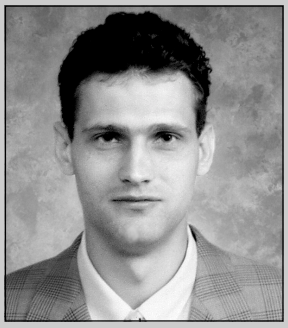
Improve Testing by Tracing ABAP Program Execution: Take a Closer Look with the Coverage Analyzer

Dr. Christian Hansen



Dr. Christian Hansen joined SAP AG in 1997 and became a member of the technology development internationalization group, where he works on the design, tools, and rollout of Unicode support for ABAP. The ABAP Coverage Analyzer is one of the tools developed and used to improve and guarantee the quality of ABAP coding during the transition to Unicode.

(complete bio appears on page 84)

Even the most extensive testing is often a hit-or-miss affair, and some elusive error always seems to be lurking in the shadows. As of Release 6.10, the days of testing the runtime behavior of your ABAP programs in the dark are over. With the ABAP Coverage Analyzer, you can at last persistently trace the system-wide execution of ABAP programs. Best of all, this new and indispensable tool comes with the standard SAP system. Whether you're a quality manager or a developer, with the Coverage Analyzer you can now:

- Globally review the testing activities on your system
- Monitor the testing activities of different user groups
- Get detailed information about the usage of different program units (procedures, modules, etc.) in your program objects
- Identify and make well-directed performance improvements, where necessary
- Detect possibly "dead" code
- Support Unicode-enablement[1] of your ABAP programs

This article shows you how to put the Coverage Analyzer to work for you. First I'll discuss what the Coverage Analyzer is and how it works. Then I'll walk you through some practical examples for using it, including how to make a system-wide test more effective and how to assess the usage data of a program or its subroutines.

[1] For more on Unicode, an international character-encoding standard, see the article "Looking Forward to the Unicode Advantage: Internationalization and Integration" in the January/February 2002 issue of *SAP Professional Journal*.

How the Coverage Analyzer Works

Quite simply, the Coverage Analyzer (transaction SCOV) is a tool for recording ABAP program execution. The Runtime Analysis tool (transaction SE30) provides detailed information about time consumption, table accesses, and more for a single transaction, but only for the duration of that transaction. In contrast, the Coverage Analyzer automatically traces the system-wide execution of any *program unit*, at any time, and stores the data persistently in the system database. You don't even need to set any explicit watchpoints.

Figure 1 shows what constitutes program units in the context of the Coverage Analyzer, along with the ABAP keywords that trigger them.

Figure 1 Traced Program Units

Program Unit	ABAP Keyword
Procedures	FORM FUNCTION METHOD
Modules	MODULE
Event blocks	AT LINE SELECTION AT PF<xx> AT SELECTION SCREEN AT USER COMMAND END-OF-PAGE END OF SELECTION GET GET LATE INITIALIZATION START-OF-SELECTION TOP-OF-PAGE
Program starts	PROGRAM REPORT

> ✓ ***Note!***
>
> *In the context of the Coverage Analyzer, you can think of any "program unit" as a "processing block," with the exception of program starts (e.g., SUBMIT REPORT), which technically are not processing blocks. However, since the Coverage Analyzer traces both program starts and processing blocks (or, more precisely, the entry into the definition part of each processing block), for simplicity, the term "processing block" will include program starts throughout this article.*

For each processing block, the Coverage Analyzer collects the following information:

- Number of executions
- Number of runtime errors inside the processing block

Once the Coverage Analyzer is started, data collection batch jobs run periodically on every application server in your system. You control the frequency of these batch jobs in the Coverage Analyzer settings.

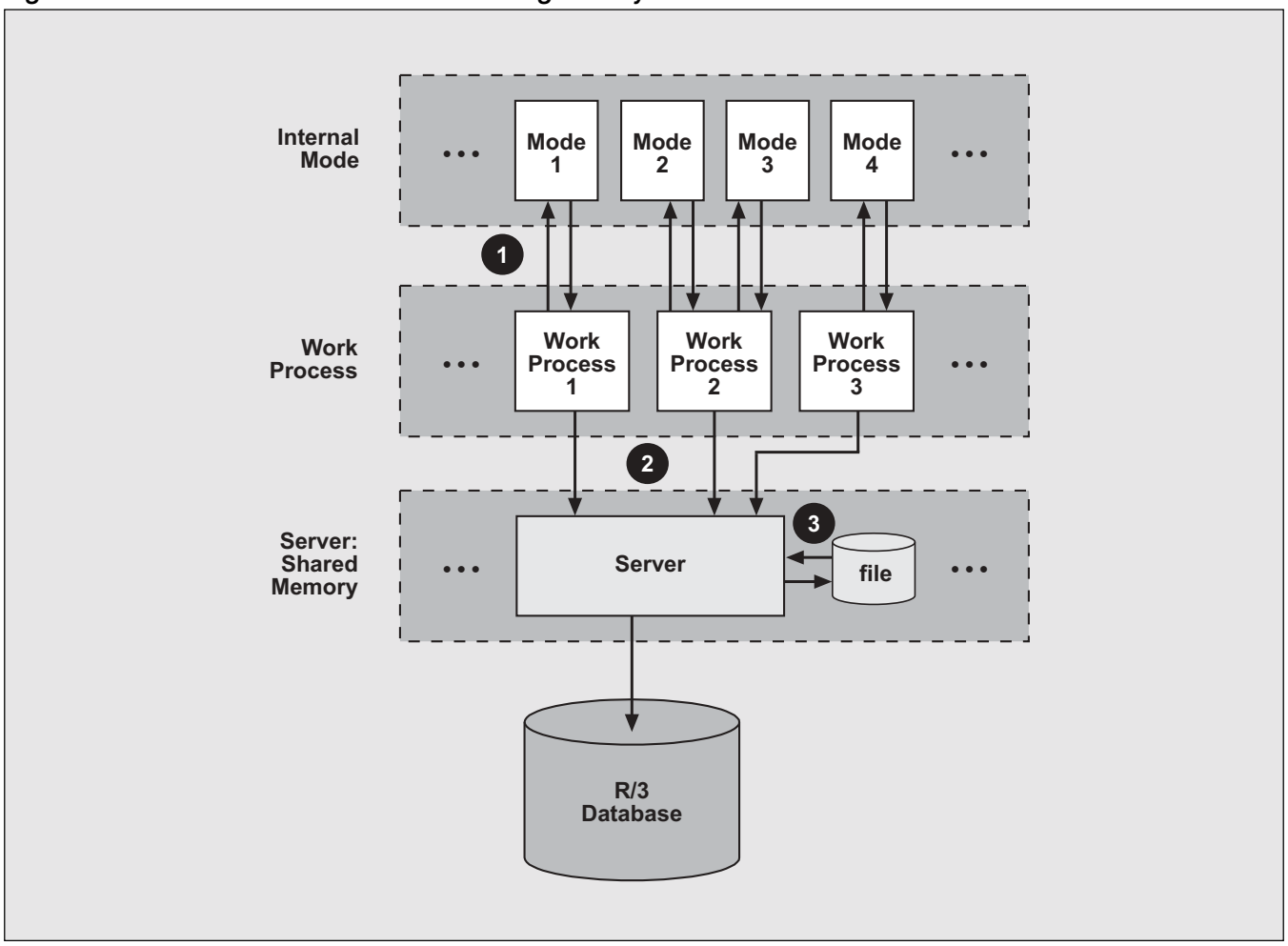
As shown in **Figure 2**, the Coverage Analyzer passes information from the execution point in the processing block to the R/3 database:

1. The Coverage Analyzer collects call frequencies for each processing block in the memory of an active internal mode running in a particular work process.
2. If the internal mode is rolled out, the coverage information is flushed to the shared memory of

> ✓ ***Note!***
>
> *The Coverage Analyzer does not count calls of transactions, dialog modules, or other superordinate entry points to program execution. It also does not resolve sub-units inside processing blocks, such as [if...then...else] branches or case statements.*

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 2 How the Coverage Analyzer Collects Information



the application server and then inserted into the R/3 database (triggered by periodic batch jobs).

3. If there is insufficient shared memory, the coverage results are written to a temporary file. The contents of this file are automatically reread and inserted into the R/3 database in subsequent runs of the data collection batch jobs.

Impact on Overall System Performance

You might be concerned that continuously tracing program executions will degrade system performance. Don't worry, the actual performance impact is minimal. The effect is equivalent to one additional statement — such as `DATA progname TYPE SYREPID. Progname = SY-REPID` — per processing block. The periodic batch job also needs a little time to insert coverage data into the database. In one of our production application development systems at SAP, we experienced a processing speed of approximately 10 programs/second, or 100 processing blocks/second. Therefore, it took only 1.5 minutes to process coverage data for 1,000 programs with 10,000 processing blocks (or 15 minutes for 10,000 programs with 100,000 processing blocks).

Shared memory consumption is typically 0.2-1.8 MB on a non-Unicode system, and 0.3-3.4 MB on a

Unicode system.[2] However, the need for more memory activates the file mechanism. In the R/3 database, the volume of data depends on the number of programs in your system and the coverage achieved. As an example, on a non-Unicode Basis system with 50,000 programs and 20% coverage, the database contained 30 MB of coverage data.

Beneficial Features for Real-World Testing

Even in a test system, software is typically not static. Individual programs are subject to changes due to development and transports, or — in a pure test system — error corrections and patches. The Coverage Analyzer resets the current coverage results for programs that are changed either explicitly (via a code modification) or implicitly (via a change to a referenced Data Dictionary definition). Data collection starts again at zero.

> **💣 *Caution!***
>
> *Be aware that even small changes to central Data Dictionary definitions could have a major impact on coverage results for the overall system. For example, consider the impact of adding a field to the central system table SYST, which is used in virtually every program.*

If you had counters that show only current coverage, trying to get good coverage on test systems would be difficult at best. To address this gap, the Coverage Analyzer provides additional counters that track ongoing processing block executions since the start of the test, including:

- The cumulative number of executions
- The cumulative number of runtime errors inside the processing block
- The number of resets

[2] A Unicode system is an SAP installation that uses Unicode on the application server and in the database.

While the Coverage Analyzer generally traces complete system activities for any processing block, sometimes you are only interested in the activities of subgroups of users. For this purpose, you can create *test groups*[3] to which you assign a particular set of users. You can then distinguish and compare the results of different test groups. For example, you can identify ways to use the CATT (Computer Aided Test Tool) more effectively by analyzing the CATT coverage alone, without the added overhead of other system activities.

So far, I have focused on how the Coverage Analyzer traces program execution on a single system. This approach can sometimes be inadequate if you need to test similar activities on several different systems. However, simply adding up the coverage in terms of percentage for different systems does not work because you can't deduce any overlap of tested program units. Instead, turn on "Data Aggregation from Different Remote Systems"[4] to condense the coverage results for all systems, which gives you a single result with the aggregate coverage.

To make it easy to interpret all of this valuable information, the Coverage Analyzer provides two views for analyzing results — the "Detail View" and the "Global View." **Figure 3** describes their contents. I'll discuss these views in more detail when we walk through some examples later in this article.

> **✓ *To Learn More…***
>
> *The online help (**http://help.sap.com**) is an excellent source of information on the Coverage Analyzer. Refer to it for detailed instructions on setting up and starting the tool, interpreting results, reading the monitor, and other administrative tools. The same information is available at transaction SCOV (**Help → Application Help**).*

[3] Choose "Test Groups" in the Coverage Analyzer's main menu.

[4] To set this parameter, select **Administration → On/Off, Status**. Refer to the online help at **http://help.sap.com** for detailed instructions.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 3 Analyzing Coverage Analyzer Results

View	What It Shows	Configurable Amount of Data?
Detail View	Results for a program object or single processing block.	No — Automatically includes everything, from local objects to SAP objects and customer objects.
Global View	Results for the whole system, sorted by package or object owner. Also provides additional derived information, such as degree of utilization.	Yes — Restricts calculation and display of coverage results to the packages or transport layers of interest. For example, if you are not interested in SAP programs, select only those packages in the customer name range.

Setting Up the Coverage Analyzer

Because the Coverage Analyzer comes with the standard R/3 system, setup is a simple task for your system administrator:

1. Provide the name of the application server where the trigger for the data collection batch jobs will run.
2. Set the execution period for the trigger.
3. Turn on the Coverage Analyzer.
4. Turn on "Automatic recording"* to activate the evaluations for the Global View.

During initialization, all programs are generated once to build a complete list of all processing blocks in the system. Thereafter, the Coverage Analyzer collects data from every program execution on the system. You don't need to turn it on specifically for your program or your package, although you may decide to configure the Global View to display only the data of interest to you.

* To set this parameter, select **Administration** → **On/Off, Status**. Refer to the online help for detailed instructions.

Using the Coverage Analyzer

Now that you understand how the Coverage Analyzer works, in the following sections I'll walk you through a few scenarios that illustrate how you might take advantage of this tool in your own environment:

☑ **Scenario 1: "What Constitutes a Good Test?"**

This scenario describes how to assess and improve the effectiveness of a system-wide test.

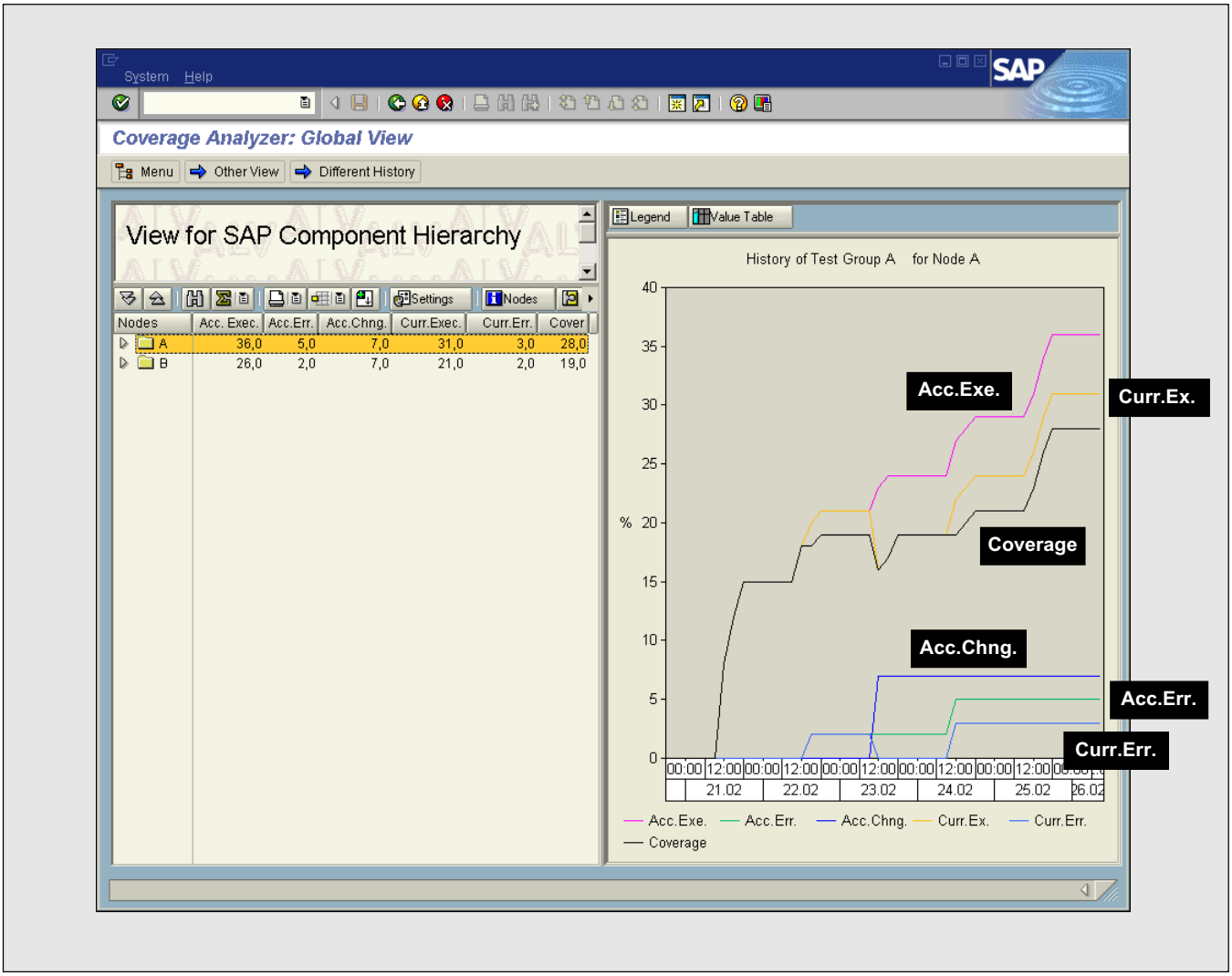
☑ **Scenario 2: "How Is This Program Being Used?"**

This scenario discusses how to extract detailed information about program usage.

☑ **Scenario 3: "Is the ABAP Unicode-Enabling Project Done?"**

This scenario examines how to monitor the progress of a Unicode-enabling project.

Figure 4 Global View: Results After One Week of Testing (in Percent)



The following sections explore these scenarios in detail.

Scenario 1: "What Constitutes a Good Test?"

Imagine you are a quality manager responsible for directing the work of 30 testers. Their task is to spend a week going through the results of your latest ABAP development effort with a fine-toothed comb. You provide an elaborate test plan. After one week of uninterrupted testing, they uncover only two errors. You could draw one of several conclusions from this result:

1. The testers didn't do their work. They were just hanging around drinking coffee at your company's expense.
2. The test plan is not very useful.
3. You have very well-written software.

Previously, you had few alternatives to help you make a conclusive decision. Using the Coverage Analyzer, you can now easily prove or disprove the first two possibilities. Based on the same one-week

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 5 *Columns in the Global View*

Label	Full Name	Contents (in Percent)
Acc.Exec.	Accumulated Executions	Processing blocks that have been executed since the Coverage Analyzer was started.
Acc.Err.	Accumulated Errors	Processing blocks in which a runtime error occurred since the Coverage Analyzer was started.
Acc.Chng.	Accumulated Changes	Processing blocks that were reset.
Curr.Exec.	Current Executions	Processing blocks that were executed in the current version, which is the last present and unchanged version of a processing block (i.e., processing blocks that were executed since the last reset).
Curr.Err.	Current Errors	Processing blocks in which a runtime error occurred in the current version.
Cover	Coverage	Processing blocks that were tested. To set the criteria, choose **Coverage Analyzer** → **Administration** → **Settings**. In the example shown in Figure 4, I used the default setting of "Has been running in the current version without runtime errors."

test scenario, you divide the group in half and validate the test results by comparing quantitative data for each group. Suppose you create two separate test groups ("A" and "B") and assign 15 testers to each group. After a week of testing, you examine the Global View in the Coverage Analyzer. **Figure 4** shows the results that you might find.

On the left side of the screen, you see the total results for each test group. **Figure 5** describes what appears in the columns. Double-clicking on a test group will display on the right side of the screen a color graph showing the test progress over time.[5] We'll look at some examples of these graphs in the next section.

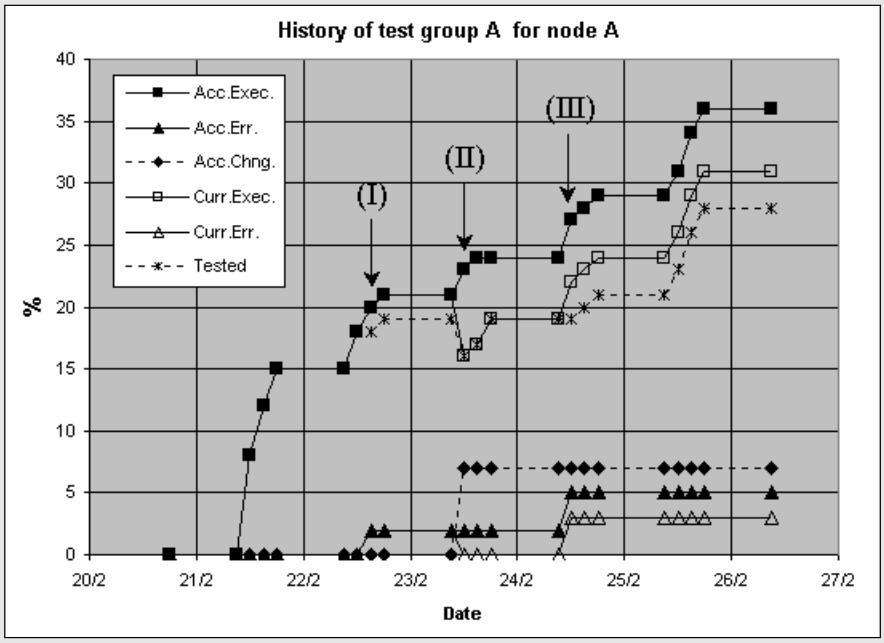
Reviewing the results shown in Figure 4, you see that Group A touched 36% of all processing blocks under investigation. In one week, they found runtime errors in 5% of those processing blocks. In the test system, 7% of the processing blocks have been changed. Of the processing blocks under investigation, 31% ran in the current version. They found runtime errors in 3% of the current versions of processing blocks. Therefore, 28% of the processing blocks under investigation are considered "tested."

In contrast, Group B touched only 26% of all processing blocks under investigation. In one week, they found runtime errors in 2% of those processing blocks. Group B, like Group A, found that 7% of the processing blocks were changed in the test system. Of the processing blocks under investigation, 21% ran in their current versions. They found runtime errors in 2% of the current versions of processing blocks. Therefore, only 19% of the processing blocks under investigation are considered "tested" — approximately one-third less than the number tested by Group A.

You can clearly see that Group A delivered better test results than did Group B, which perhaps did not work through the complete test plan.

[5] In order to improve the readability of the colored graphs in a grayscale print medium, a slightly different presentation is given in Figures 6 and 7, as you will see in a moment.

Figure 6 *Results of Test Group A*



Analyzing Progress over Time in More Detail

Let's look at an alternate representation of the Global View results shown in Figure 4. **Figure 6** and **Figure 7** show the graphs generated by the Coverage Analyzer when the results are exported to a table calculation program. Roman numerals indicate key points in time for ease of navigation and comparison.

In Figure 6, you see testing progress over time for Group A. Group A started testing on February 21.[6] On February 22 (I), they found errors in 2% of the processing blocks. They reported the errors, and received fixes on February 23 (II). These fixes reset coverage results for 7% of the processing blocks. They then retested the 2% of processing blocks that contained errors. On February 24 (III), they found errors in another 3% of the processing blocks, which were not resolved until the end of the testing phase.

You see testing progress over time for Group B in Figure 7. First, you see that they were slower than Group A. They did not find the errors first detected by Group A, because Group A found and received fixes for them before Group B reached that step in the test plan. Group B just sees that changes were made to 7% of the processing blocks on February 23 (II). At the end of the testing phase on February 25 (IV), they detected only 2% of the 3% of errors reported by Group A on February 24.

Identifying Untested Code

As you can see, the Coverage Analyzer provides valuable insight to help you resolve questions about how testing is progressing. Besides detecting runtime errors, perhaps the most important and exciting feature is the ability to identify completely untested functionality in the system. Even Group A left 64% of the code untouched and 72% untested, if you consider programs to be "tested" only if they ran error-free in the current version.

The Detail View of the Coverage Analyzer provides a list of seldom or untested program objects, which you should consider incorporating into revised test plans. In **Figure 8**, you see that the

[6] Dates appear in international format in Figures 6 and 7 (dd/mm).

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 7 Results of Test Group B

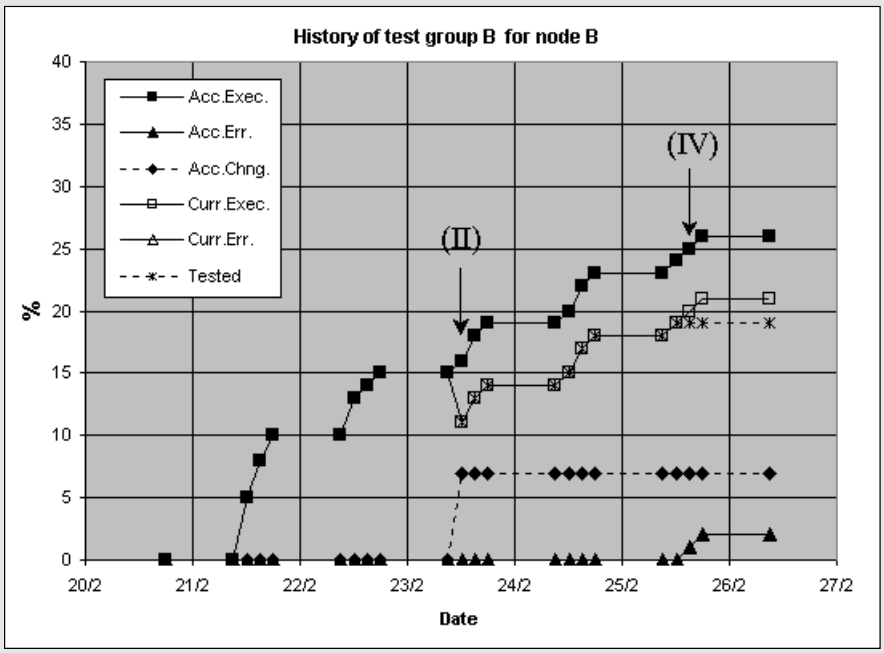
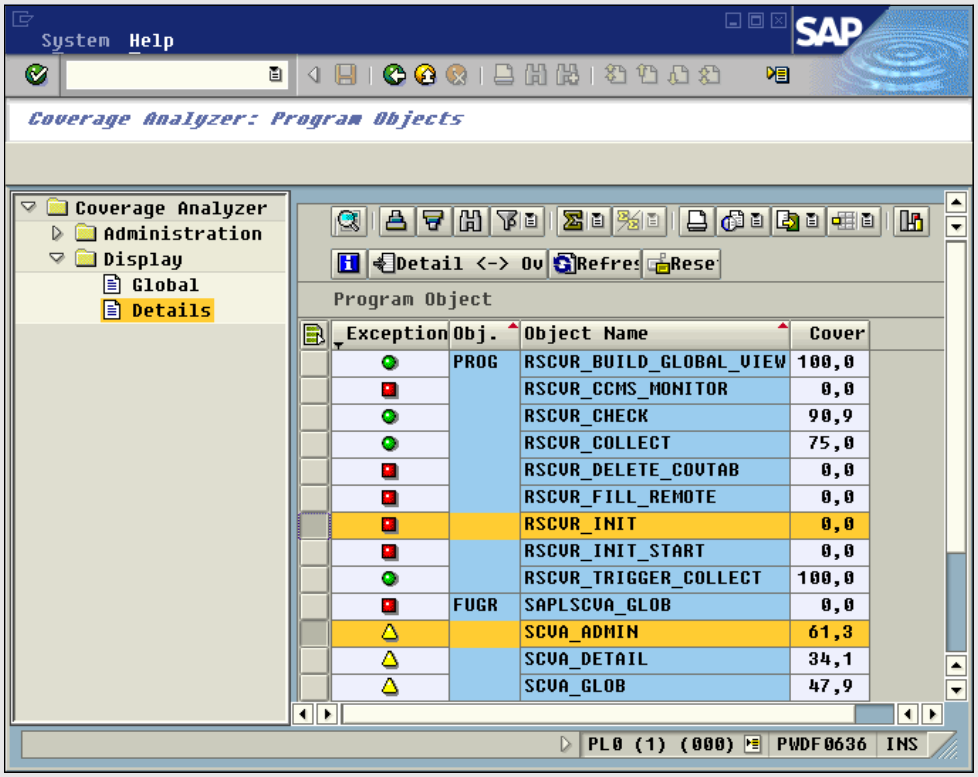


Figure 8 List of Tested and Untested Program Objects in the Detail View



program RSCVR_INIT has not been tested at all, whereas parts of the function group SCVA_ADMIN have been tested.

Leveraging the Coverage Analyzer

Don't underestimate the motivation that testers derive from the Coverage Analyzer results! At last, they can see the progress of their work — powerful encouragement to seek new paths through the code to increase coverage, and ultimately improve your test plans. While beneficial for "black-box" tests, where the tester does not examine the programming code, the Coverage Analyzer delivers even more value for "white-box" tests, where the tester has access to the programming code to analyze the results. Use testers who have enough knowledge to identify the connection between untested program objects and the corresponding actions that need to be executed to reach those objects. In their hands, the Coverage Analyzer becomes a key diagnostic tool for improving the quality of your ABAP programs.

Perhaps the best benefit of the Coverage Analyzer is that it improves your use of the CATT to perform automated repeatable tests, measure their coverage, and improve them. To do this, you create a test group (such as "CATT") and assign to it a single user (such as "CATTUSER"). Log on to the system as this user, start the CATT, and log off again. This method provides the results of the CATT without much interference from the activities of the assigned user.

At this point, you might say, "Nice feature. I will use it. But what test coverage should I aim for?" Unfortunately, the answer is, "It depends." A target of 100% is simply not realistic. During a normal test, you will not enter every single subroutine. For example, testing a subroutine that is designed to perform recovery actions after a serious system crash might be difficult. Other subroutines for error handling may not be reached during a test because the error condition does not arise. On the other hand, this untested code needs to work if this situation *does* arise, therefore you should try to find a way to test it, though this can be hard to do. Either way, error-handling routines make up only a small part of all coding in a system, and I would be doubtful if someone claimed that since the remaining untested code is all error-handling routines, 50% test coverage is adequate. Even if you exclude error-handling routines from testing, you should not be satisfied with low coverage.

However, reaching 100% coverage does not necessarily guarantee error-free code, because the Coverage Analyzer traces only processing blocks and does not resolve [if...then...else] branches or single statements (see the second "Note" on page 72). Even if the Coverage Analyzer did so, you may still encounter data-dependent errors. And syntactically correct programs could be pure nonsense in terms of the implemented application logic.

At a minimum, with the Coverage Analyzer you can be 100% certain to identify everything that is untouched and untested. Consider the value of that certainty, which you could never achieve previously.

Scenario 2: "How Is the Program Being Used?"

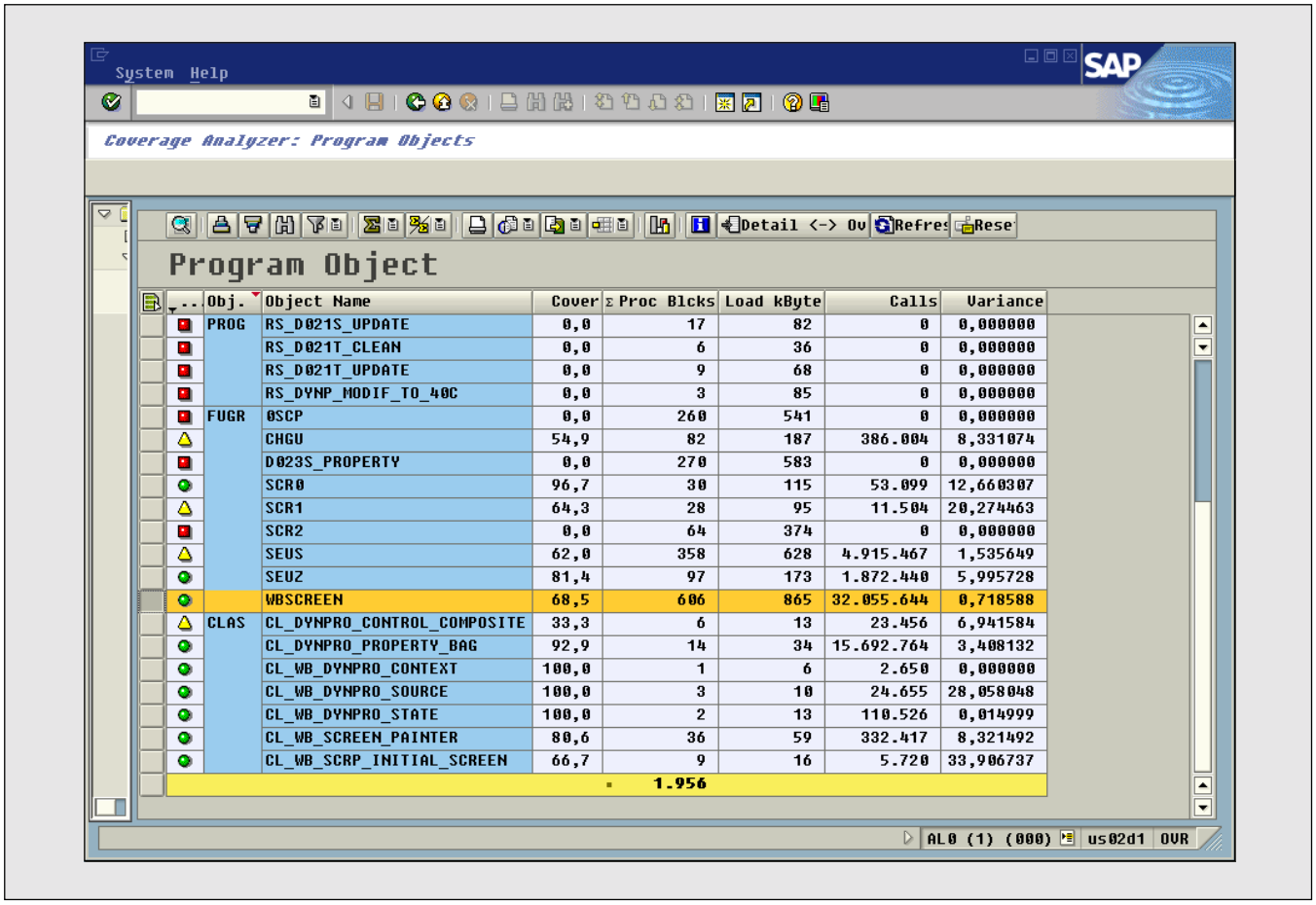
Now let's examine another way of taking advantage of the Coverage Analyzer. Suppose you are the developer responsible for a graphical layout editor, and you are also responsible for an older alphanumeric layout editor. Somebody proposes discontinuing support for the older editor, suspecting that nobody is using it. In the past, you had several options for validating this assumption, such as:

1. Survey all users of the tool.
2. During lunch, ask four colleagues about their preferences and extrapolate the result to 6,000 developers.
3. Set a breakpoint in the old code, and wait to see if anybody complains.

The Coverage Analyzer provides a more effective alternative. You can now simply examine the call frequencies of the two editors and use those figures to

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 9 **Results for Program Objects**



decide whether to withdraw support. Generally speaking, you can base such decisions on facts, rather than assumptions.

If the entrypoints to the two editors occur in different processing blocks, you can easily use the Coverage Analyzer to answer the question. Turn on the Coverage Analyzer on a system where the two editors are used, collect data for a period of time that is long enough to include sufficient hits in the usage of the two editors, and examine the results.

To better illustrate this option, let's walk through a hypothetical example. Suppose we ask the same question about the SAP layout editor (don't worry, this question is *not* being asked!). We locate an internal system at SAP that has been running for 10 months to investigate how often the editors are being used. The entrypoints to the editors are located in the function group `WBSCREEN`, in the subroutines `ENTER_ALPHA_SCREEN` and `ENTER_GRAF_SCREEN`. First we enter the name of the package that contains the function group `WBSCREEN` in the Detail Entry screen of the Coverage Analyzer.[7] As shown in **Figure 9**, we then see an overview of all program objects in that package. The `WBSCREEN` function group has good coverage with 68.5% and a high call frequency with 32,055,644 calls. Therefore, we can expect to obtain useful statistics about the usage of the two editors.

We then double-click the highlighted line to see the processing blocks of the function group

[7] Select **Coverage Analyzer → Display → Details**. Refer to the online help (**http://help.sap.com**) for detailed instructions.

Figure 10 Results for Processing Blocks

Processing Blocks

...	Type	Name	Acc. Exec.	Acc.Err.	Acc.Chng.	Curr.Exec.	Curr.Err.
	FORM	DETERMINE_INDEX_IN_T_F	0	0	50	0	0
		DETERMINE_LENG_AND_VLENG	266.965.643	0	50	12.926.989	0
		DETERMINE_MAX_LENGTH	10.514	0	50	181	0
		DETERMINE_RECTANGLE	398	0	50	20	0
		DEZ_HEADER	0	0	50	0	0
		DEZUSATZ_LIST	0	0	50	0	0
		DICINFO_RESET	3.391	0	50	7	0
		DICT_PROC_UEBERN	297	0	50	13	0
		DROPDOWN_IN	564	0	50	21	0
		DROPVAL_IN	416	0	50	21	0
		DYNAM_VAL_HELP	5.056.170	0	50	76.807	0
		DYNPRO_DOWNLOAD	96	0	50	13	0
		DYNPRO_EXPORT	45.160	4	50	610	0
		DYNPRO_REFERENCE	235	0	50	4	0
		DYNPRO_UPLOAD	128	0	50	0	0
		END_OF_BUTTON	322	0	50	2	0
		ENLARGE_DYNPRO	75	0	50	0	0
		ENTER_ALPHA_SCREEN	13.011	0	50	232	0
		ENTER_GRAF_SCREEN	64.252	0	50	887	0
		EU_SCREEN_EXIT	1.712.461	0	50	58.806	0
		EU_UPDATE	45.151	0	50	610	0
		EVT_CURSOR	45.787	0	50	864	0
		EXCLUDE_MOD_FUNCTIONS	2.223.022	0	50	103.860	0

WBSCREEN (as shown in **Figure 10**). The results show 13,011 calls to the alphanumeric layout editor and 64,252 calls to the graphical layout editor, which gives us the answer to the question about usage of the alphanumeric layout editor. The alphanumeric layout editor is used less frequently than the graphical layout editor. However, 13,011 calls (or 17% of all calls to the layout editor) is still quite high, so we should continue to support the alphanumeric layout editor, even though it is used less.

The screen in Figure 10 contains other interesting information as well:

- If you are looking for "dead" code, these results provide a first clue of where to look. For example, the subroutines DEZ_HEADER and DEZUSATZ_LIST did not run at all during this 10-month period, and might be candidates. However, be aware that the Coverage Analyzer actually detects "not dead" code, because clearly the code isn't dead if you see at least one call. Plus, code that has not yet been called could still be called the next day, proving that it is not dead.

- If you are looking for possible targets for performance improvement, look for programs that are called frequently. In this example, you might want to look closer at the subroutine DETERMINE_LENG_AND_VLENG, which had the considerable number of 266,965,643 hits in 10 months.

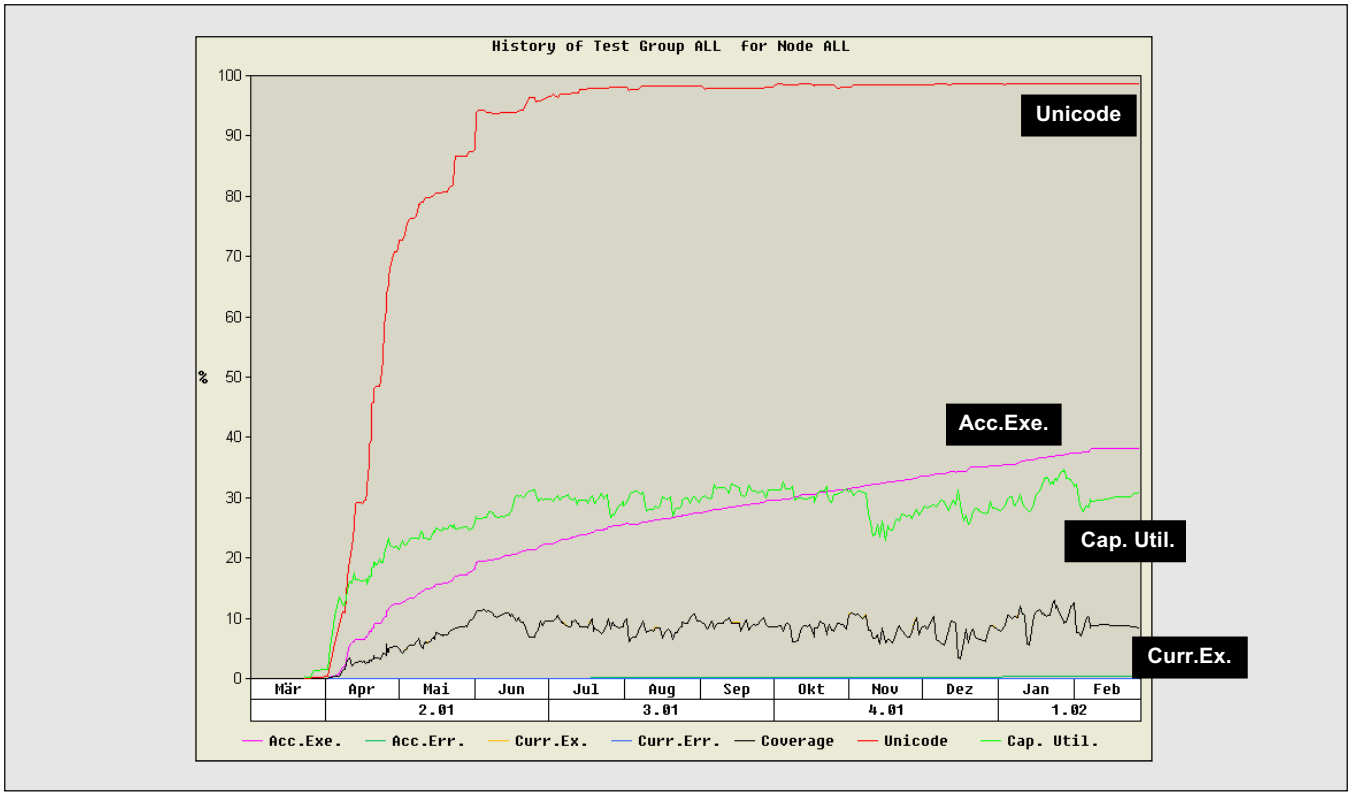
Scenario 3: "Is the ABAP Unicode-Enabling Project Done?"

Another good opportunity for using the Coverage Analyzer is tracking the progress of an ABAP Unicode-enabling project.[8] **Figure 11** shows the Global View results for a typical ABAP Unicode-enabling phase in a non-Unicode development system. In addition to the discussed values "Acc.Exe." and "Curr.Ex.," you can see the percentage of processing blocks contained in program objects that comply with Unicode syntax rules (the curve labeled "Unicode"). Although this value is a static (rather than runtime-related) property of a processing block, it is also available in the standard Coverage Analyzer.

[8] See the article "Looking Forward to the Unicode Advantage: Internationalization and Integration" in the January/February 2002 issue.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 11 *Progress of ABAP Unicode-Enabling*



Progress in the Unicode-enabling process is very fast at first, but slows in the second phase of the project. This change of pace reflects the fact that most code is already Unicode-compliant, which means you simply need to set the attribute that the program object complies with the Unicode syntax rules. Some remaining code is not Unicode-compliant from the start, and requires program modifications, which are more time-consuming.

During the Unicode-enabling process, you need to achieve both technical and mindset shifts from "most program objects are running with the old syntax" to "all relevant programs are running with the new syntax." To promote this switch, you can enforce usage of the Unicode syntax rules for all programs via the profile parameter `abap/unicode_check` in order to jumpstart developers who have not yet done their conversion work. Again, you can use the Coverage Analyzer to find used, but not yet Unicode-enabled, programs. The Detail View either proves that all used programs are already Unicode-enabled, or at least provides a prioritized list of program objects to be converted first.

Of course, checking program status is only the first step. You need to perform final runtime tests in a real Unicode system, and achieve good test coverage.

The graph in Figure 11 also shows the capacity utilization curve (labeled "Cap. Util."), which is another key metric available in the Coverage Analyzer. It shows the ratio between the number of processing blocks actually used and the number of processing blocks that were loaded into ABAP program memory. This value helps you identify badly proportioned subroutine pools, classes, or function groups in terms of ABAP program memory consumption. For example, you should never put a heavily used line-break routine into the same function group as a subroutine doing end-of-year inventory, because the end-of-year inventory subroutine will sit in program memory doing nothing most of the time.

Helpful Hints

Here are a few things to keep in mind as you set out to put the Coverage Analyzer to work in your own environment:

☑ As the Coverage Analyzer generates all programs in the system during initialization, it is a good idea to perform this initialization overnight or when the system is not heavily used.

☑ Don't create too many test groups! The more test groups you have, the bigger the performance impact on your system.

☑ If you can't find your program object in the Coverage Analyzer, it could be that it is syntactically incorrect since the start of the Coverage Analyzer. The Coverage Analyzer displays only objects that have been generated, which is never the case for programs with syntax errors.

☑ If you don't see the last change of your program in the Coverage Analyzer, even if it contains no syntax errors, it could be that the program object has been activated, but not generated (e.g., if you imported objects to the system without generation). Use the consistency checks at **Coverage Analyzer** → **Administration** → **Consistency Checks** in order to resolve such problems.

Conclusion

You have seen several examples of how using the Coverage Analyzer can improve the quality of your ABAP programs by making your testing more effective:

- Use the Global View to "test the test." Determine measurable values for your tests and set goals for your quality process.
- Apply the concept of "test groups" to analyze and improve the coverage of your automated tests.
- Use the Detail View to improve your understanding of how, and how frequently, a particular program is used.
- Examine call frequencies for different program units to identify candidates for performance improvement, or "dead wood."

As a nice side-effect, you will discover that testing actually becomes more fun for your testers. With the Coverage Analyzer, they have positive and measurable goals. Their results will now be "We have tested 50% of the code" instead of "We didn't find any errors."

I hope this article has piqued your interest in the Coverage Analyzer. As creative readers, surely you can envision many more opportunities for using it beyond those presented in this article. The challenge is up to you — put the Coverage Analyzer to work in your environment today and discover possibilities that designers never even thought of!

Acknowledgements

I would like to thank Hyo-Jun Lee, Thomas Manzke, Dr. Andreas Simon Schmitt, and Daniela Stegaru for their contributions to the design and implementation of the Coverage Analyzer, and Michael Redford for his kind review of this article.

Dr. Christian Hansen studied physics at the Albert-Ludwigs University in Freiburg. He received his doctorate at the Max-Planck-Institute for polymer research in Mainz. In 1997 he joined SAP AG and became a member of the technology development internationalization group, where he works on the design, tools, and rollout of Unicode support for ABAP. The ABAP Coverage Analyzer is one of the tools developed and used to improve and guarantee the quality of ABAP coding during the transition to Unicode. He can be reached at christian.hansen@sap.com.

 ©2002 SAP Professional Journal. Reproduction prohibited. All rights reserved.