 RetCode=00000000                  Function=PUT
Log: ReqNo=00000008 FileReqNo=00000004 Length=00133 RetCode=00000000                  Function=PUT
                                                                                     |                                |
Cur: 0000   40C1E3C7 C2C3D7F2 E3406040 C1A4A396   9481A385 8440D985 879985A2 A2899695 | ATGBCP2T - Automated Regression|
Log: 0000   40C1E3C7 C2C3D7F2 D5406040 C1A4A396   9481A385 8440D985 879985A2 A2899695 | ATGBCP2N - 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   40F2F161 F0F861F2 F0F0F040 404040E3 |            Date 21/08/2000    T|
Log: 0040   40404040 40404040 40404040 4040C481   A38540F2 F161F0F8 61F0F040 404040E3 |              Date 21/08/00    T|
                                  ****** *   ********  ***** * **                    |             *************      |
Cur: 0060   89948540 F0F67AF4 F57AF5F0 40404040   D7818785 40404040 F1404040 40404040 |ime 06:45:50    Page    1       |
Log: 0060   89948540 F0F67AF4 F57AF4F4 40404040   D7818785 40404040 F1404040 40404040 |ime 06:45:44    Page    1       |
                               * *                                                   |          **                    |
Cur: 0080   40404040 40                                                              |                                |
Log: 0080   40404040 40                                                              |                                |
                                                                                     |                                |

-------------------------------------------------------------------------------------------------------------------------
Current: Service=QSAM     Resource=SYSPRINT                  Logged: Service=QSAM     Resource=SYSPRINT
-------------------------------------------------------------------------------------------------------------------------
Cur: ReqNo=00000020 FileReqNo=00000008 Length=00133 RetCode=00000000                  Function=PUT
Log: ReqNo=00000020 FileReqNo=00000008 Length=00133 RetCode=00000000                  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 F2F161F0 F861F2F0 F0F04040 |ader record date is 21/08/2000  |
Log: 0020   81848599 40998583 96998440 8481A385   4089A240 F2F161F0 F861F0F0 40404040 |ader record date is 21/08/00    |
                                                                  *   * *              |                          * **  |
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: Service=VSAM     Resource=ATGFKSDI                  Logged: Service=VSAM     Resource=ATGFKSDI
-------------------------------------------------------------------------------------------------------------------------------
Cur: ReqNo=00000022 FileReqNo=00000005 Length=00082 RetCode=00000000 FeedBck=00000000  Function=GET
Log: ReqNo=00000022 FileReqNo=00000005 Length=00082 RetCode=00000000 FeedBck=00000000  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       |
                                                                                      |                                |
Cur: 0040   40404040 40404040 40404040 40404040   4040                                |                                |
Log: 0040   40404040 40404040 40404040 40404040   4040                                |                                |
                                                                                      |                                |
                                                                                      |                                |
. . .

ATGA0040I Program ATGBCP2T 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                  SYSADM.USRT037A.JOB00136.D0000106.?
ATGA0715I SYSPRINT Y      1      1        0       27       11       11                  SYSADM.USRT037A.JOB00136.D0000105.?
ATGA0715I ATGBAPHN Y      1      1        3        0        1        1 ATGBFPC3
ATGA0715I ATGFKSDI Y      1      1        6        0        2        2 ATGBFPC1          ATG.SMP212.ATGFKSIT.VSAM
ATGA0715I ATGFKSDO Y      2      2        1        4        2        2 ATGBFPC1          ATG.SMP212.ATGFKSOT.VSAM
ATGA0715I ATGFESDI Y      1      1        4        0        0        0                  ATG.SMP212.ATGFESIT.VSAM
ATGA0715I ATGFESDO Y      1      1        0        3        0        0                  ATG.SMP212.ATGFESOT.VSAM
ATGA0715I ATGFRRDI Y      1      1        4        0        1        1 ATGBFPC4          ATG.SMP212.ATGFRRIT.VSAM
ATGA0715I ATGFRRDO Y      1      1        0        3        1        1 ATGBFPC4          ATG.SMP212.ATGFRROT.VSAM
ATGA0715I ATGFQSMI Y      1      1        4        0        2        2 ATGBFPC2          ATG.SMP212.ATGFQSIT.QSAM
ATGA0715I ATGFQSMO Y      1      1        0        4        2        2 ATGBFPC2          ATG.SMP212.ATGFQSOT.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                  SYSADM.USRT037A.JOB00135.D0000109.?
ATGA0715I SYSPRINT N      1      1        0       27        0        0                  SYSADM.USRT037A.JOB00135.D0000108.?
ATGA0715I ATGBAPHN Y      1      1        3        0        0        0 ATGBFPC3
ATGA0715I ATGFKSDI N      1      1        6        0        0        0                  ATG.SMP212.ATGFKSIN.VSAM
ATGA0715I ATGFKSDO N      2      2        1        4        0        0 ATGBFPC1          ATG.SMP212.ATGFKSON.VSAM
ATGA0715I ATGFESDI N      1      1        4        0        0        0                  ATG.SMP212.ATGFESIN.VSAM
ATGA0715I ATGFESDO N      1      1        0        3        0        0                  ATG.SMP212.ATGFESON.VSAM
ATGA0715I ATGFRRDI N      1      1        4        0        0        0 ATGBFPC4          ATG.SMP212.ATGFRRIN.VSAM
ATGA0715I ATGFRRDO N      1      1        0        3        0        0 ATGBFPC4          ATG.SMP212.ATGFRRON.VSAM
ATGA0715I ATGFQSMI N      1      1        4        0        0        0 ATGBFPC2          ATG.SMP212.ATGFQSIN.QSAM
ATGA0715I ATGFQSMO N      1      1        0        4        0        0 ATGBFPC2          ATG.SMP212.ATGFQSON.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 complement existing testing processes
- Architected to work in an integrated manner.

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

Source Audit Assistant integration

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

1. Indicating records in which changes were made, but were not intended
2. Indicating variables that should have been found in changed lines, but were not
3. Generating a control file that Coverage Assistant can use to produce its Targeted Summary report

SAA *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 or expected (that is, a list of data elements of interest). After producing a SuperC comparison file of the differences between a baseline and its related changed compile unit, SAA can be used to analyze the comparison file. It generates a Change Validation report listing those records which did not contain any variable from the given list. 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 the changes do not have any clear relationship to the data elements of interest.

Another 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 all of the variables that were found within changed lines, SAA makes it easy to use Coverage Assistant to 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. 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:

Figure 22. Sample naming convention

	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.

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

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 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 modify 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.

Appendix A. DBCS support

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

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

DBCS characters are permitted in:

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

For more details, see the DBCS information in the *Application Testing Collection for MVS/ESA & OS/390 Version 2 Release 1 User's Guide*, SC26-9871.

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.

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:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J74/G4
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

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

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

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

Glossary

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

A

Annotated Listing report. 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*.

base run. 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 a SuperC comparison as checked 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.

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 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.

data transformation. 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).

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

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

monitor session. A distinct invocation of the monitor program.

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. 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 run. 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 run 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 job that produces the summary and Annotated Listing report reports after a test case run.

S

SAA comparison analysis. Generates a change validation report and a prototype Coverage Assistant target control file using a compare file generated by the SuperC Compare Utility and a list of seed variables that you want to monitor.

seed list. A list of seed variables.

seed variable. A variable name used as input to the Source Audit Assistant comparison analyzer. This is generally the name of a variable that was identified as one that will be involved with the planned code changes.

session. A distinct invocation of the monitor program.

Setup. The Coverage Assistant job 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.

SuperC Compare Utility. A compare program that processes two sequential data sets, two complete partitioned data sets (PDS), or members from two partitioned data sets or concatenated data sets. SuperC can compare data sets of unlimited size and record lengths at the file, line, word, or byte level. SuperC creates output listings in which you can locate data differences, including the following: delta listings, long listings, summary listings, and side-by-side line listings.

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*.

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.

VSAM. Virtual storage access method.

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

A

Annotated Listing report 17
Application Testing Collection
  Automated Regression Testing Tool 41
  Coverage Assistant 7
  Distillation Assistant 27
  integrating into testing 49
  overview 1
  Source Audit Assistant 21
  Unit Test Assistant 33
assemblers 11, 23
Automated Regression Testing Tool
  Capture mode 41
  description 41
  inputs 45
  integrating into testing 54
  log file 41, 42
  outputs 45
  problem addressed by 41
  questions and answers 41
  Replay mode 41

B

batch support
  ARTT 41
  CA 11
  UTA 34
black box testing 2
branches, executed 8, 15

C

Capture mode 41
Change Validation report 22, 24
CICS support
  ARTT 41
  CA 11
  UTA 34
code coverage 7
code, comparing 21
comparison
  file 21
  output 41
  report 25
compilers
  CA 11
  DA 28
  SAA 23
  UTA 34
control file
  CA 9, 12
control file *(continued)*
  example 11
  SAA 10, 13
  Targeted Summary 24
conversion, data 6, 42
Coverage Assistant
  control file 9, 12, 13
  description 1, 7
  environments supported 11
  inputs 13
  integrating into testing 50
  outputs 15
  problem addressed by 7
  questions and answers 7
  reports 47
  software supported 11
coverage, code 7

D

database support
  CICS
    ARTT 41
    CA 11
    UTA 34
  DB2 41
  for date testing 42
  IMS
    ARTT 41
    CA 11
    UTA 34
DB2 support 41
DBCS support 55
debugging 33
Distillation Assistant
  data distillation 6
  data sets distilled 28
  description 1, 27
  inputs 31
  integrating into testing 52
  outputs 31
  problem addressed by 27
  questions and answers 27
  regression testing 2
distillation, data 27
dynamic data warping 33—34

E

environments supported
  ARTT 42
  CA 11

environments supported *(continued)*
complex 5, 41
production 1
test 52
UTA 34, 54
executed statements 7
external testing 2

F

file comparison 21
file warping
ATC 5
UTA 33—35, 38

I

IMS support
ARTT 41
CA 11
UTA 34
inputs
ARTT 45
CA 13
DA 31
SAA 25
UTA 36
interactive support
ARTT 42
CA 11
SAA 24
UTA 34
internal testing 3

K

key list, DA 31
keys, DA 28

L

listing files, comparing 21
logging data 33, 53

M

methods, testing 2

O

online support
ARTT 42
CA 11
SAA 24
UTA 34
outputs
ARTT 45
outputs *(continued)*
CA 15
comparing 41
DA 31
SAA 25
UTA 38
overview
integrating ATC 49
using Automated Regression Testing Tool 43
using Coverage Assistant 12
using Distillation Assistant 29
using Source Audit Assistant 24
using Unit Test Assistant 35

P

postprocessor, SAA 24

R

regression testing 2, 41
Replay mode, ARTT 41
reports
Annotated Listing report, CA 17
ARTT 47
CA 15
change validation 24
coverage summary, CA 15
SAA 25
summary, ARTT 47
targeted summary, CA 19
rolling, dates 42

S

Source Audit Assistant
description 1, 21
inputs 25
integrating into testing 50
languages supported 23
outputs 25
postprocessor 24
problem addressed by 21
questions and answers 21
statements executed 7
Summary report 15, 47
support
ARTT 41
batch 11, 34
CA 11
CICS 11, 34
compiler and assembler 23, 34
CA 11
DA 28
SAA 23
UTA 34

support *(continued)*
DB2 41
DBCS 55
IMS 11, 34, 41
language 23
online 11, 42
UTA 34

T

targeted code coverage 9
Targeted Summary control file 24
Targeted Summary report 19
testing
black box 2
integrating ATC tools into 49
methods 2
process overview 49
regression 2
time, reducing 27
white box 3
tools, testing 2
transforming data 42

U

Unit Test Assistant
description 1, 33
dynamic data warping 33
file warping 33—35, 38
inputs 36
integrating into testing 53
outputs 38
problem addressed by 33
questions and answers 33

W

white box testing
tools 4
with ARTT 5, 41
with CA 7
with SAA 21

We'd Like to Hear from You

Application Testing Collection
for MVS/ESA & OS/390
General Information
Version 2 Release 1

Publication No. GC26-9870-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. NUMBER: (859) 243-4345.
- Electronic mail—Use the following network ID:
  - Internet: atchelp@us.ibm.com

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

Application Testing Collection
for MVS/ESA & OS/390
General Information
Version 2 Release 1

Publication No. GC26-9870-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

Phone No.

Readers' Comments
GC26-9870-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Application Testing Collection Development
Department TNZA / Building 962-2
745 West New Circle Road
Lexington, KY 40511-1846

Fold and Tape **Please do not staple** Fold and Tape

GC26-9870-02

Cut or Fold
Along Line

Program Number: 5655-B97

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC26-9870-02

Spine information:

ATC General Information *Version 2 Release 1*