Test Advisor 2020.12 User and Administrator Guide

Test Advisor is part of the Coverity Development Testing Platform.

Table of Contents

Part 1. Test Advisor usage

Table of Contents

Chapter 1.1. Test Advisor Overview

Table of Contents

Development organizations seldom have effective quality control while the product is being developed. Because of this, development teams typically rely on the QA organization to ensure that the product release meets the organization's quality standards. This practice has several disadvantages, including:

- A delay between the introduction and detection of a bug. This delay makes the fix costly because developers move on to other work, and might inadvertently rely on the bug.
- The bug’s presence in the source code causes problems long before the product is released.
- Long QA cycles of finding and reporting bugs can lead to a delay in the release.

Many organizations are now adopting the practice of using development testing to identify quality issues early in the development process. *Development testing* is basically any quality assurance activities that take place during the development cycle. Common assurance activities that have been integrated as essential to development testing are continuous static code analysis, code reviews, and so forth.

Perhaps the most important part of the process lies in *developer tests*, which are the automated tests created and maintained by developers that are run as part of a continuous integration cycle. Often, but not always, the tests can be classified as unit or component tests and are implemented in one of the common test frameworks such as JUnit or CPPUnit.

Successful adoption of developer tests presents several significant challenges to a development organization. Some of these challenges include:

- Motivation - Encouraging developers to create effective tests and to make best use of these tests in development, for example, by running them before check-in.
- Efficiency - Making developers more efficient in their day-to-day activities and increasing the return on investment of developer tests. Developer efficiency has two distinct components:
  1. When creating tests, focus developer effort where it matters most (for example, test the most code with the highest risk potential for bugs).
  2. When writing code, enable developers to re-run only the tests that are relevant to them.
- Effectiveness – Creating tests that are effective at stopping quality defects from escaping the development organization.
- Manageability – Focusing the testing efforts based on the risk and impact of individual code changes.
- Visibility – Enabling the technical leadership of an organization to assess the overall effectiveness of developer testing efforts.

1.1.1. The Test Advisor solution

Test Advisor can help your organizations improve their development testing practices by addressing the challenges of motivation, efficiency, effectiveness, manageability, and visibility. The Test Advisor solution is motivated by the following observations made about organizations that have successfully incorporated developer tests as part of their development process:

- They have business stakeholders that believe in and sponsor developer testing efforts.
- They have people in the organization (usually senior developers or architects) that strongly believe in the developer testing practices and watch over the testing efforts.
- They have a policy on test writing. Such a policy might dictate the use of a particular testing framework, provide guidelines for how the code should be tested, specify metrics by which the tests are measured, assert that each new feature must come with tests, and so forth.
- They use a continuous integration/testing process that gates a build’s success on its ability to pass an automated test suite. Test failures are then quickly detected and developers are motivated to repair their code/tests to ensure build’s completion.

Test Advisor is an automated tool that supports the best practices noted above in an automated tool that will help development organizations improve their developer testing practices.

1.1.2. Test Advisor features and development workflow

Test Advisor consists of three key elements:

1. Using a formal *test policy* to express the desired testing goals. A policy might contain coverage thresholds for different parts of the system, the list of critical areas of the system, constraints on the test input values, distinction between the new code and the legacy code, and so forth. Test Advisor formalizes these test policies so that an automated tool can process them.
2. Using the test policy and the collected data to:
   - Create actionable work items that direct developers at improving tests (according to the test policy)
   - Facilitate greater understanding of the existing tests (code that they reach, historical pass/fail status, etc.)
   - Focus testing efforts around the impact of the individual code changes to lower the risk and maximize the return on investment of limited resources
   - Provide full visibility into (and means for assessment of) the overall development testing efforts
3. Collecting data about testing activities through:
   - *Test instrumentation* is added to the (existing) continuous testing environment to observe the dynamic behavior relevant to policy enforcement and stores the resulting observations for later analysis.

- *Test capture* is performed by wrapping the (existing) test execution harness to simplify management of the runtime observations generated during test execution and to determine test boundaries when multiple tests are executed together.
- *Static analysis* of the tests, including relevant properties, such as reachability of code, identification of code that should not be covered due to its semantics (for example, exception handling and logging), evaluation of testability heuristics, and so on.
- *Integration incorporation* of data from other relevant sources with other data sources about test and code development activities, such as Source Control Management (SCM) systems, static code analysis, and so forth.

1.1.3. Test Advisor SCM and platform support

For information on SCM and platform support for Test Advisor, see the *Coverity 2020.12 Installation and Deployment Guide* .

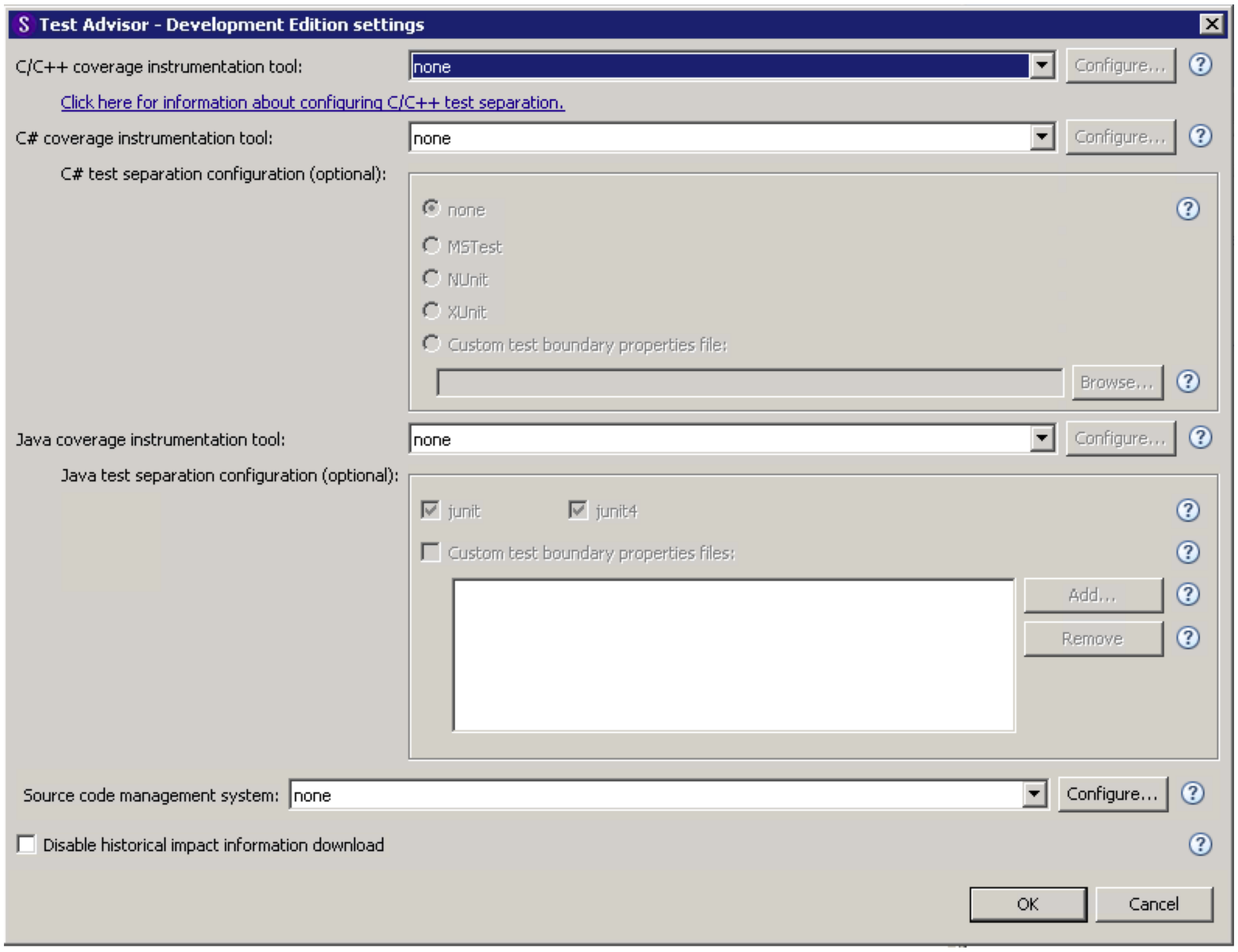
Chapter 1.2. Getting Started with Test Advisor

Table of Contents

This section provides a workflow for integrating and running Test Advisor integrated with your test environment. The main focus of this section is to suggest an order in which you should use the Coverity Analysis and Test Advisor command line tools to establish accurate results that reflect your tests' effectiveness and quality.

The workflow described in this section describes running Test Advisor tools through the command line. However, Coverity Wizard provides a complete Test Advisor workflow through a graphical user interface. For example the following screen allows you to add your Test Advisor settings for collecting SCM and coverage data:

Figure 1.2.1. Test Advisor Settings in Coverity Wizard



Test Advisor - Development Edition settings

C/C++ coverage instrumentation tool: none

Click here for information about configuring C/C++ test separation.

C# coverage instrumentation tool: none

C# test separation configuration (optional): [✓ none] MSTest NUnit XUnit Custom test boundary properties file:

Java coverage instrumentation tool: none

Java test separation configuration (optional): [✓ junit] [✓ junit4] Custom test boundary properties files:

Source code management system: none

Disable historical impact information download

For more information see Coverity Wizard 2020.12 User Guide.

It is important to note that the overall workflow is both iterative and progressive. The workflow starts with basic procedures and then adds more complex utilities as the workflow progresses. The goal is to verify that, at the completion of each procedure, Test Advisor is producing the test coverage, effectiveness, and analysis results that you expect. As you add more complexity to your system test analysis, you can adjust your test policies to produce a manageable amount of violations and coverage data.

Some of the procedures in this Getting Started section are particular to a given language (C/C++, Java. or C#). Examples for them are listed within the same steps.

1.2.1. Performing a basic commit with Test Advisor

The goal of this scenario is to provide a basic end-to-end workflow. It guides you through a successful Test Advisor build, through policy definition and application, and then committing the results to Coverity Connect, where you can view and manage Test Advisor issues.

1. Configure your compiler (if you have not already done so) using the `cov-configure` command.

   Example for C/C++:

   ```
   cov-configure --gcc
   ```

   Example for Java:

   ```
   cov-configure --java
   ```

   Example for C#:

   ```
   cov-configure --cs
   ```

2. Check out the source code repository from your SCM tool. The following example uses GIT, though you will use the checkout command utility for your source control system.

   ```
   git clone git://github.com/gitster/git.git
   ```

   The following is an example for checking out Java source:

   ```
   git clone http://github.com/voldemort/voldemort.git
   ```

3. Build the executables with coverage instrumentation to the intermediate directory. In this case, the coverage tool for C/C++ is `gcov` and it uses the make command.

   ```
   cov-build --dir idir --c-coverage=gcov make
   ```

   Note

   In addition to gcov, Test Advisor also supports the following coverage tools for C/C++:

   - BullseyeCoverage, for example:

     ```
     cov-build --dir t1 --c-coverage=bullseye \
      --bullseye-dir /opt/bullseye make build
     ```

     For full implementation information, see `cov-build`. You can also implement a Bullseye small runtime. For more information, see Section 1.3.5, “Using the Bullseye small runtime”.

     Test Advisor does not include BullseyeCoverage as part of the Test Advisor installation package. You must purchase or have an existing licensed copy of BullseyeCoverage. For more information, see http://www.bullseye.com/index.html.

   - PureCoverage (an IBM Rational PurifyPlus tool). Coverage files are generated on Windows or Linux systems by Purecov and exported to a text file format that can then be added to Test Advisor through the `cov-manage-emit` command.

   - Bullseye or CoverageScope with VxWorks. For more information, see Appendix A, *Additional Test Advisor usage notes*.

   - Function Coverage Instrumentation, for example:

```
cov-build --dir t2 --c-coverage=function make build
```

Function Coverage Instrumentation is included with Test Advisor, and provides lightweight coverage at the function call level. See Function Coverage Instrumentation for more information.

The following build command for Java uses `cobertura` for the coverage instrumentation (Test Advisor also supports JaCoCo):

```
cov-build --java-coverage=cobertura --dir idir ant build buildtest
```

The following build command for C# uses `opencover` for the coverage instrumentation:

```
cov-build --dir idir msbuild mysolution.sln /t:rebuild
```

Note

The `--cs-coverage` option is not required for the build since instrumentation is performed at runtime, not compile time. However, `.pdb` files are required to collect C# coverage. Most builds from within Visual Studio generate `.pdb` files automatically, but if the build does not, it will need to be modified to generate `.pdb` files before coverage will be collected. For an example of how to generate `.pdb` files, you visit the following MSDN blog entry: http://blogs.msdn.com/b/yash/archive/2007/10/12/pdb-files-what-are-they-and-how-to-generate-them.aspx.

For more usage details, see `cov-build --cs-coverage`.

`cov-build` automatically inserts compiler flags during the build which will cause the compiler to add coverage instrumentation code to the resulting executables. It allows you to get executables that are instrumented for coverage collection without having to modify your build system.

If you want to run your tests independently from your build, you can use `cov-build --test-capture` (see the next step) as a separate command.

4. Run the instrumented executables and store the execution counts in the emit directory. Make sure that this command runs the tests in your project.

   Example for C/C++:

```
cov-build --test-capture --dir idir --c-coverage=gcov make test
```

   Example for Java:

```
cov-build --test-capture --dir idir --java-coverage=cobertura ant junit-all
```

   Example for C#:

```
cov-build --test-capture --dir idir --cs-coverage opencover --cs-test mstest
 mstest \
 /testcontainer:mydll.dll
```

Test Advisor allows you to add instrumentation for more complicated capture scenarios, such as for tests that exist on a remote machine and longer-running tests. For more information, see Chapter 1.3, *Instrumentation options* .

Note

Note that when using Java Test Advisor, Java Test Advisor classes will be injected into the class path used to run your tests. In order for Java Test Advisor to function correctly, these classes must be loaded by the system class loader. Some mocking frameworks, such as PowerMock, are known to use their own class loader to load test classes, which in turn results in Java Test Advisor classes also being loaded by the mocking framework class loader. If you see the following messages on `stderr`, the likely cause is that the Java Test Advisor classes were not loaded by the system class loader:

```
[WARNING] Test Advisor callback classes are not properly initialized.
[WARNING] Coverage will not be recorded.
[WARNING] This can happen when using certain types of classloaders, see
 documentation for more details.
```

If you see the messages above, please consult the documentation for the mocking framework to determine how to defer loading of certain classes to the system class loader. All classes in the following packages must be loaded by the system class loader:

- com.coverity.capture.*
- com.coverity.test_separation.*
- com.coverity.util.*

To be safe, you may wish to simply defer loading of all classes in the package, `com.coverity.*`, to the system class loader. In the case of PowerMock, this can be accomplished by adding "`@PowerMockIgnore({ "com.coverity.*" })`" to your test classes.

5. Run the analysis using a sample policy file to find test policy violations.

   Example for C/C++ and C#:

```
cov-analyze --dir idir --test-advisor --strip-path <path> \
 --test-advisor-policy=<path>example_1-function_100.json
```

   Example for Java:

```
cov-analyze --dir idir  --include-java  --test-advisor --strip-path <path> \
 --test-advisor-policy=<path>example_1-function_100.json
```

   You must specify a full or relative path to your policy file location. In the examples above, `example_1-function_100.json` is a sample policy file that exists in the following location:

   `<install_dir_ca>/doc/examples/ta-policies`

This sample policy tests for 100% coverage of functions and every function must have 100% of its lines covered.

`cov-analyze` generates issues based on your policy that specify the standards of coverage. Coverage that does not meet the criteria set out in the policy generates a policy violation issue.

You need to specify the `--test-advisor` option to enable the Test Advisor components of `cov-analyze`, and `--test-advisor-policy <filename>` to specify the policy file.

Note

The `--strip-path` option is required for `cov-analyze` when used with the `--test-advisor` option.

Test Advisor will only analyze files located within a strip path directory or its subdirectory. It will ignore, for example, C++ system header files for this reason. If your build dynamically generates some source files, make sure they are located within one of the strip paths. Remember that you may specify more than one `--strip-path` argument to `cov-analyze`.

For more information, see `cov-analyze`.

6. Commit the policy violations to Coverity Connect.

   The command is the same for both C/C++ and Java. For example:

   ```
   cov-commit-defects --dir idir --host <hostname> --port <port_number> \
    --stream <stream_name> --user <user_name>  --password <password>
   ```

7. Log into Coverity Connect, and verify that the code coverage data and policy violations exist in Coverity Connect for your specified stream.

   The Coverity Connect UI provides features for viewing coverage data (lines covered, lines not covered, coverage exclusions), SCM change data, events, and test policy rules. You can also get test status, as well as coverage information, metrics, and issue counts on files and functions. For details, see Chapter 1.8, *Managing test results in Coverity Connect and Coverity Policy Manager*.

1.2.2. Adding SCM data

Source history can be an important factor in targeting tests or assessing risk. Test Advisor provides tools to incorporate historical data from Source Control Management (SCM) systems. In this procedure, you will add SCM change information for source code lines. SCM change information includes the following:

- date - The date that the changed code was checked into the SCM system.
- revision - The revision number corresponding the check-in of the changed code. Revision values depend on the SCM system.
- author - The user name of the user who checked the code in.

In this scenario, you will also choose how SCM change data is presented and used in policies.

1. Check out the code source.

   For guidance, see Step 2 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

2. Build the code.

   For guidance, see Step 3 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

3. Run the instrumented executables, and store the execution counts.

   For guidance, see Step 4 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

4. Retrieve the SCM data, and add it to the emit.

   Example:

   ```
   cov-import-scm --dir idir --scm git --log log.txt
   ```

   The `--scm` options are:

   - `accurev`
   - `clearcase`
   - `cvs`
   - `git`
   - `hg`
   - `perforce`
   - `svn`
   - `tfs{2012|2013|2015}`

     For example, `tfs2013`. This option must match the version of TFS that you are using.

   Test Advisor might produce error messages on some files, such as system include headers that are not under SCM control. Such errors will be displayed in the specified log file, or standard error if not specified. You can avoid these by setting up a more advanced SCM integration. For more detailed information about incorporating SCM data into your Test Advisor process, see Chapter 1.4, *Incorporating SCM data into Test Advisor*.

5. Copy and edit your policy file.

   Recently-changed lines of code may correlate with quality issues for a number of reasons:

   - New implementations will have less validation in the field than code that has been working for a while.

- New changes to existing code may suggest there were previous errors that needed to be corrected, or new functionality that needs to interact properly with older usage.

Test Advisor uses SCM change data in two ways: in presenting the sources in Coverity Connect, and as an input to quality rules. The settings for these are independent.

For display in Coverity Connect, copy and edit your existing policy file to add cutoff dates to the policy that are more meaningful than the defaults. For example:

```
cp example_1-function_100.json mypolicy.json
```

In your new policy file, edit the following two lines to capture your desired date range:

```
recent_date_cutoff: "2012-01-01", old_date_cutoff: "2011-01-01",
```

The cutoff dates define the age of the code as it is displayed in Coverity Connect.

6. Run the analysis with the updated policy file.

   Example for C/C++:

   ```
   cov-analyze --dir idir --test-advisor --strip-path <path> \
    --test-advisor-policy=<path>mypolicy.json
   ```

   Example for Java:

   ```
   cov-analyze --dir idir --include-java  --test-advisor --strip-path <path \
    --test-advisor-policy=<path>mypolicy.json
   ```

7. Commit the results to Coverity Connect.

   For guidance, see Step 6 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

8. Log into Coverity Connect to examine code coverage and SCM change data.

   To view these properties, see Section 1.8.2, “Displaying SCM data, line numbers, events, and Test Advisor coverage data”.

1.2.3. Adjusting the policy file

When you edit your policy file, you can change the policy settings and iterate your changes quickly. You do not have to rebuild your intermediate directory or rerun your tests. You can change your policy definitions and run the analysis command, and commit your results. For example, you can change your file and function exclusions and make sure you are excluding the right sections of code.

For more information about creating and editing your policy files, see Chapter 1.5, *Creating Test Advisor policies*.

Tip

The Coverity Wizard utility that ships with Coverity Analysis provides a graphical interface that helps you quickly build your policy file. For more information, see the *Coverity Wizard 2020.12 User Guide*.

Test Advisor provides a number of sample polices that you can use to understand the Policy language usage and how the files are constructed. Furthermore, you can use these policy files directly in your implementation, making changes that are appropriate for your code base and testing needs. The files are located in the following directory:

```
<install_dir_ca>/doc/examples/ta-policies/
```

The sample policies start with simple definitions and then build in complexity as they progress:

- `example_1-function_100.json`

  This policy requires that every function must have 100% of its lines covered. The `recent_date_cutoff` and `old_date_cutoff` attribute values control code age as it is displayed in Coverity Connect. The `violation_name` attribute defines the name that is assigned to defects.

- `example_2-file_100.json`

  This policy requires that every file must have 100% of its lines covered.

- `example_3-filename_filter.json`

  This policy requires that every function must have 100% of its lines covered. It also enforces that only the functions in files located in the `parser/src/` or `libs/src` directories should be considered for this rule.

- `example_4-ignore_annotations.json`

  This policy requires that every function must have 100% of its lines covered and that every file must have 100% of its lines covered. The lines between `begin-ignore-coverage` and `end-ignore-coverage` annotations (for example, in source code comments) are ignored for coverage purposes. The `use_filters` element allows multiple rules to share a common filter. The `define_filters` element creates a named filter that is referenced by `use_filters`.

- `example_5-modified_dates.json`

  This policy requires that every function must have 100% of its lines covered. Lines that are older than `2012-01-01` are ignored for coverage purposes. The `exclusion_reason` element adds its annotation to the output when the `--test-advisor-eval-output` option is passed to `cov-analyze`. This is useful for debugging purposes.

- `example_6_cover_all_returns.json`

  This policy example requires that all ways a function can terminate normally be covered by tests.

- `example_7-impacted_dates.json`

This policy example illustrates impact analysis. This policy uses RecentlyImpactedFunctionFilter to restrict the analysis to functions that have been impacted recently. For a workflow that uses impact analysis, see Section 1.2.4, “Getting impact analysis results”.

1.2.4. Getting impact analysis results

Test Advisor impact analysis identifies code with behavior that has been affected by a change to the program source code. A code change in one part of a program might cause other parts to change behavior. To validate that a code change does not cause the program to behave incorrectly, it is important for tests to cover all the impacted parts of a program, not just the changed code. For detailed information on this topic, including information about advanced usage options, see Part 2, “Test Advisor impact analysis”.

A Test Advisor policy focuses on impacted functions by using one or more of the following impact analysis filters:

- RecentlyImpactedFileFilter
- RecentlyImpactedFunctionFilter
- RecentlyImpactedLineFilter

Note

The policy example `example_7-impacted_dates.json` illustrates impact analysis. This policy uses the `RecentlyImpactedFunctionFilter` to restrict the analysis to recently impacted functions. It assumes that the policy file has an approriately defined value for recent_date_cutoff. This date indicates what changes should be considered recent for purposes of defining what functions are recently impacted.

Impact analysis makes use of historical information regarding the impact of functions on their callers. This information is stored as part of the data that is committed to Coverity Connect. To use impact analysis in your policies, you must retrieve this impact data from Coverity Connect *before running `cov-analyze`*. The retrieval process includes running the `cov-manage-history` command. Because this is historical data, it is not available in the first build for a project. The command should be omitted the first time Test Advisor runs.

Example of the first build:

1. Build the code.

   For guidance, see Step 3 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

2. Run the instrumented executables and store the execution counts.

   For guidance, see Step 4 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

3. Retrieve the Source Control management (SCM) data and add it to the emit.

   For example:

```
cov-import-scm --dir idir --scm git --log log.txt
```

4. Run the analysis with the policy file.

   Example for C/C++:

   ```
   cov-analyze --dir idir --test-advisor --strip-path <path> \
    --test-advisor-policy=<path>mypolicy.json
   ```

   Example for Java:

   ```
   cov-analyze --dir idir --include-java --test-advisor --strip-path <path> \
    --test-advisor-policy=<path>mypolicy.json
   ```

5. Commit the results to Coverity Connect.

   For guidance, see Step 6 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

Running subsequent builds with change impact:

The following steps are the same as the steps outlined in the previous procedure, except that `cov-manage-history` is added to the command sequence *before* the analysis.

Also note that in this example, it is assumed that the sample policy file, `mypolicy.json` has the value for `ImpactedDateLineFilter` appropriately defined for change impact. For an example of defining impact in a policy file, see Section 1.5.2.4, “To filter based on SCM change data”.

1. Build the code.

   For guidance, see Step 3 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

2. Run the instrumented executables and store the execution counts.

   For guidance, see Step 4 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

3. Retrieve the SCM data and add it to the emit.

4. Download the impact history to the intermediate directory.

   Example:

   ```
   cov-manage-history --dir idir download --host <cim-host> \
    --port 8080 --stream <stream> --user <user_name> --password <password>
   ```

   The `cov-manage-history` command queries a running Coverity Connect instance for the history of functions previously committed to a specified stream.

   Note

   `cov-download-history` is no longer supported as of the 7.5.0 release. Instead, use `cov-manage-history`.

5. Run the analysis with the policy file.

   Example for C/C++:

   ```
   cov-analyze --dir idir --test-advisor --strip-path <path> \
    --test-advisor-policy=<path>mypolicy.json
   ```

   Example for Java:

   ```
   cov-analyze --dir idir --include-java --strip-path <path> \
    --test-advisor --test-advisor-policy=<path>mypolicy.json
   ```

6. Commit the results to Coverity Connect.

   For guidance, see Step 6 in Section 1.2.1, “Performing a basic commit with Test Advisor”.

7. Log into Coverity Connect to examine the change impact.

   For examples, see Section 1.8.3.5, “Function Impact” and Section 1.8.3.4, “Line Impact”.

1.2.5. Importing history from another intermediate directory

The `cov-manage-history` command has an `import` command that acts an alternative to the `download` command. You can import impact history directly from another intermediate directory without the need to commit the history to Coverity Connect. This feature is meant to facilitate the testing and demonstration of impact analysis.

To use the import command, subsequent builds for change impact should use a different intermediate directory than the previous builds. For example, the flow described in Running subsequent builds with change impact should be changed to use `idir2` as the name of the intermediate directory to avoid conflict with the `idir` intermediate directory created in Example of the first build. Step 4 (of Running subsequent builds with change impact) can then be changed to use the import command instead of the download command of `cov-manage-history`:

```
cov-manage-history --dir idir2 import --from-dir idir
```

The commit that was performed in Example of the first build can be omitted, as history is now imported directly from the `idir` intermediate directory.

You can use the same intermediate directory for subsequent builds as the first build. This effectively makes subsequent runs analyze the same version of the code that was used to generate the impact history. Note that the import command is still necessary, however the same intermediate directory should be specified for both the `--dir` and `--from-dir` options. For example:

```
cov-manage-history --dir idir import --from-dir idir
```

1.2.6. Test boundary detection

Setting test boundaries allows Test Advisor to communicate meaningful data about individual tests in your system. This includes the coverage for each individual test (instead of coverage for all of the tests combined), the naming of tests, the grouping of tests into test suites, the status of individual tests, and the association between the tests and the code that the tests are running against.

For C/C++

There are two methods for setting up test boundaries for C/C++:

- Setting per process - allows you to set test boundaries for each individual `cov-build` process.

  Test Advisor checks several environment variables to determine each test's metadata to set test boundaries. Setting these environment variables typically requires modification to the test harness. Prior to each test's execution, the harness should set the following variables:

  - `COVERITY_TEST_NAME=<text>`
  - `COVERITY_SUITE_NAME=<text>`

- Setting within a single executable - allows you to use macros to establish test boundaries within your code for a single executable.

For more information about setting up test boundaries for C/C++, see Section 1.6.1, “Setting test boundaries for C/C++”

For Java

For Java, Test Advisor provides built-in support for common test execution frameworks, such as JUnit. It is possible to detect the boundaries of individual test cases that are executed within the same process. Furthermore, it is not necessary to set the environment variables because the test metadata can be derived from those test cases. Out of the box, Test Advisor includes test boundary detection configuration for common frameworks, such as junit3 and junit4. For example, to specify a common framework, you would specify it during the build command:

```
cov-build --java-coverage cobertura --java-test=junit --dir idir ant build …
```

It is also possible to configure Test Advisor for other text execution frameworks that follow similar patterns. For custom frameworks, you can use the following option:

`--java-config=<properties_file>`

For more information about setting up test boundaries for Java, see Section 1.6.2, “Setting test boundaries for Java”

For C#

For C#, Test Advisor provides built-in support for common test execution frameworks, such as MSTest. It is possible to detect the boundaries of individual test cases that are executed within the same process. Furthermore, it is not necessary to set the environment variables because the test metadata can be derived from those test cases. Out of the box, Test Advisor includes test boundary detection configuration for common frameworks, such as MStest, NUnit and XUnit. For example, to specify a common framework, you would specify it during the test command:

```
cov-build --test-capture --dir idir --cs-coverage opencover \
--cs-test mstest mstest /testcontainer:mydll.dll
```

By default, test frameworks such as NUnit and XUnit will copy the assemblies to a temporary location before running the tests. This can cause problems for Test Advisor since they do not copy the associated `.pdb` files. To avoid this issue you can run your tests with the `/noshadow` option to NUnit/XUnit.

It is also possible to configure Test Advisor for other text execution frameworks that follow similar patterns. For custom frameworks, you can use the `--cs-test-config=<properties_file>` option.

For more information about setting up test boundaries for C#, see Section 1.6.3, “Setting test boundaries for C#”.

Chapter 1.3. Instrumentation options

Table of Contents

With most developer-centric tests (tests that run as part of a build or through a separate subsequent command that exercises the locally-built artifacts), Test Advisor can collect runtime information by wrapping the test operations in `cov-build` command.

In this simple scenario, the `cov-build` command execution that is monitoring the tests has full access to the emit directory for both read and write processes. Because all of the testing is completed locally by the test harness (or its children), Test Advisor acquires all the data required for analysis when the test harness completes its test run. Realistically, however, not all test environments are this simple. For instance, tests might be distributed across remote machines for parallel execution. In this client/server testing scenario, the relationship between the test harness and the object under test might not fit this hierarchical order (in which every process that is executed has a child of the test runner). Longer running tests are more likely to involve multiple machines, complicated setups, and varying process lifetimes.

This chapter presents strategies for adding instrumentation for these scenarios for both C++ and Java.

1.3.1. Requirements for test capture

Before you add test instrumentation to your capture process, note the following requirements and conditions:

For C/C++:

- Executables have had instrumentation added by `cov-build` (note that instrumentation is self-contained within the executable).
- Any executables that are run have monitoring set up by a `cov-build` parent or ancestor.
- A dummy emit can be used to store unprocessed coverage data.

For Java:

- Instrumentation requirements are managed in the emit.
- Class files and Java bytecode are not instrumented until runtime, so they can be freely moved and incorporated into JAR files without worrying about instrumentation.
- `cov-build` is used to launch the JVM that runs the compiled Java classes.

- `cov-build` (and the JVM) must be able to query the emit for instrumentation requirements.

For C#:

- Executables are not instrumented until runtime so they can be freely moved without affecting the instrumentation. Note the `.pdb` files need to be moved with the executables.
- When running with NUnit/XUnit be sure to run with the `/noshadow` option. This prevents NUnit/XUnit from copying the executables to a temporary location. Test Advisor requires the `.pdb` files and NUnit/XUnit will not copy these files along with the executables.
- `cov-build` is used to launch the executable that runs the tests.

1.3.2. Executables have moved between build and test execution

Test Advisor does not assume the location of compiled artifacts.

For C/C++, `cov-build` manages any instrumented child executable that write coverage information.

For Java, `cov-build` instructs the JVM to instrument any classes it loads that the emit suggests are important to Test Advisor. In this case, where the executables may have been copied or moved but are still run on the same machine as the emit, cov-build can be run as usual. Finalization will automatically happen when the tests finish.

For C#, `cov-build` will instrument all child C# executables.

1.3.3. Collecting coverage data from remote machines

When the objects under test are run on a machine that is not the analysis machine hosting the emit, the basic strategy is to gather the runtime information on the remote machine and then merge that data into the primary emit. Once the data are merged, analysis and commit can proceed as usual. The `cov-build` command is used on the remote machine to manage data collection from the running processes (including handling per-test boundaries described in Chapter 1.6, *Configuring test boundaries*).

1.3.3.1. Collecting remote data by copying the intermediate directory

The following procedure describes the steps to collect coverage data from remote machines by copying in the intermediate directory. This procedure is primarily for C/C++, (although the workflow supports Java). However, see the preferred workflow for collecting coverage data from remote machines for Java.

1. Build and instrument (local):

   ```
   cov-build --dir original-emit make-executables
   ```

2. Prepare the remote emit (remote).

   a. For C/C++ when using `--c-coverage gcov` or `--c-coverage bullseye-small`, start with an empty emit directory.

   b. For Java, or for C/C++ with `--c-coverage bullseye`, make a copy of the existing emit and adjust to new location. On the remote host:

```
scp -r buildhost:original-emit copy-of-emit
```

```
cov-manage-emit --dir copy-of-emit reset-host-name
```

3. Run and monitor the tests; and collect (but do not process) coverage data (remote):

   ```
   cov-build --test-capture --dir copy-of-emit --leave-raw-coverage run-tests --c-
   coverage=gcov
   ```

   Note

   You must specify the coverage tool for `cov-build`.

4. Copy data from the remote host (if it is not directly available through the network) back to the local host, into a location that is separate from the original emit (local)

   ```
   scp -r remotehost:copy-of-emit copy-of-emit
   ```

5. Merge the new coverage into the original emit:

   ```
   cov-build --test-capture --dir original-emit --merge-raw-coverage copy-of-emit
   ```

6. Proceed with analysis and commit as usual.

1.3.3.2. Collecting data from a remote machine specifying a list of classes (preferred for Java)

This is the preferred method for Java because you only have to copy around a single text file, as opposed to copying the entire intermediate directory to the remote machines on which you want to run the tests.

1. Build and instrument (local):

   ```
   cov-build --dir <int_dir> ant
   ```

2. Create the list of classes (`list.csv`) on the build (local) machine:

   ```
   cov-manage-emit --java --dir <int_dir> list-compiled-classes > list.csv
   ```

3. Copy `list.csv` to the remote machine:

   ```
   scp list.csv remotehost:list.csv
   ```

4. Run `cov-build` on the remote machine:

   ```
   cov-build --test-capture --leave-raw-coverage --java-instrument-classes <list.csv>
    \
    --java-coverage=cobetura
   ```

   This allows `cov-build` to run against a nonexistent intermediate dir that will get populated.

5. Proceed with analysis and commit as usual.

1.3.3.3. Collecting data from a remote machine for C#

1. Build locally:

   ```
   % cov-build --dir idir-local msbuild /t:rebuild foo.sln
   ```

2. Copy the assemblies and their corresponding `.pdb` files to the remote machine (you do not need to copy the intermediate directory).

3. Run the tests on the remote machine, not processing the coverage.

   ```
   cov-build --test capture --leave-raw-coverage --dir idir-remote \
    --cs-coverage opencover mstest ...
   ```

4. Copy back the emit from the remote host.

   ```
   scp -r remotehost:idir-remote idir-remote
   ```

5. Merge the new coverage into the original emit.

   ```
   cov-build -test-capture --dir idir-local --merge-raw-coverage idir-remote
   ```

6. Proceed with analysis and commit as usual.

1.3.4. Collecting data from long-running processes

In the normal capture mode, `cov-build` starts a command and monitors it until it exits, at which point it assumes all processes (tests, in this case) have finished and it adds information from the command to the emit.

For some commands, the fact that the command has returned might not be a total indication that all processing is complete. Commands that start daemons or services are examples of this exception: the start command will return success (indicating that the service is running) but the object under test (the service itself) will only be at the beginning of its life cycle. External input might be used to exercise the service. Test Advisor should not collect the coverage data until the service is shut down and the coverage for the full lifetime of the service is written.

In this case, `--leave-raw-coverage` is used to delay finalization until it is explicitly requested; that is, after all processes have been shut down and the full lifetime coverage has been written. This coverage is then merged into the emit and analysis can commence. When `cov-build` is used to launch the service, it puts in place the coverage management for instrumented child executables. That coverage management is inherited by all child processes and will persist as long as those children are alive, even after the launch operation has returned and `cov-build` has exited. It is the responsibility of the user to make sure all processes have been shut down (which triggers the final write of their coverage) before merging coverage.

The pattern to handle asynchronous processes is similar to the remote machine process above, but the secondary emits are local. For example:

1. Build and instrument:

   ```
   cov-build -dir main-emit
   ```

2. For Java only, prepare the emit.

   ```
   cp -r main-emit secondary-emit
   ```

3. Run and monitor the tests; and collect (but do not process) coverage data (local):

   ```
   cov-build --test-capture --dir secondary-emit --leave-raw-coverage launch_daemon
   ```

4. exercise_daemon
5. shutdown_daemon
6. Merge the new coverage:

   ```
   cov-build --test-capture --dir main-emit --merge-raw-coverage secondary-emit
   ```

7. Proceed with analysis and commit as usual.

1.3.5. Using the Bullseye small runtime

In order to use `--c-coverage bullseye-small` you must first build the small runtime that will be linked into your build. Bullseye provides source for building a small runtime, but it must be modified for use with Test Advisor:

1. Test Advisor provides a tool to make changes to the small runtime easily. Run this tool giving it the Bullseye installation directory, and an output directory:

   ```
   cov-patch-bullseye --bullseye-dir /path/to/bullseye  --output-dir <output_dir>
   ```

   By default if no `--output-dir` is specified the tool will default to:

   ```
   /path/to/bullseye/coverity
   ```

   `cov-patch-bullseye` will not modify or remove any existing files. All changes are made either in the `<output-dir>` directory (or in the default directory if `<output-dir>` is not specified). Files are copied to this location and patched in place. The contents of the `--bullseye-dir` directory are left unchanged.

   ```
   cov-patch-bullseye --bullseye-dir /path/to/bullseye
   ```

   This will not modify/remove any existing files. All of the changes will be made in the `/path/to/bullseye/coverity` directory. Files will be copied into this location and patched in place. The original Bullseye directory will remain unchanged (except for the new directory).

2. Build the small runtime environment.

   When the `cov-patch-bullseye` tool completes you can build the small runtime. In addition to the making the necessary source modifications `cov-patch-bullseye` will also generate a Makefile to make building easier.

   Change to the output directory that you specified and run make:

   For Linux:

```
cd <output_dir>
make
```

For Windows:

```
cd <output_dir>
nmake
```

The makefile will provide more instructions about the targets that you should run in order to execute the build.

Note

- For Linux 32- and 64-bit, and Windows 32-bit, you only need to run one target (either `linux32`, `linux64`, or `win32`).
- For win64, what you build depends on whether or not you build 64-bit binaries:
  - If you only build 32-bit binaries, then only the `win32` target is necessary.
  - If you build 64-bit binaries, then you need to build `win64` as well.
  - Note that `win32` must be built from an X86 Visual Studio command prompt and `win64` must be built from an X64 Visual Studio command prompt.

3. Use the built small runtime.

   After the `make` completes successfully, the small runtime is ready for use. To use it simply provide this output directory to the `--bullseye-lib-dir` argument of `cov-build`. For example:

```
cov-build --dir idir --c-coverage bullseye-small --bullseye-dir <output-dir  \
 --bullseye-lib-dir <replaceable>output-dir</replaceable>/lib make build
```

```
cov-build --test-capture --dir idir --c-coverage bullseye-small --bullseye-dir
 <output-dir> \
 --bullseye-lib-dir /<replaceable>output-dir</replaceable>/lib make test
```

1.3.6. Collecting C/C++ coverage data without `cov-build`

In certain situations, it might not be possible to run your tests with `cov-build`. For example:

- If you run on a platform where `cov-build` does not work.
- If you use a coverage tool (such as Purecov or VxWorks CoverageScope) that does not have `cov-build` integration.

When `cov-build` cannot be used to wrap the tests you need to manually add the coverage to the emit. The following procedure describes the steps to use collecting coverage data without using the `cov-build` command:

1. Build your instrumented binaries.

Before you add coverage data, you need to create the emit directory (intermediate directory) for your source code. You can perform this build either with or without coverage instrumentation. For example:

Wrap a regular build:

```
cov-build --dir idir make
```

Wrap a regular build, but add gcov instrumentation:

```
cov-build --dir idir --c-coverage gcov make
```

Wrap a regular build, but add Bullseye instrumentation:

```
cov-build --dir idir --c-coverage bullseye \
 --bullseye-dir /path/to/bullseye make
```

Wrap a regular build, but add the Bullseye small runtime instrumentation:

```
cov-build --dir idir --c-coverage bullseye-small \
 --bullseye-lib-dir /path/to/small/runtime/lib \
 --bullseye-dir /path/to/bullseye make
```

Wrap a regular build with Function Coverage Instrumentation:

```
cov-build --dir idir --c-coverage=function make
```

This step is dependent on both the coverage tool you are using and your build system. This step results in the binaries that (when they are run) will collect coverage and produce data files for that coverage tool:

- Binaries built with Gcov should produce `.gcda` files.
- Binaries built with Bullseye should update the Bullseye file pointed to by the `COVFILE` environment variable.
- Binaries built with the Bullseye small runtime will create `BullseyeCoverage.data-<pid>` files in the current working directory when the test is run.
- Binaries built with VxWorks coverage tools (Bullseye and CoverageScope). For more information, see Appendix A, *Additional Test Advisor usage notes*.
- Binaries built with Function Coverage Instrumentation can write coverage data to a network connection or to a data file, as described in the examples below and in Function Coverage Instrumentation.

2. Run your tests to gather data.

   In this step, you run the command that will trigger the running of your tests, for example:

   Tests wrapped in make:

```
make test
```

Unit tests in a binary:

```
/path/to/bin/program-unit-test
```

Depending on the coverage tool that you are using, you may need to tell the coverage tool where to place the coverage data:

Gcov:
: Nothing special needs to be done before running your Gcov instrumented binaries, although there are a few notes about the gcov coverage files that you should keep in mind:

- By default, the coverage data will be placed beside each object file in the build.
- For each object file compiled with gcov, two files will be produced:

  `myobj.gcno` (at compilation time)

  `myobj.gcda` (at test run time)
- These files will be dispersed across your build tree in the same way as your object files.

Bullseye:
: Before you run the Bullseye tests, you have to tell Bullseye where the coverage data file can be found:

- To specify the coverage file to use for Bullseye, set the `COVFILE` environment variable to the location of the coverage file generated when you built the binaries.

  If you generated your instrumented binaries with `cov-build`, this file can be found in the intermediate directory:

  `<int_dir>/coverage/bullseye.cov`

  If you generated your instrumented binaries outside of `cov-build`, then the file will be at the location specified by the `COVFILE` environment variable, so you should not need to do anything special.

Bullseye small runtime:
: Nothing special needs to be done before you run the binaries instrumented with the Bullseye small runtime. However, you should note the following:

- There will be a `BullseyeCoverage.data-<pid>` file created in the current directory when the test binary is run. These files could be spread out depending on how your tests are run.
- If no coverage files are being written, you might need to modify your source code to flush the coverage data with the `cov_dumpData()` function. For more information, see http://www.bullseye.com/help/ref-cov_dumpData.html.

Function Coverage Instrumentation

To flush coverage data over the network, set the COVERITY_EMIT_SERVER_ADDR environment variable to the network address of the server. Example:

```
# On the server hosting the emit database:
  cov-manage-emit --dir idir start-server --port 15772

# On the test machine:
  PATH="<install_dir>/lib:$PATH" \
  COVERITY_EMIT_SERVER_ADDR=remote-host-name:15772 \
  COVERITY_TEST_CAPTURE_RUN_TIMESTAMP='date +%s' \
  make test

# On the server hosting the emit database, once tests are complete:
  cov-manage-emit --dir idir stop-server
```

- The `cov-manage-emit` command can start an emit server on a machine separate from where the tests are run.
- Network flush is the preferred way of gathering coverage data. See Network flushing for more information.

To write data to a file, set the COVERITY_COVERAGE_FILE environment variable to the file name and run the tests. Example:

```
PATH="<install_dir>/lib:$PATH" \
  COVERITY_COVERAGE_FILE=/tmp/coverage-data \
  COVERITY_TEST_CAPTURE_RUN_TIMESTAMP='date +%s' \
  make test
```

When executables with function coverage instrumentation are run without `cov-build`:

- Either COVERITY_EMIT_SERVER_ADDR or COVERITY_COVERAGE_FILE must be defined, but not both. See Function Coverage Instrumentation for more information.
- The lib directory must be added to the PATH variable. Without this, the instrumented programs will fail with an error that it cannot locate `libci-runtime.so` (on linux) or `libci-runtime.dll` (on windows).

PureCoverage:

The following information is provided as a general guideline. For specific instructions, see the official PureCoverage documentation.

For Linux/Unix:

- Set the `PURECOVOPTIONS` environment variable to include at least `-counts-file=/path/to/coverage.pcv` as well as any other options you might require.
- Run your test binary as usual. This will generate the `/path/to/coverage.pcv` file.

For Windows:

- Run your test binary under PureCoverage, for example:

  ```
  coverage /SaveMergeTextData=c:/coverage.txt /AutoMergeData=yes \
   mytest.exe arg1 arg2
  ```

  Note

  Purecov on Windows is slightly different than all other coverage tools, as you must specify the coverage file location on the same command line where you execute your tests.

3. Convert the coverage.

   After the tests have completed you should have collected some coverage data in the files associated with your particular coverage tool. Before you can add the data to the emit, you must convert this data into a format that the Coverity tools can understand. The conversion that needs to be done depends on the coverage tool:

   Gcov:
   No conversion is necessary. Test Advisor can read and parse the `.gcno` and `.gcda` files.

   Bullseye:
   Convert the Bullseye coverage file into a CSV format. This can be done using the `covbr` tool from Bullseye, for example:

   ```
   covbr --no-banner --quiet --csv --file bullseye.cov --output bullseye.cov.csv
   ```

   This `bullseye.cov` file will be the file that was specified in the `COVFILE` environment variable.

   Bullseye small runtime:
   The `BullseyeCoverage.data` files must be post-processed before Test Advisor can understand the format. For post-processing, Bullseye provides a tool named `covpost`. The following example demonstrates its basic usage:

   ```
   covpost -f mycovfile.cov BullseyeCoverage.data-<pid>
   ```

   Where `mycovfile.cov` is the file that you pointed COVFILE to when you performed the build. If you used `cov-build` to perform the build, this file can be found in the following location:

   ```
   <int_dir>/coverage/bullseye.cov
   ```

   `covpost` accepts multiple `.data` files on the command line, so you can specify multiple files to merge in at once. The important thing is to ensure that all `BullseyeCoverage.data-<pid>` files are post-processed.

   After the `.data` files are post-processed, the process is similar to the regular Bullseye process. Convert the Bullseye coverage file into a CSV format using the `covbr` tool from Bullseye, for example:

   ```
   covbr --no-banner --quiet --csv --file bullseye.cov --output bullseye.cov.csv
   ```

   This `bullseye.cov` file should be the file that you specified to the `covpost` command above.

Function Coverage Instrumentation
   No conversion is necessary.

PureCoverage:
   On Unix/Linux the `.pcv` file must be converted to a `.txt` file before you can add it to the coverage. For example:

```
purecov -export=output.txt /path/to/coverage.pcv
```

   On Windows, the output should already be in the appropriate .txt format.

4. Add the coverage data.

   Coverage data can be added to the emit with `cov-manage-emit`. For example:

```
cov-manage-emit --dir idir add-coverage <coverage-specific-options>
```

   The options that you will need to provide will depend on the coverage tool in use:

   Gcov:
      For each `.gcda`/`.gcno` pair:

```
cov-manage-emit --dir idir add-coverage --gcda myobject.gcda \
 --gcno myobject.gcno
```

      The `.gcda`/`.gcno` files are be spread throughout the build directory, so an effective way to find all of them is to run a find command:

```
find -type f -name \*.gcda
```

```
find -type f -name \*.gcno
```

   Bullseye and Bullseye small runtime:
      For the single generated `.csv` file:

```
cov-manage-emit --dir idir add-coverage --bullseye-csv foo.csv
```

   Function Coverage Instrumentation
      For networking, the coverage data has already been added.

      For the coverage file created by the test, use `cov-manage-emit` to add the coverage data. Example:

```
cov-manage-emit --dir idir add-coverage \
  --coverage-file=/tmp/coverage-data
```

   PureCoverage:
      For the single generated `.txt` file:

```
cov-manage-emit --dir idir add-coverage --purecov-text foo.txt
```

For all of the previous sample `cov-manage-emit` commands, you can also provide arguments to specify the test metadata such as `--testname`. For example:

```
cov-manage-emit --dir idir add-coverage --gcda myobject.gcda \
 --gcno myobject.gcno --testname mytest\
```

See the documentation for `cov-manage-emit` for a full list of arguments available to `cov-manage-emit add-coverage`.

At this point the intermediate directory is ready to be analyzed, just as if the tests were run under `cov-build`.

1.3.6.1. Manually adding test boundaries without `cov-build`

The information provided in Section 1.3.6, “Collecting C/C++ coverage data without cov-build” describes how coverage can be collected and added without the use of `cov-build`, however you can still add coverage manually to set test boundaries. It is important to note the following:

- If you do not use `cov-build`, the test boundary environment variables have no affect.
- If you do not use `cov-build`, the test boundary API can not be used.

For more information about setting test boundaries, see Chapter 1.6, *Configuring test boundaries* .

The general process is as follows (note that the process for each coverage tool will vary slightly, which is described at the end of this section):

1. Run a test.
2. Gather and add the coverage information generated by that test.
3. Clean up the coverage data before running another test.

The following example uses Gcov and demonstrates the basic process flow:

1. Create your intermediate directory:

   ```
   cov-build --dir idir make mytest
   ```

2. Build your program with coverage.

   This will produce `.gcno` files for each object file. For example:

   ```
   mytest mytest.o mytest.gcno
   ```

3. Run the first test:

   ```
   mytest test1
   ```

4. This will produce a `.gcda` file beside the object file. For example:

```
mytest mytest.o mytest.gcno mytest.gcda
```

5. Add the coverage for `test1` to the emit.

   Note that in this example, there is only one `.gcda` file. Your actual build will likely have multiple `.gcda` files (one per object). The following command will need to be repeated for each `.gcda`/`.gcno` pair

   ```
   cov-manage-emit --dir idir add-coverage --gcda mytest.gcda \
    --gcno mytest.gcno --testname test1
   ```

6. Clean up the existing coverage data before running the next test.

   Note that in this example only one `.gcda` file is removed. Be sure to remove all `.gcda` files that are generated by your test. Do not remove the `.gcno` files. For example:

   ```
   rm mytest.gcda
   ```

7. Run `test2`:

   ```
   mytest test2
   ```

8. Again, this will produce a `.gcda` file beside the object file:

   ```
   mytest mytest.o mytest.gcno mytest.gcda
   ```

9. Add the coverage for `test2` to the emit.

   Note the different `--testname` given to `cov-manage-emit`:

   ```
   cov-manage-emit --dir idir add-coverage --gcda mytest.gcda \
    --gcno mytest.gcno --testname test2
   ```

The example above demonstrated the general flow for achieving separated tests using `cov-manage-emit`, but there are some important notes to be aware for each coverage tool:

Gcov;

- Gcov will always write the `.gcda` files next to the object file. So it is important that you clean up all `.gcda` files between each test.
- This also means that you cannot run tests in parallel because separate `.gcda`. files are needed for each test.
- Depending on the version of GCC used, you might be able to use the `GCOV_PREFIX` environment variable to help with this. For more information, see the official GCC documentation.

Bullseye:

- Bullseye always writes coverage to the file specified by the `COVFILE` environment variable. So it should be possible to set `COVFILE` per test to get a separate file per test.

Note that you will need to start with the `COVFILE` that was generated by the build as Bullseye updates the information in this file. It will fail if its empty. For example:

```
# test1
cp <COVFILE-from-build> <COVFILE-for-test1>
export COVFILE=<COVFILE-for-test1>
runtest1
```

```
# test2
  % cp <COVFILE-from-build> <COVFILE-for-test2>
  % export COVFILE=<COVFILE-for-test2>
  % runtest2
```

Function Coverage Instrumentation
: Function Coverage Instrumentation does not support the COVERITY_TEST_SOURCE and COVERITY_TEST_SOURCE_ENCODING environment variables.

PureCoverage:

- Because you specify the `.txt` file for Purecov to write to, you just specify a different file for each test and then add that with `cov-manage-emit` specifying the appropriate test name.

You can also specify a suitename to the `cov-manage-emit`, by adding `--suitename mysuitename` to specify a suitename for the coverage you are adding.

1.3.7. Collecting Java coverage data using JaCoCo without `cov-build`

In certain situations, it might not be possible to run your Java tests with `cov-build`. For example, you might need to run your Java tests on a platform that is not supported by `cov-build`, such as on an Android device. In this situation, you can manually add JaCoCo coverage data to the emit. The procedure described in this section is only for JaCoCo and NOT for other Java coverage tools (Cobertura).

To collect JaCoCo coverage data without using `cov-build`:

1. Build your Java application and test class files using `cov-build`:

   ```
   cov-build --dir idir mvn compile
   ```

   This step creates an intermediate directory with the required structure and adds class files to the emit for which coverage should be generated.

   Note

   You do not need to specify the `--java-coverage` for `cov-build`, because no tests were run by the build command. Furthermore, the subsequent steps for this workflow do not require the `--java-coverage` option because the tests were run outside of `cov-build`.

2. Run `cov-manage-emit` to compute the default coverability.

```
cov-manage-emit --dir idir compute-coverability
```

3. Run your tests to gather coverage data.

   You must ensure that you have previously configured test coverage to be collected using JaCoCo. Also, you will need to check that the JaCoCo `.exec` file can be read using JaCoCo version 0.7.1 (the version supported by Test Advisor). For example:

   ```
   mvn verify
   ```

4. Copy the JaCoCo `.exec` file to a location so that it can be accessed by the machine on which you have completed the build.

   Note that this step is dependent upon how your tests were run. For example, if the tests were run on an Android device, you should consult the Android documentation for information about the location of the JaCoCo `.exec` file and how to copy the file.

5. Merge the coverage into the emit. For example:

   ```
   cov-build --dir idir --merge-raw-coverage-file jacoco:/path/to/jacoco.exec
   ```

   This command adds all coverage found in the given JaCoCo `.exec` file to the emit.

   The argument to the `--merge-raw-coverage-file` option specifies both the coverage tool and the file from which the coverage is read. The syntax is:

   ```
   <coverage_tool>:<filename>
   ```

   The `--merge-raw-coverage-file` option is only supported for JaCoCo.

   At this point, the intermediate directory can be analyzed as if the tests were run with `cov-build`.

   It is important that the class files built in Step 1 are the same class files that are present in the emit for this step (Step 5). The coverage results might be unpredictable if the JaCoCo coverage data is for different class files than the class files that are under test.

   All of the coverage data that is stored in the emit is attributed to a test suite named `default` and a test named `default`. This can be changed by specifying the `--suitename` and/or `--testname` arguments to `cov-build`, as required.

   For example, a command to specify the suite name and test name is as follows:

   ```
   cov-build --dir idir --java-coverage jacoco --suitename suite1 --testname test 1 \
    --merge-raw-coverage-file jacoco:/path/to/jacoco.exec
   ```

   In addition to specifying the suite name and test name, it is also possible to specify additional test properties such as test start date/time, test duration and test status. The options are:

   - `--teststart <date/time_when_the_test_started>` - The date/time must be in the format `yyyy-MM-SS hh:mm:ss`.

- `--testduration <test_duration_in_milliseconds>` - The test duration must be a valid non-negative integer.
- `--teststatus <test status>` - The test status can be one of the values `pass`, `fail`, or `unknown`.

If the `--testduration` option is specified without the `--teststart` option, the value of the `--testduration` option will be ignored and a warning will be displayed. The following example shows all of the specified options:

```
cov-build --dir idir --java-coverage jacoco me suite1 --testname test 1 \
 --teststart "2013-10-30 13:00:00" --testduration 1800 --teststatus pass \
 --merge-raw-coverage-file jacoco:/path/to/jacoco.exec
```

Chapter 1.4. Incorporating SCM data into Test Advisor

Table of Contents

Adding SCM change data to the Test Advisor build and analysis process provides important information about the quality and policy coverage of your tests. This information can be utilized in your test policy file to control which defects are to be reported and then displayed in Coverity Connect. In Test Advisor, SCM change data includes the following for each line of the source code:

- date - The date that the changed code was checked into the SCM system.
- revision - The revision number corresponding the check-in of the changed code. See Section 1.4.4, “SCM revision format” for more information.
- author - The username of the user who checked the code in. Author is limited to a maximum of 1024 characters.

Test Advisor provides several tools that enable you to extract and import accurate and effective change data from your SCM. SCM change data is only maintained for files that appear in the emit as a result of being used in the build. If a file is no longer used, the SCM change data is discarded and will not be retained in the event that a new build starts to reference the file.

1.4.1. Using coverage data import and extract utilities

After you build your tests, the resulting emit describes source files that were used in the build. SCM change data is not added to these sources during the build process but are added later in a specific SCM operation. You must add SCM change data before you perform an analysis, or before you commit your code to Coverity Connect.

This can be achieved using `cov-import-scm`. For example:

```
cov-import-scm --dir idir --scm git --log log.txt
```

While `cov-import-scm` can simplify the process, there are some limitations:

- Any of the files encountered during parsing (such as header files) might not be under SCM control, and querying the SCM for these files might be inefficient.
- The build might combine files from multiple repositories or SCM systems for which a single invocation of `cov-extract-scm` might not be appropriate.
- For files that have not changed from a previous analysis, there might be an advantage to re-using previously gathered data.

In these cases, you should use the underlying workflow and Test Advisor commands described Section 1.4.2, “Adding files under a directory hierarchy”.

1.4.2. Adding files under a directory hierarchy

Some of the limitations of `cov-import-scm` can be mitigated by using `--filename-regex`. This option allows finer control over the gathering of SCM information. Information is gathered only for filenames that match the regular expression. For example, if all of the source under SCM control is in a directory named `src`, it can be imported using the following command:

```
cov-import-scm --scm git --dir idir --filename-regex "/src/" --log log.txt
```

1.4.2.1. Generalized workflow

The basic workflow for `cov-import-scm` and the underlying commands invoked are as follows:

1. Identify the files that need SCM change data added.

   ```
   cov-manage-emit --dir idir list-scm-unknown --output needed-files.txt
   ```

2. Get the SCM change data for the source files.

   ```
   cov-extract-scm --scm git --input needed-files.txt --output scm-data.txt
   ```

3. Add the SCM change data to the emit.

   ```
   cov-manage-emit --dir idir add-scm-annotations --input scm-data.txt
   ```

If you cannot use `cov-import-scm` because of one of the limitations, the underlying commands can be invoked directly. For example, if you want to import all of the unknown files except the files in a directory named `/usr/include` (which are not under SCM control), you can use `cov-manage-emit list-scm-unknown` and then the following command:

```
grep -v /usr/include needed-files.txt > subset-needed-files.txt
```

You would then proceed with `cov-extract-scm` and `cov-manage-emit add-scm-annotations` commands as outlined above.

1.4.3. Caching files to leverage earlier SCM processes

Collecting SCM information is potentially time consuming. One approach for acceleration is to use caching. After running your build and capturing your tests, and before you execute queries of the SCM system, you can use caching whereby SCM change data from a previous build is reused. To cache files:

1. From an older emit, use `cov-manage-emit dump-scm-annotations` to export all annotations to a file. For example:

   ```
   cov-manage-emit --dir previous_emitdir dump-scm-annotations --output scm_cache.txt
   ```

2. For the new emit, use `cov-manage-emit add-scm-annotations` to incorporate the cached data. For example:

```
cov-manage-emit --dir current_emit add-scm-annotations --input scm_cache.txt
```

The SCM change data from the original build is used for all files with the same name and content signature. The last SCM change data is not added to new or newly-modified files.

After the cache has been established, you can perform additional SCM operations to provide any needed SCM change data using the same steps used without the cache.

1.4.4. SCM revision format

Revision values depend on the SCM system, Test Advisor uses the following definitions of revision:

- accurev - `"revision_id-transaction"`
- clearcase - `"extended_name"`
- cvs - `"revision"`
- git - `"revision"`
- hg - `"changeset"`
- perforce - `"change_set"`
- svn - `"revision"`
- tfs - `"changesetId"`

Revision is limited to a maximum of 64 characters.

1.4.5. Files without SCM data

It is possible that some files which are compiled when building your project will not have SCM data associated with them. Most commonly, this occurs for one of the following reasons:

- The file was newly created, and has not yet been checked into SCM.
- The file belongs to the underlying operating system, or other related system files. For example, C++ templates for STL exist in system header files which are typically not under SCM control.
- The file was automatically generated by the build, and was not checked into SCM. For example, C sources generated using the `yacc` utility might not have SCM data if only the `yacc` grammar files are kept under SCM.

Test Advisor treat files that do not have SCM information as if they were newly created. Policy files which include/exclude code based on SCM modification date (for example, `example_5-modified_dates.json` described in "Adjusting the policy file") will thus normally include such files when performing analysis.

As an exception to the above, Test Advisor will ignore all files which lie outside of the directories for the project being analyzed. The project directories are taken to be the arguments to the `--strip-path` option given to `cov-analyze`. This exception is meant to exclude files which belong to the system, rather than user-written code, since Test Advisor results for such files are typically not interesting.

Because files which lie outside of the specified `--strip-path` are ignored, you should choose the `--strip-path` argument carefully to ensure all your sources are analyzed by Test Advisor. The `--strip-path` argument can be specified multiple times, if your sources lie in distinct subdirectories.

If you wish to include all system files in the Test Advisor analysis, you may add the option `--strip-path /` to the `cov-analyze` options. This will cause all files to be included for analysis.

Normally, automatically-generated source files will be included for analysis, due to the behavior described above. If you wish to exclude such code from analysis, you can use the `HasSCMDataLineFilter` in the Test Advisor policy to exclude lines which do not have SCM data. This can be used in conjunction with `ContainsLineFileFilter` or `ContainsLineFunctionFilter` to exclude entire files or functions without SCM data, respectively.

See "Creating Test Advisor policies" for more details about the policy language, including descriptions of the filters that use SCM data. See `cov-analyze` in the *Coverity 2020.12 Command Reference* for details about the `--strip-path` option.

Chapter 1.5. Creating Test Advisor policies

Table of Contents

Test Advisor policies are used to determine the adequacy of developer tests in the code. A test policy is often specific to the system under test. Test policies perform dynamic observations of your system's behavior under tests, including policy coverage, value coverage, dynamic call graph, and so forth; including the origin of the data (the data that comes from the unit test suite and the data that comes from the integration test suite will possibly be handled differently by a policy).

Additionally, polices check program structure, such as AST, static call graph information gathered during static analysis, as well as SCM data, such as the last modification date for a line of code.

For example, a you might construct a policy that states that in component X (a component of your code base), each function that was modified during the last release must have at least 80% line coverage from unit tests, not counting the exception-handling code.

Test Advisor provides a policy language that allows you to create customized test policies for your system. The language allows you to create policies that target:

- By location (file/function)
- By SCM change data
- By complexity threshold (CCM)
- By impact (yes or no in a given snapshot with respect to a previous analysis snapshot)
- By impact age (recency of impact)

This chapter provides the following sections to help you create your policy files:

- Policy concepts - gives you an overview of a policy file's structure.
- Using Coverity-provided sample policy files to demonstrate comment policy tasks and usage.
- Policy language reference - contains descriptions the policy language elements and element attributes, syntax rules, policy evaluation rules, and so forth.
- Policy language example - provides example policy files. It is recommended that you study the examples to help you understand how to use the policy language.

Tip

The Coverity Wizard utility that ships with Coverity Analysis provides a graphical interface that helps you quickly build your policy file. For more information see Coverity Wizard 2020.12 User Guide.

1.5.1. Policy concepts

In the Test Advisor policy language, a *policy* is a set of rules defined in a single policy file using the JSON (JavaScript Object Notation). In Test Advisor all policy files use the `.json` file extension. Policy files are ASCII-encoded or UTF-8-encoded text files.

A *rule* contains a set of *filters*. Each rule is evaluated on all of the inputs that are necessary to determine compliance to the policy. A Policy reports violations on the particular entity to which it applies, such as a function or file. When violations are discovered, they are reported to Coverity Connect and include all the information necessary for developers to resolve the violation. Policy files are saved with a `.json` extension and are specified through the `cov-analyze --test-advisor-policy` option. For more information, see `cov-analyze`.

Filters define the scope of one or more rules through granularity and threshold settings. These settings, in turn, define when the coverage of code is sufficient. Granularity and threshold settings define how policy rules measure compliance based on the following:

- Killpath
- Dead code
- Analysis annotations
- SCM change data
- Syntactic constructs

1.5.2. Using the sample policy files to build common tasks

Test Advisor provides sample policies that represent common adherence conditions that will help you determine the quality and coverage of your tests. The goals of this section are:

1. To provide examples so you can understand the structure of the Policy language.
2. To use this file as template so you can modify and activate, for your policy integration.

This section primarily focuses on the `ta-policy-sample.json` file, which contains rules and filters for common policy tasks. Before you review the tasks presented later in this section, it is important to consider the following usage notes:

- In its original state, `ta-policy-sample.json` defines a simple general policy, however it contains additional filters that illustrate more complex tasks. You can activate these filers by removing the comment tags that precede them.

- The `ta-policy-sample.json` file is included at the end of this chapter for reference. The filters described in the task descriptions link to their location in the file reference. The steps also explain where to uncomment the filters in order to activate them.
- The actual editable file is located in the following directory:

  `<install_dir_ca>/doc/examples/ta-policies/`
- Descriptions of additional sample policies are described at the end of this section are described in Section 1.2.3, “Adjusting the policy file”.

1.5.2.1. To define function coverage

This is the basic task covered by `ta-policy-sample.json`. This policy enforces that each function must have at least 1% of it's lines executed by tests; generally this means that the function is at least partially covered by some test. The violation name for this rule is FUNCTION_INSUFFICIENTLY_TESTED, and this is how the violation will be named in Coverity Connect. In addition, this policy rule contains the following filters:

- `special-comments` ignores code marked with special comments that indicate that it does not need to be covered. See Section 1.5.2.3, “To filter based on comments”.
- `function-modified-recently` ignores functions which were modified before the `recent_date_cutoff` specified in the policy file. See Section 1.5.2.4, “To filter based on SCM change data”.

These filters are "enabled" by default, meaning that you do not have to uncomment them in any part of the policy file.

1.5.2.2. To filter on impossible paths

This task uses filters to exclude code that appears to be difficult, or impossible, to execute and when the code is executed, it often terminates the program. This type of code is usually not worth trying to test. This tasks requires the following filters:

- `abnormal-termination` excludes lines that are only executed if the program is about to abnormally terminate. For example:

  ```
  if (some_impossible_condition()) {
      log("Inconceivable!");       // EXCLUDED
      clean_up_as_best_we_can();   // EXCLUDED
      abort();                     // EXCLUDED
  }
  ```

  The call to `abort()` will abnormally terminate the program. That line, and the two preceding lines are excluded.

  You do not need to uncomment this filter. It is enabled by default.
- `runtime-exception` regards throwing a `RuntimeException` as being similar to calling `abort()` (from the example above), and it ignores such throws. It also ignores all code that is inevitably followed by that throw.

To enable this filter, remove the comments preceding the filter name.

- `ignore-catch-blocks` ignores everything inside of a `catch` block.

  To enable this filter, remove the comments preceding the filter name.

1.5.2.3. To filter based on comments

This task involves enabling one filter:

- `special-comments` excludes lines that have been marked with certain special comments in the source code. The choice of what comments to treat as special is arbitrary.

  You do not need to uncomment this filter. It is enabled by default.

1.5.2.4. To filter based on SCM change data

Policies that report violations based on SCM changes require that you have imported data from your SCM. For more information, see Chapter 1.4, *Incorporating SCM data into Test Advisor*. The following filter allows you to filter based on how recently your code has been changed:

- `function-modified-recently` passes functions that contain at least one line that was modified since the `recent_date_cutoff`.

  You do not need to uncomment this filter. It is enabled by default.

1.5.2.5. To filter debugging code

The following filters exclude debug code, logging code, and straight-line C++ print function from your tests:

- `debug-block` this filter excludes blocks of code are inside one of:

  ```
  This filter excludes blocks of code that are inside one of:
  "if (debug) {...}"
  "if (isDebugEnabled()) {...}"
  "if (isTraceEnabled()) {...}"
  ```

  This represents debug code that will not be included.

  To enable this filter, remove the comments preceding the filter name.

- `ignore-logging` ignores lines that call a `log4j` logger. Lines like these are uncovered because the are on debug path. Even when such lines aren't on a debug path, they are not generally important enough to enforce test coverage.

  To enable this filter, remove the comments preceding the filter name.

- `ignore-simple-cpp-print-functions` excludes straight-line C++ print functions, which are generally correct, and might only be called from debug routines.

The straight-line aspect is detected using the Cyclomatic Complexity (CCM). A CCM of 1 means that there are no branches.

To enable this filter, remove the comments preceding the filter name.

1.5.2.6. To filter based on filenames in Java source

The following filter focuses on Java source files that have an associated unit test. It relies on file naming conventions that are common in Java projects:

- java-file-has-unit-test uses a regular expression in the `file_name_regex` attribute.

  To enable this filter, remove the comments preceding the filter name.

1.5.2.7. To use "when" filters

The following "when" filter modifier causes a filter to take effect (to potentially exclude items) when some other condition is true:

Note

This portion of the policy file uses arbitrary names as examples of the filter's function. You will need to change the names to make of these filters in your code base.

- ignore-serialization-printers ignores print functions, but only within the `/serialization` subsystem.
- ignore-rpc-exception excludes code that throws exceptions in RPC (Remote Procedure Call) functions.
- ta_ignore-parser-generated-code excludes lines delimited by `"BEGIN GENERATED CODE"` and `"END GENERATED CODE` in the parser subsystem.

1.5.3. Policy language reference

This section provides a reference for all of the elements and their attribute values that you can use to construct a test policy.

1.5.3.1. JSON syntax

The Policy language is based on JSON. However, the language also uses the following extensions, all of which retain the property that the file format is a subset of JavaScript:

- Comments are allowed, both starting with `//` and extending to the end of the line, and starting with `/*` and ending with `*/`.
- Field names are not required to be quoted when they match the following regular expression:

  ```
  ^[a-zA-Z_][a-zA-Z0-9_]*$
  ```

  All field names in this file format conform to that regex, so none of them must be quoted in the JSON syntax. (However, quoting them is permissible based on standard JSON.)

- String literals may be extended across multiple lines (without introducing newlines into the string contents) by joining quoted string literal fragments with the + token, optionally surrounded by whitespace (including newlines). A string value may be composed of any number of concatenated fragments. This syntax follows that of JavaScript string concatenation.
- Objects and arrays may have a final `,` (comma).

1.5.3.2. Policy language implementation notes

In addition to the Policy language's extensions, it is important to understand the following:

- Regular expressions (regex) are used in Perl syntax. In JSON, it is represented as a string. It can match a substring of the target string unless anchors are explicitly used. Because the backslash is a metacharacter in both JSON and Perl regex, they must often be doubled. The name of a file is the sequence of directory names from the root, each pre-pended by "/", then "/" and finally the name of the file itself. The names are lowercased if they built on a Windows machine. Do not use a drive letter.
- All dates will conform to one of the following patterns:

```
YYYY-MM-DD                              Midnight, local time zone
YYYY-MM-DD[ T]hh:mm(:ss)?               Local time zone
YYYY-MM-DD[ T]hh:mm(:ss)?Z              UTC time zone
YYYY-MM-DD[ T]hh:mm(:ss)?[+-]hh:mm      Time zone explicitly specified

*Dates before 1970 are not allowed.

Examples:
2014-07-01                Midnight on July 1, 2014, local time zone
2014-07-01 13:00          1pm on July 1, 2014, local time zone
2014-07-01T13:00:30       30 sec after 1pm on July 1, 2014, local time zone
2014-07-01 13:00Z         1pm on July 1, 2014, UTC time zone
2014-07-01 13:00-07:00    1pm on July 1, 2014, Pacific Daylight Time
2014-07-01T13:00-08:00    1pm on July 1, 2014, Pacific Standard Time
2014-07-01T13:00+09:00    1pm on July 1, 2014, Japan Standard Time
```

- In the following reference the elements are grouped and arranged hierarchically by "type" so you can identify elements that are similarly grouped. JSON is not a hierarchically structured language, nor does it contain the concept of typed objects, but this document uses typed object in order to discuss the structure of the Policy Language.
- In the reference document, "container" types are denoted with square brackets ([]), meaning that the element expects an array, such a list of "sub-elements".

1.5.3.3. Demangled names

A demangled name is a string representing the fully-qualified name of a function. The format of the demangled name of a function depends on the language in which that function is written.

- For C: The demangled name of a function in C is simply the identifier for the function. For example, a function declared as `int foo(int x)` has a demangled name `foo`.

- For C++: Demangled names for C++ generally follow the grammar below (using regex-style notation):

  ```
  demangled_name     ::= qualified_id "(" arg_list? ")"
  qualified_id       ::= (qualifier "::")* identifier
  qualifier          ::= namespace | classname
  arg_list           ::= (arg ", ")* arg
  arg                ::= qualified_typename " "? ("*" const_opt)* "&"?
  qualified_typename ::= (qualifier "::")* typename
  const_opt          ::= " const "?
  ```

  For operator overloads, the function identifier is the string `operator`, followed by a space, followed by the operator that is being overloaded. Templated typenames have the following form with spaces used to separate successive greater-than signs:

  ```
  typename      ::= base_typename template_args
  template_args ::= "<" arg-list " "? ">"
  ```

  As an exception to the above, functions that are template instantiations have a demangled name that includes the return type of the function, as well as the template arguments to the function:

  ```
  demangled_name ::= qualified_typename " " qualified_id template_args
  "(" arg_list? ")"
  ```

- For Java: Demangled names generally follow the grammar outlined in the description of the MethodName value.

- For C#: Demangled names for C# generally follow the grammar below (using regex-style notation):

  ```
  demangled_name         ::= qualified_id "(" param_list ")" return_type
  qualified_id           ::= qualified_class_name "::" method_name
   number_of_type_params
  qualified_class_name  ::= qualifiers class_name
  qualifiers             ::= (namespace ".")? (outer_class_name "/")*
  outer_class_name       ::= class_name
  class_name             ::= identifier number_of_type_params
  number_of type_params ::= ("`" integer)?
  param_list             ::= (param_type ",")* param_type
  param_type             ::= type_name
  return_type            ::= type_name
  ```

  Constructors have a method name consisting of the string `.ctor`.

  Built-in types are represented by the `System` namespace types that they alias. For example, `int` is represented by `System.Int32`. Pointer types are appended by a `*` (for example, `System.Int32*` for a pointer to `int`). Reference and out parameter types are appended by a `&` (for example, `System.Int32&` for a reference to `int`).

Note that while "`.`" and "`$`" are characters that can appear in demangled names in Java and C#, they are also regex metacharacters. Such characters must be properly escaped when used as literals in a regex context. This can be done by prefixing the character with two backslashes in JSON. For example, the following filter matches a call to a function named `foo.bar` but not a call to a function named `fooxbar`:

```
{ call_to_named_function : "foo\\.bar" }
```

Two backslashes are required in JSON because it is necessary to insert a backslash in the regex to escape the metacharacter, but to obtain a single backslash in a string, you must enter a double backslash in the JSON file.

1.5.3.4. TAPolicyFile

A `TAPolicyFile` defines a policy, plus a few miscellaneous attributes explained in the following sections. This is the outermost entity in the policy file. Consequently, the policy file concrete syntax begins (modulo whitespace and comments) with the "{" token marking the start of the singular TAPolicyFile JSON object, and ends with the corresponding "}".

Attributes

- `type`

  A string identifying the type of file this is.

  ```
  type: "Coverity test policy definition",
  ```

- `format_version`

  A number identifying the revision of the file type.

  ```
  format_version: 1,
  ```

- `recent_date_cutoff`

  The date cutoff for lines to be marked as "recent" in the Coverity Connect UI. For example:

  ```
  recent_date_cutoff: "2012-01-01",
  ```

- `old_date_cutoff`

  The date cutoff for lines to be marked as "old" in the Coverity Connect UI. For example:

  ```
  old_date_cutoff: "2010-12-01",
  ```

- `rules:[]`

  The set of rules that make up the policy.

- `define_filters:[]`

  A set of named filters that may be referenced elsewhere using `use_filters`.

1.5.3.5. FilterContainer

A `FilterContainer` is an object type that contains a set of filters.

Attributes

- `use_filters:[]`

A set of filter names. Each name must match the `filter_name` of a `NamedFilter` in the same policy file. The named filter is logically incorporated into this one; for example, a line passes the present filter if it passes all of its own line_filters, and all of the `line_filters` of every referenced named filter.

- `file_filters:[]`

  A set of file filters. For file-granularity rules, only files that satisfy all filters are considered for violations. For function-granularity rules, only functions whose definition file satisfies all the file filters are considered.

- `function_filters:[]`

  A set of function filters. For file-granularity rules, every function defined in the file is passed through the filters; any function that does not pass all of the filters causes all of the lines in its definition to be excluded. For function-granularity rules, only functions that pass all filters are considered.

- `line_filters:[]`

  A set of line filters. A line must pass all the line filters to be considered.

- `test_suite_filters:[]`

  A set of filters to be applied to the test suites to restrict which tests count toward satisfying the coverage goal.

1.5.3.6. Rule

A `Rule` combines a set of filters with a coverage threshold and granularity. A `Rule` is evaluated to determine violations. It is a specialization of `FilterContainer`.

Attributes

- `violation_name`

  When a violation is reported, it is reported using the specified name as the "checker" name in the violation report.

  The name does not need to be unique; two different rules can produce violations with the same name.

- `rule_name`

  Optional. A unique identifier for this rule. It is an error for two rules in a policy to have the same `rule_name`. If not specified, then a unique name is generated based on the rule's `violation_name`.

- `suppressed_by_rule_regex`

  Specifies a regular expression which identifies one or more rules that suppress evaluation of this rule. That is, if a rule with a `rule_name` matching this regex generates a violation for a file or function, then this rule will not generate a violation for that file or function.

It is an error for a cycle to exist in the rule references generated by this attribute, i.e. a rule cannot suppress itself either directly or indirectly. It is also an error for a rule with file-level granularity to reference a rule with function-level granularity via this attribute, or vice versa. It is also an error to specify a regex which does not match any rule in the policy.

- `aggregation_granularity`

  Prior to applying the coverage threshold, the set of non-excluded lines are aggregated at one of two granularities: file or function.

- `applies_when_line_coverage_pct_is_at_least`

  Specifies, as a percentage, a minimum threshold for the ratio of covered to coverable lines required for this rule to generate a defect. If the coverage ratio is below the given value, the rule does not generate a violation. If not specified, the default value is 0.

  It is an error for a rule to have its `applies_when_line_coverage_pct_is_at_least` greater than or equal to its `minimum_line_coverage_pct`.

  For example, the following policy will report defects for functions with line coverage at least 10% but less than 80%. Users may find this a useful tool in improving coverage in areas of the code that already have at least some coverage.

  ```
  {
    type: "Coverity test policy definition",
    format_version: 1,
    recent_date_cutoff: "2014-01-01",
    old_date_cutoff: "2013-01-01",

    rules: [
      {
        violation_name: "Some coverage but not enough",
        aggregation_granularity: "function",
        minimum_line_coverage_pct: 80,
        applies_when_line_coverage_pct_is_at_least: 10
      }
    ]
  }
  ```

- `issue_category`

  When provided, this attribute specifies the category that Coverity Connect will assign to violations generated by the rule. A string between 1 and 100 characters long must be used. If this attribute is not specified, the default value is "Test advisor issues".

  By default, the issue category is displayed in Coverity Connect as a column in the List of CIDs pane.

  However, this column may be disabled in the view; see View Management.

- `issue_cwe`

When provided, this attribute specifies the CWE (Common Weakness Enumeration) identifier that Coverity Connect will assign to violations generated by the rule. A non-negative integer must be used. This attribute is optional and is omitted by default.

The issue CWE is displayed in Coverity Connect in the upper part of the Triage pane.

- `issue_impact`

  When provided, this attribute specifies the issue impact level that Coverity Connect will assign to violations generated by the rule. Acceptable values are "low", "medium", and "high". If this attribute is not specified, the default value is "low".

  Note that this attribute controls the issue impact level in Coverity Connect, and is not related to impact analysis. See the *Coverity Platform 2020.12 User and Administrator Guide* for more information about issue impact levels.

  By default, the issue impact is displayed in Coverity Connect as a column in the List of CIDs pane.

  However, this column may be disabled in the view; see View Management.

- `issue_local_effect`

  When provided, this attribute specifies the local effect text that Coverity Connect will assign to violations generated by the rule. A non-empty string must be used. If this attribute is not specified, the default value is "Test advisor issues".

  The issue local effect is displayed in Coverity Connect in the upper part of the Triage pane.

- `issue_long_description_html`

  When provided, this attribute specifies the long description that Coverity Connect will assign to violations generated by the rule. A non-empty string must be used. If this attribute is not specified, the default value is "File does not reach coverage threshold required by the policy" for rules with file-level granularity, and "Function does not reach coverage threshold required by the policy" for rules with function-level granularity.

  The value for this attribute may contain the following HTML tags, which are interpreted and displayed accordingly:

  - `<a>`
  - `<br>`
  - `<code>`
  - `<em>`
  - `<i>`

Special characters for HTML should be escaped accordingly to have them appear in the long description verbatim.

The issue long description is displayed in Coverity Connect in the upper part of the Triage pane.

- `issue_type`

  When provided, this attribute specifies the type that Coverity Connect will assign to violations generated by the rule. A string between 1 and 100 characters long must be used. If this attribute is not specified, the default value is "Insufficient file coverage" for rules with file-level granularity, and "Insufficient function coverage" for rules with function-level granularity.

  The issue type is displayed in Coverity Connect in the upper part of the Triage pane, as well as a column in the List of CIDs pane.

  However, this column may be disabled in the view; see View Management.

- `minimum_line_coverage_pct`

  Specifies, as a percentage, the minimum ratio of covered to coverable lines according to the rule. In any case where the actual ratio is less than this threshold, a violation is reported.

  This attribute takes an integer to represent the percentage. For example, `50` represents fifty percent.

  Values greater than 100 can be specified, in which case a violation is generated for any file or function which passes the filters for the rule, regardless of the measured coverage. This can be used to test the behavior of a policy, as this allows verification that the filters include/exclude the correct sets of files/ functions.

  For example, the policy below will identify (generate a violation) for every function which was modified on or after 2014-01-01:

```
{
  type: "Coverity test policy definition",
  format_version: 1,
  recent_date_cutoff: "2014-01-01",
  old_date_cutoff: "2013-01-01",

  rules: [
    {
      violation_name: "recently-modified",
      aggregation_granularity: "function",
      minimum_line_coverage_pct: 101,
      line_filters: [
        { recently_modified: true }
      ]
    }
  ]
}
```

- `old_date_cutoff`

The old date cutoff for this rule, used for evaluating filters based on this date for this rule. If this value is not specified, the global `old_date_cutoff` attribute is used, if it exists.

This allows each rule to use its own cutoff date. This may be useful to limit the number of defects reported by a rule that uses, for example, `MediumAgeModifiedLineFilter`. For instance, if one rule generates too many defects, the user may want to focus only on more recent defects by choosing a more recent cutoff date. As Test Advisor defects are resolved, the cutoff date can be gradually adjusted to report more defects.

- `recent_date_cutoff`

  The recent date cutoff for this rule, used for evaluating filters based on this date for this rule. If this value is not specified, the global `recent_date_cutoff` attribute is used, if it exists.

  This allows each rule to use its own cutoff date. This may be useful to limit the number of defects reported by a rule that uses, for example, `RecentlyModifiedLineFilter`. For instance, if one rule generates too many defects, the user may want to focus only on more recent defects by choosing a more recent cutoff date. As Test Advisor defects are resolved, the cutoff date can be gradually adjusted to report more defects.

1.5.3.7. NamedFilter

A `NamedFilter` is a filter (or set of filters) that has a name. It is a specialization of `FilterContainer`. The reason to give a filter a name is so that it can be referred to by the policy file in multiple places without duplicating the filter definitions.

Attributes

- `filter_name`

  The name given to the filter. It must be unique among all the named filters in the policy file. For example:

  ```
  filter_name: "file-has-tests",
  ```

1.5.3.8. Filter

A `Filter` is a condition that can be applied to some element of the data model and evaluates to true or false. If the filter evaluates to false, then the element is excluded from consideration. For a source code element, that means that its lines are not included in the coverage calculation. The meaning of excluding other elements is described in the following sections, along with their filters. If the filter evaluates to true, then the element passes the filter.

Attributes

- `exclusion_reason`

  Provides a string to use to identify the reason for a line being excluded from the rule, instead of resorting to naming it using the line in the policy file.

If you provide an exclusion reason, any lines thats are excluded will display the exclusion reason.

1.5.4. FileFilter

A `FileFilter` is a specialization of `Filter`, and allows filtering files. It, in turn, has a number of specializations, which are described in the related sections below.

A file filter is one of the following:

`NameFileFilter`

`AndFileFilter`

`OrFileFilter`

`NotFileFilter`

`ContainsFunctionFilter`

`ContainsLineFileFilter`

`ModifiedFileFilter [Deprecated]`

`ImpactedFileFilter [Deprecated]`

`RecentlyModifiedFileFilter`

`RecentlyImpactedFileFilter`

`RecentlyDirectlyImpactedFileFilter`

1.5.4.1. NameFileFilter

`NameFileFilter` filters files by their file name, and optionally by the existence of a file with a related name.

Attributes

- `file_name_regex`

  To pass this filter, the filename must match the specified regex. For example:

  ```
  file_name_regex: "^(.*)/main/java/(.*)\\.java$",
  ```

  The example above matches filename that has `/main` or `/java` and ends with `.java`. It remembers the part before `/main /java` as matched subgroup1 and the part between `/main`, `/java`, and `.java` as matched in subgroup2.

The regex may use grouping parentheses to define substring groups for use by `other_file_exists`.

`NameFileFilter` performs filename matching in a case-insensitive fashion.

- `other_file_exists`

  If this attribute is present, then to pass this filter, the file name must not only match `file_name_regex`, but there must also be another file that was compiled during the build whose name is exactly `other_file_exists`. The latter may use the notation `${N}`, where `N` is a decimal integer, to refer to substrings matched by `file_name_regex`. `${0}` refers to the entire matched portion of the name, while `${1}` and up refer to the substrings matching the parenthesized groups. For example:

  ```
  other_file_exists: "${1}/test/java/${2}Test.java",
  ```

1.5.4.2. AndFileFilter

A file passes this filter if it passes all of its constituent filters.

Attributes

- `and`

  Set of ANDed filters. If there are none, the file passes.

1.5.4.3. OrFileFilter

A file passes this filter if it passes at least one of its filters.

Attributes

- `or`

  Set of ORed filters. If there are none, the file does not pass the filter.

1.5.4.4. NotFileFilter

A file passes this filter if and only if it does not pass the filter specified as the value of the `not` attribute.

Attributes

- `not`

  Filter to invert.

1.5.4.5. ContainsFunctionFileFilter

A file passes the filter if there is at least one function in the file that passes the `contains_function` function filter.

Attributes

- `contains_function`

  The filter to be applied to the contained function definitions.

1.5.4.6. ContainsLineFileFilter

A file passes this filter if there is at least one line in the file that passes the `contains_line` line filter. Optionally, a minimum line count and/or percentage threshold can be specified to increase the number of lines which pass the inner filter needed to cause a file to pass.

Attributes

- `contains_line`

  The filter to be applied to the contained lines.

- `min_line_count`

  Specifies the number of lines which pass the inner filter (as an absolute count) needed to cause this filter to pass. Defaults to 1 if not specified.

- `min_line_pct`

  Specifies the number of lines which pass the inner filter (as a percentage of the total number of lines for the file) needed to cause this filter to pass. Defaults to 0 if not specified. Both the line count and line percentage thresholds need to be satisfied for a file to pass this filter.

1.5.4.7. ModifiedFileFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyModifiedFileFilter.

A file passes passes this filter if the comparison between the `modified_date` and the specified date matches the relation (before/on/after) specified by the filter The file passes the filter if all lines in all functions in the file do not have a `modified_date` (that is, if SCM data for the file is not present). The filter has one of the following attributes.

Attributes

- `min_modified_date`

  The minimum modified date.

- `modified_on_or_after`

  The minimum modification date (same as `min_modified_date`).

- `modified_on_or_before`

Modified on or before the specified date.

- `modified_after`

  Modified after the specified date.

- `modified_before`

  Modified before the specified date.

1.5.4.8. ImpactedFileFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyImpactedFileFilter.

A file passes a filter if the comparison between the `file_summary_date` and the specified date matches the relation (before/on/after) specified by the filter. `file_summary_date` is the most recent date when any function in the file has changed its summary, as computed by Coverity Analysis. The filter has one of the following attributes:

Attributes

- `min_impacted_date`

  The minimum impact date.

- `impacted_on_or_after`

  The minimum impact date (same as `min_impacted_date`).

- `impacted_on_or_before`

  Impacted on or before the specified date.

- `impacted_after`

  Impacted after the specified date.

- `impacted_before`

  Impacted before the specified date.

1.5.4.9. RecentlyModifiedFileFilter

A file passes this filter depending on the `modified_date` of the line of source code with the most recent `modified_date` in the file. The most recent `modified_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter is not sensitive to lines that are deleted from the source code, however, it is sensitive to changes in comments and whitespace. Consider using `RecentlyDirectlyImpactedFileFilter` instead to detect semantic changes to the source code, including deleted lines.

The filter requires SCM data. You can import SCM data with the `cov-import-scm` command.

Attributes

- `recently_modified`

  When true, a file passes if the most recent `modified_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if no line in the file has a `modified_date`, the file will pass this filter.

1.5.4.10. RecentlyImpactedFileFilter

A file passes this filter depending on the `impacted_date` of the function with the most recent `impacted_date` in the file. The most recent `impacted_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attributes

- `recently_impacted`

  When true, a file passes if the most recent `impacted_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if there are no functions in the file, the file will not pass the filter.

1.5.4.11. RecentlyDirectlyImpactedFileFilter

A file passes this filter depending on the `directly_impacted_date` of the function with the most recent `directly_impacted_date` in the file. The most recent `directly_impacted_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter ignores changes to comments and whitespace, but it is sensitive to deleted lines of source code. See also `RecentlyModifiedFileFilter`, which uses imported SCM data to filter based on the modification date of each line of source code. That filter is sensitive to changes in code comments, but it is unaware of deleted lines. See Direct impact for more information.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attributes

- `recently_directly_impacted`

When true, a file passes if the most recent `directly_impacted_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if there are no functions in the file, the file will not pass the filter.

1.5.5. FunctionFilter

A `FunctionFilter` is a specialization of `Filter` that evaluates to true or false for a given function. It, in turn, has specializations that are explained in the following related sections.

Every `FunctionFilter` has the following optional attribute:

- `when_in_file`

  When present, this function filter (the function filter in which this attribute appears) is evaluated as follows:

  1. Given a function, get its definition file and evaluate the `when_in_file` filter.
  2. If that filter evaluates to false, then this function filter evaluates to true.
  3. If instead that file filter evaluates to true, then evaluate this function filter normally.

  When the `when_in_file` attribute is present, the containing function filter only takes effect when the function is in a file satisfying the file filter. For example, this allows you to exclude functions with a certain pattern in their name, but only in one part of the code base (by using a filter on file name).

A FunctionFilter is one of the following:

`GetterFunctionFilter`

`SetterFunctionFilter`

`NameFunctionFilter`

`ReachableFunctionFilter`

`ExecutedFunctionFilter`

`CCMFunctionFilter`

`NumLinesFunctionFilter`

`AndFunctionFilter`

`OrFunctionFilter`

`NotFunctionFilter`

`ContainsLineFileFilter`

`ContainsASTNodeFunctionFilter`

`HasParamTypeRegexFunctionFilter`

`ModifiedFunctionFilter [Deprecated]`

`ImpactedFunctionFilter [Deprecated]`

`CallerCountFunctionFilter`

`AccessModifierFunctionFilter`

`ClassMemberFunctionFilter`

`RecentlyModifiedFunctionFilter`

`RecentlyImpactedFunctionFilter`

`RecentlyDirectlyImpactedFunctionFilter`

`AnnotatedFunctionFilter`

1.5.5.1. GetterFunctionFilter

A function passes this filter depending on whether it is a "getter" method, meaning its AST matches one of these patterns:

```
  { return this->$INSTANCE_FIELD; }    (C++)
  { return this.$INSTANCE_FIELD; }     (Java, C#)
  { return $STATIC_FIELD; }
  { return $GLOBAL; }                  (C++)
  { return $CONSTANT; }
```

`$INSTANCE_FIELD` is the name of an instance field, `$STATIC_FIELD` is the name of a static field, and `$GLOBAL` is the name of a global, namespace-scope or file-scope variable, and `$CONSTANT` is a constant expression. `$INSTANCE_FIELD`, `$STATIC_FIELD`, and `$GLOBAL` are possibly qualified with namespace and class names.

- `is_getter_method`

  If true, then only getters pass the filter; otherwise, only non-getters pass the filter.

1.5.5.2. SetterFunctionFilter

A function passes this filter depending on whether it is a "setter" method, meaning its AST matches one of these patterns:

```
  { this->$INSTANCE_FIELD = $PARAMETER; }  (C++)
  { this.$INSTANCE_FIELD = $PARAMETER; }   (Java, C#)
  { $STATIC_FIELD = $PARAMETER; }
  { $GLOBAL = $PARAMETER; }                (C++)
```

`$PARAMETER` is the name of a method parameter and the other placeholders are as defined in GetterFunctionFilter.

- `is_setter_method`

  If true, then only setters pass the filter; otherwise, only non-setters pass the filter.

1.5.5.3. NameFunctionFilter

A function passes this filter if its demangled name matches the specified regular expression.

See Section 1.5.3.3, “Demangled names” for information on the usage of demangled names.

- `func_name_regex`

  Regex to compare to the function name.

1.5.5.4. ReachableFunctionFilter

A function passes this filter depending on whether it is "reachable". A function is "reachable" if, starting from the set of functions that are called at least once during the tests, there is a path in the static call graph to the function.

- `is_reachable`

  When true, only reachable functions pass the filter; when false, only unreachable functions pass.

1.5.5.5. ExecutedFunctionFilter

A function passes this filter depending on whether there is at least one test that calls into the function.

Attributes

- `is_executed`

  When true, only executed functions pass the filter; when false, only non-executed functions pass.

1.5.5.6. CCMFunctionFilter

Cyclomatic complexity (CCM) measures the complexity of a function's code. A function passes this filter if its cyclomatic complexity (CCM) is, at least, of the specified value.

Attributes

- `min_ccm`

  Minimum CCM to pass the filter.

For more information about how CCM is calculated, please review the cyclomatic complexity metric definition.

1.5.5.7. NumLinesFunctionFilter

A function passes this filter if the number of lines in its definition is at least the specified value. The number of lines is the total number of lines from the first character of the function name in the function definition to the final closing brace. It does not exclude comments and blank lines.

Attributes

- `min_num_lines`

  Minimum number of lines to pass the filter.

1.5.5.8. AndFunctionFilter

A function passes this filter if it passes all of its constituent filters.

Attributes

- `and`

  Set of ANDed filters. If there are none, the function passes.

1.5.5.9. OrFunctionFilter

A function passes this filter if it passes at least one of its filters.

Attributes

- `or`

  Set of ORed filters. If there are none, the function does not pass the filter.

1.5.5.10. NotFunctionFilter

A function passes this filter if and only if it does not pass the filter specified as the value of the `not` attribute.

Attributes

- `not`

  Filter to invert.

1.5.5.11. ContainsLineFunctionFilter

A function passes this filter if its definition contains at least one line that passes the `contains_line` filter. Optionally, a minimum line count and/or percentage threshold can be specified to increase the number of lines which pass the inner filter needed to cause a function to pass.

Attributes

- `contains_line`

  The line filter.

- `min_line_count`

  Specifies the number of lines which pass the inner filter (as an absolute count) needed to cause this filter to pass. Defaults to 1 if not specified.

- `min_line_pct`

  Specifies the number of lines which pass the inner filter (as a percentage of the total number of lines for the function) needed to cause this filter to pass. Defaults to 0 if not specified. Both the line count and line percentage thresholds need to be satisfied for a function to pass this filter.

1.5.5.12. ContainsASTNodeFunctionFilter

A function passes this filter if its definition contains at least one AST node that passes the `contains_astnode` filter.

Attributes

- `contains_astnode`

  The AST node filter (see Section 1.5.7, “ASTNodeFilter”).

1.5.5.13. HasParamTypeRegexFunctionFilter

A function passes this filter if it has a parameter whose type, when rendered as a string in a manner conventional for the source language, is matched by the specified regex.

Attributes

- `has_param_type_regex`

  Regex to match the parameter type.

1.5.5.14. ModifiedFunctionFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyModifiedFunctionFilter.

A function passes this filter if the comparison between the its lines with the most recent `modified_date` and the specified date matches the relation (before/on/after) specified by the filter. If all lines of the function do not have a modified date, then the function passes the filter. This filter has one of the following attributes.

`modified_date` is the last time the line was physically modified according the SCM.

Attributes

- `min_modified_date`

  The minimum modification date for the function.

- `modified_on_or_after`

  The minimum modification date (same as `min_modified_date`).

- `modified_on_or_before`

  Modified on or before the specified date.

- `modified_after`

  Modified after the specified date.

- `modified_before`

  Modified before the specified date.

1.5.5.15. ImpactedFunctionFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyImpactedFunctionFilter.

A function passes this filter if the comparison between the `function_summary_date` and the specified date matches the relation (before/on/after) specified by the filter. This represents how Test Advisor determines the approximation of a function's externally-observable behavior being "impacted" by changes made (potentially elsewhere).

`function_summary_date` is the most recent date when the function summary as computed by Coverity Analysis changed. This filter has one of the following attributes.

Attribute

- `min_impacted_date`

  Minimum impact date for the function.

- `impacted_on_or_after`

  The minimum impact date (same as `min_impacted_date`).

- `impacted_on_or_before`

  Impacted on or before the specified date.

- `impacted_after`

Impacted after the specified date.

- `impacted_before`

  Impacted before the specified date.

1.5.5.16. CallerCountFunctionFilter

A function passes this filter if the number of functions of which it is in the callees set is greater than or equal to the `min_caller_count`.

Attribute

- `min_caller_count`

1.5.5.17. AccessModifierFunctionFilter

A function passes this filter if its access modifier attribute equals `has_access_modifier`.

Attribute

- `has_access_modifier`

  The required access modifier. This can be the empty string, in which case only functions whose access modifier is also the empty string will pass the filter. Valid values are:

  - `"" (empty string)` : function is not a method
  - `"public"` : public method
  - `"protected"` : protected method
  - `"private"` : private method
  - `"package-protected"` : Java default "package-protected" access
  - `"internal"` : C# "internal" access
  - `"protected-internal"` : C# "protected internal" access

  Note that "internal" and "package-protected" are synonymous and can be used interchangeably.

1.5.5.18. ClassMemberFunctionFilter

A function passes this filter if it is a member function of a class that passes the `member_of_class_type` filter.

Attribute

- `member_of_class_type`

The class filter.

1.5.5.19. RecentlyModifiedFunctionFilter

A function passes this filter depending on the `modified_date` of the line with the most recent `modified_date` in the function. The most recent `modified_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter is not sensitive to lines that are deleted from the source code, however, it is sensitive to changes in comments and whitespace. Consider using `RecentlyDirectlyImpactedFunctionFilter` instead to detect semantic changes to the source code, including deleted lines.

The filter requires SCM data. You can import SCM data with the `cov-import-scm` command.

Attribute

- `recently_modified`

  When true, a function passes if the most recent `modified_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if no line in the function has a `modified_date`, the function will pass this filter.

1.5.5.20. RecentlyImpactedFunctionFilter

A function passes this filter depending on the `impacted_date` of the function. The `impacted_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attribute

- `recently_impacted`

  When true, a function passes if its `impacted_date` is on or after the `recent_date_cutoff`. When false, this condition is negated.

1.5.5.21. RecentlyDirectlyImpactedFunctionFilter

A function passes this filter depending on the `directly_impacted_date` of the function. The `directly_impacted_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter ignores changes to comments and whitespace, but it is sensitive to deleted lines of source code. See also `RecentlyModifiedFunctionFilter`, which uses imported SCM data to filter based on the modification date of each line of source code. That filter is sensitive to changes in code comments, but it is unaware of deleted lines. See Direct impact for more information.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attribute

- `recently_directly_impacted`

  When true, a function passes if its `directly_impacted_date` is on or after the `recent_date_cutoff`. When false, this condition is negated.

1.5.5.22. AnnotatedFunctionFilter

A Java function passes this filter if it has an annotation whose name matches the specified regular expression.

Attribute

- `has_annotation_regex`

  The regular expression that must match the Java function's annotation name.

Note

The regular expression is matched against a Java annotation's fully-qualified name, including package name. If an annotation is declared inside a class, then its name will have nested-class syntax. See the following examples:

- `MyPackage.Annotation`
- `MyPackage.ParentClass$NestedAnnotation`
- `java.lang.Deprecated`
- `java.lang.annotation.Documented`

See also, ClassName value.

1.5.6. LineFilter

`LineFilter` is a specialization of `Filter` that passes or excludes source code lines. It, in turn, has several specializations, which are described in the following sections.

All `LineFilters` have the following optional attributes:

- `when_in_file`

When present, the line filter passes if the containing file passes the specified file filter and the normal logic of the line filter passes, or the file filter does not pass.

- `when_in_function`

  When present, the normal line filter "takes effect" only if the line is in a function that passes the specified filter. The exact behavior depends on what kind of rule is being evaluated:

  - In a file-granular rule, if there is a function that is (primarily) defined in the file under consideration, passes the function filter, and contains the line under consideration, then apply the line filter. Otherwise the line passes.
  - In a function-granular rule, if the function whose coverage is being evaluated passes the function filter, then apply the line filter. Otherwise the line passes.

Every line filter is one the following:

`ModifiedDateLineFilter [Deprecated]`

`ImpactedDateLineFilter [Deprecated]`

`AffectedDateLineFilter [Deprecated]`

`RegexLineFilter`

`RegexSectionLineFilter`

`AndLineFilter`

`OrLineFilter`

`NotLineFilter`

`ContainsASTNodeLineFilter`

`RecentlyModifiedLineFilter`

`RecentlyImpactedLineFilter`

`MediumAgeModifiedLineFilter`

`HasSCMDataLineFilter`

`LastModifiedByLineFilter`

1.5.6.1. ModifiedDateLineFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyModifiedLineFilter or MediumAgeModifiedLineFilter.

A line passes this filter if the comparison between the modified date and the specified date matches the relation (before/on/after) specified by the filter. If the line does not have a modified_date, then it also passes the filter. This filter has one of the following attributes.

Attributes

- `min_modified_date`

  The minimum date for a line to pass.

- `modified_on_or_after`

  The minimum modification date (same as `min_modified_date`).

- `modified_on_or_before`

  Modified on or before the specified date.

- `modified_after`

  Modified after the specified date.

- `modified_before`

  Modified before the specified date.

1.5.6.2. ImpactedDateLineFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyImpactedLineFilter.

A line passes this filter if the comparison between the impact date and the specified date matches the relation (before/on/after) specified by the filter. The impacted date for a line is determined by taking all function calls on the line, finding the date when the externally visible behavior for each call was last observed to change, then taking the latest of all such dates. If a line does not have an impacted date it does not pass the filter. This filter has one of the following attributes.

Attributes

- `min_impacted_date`

  The minimum impacted date to pass.

- `impacted_on_or_after`

  The minimum impacted date (same as `min_impacted_date`).

- `impacted_on_or_before`

  Impacted on or before the specified date.

- `impacted_after`

  Impacted after the specified date.

- `impacted_before`

  Impacted before the specified date.

1.5.6.3. AffectedDateLineFilter [Deprecated]

This filter is no longer supported as of version 7.6.0. Use of this filter will be treated as an error in the policy file. Instead, please use RecentlyModifiedLineFilter, MediumAgeModifiedLineFilter, or RecentlyImpactedLineFilter.

A combination of `ModifiedDateLineFilter [Deprecated]` and `ImpactedDateLineFilter [Deprecated]`. The affected date of a line is the latest of the modified date and the impacted date for that line. If only one of these attributes exists, then the existing date is used as the affected date and the missing attribute is ignored. If both of these attributes are missing, then the line passes this filter. This filter has one of the following attributes.

Attributes

- `min_affected_date`

  The minimum affected date to pass.

- `affected_on_or_after`

  The minimum affected date to pass (same as `min_affected_date`).

- `affected_on_or_before`

  Affected on or before the specified date.

- `affected_after`

  Affected after the specified date.

- `affected_before`

  Affected before the specified date.

1.5.6.4. RegexLineFilter

A line passes this filter if it (or an offset) matches the specified regex.

Attributes

- `line_regex`

The regex to apply to the line.

- `line_offset`

  When specified, line N passes the filter if line N+`line_offset` matches the regex. That is, an offset of 1 means Test Advisor checks the next line for the regex to decide if this line passes, and an offset of -1 means to check the previous line. When absent, line N itself is checked (offset 0).

  For lines near the ends of the file with non-zero offsets, Test Advisor logically behaves as if the lines beyond the ends are all empty strings. Therefore, if line N+`line_offset` is beyond the file ends, then line N passes the filter if and only if `line_regex` matches the empty string.

1.5.6.5. RegexSectionLineFilter

A line passes this filter if it matches the start regex or resides between the start regex and the end regex.

A common use of this filter is to exclude lines between two specific comments in the code. You might add those comments around sections of code that are known to be untestable or where you want to ignore the coverage for other reasons. Because `RegexSectionLineFilter` passes lines between the specified regexes, negation of this filter (using `NotLineFilter`) is needed to perform this task.

Example:

```
{ not: { start_section_regex: "cov-begin-ignore",
        end_section_regex: "cov-end-ignore" } }
```

If you apply the line filter shown in the example above to the code shown below, Test Advisor will ignore coverage for the `then` block.

```
some_code();
if (impossible)
  // cov-begin-ignore
  {
    lines_we_dont_want();
  }
  // cov-end-ignore
more_code();
```

Attributes

- `start_section_regex`

  The start-line regex.

- `end_section_regex`

  The end-line regex.

1.5.6.6. AndLineFilter

A line passes this filter if it passes all of its constituent filters.

Attributes

- `and`

  Set of ANDed filters. If there are none, the line passes.

1.5.6.7. OrLineFilter

A function passes this filter if it passes at least one of its filters.

Attributes

- `or`

  Set of ORed filters. If there are none, the line does not pass the filter.

1.5.6.8. NotLineFilter

A line passes this filter if and only if it does not pass the filter specified as the value of the `not` attribute.

Attributes

- `not`

  Filter to invert.

1.5.6.9. ContainsASTNodeLineFilter

A line passes this filter if there is an AST node on that line that passes the `contains_astnode` filter.

Attributes

- `contains_astnode`

  The node filter.

1.5.6.10. RecentlyModifiedLineFilter

A line passes this filter depending on its `modified_date`. The `modified_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

The filter requires SCM data. You can import SCM data with the `cov-import-scm` command.

Attributes

- `recently_modified`

  When true, a line passes if its `modified_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if the line has no `modified_date`, the line will pass this filter.

1.5.6.11. RecentlyImpactedLineFilter

A line passes this filter depending on the `impacted_date` of the line. The `impacted_date` is compared against the `recent_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have a `recent_date_cutoff`.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attributes

- `recently_impacted`

  When true, a line passes if its `impacted_date` is on or after the `recent_date_cutoff`. When false, this condition is negated. In either case, if the line has no `impacted_date`, the line will not pass this filter.

1.5.6.12. MediumAgeModifiedLineFilter

A line passes this filter depending on its `modified_date`. The `modified_date` is compared against the `old_date_cutoff` attribute of the Rule containing this filter. The policy is invalid if it uses this filter and neither the Rule nor the policy have an `old_date_cutoff`.

This filter requires impact history. You can download impact history using the `cov-manage-history` command. See Impact Dates for more information on how impact dates are calculated.

Attributes

- `medium_age_modified`

  When true, a line passes if its `modified_date` is on or after the `old_date_cutoff`. When false, this condition is negated. In either case, if the file has no `modified_date`, the line will pass this filter.

1.5.6.13. HasSCMDataLineFilter

A line passes this filter depending on whether it has SCM data or not.

Attributes

- `has_scm_data`

  When true, a line passes if it has SCM data. When false, the line passes if it does not have SCM data.

1.5.6.14. LastModifiedByLineFilter

A line passes this filter if the username of the line's most recent author matches a regular expression.

Attributes

- `last_modified_by_regex`

The regex to apply to the line's username.

1.5.7. ASTNodeFilter

`ASTNodeFilter` is a specialization of `Filter` that maps an AST node to true (pass) or false (reject). `ASTNodeFilter`, in turn, has specializations that are described the following related sections.

All `ASTNodeFilters` have the following optional attributes:

- `when_in_file`

  When present, the `ASTNodeFilter` filter passes if the containing file passes the specified file filter and the normal logic of the AST filter passes, or the file filter does not pass.

- `when_in_function`

  When present, the normal AST filter "takes effect" only if the containing function passes the specified function filter.

Every ASTNodeFilter is one of the following:

`StatementKindASTNodeFilter`

`ExpressionKindASTNodeFilter`

`VariableNameASTNodeFilter`

`IfStatementConditionASTNodeFilter`

`ContainedByASTNodeFilter`

`ContainsASTNodeFilter`

`CatchBlockTypeASTNodeFilter`

`AndASTNodeFilter`

`OrASTNodeFilter`

`NotASTNodeFilter`

`ThrowExpressionTypeASTNodeFilter`

`DominatedByASTNodeFilter`

`CallToNamedFunctionASTNodeFilter`

`UnconditionalTerminateCallASTNodeFilter`

`VariableKindASTNodeFilter`

`ContainedByLineASTNodeFilter`

`FunctionEndASTNodeFilter`

`DeadcodeASTNodeFilter`

1.5.7.1. StatementKindASTNodeFilter

An AST node passes this filter if it is a statement with the specified `kind`.

Attributes

- `statement_kind`

  The required value for `kind`.

  The statement `kind` types are:

  - skip - Empty statement (";").
  - label - Label for a goto.
  - case - A "case" statement in a switch.
  - default - The "default" case in a switch statement.
  - expression - A statement that evaluates an expression to perform computation.
  - compound - A brace-enclosed sequence of statements.
  - if - An "if" conditional statement.
  - switch - A "switch" conditional statement.
  - while - A "while" loop statement.
  - do_while - A "do" ... "while" loop statement.
  - for - A conventional "for" statement.
  - enhanced_for - A Java "for" statement that automatically iterates.
  - range_based_for - A C++11 "for" statement that automatically iterates.
  - break - A "break" statement in a loop or switch.
  - continue - A "continue" statement in a loop.
  - return - A "return" statement.
  - goto - A "goto" statement.
  - declaration - A statement that declares one or more variables.

- try - A "try" statement.
- catch - A "catch" handler.
- finally - A "finally" handler.
- synchronized - A "synchronized" statement (critical section).
- asm - An "asm" statement (inline assembly).
- java_assert - A Java "assert" statement.

1.5.7.2. ExpressionKindASTNodeFilter

An AST node passes this filter if it is an `Expression` with the specified `kind`.

Attributes

- `expression_kind`

  The required value for `kind`.

  The expression `kind` types are:

  - integer_constant: A literal or compile-time constant integer.
  - float_constant - A literal or compile-time constant floating-point value.
  - string_constant - A literal or compile-time constant string.
  - variable - A use of a variable, parameter or field.
  - assign: An assignment expression, including compound assignment (for example, `"+="`).
  - member_access - An expression denoting use of a field of an object.
  - sizeof - A use of the `"sizeof"` operator on either an expression or type.
  - noexcept - A use of the `"noexcept"` operator on an expression.
  - unary - A unary operator other than "&" or "*" applied to a sub-expression.
  - binary - A binary operator applied to sub-expressions.
  - address_of - The "&" operator applied to a sub-expression.
  - dereference - The "*" operator applied to a sub-expression.
  - array_length - An expression that retrieves the length of a Java array (for example, `"arr.length"`).
  - cast - A type cast expression.

- instanceof - An `"instanceof"` expression.
- class_literal - A Java class literal expression, for example, `"String.class"`.
- condition - A use of the `"?:"` ternary operator.
- call - A call to a function.
- new - A use of the `"new"` operator.
- delete - A use of the `"delete"` operator.
- throw - A `"throw"` expression.
- typeid - A use of the `"typeid"` operator on either an expression or type.
- va_arg - A use of the `"va_arg"` macro (not always recognized).
- statement - An expression that executes a statement (GNU extension).

1.5.7.3. VariableNameASTNodeFilter

An AST node passes this filter if it is an expression that refers to a variable whose name matches the specified regex.

Attributes

- `var_name_regex`

  The regex that the name must match.

1.5.7.4. IfStatementConditionASTNodeFilter

An AST node passes this filter if it is an "if" statement whose condition (the boolean expression that controls which branch is taken) passes the specified inner filter.

Attributes

- `if_statement_condition`

  The filter that the condition must pass.

1.5.7.5. ContainedByASTNodeFilter

An AST node passes this filter if any of its ancestor AST nodes, including the node itself and going up to the compound statement that is the function body, satisfies the specified inner filter.

Attributes

- `contained_by_astnode`

  The filter that the ancestor node must pass.

1.5.7.6. ContainsASTNodeFilter

An AST node passes this filter if any of its descendant nodes, including the node itself, passes the inner filter.

Attributes

- `contains_astnode`

  The filter that the descendant node must pass.

1.5.7.7. CatchBlockTypeASTNodeFilter

An AST node passes this filter if it is a catch statement whose type passes the inner filter, or whose type is a PointerType/ReferenceType where the pointee/referent passes the inner filter.

Attributes

- `catches_type`

  The filter that the caught type must pass.

1.5.7.8. AndASTNodeFilter

An AST node passes this filter if it passes all of its constituent filters.

Attributes

- `and`

  Set of ANDed filters. If there are none, the AST node passes.

1.5.7.9. OrASTNodeFilter

An AST node passes this filter if it passes at least one of its filters.

Attributes

- `or`

  Set of ORed filters. If there are none, the AST node does not pass the filter.

1.5.7.10. NotASTNodeFilter

An AST node passes this filter if and only if it does not pass the filter specified as the value of the `not` attribute.

Attributes

- `not`

  Filter to invert.

1.5.7.11. ThrowExpressionTypeASTNodeFilter

An AST node passes this filter if it is a throw expression whose type passes the inner filter, or whose type is a PointerType/ReferenceType where the pointee/referent passes the inner filter. If the AST node is a ThrowExpression with no subexpression, the node is rejected.

Attributes

- `throws_type`

  The filter that the thrown type must pass.

1.5.7.12. DominatedByASTNodeFilter

An AST node passes this filter if it is flow-dominated by the nodes which pass the inner filter.

A set of nodes S flow-dominates node X if every path from the function beginning to any exit point (return or throw) that executes X must also execute some node in set S (either before or after X).

Attributes

- `dominated_by_astnode`

  The filter used to determine the set of dominating nodes.

This filter can be used to exclude code which must always be executed whenever the code specified by the inner filter is executed. This is useful when the inner filter defines some code which cannot by easily tested; in this case, `DominatedByASTNodeFilter` can extend the exclusion to related code. For example, consider the code:

```
int foo(int x) {
    if(x > 0) return x;
    ++x;
    exit(0);
}
```

The filter:

```
{ not: { unconditionally_terminates: true } }
```

will exclude only the `exit(0)` statement. However, the filter:

```
{ not: { dominated_by_astnode: { unconditionally_terminates: true } } }
```

will exclude both the `exit(0)` statement and the `++x` statement.

1.5.7.13. CallToNamedFunctionASTNodeFilter

An AST node passes this filter if it is a call to a function with a demangled name that matches the given regex. See Section 1.5.3.3, “Demangled names” for information on the usage of demangled names.

Attributes

- `call_to_named_function`

The regex that the function name must match.

1.5.7.14. UnconditionalTerminateCallASTNodeFilter

An AST node passes this filter depending on whether it is a function call that unconditionally terminates the program.

Attributes

- `unconditionally_terminates`

  When true, a node passes the filter when it is a call that terminates the program. When false, a node passes the filter if it is not a call, or does not terminate the program.

1.5.7.15. DeadcodeASTNodeFilter

An AST node passes this filter depending on whether it is a Statement whose deadcode attribute equals true.

Attributes

- `is_deadcode`

  When true, a node passes the filter if it is a Statement and deadcode is true. When false, a node passes the filter if it is not a Statement or deadcode is false.

Note that setting `is_deadcode` to false is not effective, as any given line of source code will contain a statement which is not deadcode (specifically, the compound statement comprising the complete body of the function which contains that line will not be deadcode). More concretely, the following filter:

```
line_filters: [ { not: { contains_astnode: { is_deadcode: true } } } ]
```

will pass only lines that are not deadcode, while the following filterwill simply pass all lines:

```
line_filters: [ { contains_astnode: { is_deadcode: false } }
```

1.5.7.16. ContainedByLineASTNodeFilter

An AST node passes this filter if it is neither a CompoundStatement nor a SkipStatement, and any line which contains that node passes the inner filter. The containment relationship between AST nodes and lines is described in ContainsASTNodeLineFilter.

Attributes

- `contained_by_line`

  The filter that a containing line must pass.

1.5.7.17. VariableKindASTNodeFilter

An AST node passes this filter if it is a variable expression whose `kind` is the specified string.

Attributes

- The required variable `kind` value.

  The variable `kind` types are:

  - local - Local to a function.
  - parameter - Function parameter.
  - field - Non-static data member of a class.
  - static_field - Static data member.
  - global - Global, namespace or file-scoped variable.

1.5.7.18. FunctionEndASTNodeFilter

An AST node passes this filter depending on whether it is a statement that is either a "return" statement or is a last executable statement in the function on a path that does not have an explicit "return" statement. The intent is to recognize all ways that a function could terminate normally.

Attributes

- `is_function_end_statement`

  When true, the filter passes return statements and end of path statements. When false, the filter passes every other kind of AST node.

For example, the following filter:

```
{
    contains_astnode: {
      is_function_end_statement: true,
    },
 }
```

will pass the lines marked "PASS" in this code:

```
void f1(int x)
  {
    if (x) {
      return;      // PASS
    }
    x++;           // PASS
  }

  void f2(int x)
  {
    if (x) {
      x++;
      return;      // PASS
    }
    else {
      x--;         // PASS
    }
  }

  void f3(int x)
  {
    while (x--) {  // PASS (while loop, maps to header line)
      if (x==5) {
        return;    // PASS
      }
      if (x==7) {
        x++;       // PASS
        break;
      }
      if (x==9) {  // PASS (if statement, maps to header line)
        break;
      }
     }
  }
```

1.5.8. TypeFilter

TypeFilter is a specialization of `Filter` that maps a type to true or false.

A Type filter is one of the following:

- NameTypeFilter
- DescendsFromTypeFilter
- AndTypeFilter
- OrTypeFilter
- NotTypeFilter
- AnnotatedTypeFilter

1.5.8.1. NameTypeFilter

A Type passes this filter if it's name matches that the specified `regex` matches.

Attributes

- `type_name_regex`

  The regex that must match the name.

1.5.8.2. DescendsFromTypeFilter

A Type passes this filter if it is a class type that inherits from the type that matches the contained filter.

Attributes

- `descends_from`

The filter to apply to superclasses/interfaces.

1.5.8.3. AndTypeFilter

A type passes this filter if it passes all of its constituent filters.

Attributes

- `and`

  Set of ANDed filters. If there are none, the type passes.

1.5.8.4. OrTypeFilter

A type passes this filter if it passes at least one of its filters.

Attributes

- `or`

  Set of ORed filters. If there are none, the type does not pass the filter.

1.5.8.5. NotTypeFilter

A type passes this filter if and only if it does not pass the filter specified as the value of the `not` attribute.

Attributes

- `not`

  Filter to invert.

1.5.8.6. AnnotatedTypeFilter

A Java Type passes this filter if it is a class type with an annotation whose name matches the specified regular expression.

Attributes

- `has_annotation_regex`

  The regular expression that must match the Java class's annotation name.

Note

The regular expression is matched against a Java annotation's fully-qualified name, including package name. If an annotation is declared inside a class, then its name will have nested-class syntax. See the following examples:

- `MyPackage.Annotation`
- `MyPackage.ParentClass$NestedAnnotation`

- `java.lang.Deprecated`
- `java.lang.annotation.Documented`

See also, ClassName value.

1.5.9. Policy examples

This section contains an example of a policy constructed with the Test Advisor policy language. This policy requires every function to cover 1% of its lines; in most cases this means that some test calls the function. Functions that are modified before 2015-04-01 are ignored for coverage purposes.

This file is included in the Test Advisor installation. It is located at `<install_dir_ca>/doc/examples/ta-policies/ta-policy-sample.json`.

```
// ta-policy-sample.json
// Sample Test Advisor policy file.

// This file contains a sample TA policy.  In its original state, it
// defines a simple general policy:
//
//   Each function must have 1% of its lines executed by tests,
//   ignoring functions which were last modified before 2015-04-01,
//   and ignoring code marked with special comments indicating the code
//   does not need to be covered.
//
// This policy also contains additional (initially inactive) filters
// that illustrate various common tasks that can be done using a
// policy file.  It is intended that users can activate and modify
// sections as needed.
//
// Hint: Edit this file using an editor that does C/C++/Java syntax
// highlighting and delimiter balancing.

{
  // These two lines of preamble identify the file format.  Do not
  // change them.
  type: "Coverity test policy definition",
  format_version: 1,

  // These dates divide time into three "buckets": recent, old, and in
  // between.  The buckets are used by the defect presentation layer,
  // and for date-based filtering.
  recent_date_cutoff: "2015-04-01",
  old_date_cutoff:    "2010-01-01",

  // This section has the set of rules.  Each rule contributes a set of
  // violations independently of other rules.
  rules: [
    // Generate defects for all functions modified recently with insufficient
    // coverage ignoring souce constructs that do not interest us.
    {
      violation_name: "FUNCTION_INSUFFICIENTLY_TESTED",
      aggregation_granularity: "function",
      minimum_line_coverage_pct: 1,
      use_filters: [ "function-modified-recently",
                     "common-filters" ]
    },

    // This rule will not generate any defects. It will compute all policy line
    // counts so they are correctly loaded into Coverity Connect for use in reports.
    {
      violation_name: "FUNCTIONS_ALL",
      aggregation_granularity: "function",
      minimum_line_coverage_pct: 0,
      use_filters: [ "common-filters" ]
    }
  ], // end of rules

  // This section defines filters that can be used by rules.
  define_filters: [

    // --------------------------------------------------------------
    // --------------- Filtering applied to all rules ---------------
    // --------------------------------------------------------------

    // If you have multiple rules that use the same filters, you can
    // enable them here and add "common-filters" to the use_filters
    // section of each rule, to avoid repetition.
    {
      filter_name: "common-filters",

      use_filters: [
        //"abnormal-termination",
        //"runtime-exception",
        //"ignore-catch-blocks",
        "special-comments",
        //"ignore-simple-functions",
        //"debug-block",
        //"ignore-logging",
        //"ignore-simple-cpp-print-functions",
        //"java-file-has-unit-test",
        //"ignore-serialization-printers",
        //"ignore-rpc-exceptions",
        //"ignore-parser-generated-code",
      ]
    },

    // --------------------------------------------------------------
    // --------------- Filtering based on execution -----------------
    // --------------------------------------------------------------

    // This should be added to the use_filters section of the
    // applicable rule(s).
    {
```

```
    filter_name: "function-uncovered",

    function_filters: [{      // Only include functions that are not run
      is_executed: false
    }]
  },


  // --------------------------------------------------------------
  // ---------------- Filtering impossible paths ------------------
  // --------------------------------------------------------------

  // The filters in this section exclude code that appears to be
  // difficult or impossible to execute, and when executed, often
  // just terminates the program.  Such code is usually not worth
  // trying to test.

  // This filter excludes lines that are only executed if the
  // program is about to terminate abnormally.  For example, in
  // this code example:
  //
  //   if (some_impossible_condition()) {
  //     log("Inconceivable!");       // EXCLUDED
  //     clean_up_as_best_we_can();   // EXCLUDED
  //     abort();                     // EXCLUDED
  //   }
  //
  // The call to 'abort()' will terminate the program abnormally.
  // It, as well as the two preceding lines, are excluded by this
  // filter.
  {
    filter_name: "abnormal-termination",

    line_filters: [{
      // This string will appear in CC, associated with lines
      // that are excluded by this filter.
      exclusion_reason: "abnormal termination path",

      not: {                             // Exclude lines that
        contains_astnode: {                // contain an AST node that
          dominated_by_astnode: {            // is flow-dominated by
            unconditionally_terminates: true   // a call that terminates.
          }
        }
      }
    }]
  },

  // This filter is similar to "abnormal-termination".  It regards
  // throwing a RuntimeException as being similar to calling
  // abort(), and ignores such throws, as well as all code that
  // inevitably is followed by that throw.
  {
      filter_name: "runtime-exception",

      line_filters: [{
        exclusion_reason: "on path to throw of RuntimeException",
        not: {                               // Ignore any line that
          contains_astnode: {                  // contains a node that
            dominated_by_astnode: {              // is flow-dominated by a
              throws_type: {                       // throw of a type
                descends_from: {                     // that descends from
                  type_name_regex:                     // a type named
                    "^java\\.lang\\.RuntimeException$"   //
"java.lang.RuntimeException".
                }
              }
            }
          }
        }
      }]
    },

    // This filter ignores everything inside a "catch" block.
    //
    // It is in the "impossible" section just because it is most
    // closely associated with the others here.
    {
      filter_name: "ignore-catch-blocks",

      line_filters: [{
        exclusion_reason: "catch block",
        not: {                               // Ignore any line that
          contains_astnode: {                  // contains a node that
            contained_by_astnode: {              // is contained by
              statement_kind: "catch"              // a "catch" statement.
            }
          }
        }
      }]
    },


    // -------------------------------------------------------------
    // --------------- Filtering based on comments ------------------
    // -------------------------------------------------------------

    // This filter excludes lines that have been marked with certain
    // special comments in the source code.  The choice of what
    // comments to treat as special is arbitrary.
    {
      filter_name: "special-comments",

      line_filters: [
        {
          exclusion_reason: "marked with \"cov-ignore\"",
        not: {                            // Exclude lines where
          line_regex: "//.*cov-ignore"       // the line's text matches
        }                                    // the regex "//.*cov-ignore".
      },

      {
        exclusion_reason: "between \"cov-begin-ignore\" and \"cov-end-ignore\"",
        not: {                           // Exclude lines that
          start_section_regex:           // are between a line matching
            "//.*cov-begin-ignore",      // the regex "//.*cov-begin-ignore"
          end_section_regex:             // and a line matching
            "//.*cov-end-ignore"         // the regex "//.*cov-end-ignore".
        }
      }
    ]
  },


  // ----------------------------------------------------------------
  // -------------------- Age-based filtering -----------------------
  // ----------------------------------------------------------------

  // This filter only passes functions that contain at least
  // one line modified since the recent_date_cutoff (see defined above).
  // Lines without SCM data (e.g. automatically-generated code)
  // are ignored.
  {
    filter_name: "function-modified-recently",

    function_filters: [{                  // Only include functions that
      contains_line: {                    // contain a line that
        and: [
          { recently_modified: true },    // was modified since recent cutoff
          { has_scm_data: true } ]        // date and has SCM data.
      },
      exclusion_reason: "function not modified recently (recent cutoff date)"
    }]
  },


  // ----------------------------------------------------------------
  // ------------------ Filtering simple functions ------------------
  // ----------------------------------------------------------------

  // This filter only passes functions which involve at least one
  // decision (i.e. functions which have more than one path through
  // them).
  {
    filter_name: "ignore-simple-functions",

    function_filters: [{
      min_ccm: 2,
      exclusion_reason: "function is simple"
    }]
  },


  // ------------------------------------------------------------
  // ------------------- Filtering debug code ---------------------
  // ------------------------------------------------------------

  // This filter excludes blocks of code that are inside one of:
  //   "if (debug) {...}"
  //   "if (isDebugEnabled()) {...}"
  //   "if (isTraceEnabled()) {...}"
  // since that is debug code we don't care about testing.
  {
    filter_name: "debug-block",

    line_filters: [{
      exclusion_reason: "inside \"if (debug) {...}\"",
      not: {                              // Ignore a line if it
        contains_astnode: {                 // contains a node that
          contained_by_astnode: {              // itself is contained in
            if_statement_condition: {            // an "if" whose condition
              contains_astnode: {                  // contains a node that
                or: [                                // either
                  {
                    var_name_regex:                    // is a variable with
                      "debug"                            // "debug" in its name, or
                  },
                  {
                    call_to_named_function:            // is a call to
                      "is(Debug|Trace)Enabled"           // isDebugEnabled or
isTraceEnabled.
                  }
                ]
              }
            }
          }
        }
      }
    }]
  },

  // This filter ignores lines that call a log4j logger.  Often,
  // when such lines are uncovered, it is because they are on a
  // debug path.  Even when that isn't the case, they are rarely
  // interesting to cover in their own right.
  {
    filter_name: "ignore-logging",

    line_filters: [{
      exclusion_reason: "logging call",
      not: {                              // Ignore any line that
        contains_astnode: {                 // contains a node that
```

```
          call_to_named_function:               // is a call to a function whose
            "Logger\\.(warn|error|info|debug)"   // name matches the given regex.
        }
      }
    }]
  },

  // This filter excludes straight-line C++ print functions, which are
  // usually correct, and may be only called from debug routines.
  //
  // The straight-line aspect is detected using the Cyclomatic
  // Complexity (CCM).  A CCM of 1 means there are no branches.
  {
    filter_name: "ignore-simple-cpp-print-functions",

    function_filters: [{
      exclusion_reason: "straight-line print function",
      not: {                               // Ignore a function if it
        and: [
          {                                  // has a print-like name and
            func_name_regex: "(operator <<|print|write)"
          },
          {                                  // accepts an ostream& and
            has_param_type_regex: "^std::ostream &"
          },
          {
            not: {                           // does not
              min_ccm: 2                       // have a CCM of 2 or more.
            }
          }
        ]
      }
    }]
  },


  // -------------------------------------------------------------
  // ------------------ Filtering by file name --------------------
  // -------------------------------------------------------------

  // This filter focuses on Java source files that have an
  // associated unit test file.  It relies on file naming
  // conventions that are common in Java projects.
  {
    filter_name: "java-file-has-unit-test",

    file_filters: [
      {
        // First, we require that the file being tested match a
        // certain regular expression.  Note the grouping
        // parentheses.  The corresponding matched substrings will
        // be used next.
        file_name_regex: "^(.*)/main/java/(.*)\\.java$",
        //              ^^^^           ^^^^
        //              ${1}           ${2}

      // Then, additionally test that a certain other file exists,
      // where that other file depends on what we matched for the
      // file being tested.  "${1}" is replaced by the substring
      // that matches the first parenthesized group, and "${2}" is
      // replaced by the substring that matches the second group.
      // Then the resulting file name is checked for existence.
      // If it exists, the file being tested passes the filter.
      other_file_exists: "${1}/test/java/${2}Test.java"
    }
  ]
},


// -------------------------------------------------------------
// ---------------- Examples of "when" filters ------------------
// -------------------------------------------------------------

// The filters in this section illustrate the use of the "when"
// filter modifier, which causes a filter to only take effect
// (i.e., potentially exclude things) when some other condition is
// true.  They are just examples with some arbitrary names.  You
// would need to change the names to make use of them in your code
// base.

// This filter ignores print functions, but only within the
// "serializaton" subsystem.
{
  filter_name: "ignore-serialization-printers",

  function_filters: [{
    not: {                             // Exclude any function that
      func_name_regex: "print"           // has "print" in its name
    },
    when_in_file: {                    // if the containing file
      file_name_regex: "/serialization/"   // has "/serialization/" in its name.
    }
  }]
},

// This filter excludes code that throws exceptions in Remote
// Procedure Call (RPC) functions.
{
  filter_name: "ignore-rpc-exceptions",

  line_filters: [{
    not: {                             // Exclude any line that
      contains_astnode: {                // contains an AST node that
        dominated_by_astnode: {            // is flow-dominated by a
          expression_kind: "throw"           // 'throw'
        }
```

```
          }
        },
        when_in_function: {                     // if the line is in a function that
          func_name_regex: "^rpc_"                // has a name beginning with "rpc_".
        }
      }]
    },

    // This filter excludes lines delimited by "BEGIN GENERATED CODE"
    // and "END GENERATED CODE" in the "parser" subsystem.
    {
      filter_name: "ignore-parser-generated-code",

      line_filters: [{
        not: {                                // Exclude any line that
          start_section_regex:                  // is between a line that
            "BEGIN GENERATED CODE",               // matches "BEGIN GENERATED CODE"
          end_section_regex:                    // and a line that
            "END GENERATED CODE"                  // matches "END GENERATED CODE"
        },
        when_in_file: {                         // if the containing file
          file_name_regex: "/parser/"             // has "/parser/" in its name.
        }
      }]
    }

  ]  // end of defined_filters
} // end of policy definition

// EOF
```

Chapter 1.6. Configuring test boundaries

Table of Contents

Setting test boundaries allows Test Advisor to communicate meaningful data about individual tests in your system. This includes the coverage for each individual test (instead of coverage for all of the tests combined), the naming of tests, the grouping of tests into test suites, the status of individual tests, and the association between the tests and the code that the tests are running against.

Test Advisor reports these test boundaries to Coverity Connect so you can get a better visualization of how your individual tests, or test suites, provide policy coverage for your code. For more information about how test boundaries are represented in Coverity Connect, see Chapter 1.8, *Managing test results in Coverity Connect and Coverity Policy Manager*.

The process for setting up test boundaries for C/C++ and Java differ, but there are some common tasks. This chapter gives you examples for setting up test boundaries for each language.

1.6.1. Setting test boundaries for C/C++

There are two methods for setting up test boundaries for C/C++:

- Setting per process - allows you to set test boundaries for each individual `cov-build` process.
- Setting within a single executable - allows you to use macros to establish test boundaries within your code for a single executable.

1.6.1.1. Setting per-process test boundaries

Setting test boundaries in your C/C++ code can be done at the process level. Test Advisor accepts several environment variables that are checked when the test executable exits. Because of this, you cannot change the environment variable values within individual processes.

The start time of each test is taken from the start time of the `cov-build` command that ran the test. This means that all tests run by a single invocation of `cov-build` will share the same start time.

To help configure test separation, this section provides the following:

- Environment variable definitions

- A basic test separation example that shows how to implement the environment variables into your test executables.

1.6.1.1.1. Test boundary environment variables

Test separation is defined by the following environment variables and must be included in your build file before the test executable is called:

COVERITY_SUITE_NAME=`<suite_name>`

The name for a suite (or set) of tests. If this variable is not set, the name is assumed to be an anonymous test suite called `default`. If you set this variable, COVERITY_TEST_NAME must also be set. This variable can be set in each test build file, but it only needs to be set once during the process (see the example below.)

The variable accepts any string up to 256 characters. The test suite name "default" denotes the anonymous test suite and is given special treatment. For best results avoid using "default" as a test suite name.

COVERITY_TEST_NAME=`<test_name>`

The name of particular test. If this variable is not set, the name is assumed to be an anonymous test called `default`. If you set this variable, COVERITY_SUITE_NAME must also be set.

The variable accepts any string up to 2048 characters. The test name "default" denotes the anonymous test and is given special treatment. For best results avoid using "default" as a test name.

The following environment variables can be used to store additional test meta-data:

COVERITY_TEST_SOURCE=`<path_to_source>`

Sets the test source file. In C/C++, this is typically a shell script or makefile. `<path_to_source>` can be an absolute or relative path to the test source file, but the path must be valid at the time when the text executable process exits.

If the value ends with a colon (:) followed by one or more digits, then the digits shall be taken as the test source line number for the test, and only the portion of the value before the colon shall be used for the test source filename. A default line number of 1 is used if no line number is provided.

The test source file defined in this variable is displayed in the pop-up window for test coverage lines in Coverity Connect. For more information, see Chapter 1.8, *Managing test results in Coverity Connect and Coverity Policy Manager*.

Additionally, the test source is only recorded if an exit function (e.g., `exit()`) is called. So, if the following environment variables are set:

```
COVERITY_TEST_SOURCE=</path/to/test/source/file>
COVERITY_TEST_SOURCE_ENCODING=<encoding>
```

Then the following test code will not cause the test source to be recorded:

```
int main() { return 0; }
```

COVERITY_TEST_SOURCE_ENCODING=`<encoding>`

This variable is optional and is only used if COVERITY_TEST_SOURCE is also set. This variable sets the encoding to be used for the test source. If it is not set, the encoding is set to "US-ASCII". For the list of possible encoding options, see the `cov-emit --encoding` option.

COVERITY_TEST_STATUS=`<pass|fail|unknown>`

At the completion of each test's execution, the test should set this variable if you want the test to report explicit status to Test Advisor. The exit status code is checked as each process is executed as part of the test. If this variable is not included, Test Advisor infers the status of the test from the exit status code of the last process that is executed as part of the test.

By default, Test Advisor will determine the status of a test by the exit code of the last process that was run for that test: 0 (zero) is pass, non-zero is fail.

You can set this environment variable to override the process exit code. The valid values are pass, fail or unknown. Any value other than a valid one will use unknown. This variable only needs to be set if the process return code is not returning the status that you want or expect.

Test status is ignored if the test returns from `main()`. Calling `exit()` is needed to set the test status, even when using the `COVERITY_TEST_STATUS` environment variable.

1.6.1.1.2. Example: Setting up test boundaries

Test boundaries must be set at the appropriate location in your test harness and they will greatly vary for each test harness, but the following section provides a basic example how to use the environment variables for setting test boundaries.

Assume that you have the following makefile (the test harness is Make):

```
build:
    gcc -o /tmp/prog1 prog1.c
    gcc -o /tmp/prog2 prog2.c

test: test-prog1 test-prog2

test-prog1:
    /tmp/run-test1.sh

test-prog2:
    /tmp/run-test2.sh
```

1. Before running the tests, add test boundaries to the test harness.

   In the following file, the COVERITY_TEST_NAME and COVERITY_TEST_SOURCE environment variables are set to delineate test boundaries for `test-prog1` and `test-prog2` respectively.

   ```
   build:
       gcc -o /tmp/prog1 prog1.c
       gcc -o /tmp/prog2 prog2.c

   test: test-prog1 test-prog2
   ```

```
test-prog1:
    COVERITY_TEST_NAME=test1 \
    COVERITY_TEST_SOURCE="/tmp/run-test1.sh" \
    /tmp/run-test1.sh

test-prog2:
    COVERITY_TEST_NAME=test2 \
    COVERITY_TEST_SOURCE="/tmp/run-test2.sh" \
    /tmp/run-test2.sh
```

2. Set the COVERITY_SUITE_NAME environment variable for this analysis run.

   ```
   export COVERITY_SUITE_NAME="suite-run-2012-10-31"
   ```

   Note that this variable can alternatively be defined in the test file for each test. For this example, it only needs to be set once.

3. Run `cov-build` to build the test files with test boundaries.

   ```
   cov-build --test-capture --dir t1 --c-coverage gcov make test
   ```

4. Optional - verify that the test boundaries were set correctly.

   Use the `list-tests` option for `cov-manage-emit`:

   ```
   cov-manage-emit --dir t1 list-tests
   ```

   The output from the command will list the tests and test suites, as well as the test source file and the exit status. For example:

   ```
   Test 1:
     Name:     test1
     Suite:    suite-run-2012-10-31
     Date:     2012-10-31 07:03:00
     Status:   pass
     Duration: 23 ms
     Source:   /tmp/run-test1.sh (US-ASCII)
     Line:     1
   Test 2:
     Name:     test2
     Suite:    suite-run-2012-10-31
     Date:     2012-10-31 07:03:00
     Status:   pass
     Duration: 123 ms
     Source:   /tmp/run-test2.sh (US-ASCII)
     Line:     1
   ```

Note

The previous example does not show the use of COVERITY_TEST_STATUS. Typically, this variable would need to be set inside of the test shell script before it exits. It is only necessary if the return code is not sufficient, or incorrect (for example, if a test pass status returned 1).

1.6.1.2. Setting test boundaries for C/C++ within a single executable

The `COVERITY_*` environment variables can be used to set test boundaries at the process level. However, there are cases in which this approach is not favorable. For example:

- If you have long running daemon processes
- If you have unit test-like binaries where a number of tests are contained in a single executable

As a solution, Test Advisor provides an API that can be used to specify test boundaries by modifying the source code. The API is a single header file that can be included in your source. This allows you to call the various macros to control the test boundaries. This API can be found in the Coverity Analysis installation directory:

`<install_dir>/library/coverity-test-separation.h`

Note that typically you will want to use either the test boundary environment variables or the test boundary API, but not both. Using both at the same time may produce unexpected results.

Important

This test separation header is compatible with both the `--c-coverage gcov` (using gcc 3.4.6 and up) and `--c-coverage bullseye-small` options for `cov-build`. This will **not** work with the `--c-coverage bullseye` or `--c-coverage function` options. If you are using the Bullseye small runtime API, you will first need to build the modified Bullseye small runtime library. For more information, see Section 1.3.5, “Using the Bullseye small runtime”.

1.6.1.2.1. Using the test boundary API

To use the test boundary API:

1. Include the following header in your source code:

   `#include "coverity-test-separation.h"`

2. Modify your compilation command line to specify the include location where the header can be found. For example, if your original `gcc` command line was:

   ```
   gcc -c mysource.c
   ```

   It will now be:

   ```
   gcc -I <install_dir_ca>/library -c mysource.c
   ```

   Alternatively you can copy the `coverity-test-separation.h` header to an include directory already specified in your build.

After the API header is included, you can start defining test boundaries within your executable. Consider the following example test program:

```
mytest.c:
```

```
#include <stdio.h>

void testOne() {
     printf("testOne() called!\n");
}

void testTwo() {
     printf("testTwo() called!\n");
}

int main(int argc, char* argv[]) {
    testOne();
    testTwo();
    return (0);
}
```

With the API, you can define the test boundaries around each of the two functions. For example, a use of the API is as follows:

```
#include <stdio.h>
#include "coverity-test-separation.h"

void testOne() {
     printf("testOne() called!\n");
}

void testTwo() {
     printf("testTwo() called!\n");
}

int main(int argc, char* argv[]) {

    COVERITY_TS_START_TEST("test1");
    testOne();
    COVERITY_TS_END_TEST();

    COVERITY_TS_START_TEST("test2");
    testTwo();
    COVERITY_TS_END_TEST();

    return (0);
}
```

In the above case, coverage for `testOne()` will belong to the test 'test1', while coverage for `testTwo()` will belong to 'test2'.

It is important to note that these macros will only have an effect on a coverage build. If you are not building for Gcov or Bullseye, these will define away to nothing so your native build will continue to work as it previously has.

While this is a simple example, it shows how the source can be modified to define test boundaries within an executable. Thus, the typical flow would be:

```
COVERITY_TS_START_TEST("test1");
...
COVERITY_TS_END_TEST();
```

1.6.1.2.2. Test boundary macros

The API provides a number of different macros that can be used to define the desired test boundaries. This section provides descriptions of each of the macros and how they can be used. This part of the documentation can also be found in the API header file itself.

Note

The test boundary macros will only affect the process in which they are executed, and have no effect in subprocesses. For example, using the COVERITY_TS_START_TEST macro in one process will cause subsequent coverage within that process to be attributed to the named test, however coverage in subprocesses will be attributed to the default test. The environment variables described in Test boundary environment variables can be used to propagate test boundaries to subprocesses, if needed.

1.6.1.2.3. Macros for setting the suite name

COVERITY_TS_SET_SUITE_NAME(covsuite)
Set the suite name for tests. Typically this will only be called once per executable.

Requires:

- covsuite - (char*) the suite name to use

1.6.1.2.4. Macros for starting a test

For all of the macros to start a test, they will do the following:

- flush any existing coverage
- set the testname
- log the test start time

Depending on the macro used, it may also log additional information such as the test source file and encoding.

COVERITY_TS_START_TEST(covtest)
Start a test using the provided name.

Requires:

- covtest - (char*) the test name for this test

COVERITY_TS_START_TEST_SOURCE(covtest,covsource,covloc)
Start a test using the provided name and test source.

Requires:

- covtest - (char*) the test name for this test
- covsource - (char*) the filename to use as the source for this test
- covloc - (int) the line number to use for this source

COVERITY_TS_START_TEST_SOURCE_NOLINE(covtest,covsource)
: Start a test using the provided name and test source, not specifying a line for the test source.

  Requires:

  - covtest - (char*) the test name for this test
  - covsource - (char*) the filename to use as the source for this test

COVERITY_TS_START_TEST_SOURCE_ENCODING(covtest,covsrc,covloc,covenc)
: Start a test using the provided name, source and encoding.

  Requires:

  - covtest - (char*) the test name for this test
  - covtest - (char*) the test name for this test
  - covloc - (int) the line number to use for this source
  - covenc - (char*) the encoding to use for this source

COVERITY_TS_START_TEST_CURRENT_SOURCE(covtest)
: Start a test using the provided name, using the current source location as the test source.

  Requires:

  - covtest - (char*) the test name for this test

1.6.1.2.5. Macros for ending a test

For all of the macros to end a test, they will do the following:

- flush any existing coverage for the current test
- reset the testname to 'default'
- log the test end time

Depending on the macro used, it may also log additional information such as the result (pass/fail) of the test.

COVERITY_TS_END_TEST()
: End the current test.

COVERITY_TS_END_TEST_PASS()
: End the current test with a passing status.

COVERITY_TS_END_TEST_FAIL()
  End the current test with a failing status.

COVERITY_TS_END_TEST_RETURN_CODE(covretcode)
  End the current test with the status based on the specified return code.

  Requires:

  - covretcode - (int) the return code to set the pass/fail status from 0 = pass, non-0 = fail

COVERITY_TS_END_TEST_BOOL(covbool)
  End the current test with the status based on the specified boolean.

  Requires:

  - covbool - the boolean value to set the pass/fail status from

1.6.1.2.6. Helper macros

If the start/end flow does not fit easily into your code, you might get better results by doing things on a more granular level with the underlying helper macros, rather than using the `START_TEST_*` and `END_TEST_*` macros.

If you want to use the helper macros it is highly recommended to read through the following descriptions to ensure that they will accomplish what you intend. The `START_TEST_*` and `END_TEST_*` macros should be used unless there is a particular reason they cannot be.

Note

Note that all of the `START/END_TEST_*` macros are implemented by calling these helper macros; you can look at the `START/END_TEST_*` macros to understand what they do. This should help give you a better idea of what helper macros you might want to call if the `START/END_TEST_*` macros will not work for your purposes.

COVERITY_TS_SET_TEST_SOURCE(covsource,covline)
  Helper macro to set the test source file and line number.

  Requires:

  - covsource - (char*) the filename to use as the source for this test
  - covline - (int) the line number to use for this source

  Note

  This macro mostly behaves as expected. It will change the test source for the current test.

COVERITY_TS_SET_TEST_SOURCE_ENCODING(covencoding)
  Helper macro to set the test source encoding.

  Requires:

- covencoding - (char*) the encoding to use for this test's source

Note

This macro mostly behaves as expected. It will change the test source encoding for the current test.

COVERITY_TS_SET_TEST_STATUS(covstatus)
Helper macro to set the test status.

Requires:

- covstatus - (int) the return code to set the pass/fail status from 0 = pass, non-0 = fail

Note

This macro mostly behaves as expected. It will change the test status for the current test. However, if a subsequent `END_TEST_*` macro is called for this test that sets a status, the value may be overwritten.

COVERITY_TS_SET_TEST_START_TIME()
Helper macro to set the test start time.

Note

This macro will log the start time for the current test. It is important to note that it will only succeed the first time it is called for a test. Once the start time has been set. Subsequent calls will have no effect.

COVERITY_TS_SET_TEST_END_TIME()
Helper macro to set the test end time.

Note

This macro will log the end time for the current test. Unlike start time, this will work every time it is called, so it is safe to repeatedly call this - the last value will win.

COVERITY_FLUSH_COVERAGE()
Helper macro to flush the currently collected coverage.

Note

This macro works mostly as expected. It will flush the current coverage to the current test name.

For more information about Gcov coverage flushing, see Section A.1.3, “Gcov and flushing coverage”.

COVERITY_TS_SET_TEST_NAME(covtest)
Helper macro to set the test name.

Requires:

- covtest - (char*) the test name for this test

> **Note**
>
> This macro will only change the currently set test name. This means that the additional information logged by the `START_TEST_*` macros will not be logged. This includes:
>
> - start time
> - test source file
> - test source file encoding
>
> If setting these for this test is desirable you must set these after the testname has changed using the appropriate macro.

1.6.2. Setting test boundaries for Java

This section describes the following methods for establishing test boundaries for Java:

- Setting per-process test boundaries with environment variables
- Setting test boundaries for Java with the API

1.6.2.1. Setting per-process test boundaries for Java with environment variables

The COVERITY_SUITE_NAME and COVERITY_TEST_NAME environment variables can be used to group all tests executed by an instance of JVM into a single test (corresponding to COVERITY_TEST_NAME set prior to JVM invocation) and then assign them to the appropriate test suite (corresponding to COVERITY_SUITE_NAME set prior to JVM invocation). However, it is possible to further subdivide tests that are executed by a single JVM instance.

1.6.2.1.1. Test boundary properties file

Test boundary detection for Java is configured through a properties file specified using the `--java-test-config` option. For example:

```
cov-build --test-capture --dir idir --java-test-config myconfig.properties
 <build_or_test_command>
```

> **Note**
>
> The `--java-test-config` should be used with the command that is responsible for executing the tests. For example, use the option with `cov-build` if the tests are run as part of the build. Use the option with `cov-build` if tests are executed separately from the build process.
>
> The `cov-build` command can accept multiple test boundary properties files on the command line. See the following examples:

```
cov-build --java-test junit --java-test junit4
```

```
cov-build --java-test-config test-config1.properties --java-test-config test-
config2.properties
```

```
cov-build --java-test junit --java-test-config test-config.properties
```

A test is recognized when it matches any of the specified property files.

The following properties can be set in this properties file:

`coverity.test.class.regex`

This property configures a regular expression for matching against test class names. When this property is set, only those classes whose name matches the specified regular expression are considered to be potential test classes (whose methods are further examined to determine if they represent test boundaries). When this property is not set, class name does not play role in determining which classes are considered to be potential test classes.

Example:

```
coverity.test.class.regex=pkg/.*/TestClass.*
```

`coverity.test.class.annotation.regex`

This property configures a regular expression for matching against test class annotations. When this property is set, only those classes that are annotated with a Java annotation whose name matches the specified regular expression are considered to be potential test classes (whose methods are further examined to determine if they represent test boundaries). When this property is not set, class annotations do not play role in determining which classes are considered to be potential test classes.

Example:

```
coverity.test.class.annotation.regex=pkg/meta/TestClassMarker
```

Note

When both `coverity.test.class.regex` and `coverity.text.class.annotation.regex` are set, a class must meet both criteria to be considered a test class. When neither property is set, no class is considered to be a test class and further test boundary detection is not performed.

`coverity.test.method.regex`

This property configures a regular expression for matching against test method names in classes that are considered to be test classes. When this property is set, the methods whose name matches the specified regular expression are considered to be test entry points. When this property is not set, method name does not play a role in determining which methods are considered to be potential test entry points.

Example:

```
coverity.test.method.regex=test.*
```

`coverity.test.method.annotation.regex`

This property configures a regular expression for matching against test method annotations in classes that are considered to be test classes. When this property is set, those methods that are annotated with a Java annotation whose name matches the specified regular expression are considered to be test entry points. When this property is not set, method annotations do not play a role in determining which methods are considered to be potential test entry points.

Example:

```
coverity.test.method.annotation.regex=pkg/meta/TestMethodMarker
```

Note

when both `coverity.test.method.regex` and `coverity.text.method.annotation.regex` are set, a method must meet both criteria to be considered a test entry point. When neither property is set, no method is considered to be a test entry point and test boundary detection is not performed.

Test boundary detection is performed only at the top level and not within other tests. In other words, when the test execution framework calls `testA` that subsequently calls `testB` and both tests would be considered test entry points by the above rules, only `testA` is recorded as the boundary for the entire test.

Properties that represent regular expressions that are matched against Java type names (`coverity.test.class.regex`, `coverity.test.class.annotation.regex`, and `coverity.test.method.annotation.regex`) are matched against Java binary class names, for example, `java/lang/String` as opposed to `java.lang.String`.

When test boundary detection is performed using the above annotations, Test Advisor determines the status of a test by checking whether a test has thrown an exception. A test method that successfully completes without throwing an exception is considered to be a passing test. A test method that throws an exception is considered to be a failing test. For test execution frameworks that support the use of annotations to specify an expected exception (such as JUnit 4), `coverity.test.method.annotation.expected.field.regex` property can be used to inform Test Advisor which annotation field (element) is used to specify the expected exception class. For example:

```
coverity.test.method.annotation.expected.field.regex=expected
```

1.6.2.1.2. Sample property files

Test Advisor provides two configuration files that specify test boundary configuration for the popular JUnit test execution framework (one for JUnit 4 and one for earlier versions of JUnit).

These properties files are located in the following directory:

```
<install_dir_ca>/config/
```

For convenience, the `cov-build --java-test <config-name>` command line option can be used to point to these files instead of specifying the full path to the `config/` directory. The following values are allowed for `<config-name>`:

- `junit4` uses `method.annotation.regex` to identify JUnit4 test methods based on @Test annotation.
- `junit` uses `method.regex` to identify tests based on method names starting with 'test'.

1.6.2.2. Setting test boundaries for Java with the API

The `TestManager` class defines the Coverity API for test separation. `TestManager` is responsible for dispatching API calls to the Test Advisor runtime through reflection, eliminating the dependency on having Test Advisor runtime classes in the class path during compilation.

The `TestManager` class representing the API is compiled into `coverity-test-separation.jar` and installed under `<install_dir_ca>/library` (along with other public jars). This class must be linked into your application.

`TestManager` correctly handles the absence of Test Advisor runtime classes on the class path (that is, when not running under `cov-analysis` tools) and turns all API invocations into no-ops. All of the binding of runtime methods and classes is handled in the static constructor using reflection. In case of errors during initialization, the field `initializationError` will contain the exception that caused the error.

1.6.2.2.1. Using the test boundary API

This section describes the usage flow for the test boundary API for Java. This section refers to methods that are contained in the `TestManager` class. Reference information for these methods are provided in the JavaDoc for the class in the following location:

`<install_dir_ca>/library/coverity-test-separation-javadoc.jar`.

The source for the class is in the following location:

`<install_dir_ca>/library/coverity-test-separation-sources.jar`.

The basic usage flow to achieve test separation consists of calls to `startTest()` and `endTest()` methods, which you can augment with calls to set various parameters of a test (including name, source, and status).

- A call to `startTest()` resets coverage counters.
- A call to `endTest()` causes the coverage to be written out to the data file corresponding to the test.

Note

Coverage collection is not thread-specific and all coverage data collected between the pairs of `startTest()` and `endTest()` is attributed to the test, regardless of the thread that produces it.

When the JVM is initialized, all coverage is directed to a special "global" test, whose name is derived from the setting of `COVERITY_TEST_NAME` and `COVERITY_SUITE_NAME` environment variables. When a test is signaled using the `startTest()` and `endTest()` methods, the coverage is attributed to that test instead. Any coverage collected between the end of one test and the start of the next test (or after the last `endTest()` call) is attributed to the "global" test. The name that is associated with a test is the name assigned to the test at the time `endTest()` is called, either by calling `setTestName()` method prior to `endTest()`, or by specifying a name to `startTest()`. The `startTest()` and `endTest()` methods support re-entrancy based on the test and suite name. When a test is started, all other attempts to start or end a test with a different test and suite name are ignored. This allows, for example, for an outer test to invoke an inner test and have all coverage get attributed to the outer test. The `startTest()` and `endTest()` methods also correctly handle recursion by matching the `startTest()` and `endTest()` pairs for the same test.

When the JVM is shut down, the coverage associated with the "global" test is flushed along with the additional metadata collected during coverage instrumentation. This shutdown procedure *must* complete. The shut down is intercepted through a JVM shut down hook (see `{@link Runtime#addShutdownHook(Thread)}`) registered by the Test Advisor runtime. In case the JVM is expected to be shutdown forcefully via an OS signal or by calling `{@link Runtime#halt(int)}`, the application must explicitly call `aboutToTerminate()`.

The following example shows the signalling of the start and end of a test:

```
startTest("integration", "mytest");
...
endTest("integration", "mytest");
```

The following example shows setting parameters for a test during test:

```
startTest();
...
setTestName("mytest");
setTestSource(new File("/path/to/my/test"));
setTestStatus(STATUS_PASS);
endTest();
```

Note

When Java test execution is using an "isolating" class loader that does not attempt to load classes from JVM's boot class path, please make sure that Coverity runtime support classes (`<install_dir_ca>/jars/capture-rt.jar`) are available on that class loader's class path.

When a class loaded by an isolating class loader is loaded and the class loader is unable to locate the Coverity runtime support classes, the class will not be instrumented for coverage collection and the coverage for any methods in that class will not be recorded.

1.6.3. Setting test boundaries for C#

1.6.3.1. Setting per-process test boundaries for C# with environment variables

The COVERITY_SUITE_NAME and COVERITY_TEST_NAME environment variables can be used to group all tests executed by a C# executable into a single test (corresponding to COVERITY_TEST_NAME set prior to the executable invocation) and then assign them to the appropriate test suite (corresponding to COVERITY_SUITE_NAME set prior to the executable invocation). However, it is possible to further subdivide tests that are executed by a single executable.

1.6.3.1.1. Test boundary properties file

Test boundary detection for C# is configured through a properties XML file specified using the `--cs-test-config` option. For example:

```
cov-build --test-capture --dir idir --cs-coverage opencover --cs-test-config
 myconfig.properties.xml <test_command>
```

Note

The `--cs-test-config` should be used with the command that is responsible for executing the tests.

The following properties can be set in this properties file:

`coverity.test.class.regex`
: This property configures a regular expression for matching against test class names. When this property is set, only those classes whose name matches the specified regular expression are considered to be potential test classes. When this property is not set, class name does not play role in determining which classes are considered to be potential test classes.

Example:

```
<properties>
  <property>
    <name>coverity.test.class.regex</name>
    <value>^.*TestClass.*$</value>
  </property>
</properties>
```

`coverity.test.class.attribute.regex`
: This property configures a regular expression for matching against test class attributes. When this property is set, only those classes that have an attribute whose name matches the specified regular expression are considered to be potential test classes. When this property is not set, attributes do not play role in determining which classes are considered to be potential test classes.

Example:

```
<properties>
  <property>
    <name>coverity.test.class.attribute.regex</name>
    <value>^.*TestClassAttribute.*$</value>
  </property>
</properties>
```

Note

When both `coverity.test.class.regex` and `coverity.test.class.attribute.regex` are set, a class must meet both criteria to be considered a test class. When neither property is set, no class is considered to be a test class and further test boundary detection is not performed.

`coverity.test.method.regex`

This property configures a regular expression for matching against test method names in classes that are considered to be test classes. When this property is set, the methods whose name matches the specified regular expression are considered to be test entry points. When this property is not set, method name does not play a role in determining which methods are considered to be potential test entry points.

Example:

```
<properties>
  <property>
    <name>coverity.test.method.regex</name>
    <value>^.*TestMethod.*$</value>
  </property>
</properties>
```

`coverity.test.method.attribute.regex`

This property configures a regular expression for matching against test method attributes in classes that are considered to be test classes. When this property is set, those methods that have an attribute whose name matches the specified regular expression are considered to be test entry points. When this property is not set, method attributes do not play a role in determining which methods are considered to be potential test entry points.

Example:

```
<properties>
  <property>
    <name>coverity.test.method.attribute.regex</name>
    <value>^.*TestMethodAttribute.*$</value>
  </property>
</properties>
```

Note

When both coverity.test.method.regex and `coverity.test.method.attribute.regex` are set, a method must meet both criteria to be considered a test entry point. When neither property is set, no method is considered to be a test entry point and test boundary detection is not performed.

Test boundary detection is performed only at the top level and not within other tests. In other words, when the test execution framework calls `testA` that subsequently calls `testB` and both tests would be considered test entry points by the above rules, only `testA` is recorded as the boundary for the entire test.

When test boundary detection is performed using the above annotations, Test Advisor determines the status of a test by checking whether a test has thrown an exception. A test method that successfully completes without throwing an exception is considered to be a passing test. A test method that throws an exception is considered to be a failing test. For test execution frameworks that support the use of annotations to specify an expected exception (such as MSTest), the following properties can be used to inform Test Advisor which annotation field is used to specify the expected exception class.

coverity.test.expectedexception.attribute.regex

This property configures a regular expression for matching against an attribute name that will specify the expected exception.

Example:

```
<properties>
  <property>
    <name>coverity.test.expectedexception.attribute.regex</name>
    <value>^Microsoft\.VisualStudio\.TestTools\.UnitTesting
\.ExpectedExceptionAttribute$</value>
  </property>
</properties>
```

coverity.test.expectedexceptionclass.property.regex

This property configures a regular expression for matching against a property name that will specify the expected exception class.

Example:

```
<properties>
  <property>
    <name>coverity.test.expectedexceptionclass.property.regex</name>
    <value>^ExpectedException$</value>
  </property>
</properties>
```

coverity.test.expectedexceptionname.property.regex

This property configures a regular expression for matching against a property name that will specify the name of the expected exception.

Example:

```
<properties>
  <property>
    <name>coverity.test.expectedexceptionname.property.regex</name>
    <value>^ExpectedExceptionName$</value>
  </property>
</properties>
```

1.6.3.1.2. Sample property files

Test Advisor provides three configuration files that specify test boundary configuration for the popular test execution frameworks: `MSTest`, `NUnit` and `XUnit`.

These properties files are located in the following directory:

```
<install_dir_ca>/config/
```

For convenience, the `--cs-test <config-name>` command line option `to cov-build` can be used to point to these files instead of specifying the full path to the config/ directory. The following values are allowed for `<config-name>`:

- `mstest` - Uses `method.attribute.regex` to identify MSTest test methods based on the attribute `Microsoft.VisualStudio.TestTools.UnitTesting.TestMethodAttribute`.
- `nunit` - Uses `method.attribute.regex` to identify NUnit test methods based on the attributes `NUnit.Framework.TestAttribute` or `NUnit.Framework.TestCaseAttribute`.
- `xunit` - Uses the `method.attribute.regex` to identify XUnit test methods based on the attribute `Xunit.FactAttribute`.

Additionally each of these properties files specify the appropriate expected exception values for the framework.

1.6.3.1.3. Special note on test boundary detection

NUnit has the ability to write a test such as the following:

```
[TestCase(1, 2, Result = 3)]
[TestCase(2, 3, Result = 5)]
[TestCase(-1, 2, Result = 1)]
public int TestAdd(int arg0, int arg1)
{
    return (arg0 + arg1);
}
```

In this case NUnit actually runs three tests with the various arguments. Note that in this case Test Advisor will only show this as a single test.

1.6.4. Remote Test Separation

A feature of Test Advisor called Remote Test Separation provides a way to configure test boundaries for long-running processes such as daemons. Remote Test Separation has the following advantages:

- Tests can be started and stopped without having to stop the long-running process (unlike using environment variables to configure test boundaries).
- No manual code changes need to be made (unlike using the test separation API).

1.6.4.1. Remote Test Separation Requirements

Linux:

- Only Linux 32/64 bit platforms are supported.
- Only the gcc/g++ compilers are supported.
- The pthreads library must be available on the target platform, and the build compiler must support compilation with pthreads.

Windows:
No additional limitations for Windows.

See *Coverity 2020.12 Installation and Deployment Guide* for general Test Advisor product requirements.

1.6.4.2. Limitations of Remote Test Separation

Currently the following limitations are placed on the use of Remote Test Separation:

- At any given time, all coverage being collected under a single emit server must be assigned to the same test (suitename/testname). That is, there is no way for a single emit server to collect coverage for different tests simultaneously, even when the coverage comes from different processes.
- Neither the test boundary environment variables nor the API for configuring test boundaries should be used when Remote Test Separation is being used. Trying to combine these techniques will result in indeterminate behavior.
- Function Coverage Instrumentation must be used to collect coverage. Additionally, the network flush workflow must be used. See Function Coverage Instrumentation for details about Function Coverage Instrumentation.

1.6.4.3. Using Remote Test Separation

By default, when using Function Coverage Instrumentation on a supported platform, support for Remote Test Separation is automatically built into the binaries generated by the `cov-build` command.

The `cov-emit-server-control` command is used to start and stop tests using Remote Test Separation. This command contacts the emit server instance which is collecting coverage for the network flush workflow, and indicates that a new test is to be started, or a running test is to be stopped. Coverage collected from the running process will be assigned to the appropriate test, as specified.

As an example, suppose an emit server is running on IP address 1.2.3.4 on port 7890, collecting coverage for a process using Remote Test Separation. The following commands can be used to tell the emit server that a test with suitename "mysuite" and testname "test1" is to be started, and that further coverage from the test process should be assigned to this test:

```
> cov-emit-server-control --interface 1.2.3.4 --port 7890 --start-suite mysuite
> cov-emit-server-control --interface 1.2.3.4 --port 7890 --start-test test1
```

Coverage from the test process will now be assigned to mysuite/test1. When the user is satisfied that test1 is complete, and that test2 should be started, the following commands should be used:

```
> cov-emit-server-control --interface 1.2.3.4 --port 7890 --end-test test1
> cov-emit-server-control --interface 1.2.3.4 --port 7890 --start-test test2
```

At this point, further coverage from the test process will be assigned to mysuite/test2.

Note

Note that the user is responsible for determining when tests should be started and stopped. Incorrectly doing so may lead to test coverage being attributed to the wrong test.

1.6.4.4. Remote Test Separation Example

To demonstrate the use of Remote Test Separation, sample code for a simple application is provided at `<install_dir_ca>/doc/examples/remote-test-separation/simple-server.c`. This program simply accepts input on stdin and generates output based on the presence of the digits '1', '2', and '3' in the input. The digit '0' terminates the program. This program mimics the abstract behavior of a long-running server. We will use this program with Remote Test Separation.

This server code runs on any Linux platform. First, make sure your platform has a working gcc and supports the other Remote Test Separation requirements; see Section 1.6.4.2, “Limitations of Remote Test Separation” for a list of these requirements. To build the server, create and change to a suitable work directory, copy the file there, and issue the command:

```
> gcc -o simple-server simple-server.c
```

You can start the server and try various inputs to it with:

```
> ./simple-server
```

Entering '0' will cause the program to exit.

Now we will build the program with Remote Test Separation support. This assumes Coverity Analysis has already been installed and configured for gcc; see the *Coverity Analysis 2020.12 User and Administrator Guide* for details. Issue the commands:

```
> rm simple-server
> cov-build --dir idir --c-coverage function gcc -o simple-server simple-server.c
```

The output should contain the line:

```
4 function-coverage sites (100%) have been instrumented
```

This indicates that Function Coverage Instrumentation was successful in instrumenting the code. The resulting executable now contains support for Remote Test Separation.

We will now start an instance of `cov-emit-server`:

```
> cov-manage-emit --dir idir start-server --port 0
```

The `--port 0` option indicates that an unused port will be selected. The port which is used can be determined by:

```
> cov-manage-emit --dir idir query-server
```

Now run the simple server, sending the collected coverage data to the running emit server; replace {PORT_NUMBER} with the port used by the emit server:

```
> export PATH="<install_dir>/lib:$PATH"
> export COVERITY_COVERAGE_FILE=
> export COVERITY_EMIT_SERVER_ADDR=127.0.0.1:{PORT_NUMBER}
> export COVERITY_TEST_CAPTURE_RUN_TIMESTAMP='date +%s'
> ./simple-server
```

We will now run two tests. The first test will verify the behavior of the simple server for the input "1", and the second test will verify the behavior for the input "2".

Open another terminal on the same machine, and issue this command to start a test named "test1"; again, replace {PORT_NUMBER} with the port used by the emit server:

```
> cov-emit-server-control --interface 127.0.0.1 --port {PORT_NUMBER} --start-test
 test1
```

Go back to the terminal running the simple server, and enter "1". This represents the input stimulus for the first test. Now return to the terminal used for controlling the emit server, and issue the commands:

```
> cov-emit-server-control --interface 127.0.0.1 --port {PORT_NUMBER} --stop-test test1
> cov-emit-server-control --interface 127.0.0.1 --port {PORT_NUMBER} --start-test
 test2
```

Go back to the terminal running the simple server and enter "2". Return to the control terminal and issue the command:

```
> cov-emit-server-control --interface 127.0.0.1 --port {PORT_NUMBER} --stop-test test2
```

Go back to the terminal running the simple server and enter "0". This causes the application to exit. Stop the running emit server by using the command:

```
> cov-manage-emit --dir idir stop-server
```

Now the emit directory will contain the coverage gathered during the tests. You can inspect this coverage using the `cov-manage-emit` command:

```
> cov-manage-emit --dir idir --tu 1 print-source --coverage
> cov-manage-emit --dir idir --tu 1 print-source --coverage --testname test1
> cov-manage-emit --dir idir --tu 1 print-source --coverage --testname test2
```

The first of these commands shows the coverage collected across all tests. The second and third commands shows coverage specific to a particular test. Coverage is displayed as the second character of each line; a '-' indicates an uncovered function, while a '+' indicates a covered function. Function `foo1()` should be covered only by test1, while `foo2()` should be covered only by test2.

If you wish to repeat these tests, keep in mind that coverage is accumulated in the emit directory. You should issue this command to remove the coverage which has already been collected and start fresh:

```
> cov-manage-emit --dir idir delete-coverage
```

As mentioned, this simple server is intended to mimic the abstract behavior of a long-running server process. The same general procedure can be used to configure test boundaries for real applications and tests:

- Build the application with support for Remote Test Separation
- Start an emit server
- Start the application, directing coverage collection to the emit server

- For each test:
  1. Use `cov-emit-server-control` to start the test
  2. Execute the test by sending test stimulus to the application
  3. Use `cov-emit-server-control` to stop the test
- Stop the application
- Stop the emit server

Chapter 1.7. Additional coverage collection methods

Table of Contents

In previous releases, test runs were stored only by suitename and testname. So, if you were to run the same test multiple times, the coverage would only accumulate for that one test. In this release, if you run the same test multiple times, Test Advisor will store tests as separate entities in the intermediate directory.

It is important to note, that these changes **will not affect your current Test Advisor workflow**. There is no need to updated your scripts or Test Advisor processes.

This enhancement was developed primarily to support the following additional test coverage features:

- Network flushing - For Gcov and Function Coverage Instrumentation only, this feature allows you to flush coverage over the network, instead of to the file system, which can drastically improve performance.

- Coverage merging - Allows coverage contained in an intermediate directory library to be merged together into a single target intermediate directory. This is useful for combining coverage from different builds.

  To support these features, Test Advisor introduces the concepts of test capture runs, build IDs, and intermediate directory libraries. The new features, including workflow examples, the associated concepts, and additions to command line options are described in Chapter 1.7, *Additional coverage collection methods*.

  For a concise list of new options to the `cov-build` and `cov-manage-emit` commands, see the "Commands related to Test Advisor" section of the *Coverity 2020.12 Release Notes* .

In order to utilize these new features, you should first read the important concepts section.

1.7.1. Important concepts

Before you use the network flush and/or coverage merging features, it is important that you understand the following feature related concepts:

- Test capture runs
- Build IDs
- Intermediate directory libraries

Note that the items described in this section are not necessarily exclusive to the network flush or coverage merging features, however they are integral parts of the respective feature's operation.

1.7.1.1. Test capture runs

A test capture run is a run of your tests under `cov-build`. Test capture runs provide a way for you to uniquely tag and identify a run of your tests. A test capture run is uniquely identified by:

- a build id
- a timestamp of when it was run
- a user-specified tag

Both the build ID and the timestamp are automatically generated when you run `cov-build`. The test capture run tag is user-specified and is given to `cov-build` at run time. For example:

```
cov-build --test-capture --emit-server localhost --c-coverage gcov --build-id-file my-
build-id.txt      \
--test-capture-run-tag "build-version1" make test
```

In the example above, the test capture run is tagged with `"build-version1"`. This tag is intended to make it easier to recognize test runs within the intermediate directory, which is important for coverage merging.

Test capture runs can be displayed and manipulated using `cov-manage-emit`. For example, to list test capture runs:

```
cov-manage-emit --dir idir list-test-capture-runs
Test Capture Run 1:
  Build ID:   default-2db80a6001c08bb6f68abda0af411d45
  Timestamp:  2014-05-31T10:57:46-06:00
  Tag:        build-version1
  Successful: unknown
```

The most common ways in which you will interact with a test capture run are as follows:

- Specifying a test capture run tag to `cov-build` (as shown in the `cov-build` command above).
- Updating the status of a test capture run. This can be done with either `cov-build` or `cov-manage-emit`.

  For `cov-build`, the following command example specifies that the file `tcr-status-is-here.txt` will contain the proper test capture run status to use at the end of the run.

  ```
  cov-build --test-capture --emit-server localhost --c-coverage gcov     \
    --build-id-file my-build-id.txt  --test-capture-run-tag mytag   \
    --test-capture-run-status-file tcr-status-is-here.txt make test
  ```

  For `cov-manage-emit`, the following command example updates the success status of a test capture run:

  ```
  cov-manage-emit --dir idir update-test-capture-run --test-capture-run-id 1    \
  ```

```
  --success true
```

Test capture run information (coverage) is merged when it is sent to Coverity Connect. This means that you will not see any test capture run related information in Coverity Connect. Coverage information will look the same as in previous releases.

For more information on the arguments related to test capture runs see `cov-build` and `cov-manage-emit`.

1.7.1.2. Build IDs

A build ID is a way to uniquely identify an intermediate directory. A build ID will generally have the following syntax:

```
<user-specified-string>-<internal md5 checksum>
```

For example:

```
linuxbuild-version1234-2db80a6001c08bb6f68abda0af411d45
```

The `<user-specified-string>` portion of the build ID comes from a `cov-build` command line argument. For example:

```
cov-build --dir idir --c-coverage gcov --build-description "linuxbuild-version1234"
 make build
```

The command above will result in a build ID similar to the example above. A build description is not required, and if it is not included, the user specified portion defaults to `default`, so in this case, it will display as follows:

```
default-2db80a6001c08bb6f68abda0af411d45
```

The build ID is an important part of using the network flush feature. It ensures that Test Advisor adds coverage to the correct intermediate directory. Both `cov-build` and `cov-manage-emit` have multiple arguments to be used with the build ID. The most important options of note are as follows:

For `cov-build`:

- `--build-id <build-id>`

  This allows you to specify the build ID on the command line as a string.

- `--build-id-file <filename>`

  This allows you to specify the name of a file that contains the build ID, which is often easier than typing out the build ID.

- `--build-id-output <filename>`

  This allows you to have `cov-build` output the build ID for a build to the specified file. This will save you from having to query the build ID later.

For `cov-manage-emit`

- `query-build-id`

  Displays the build ID for the specified intermediate directory.

For more information on the arguments related to build IDs, see `cov-build` and `cov-manage-emit`.

1.7.1.3. Intermediate directory libraries

An intermediate directory library is a collection of intermediate directories and is created and managed using `cov-manage-emit`. For example:

To add an intermediate directory to an intermediate directory library:

```
cov-manage-emit --idir-library /path/to/my/idir-lib add-to-library --dir addme
```

To remove an intermediate directory from the intermediate directory library:

```
cov-manage-emit --idir-library /path/to/my/idir-lib remove-from-library --dir removeme
```

The main usage of an intermediate directory library is (currently) for use with the coverage merging feature. One advantage of using an intermediate directory library is that you can run a single emit server on the library and that can collect coverage for multiple and different builds.

1.7.2. Network flushing (Gcov only)

By default, the `cov-build` command flushes coverage over the network instead of to the file system. Prior to version 7.6.0, the default method for adding Gcov coverage relied on writing `.gcda` files to the disk, but there are cases where doing this is not optimal, for example:

- When you have a large number of tests and object files. Adding all of these files at the end of the run can use a large amount of system resources and take a long time.
- If you want to gather coverage from a remote machine, you need to copy the coverage files from the remote machine to the build machine, and manually merge them in with `--merge-raw-coverage`.

Flushing coverage over the network handles such cases better.

This section describes two ways to perform network flushing:

- On a local machine
- On a remote machine

1.7.2.1. Flushing coverage over the network: Local machine example

Recommended workflow:

1. Run the build.

```
cov-build --dir idir --c-coverage gcov make build
```

2. Execute the tests.

   ```
   cov-build --test-capture --dir idir --c-coverage gcov make test
   ```

In the past, the workflow for using the network to flush coverage on a single machine (over the loopback) involved more steps. You can continue to use that process (which required you to start and stop the emit server manually and to use `cov-build` to run your tests and send the coverage to the emit server), but it is no longer necessary.

Note that you can also fall back to the file system mode by specifying the `--no-network-coverage` option to `cov-build`.

1.7.2.2. Flushing coverage over the network: Remote machine example

The example in the previous section showed using the network flush method on a single machine over the loopback. This method can improve performance for some builds, but is only part of the functionality that network flushing provides.

Assume that you build on `machine1`, but run your tests on `machine2`. Previously, you would have used the `cov-build` `--leave-raw-coverage`/`--merge-raw-coverage` options and then copy a large number of files between the two machines. Using this new network flush method can simplify this. For example:

1. Build on `machine1`.

   ```
   cov-build --dir idir --c-coverage gcov make build
   ```

2. Start the emit server on the intermediate directory for `machine1`.

   ```
   cov-manage-emit --dir idir start-server
   ```

   In order to run the binaries on `machine2` you will need the following:

   - The build ID for this intermediate directory

     The most efficient way to do this is to generate a file containing the build ID and copy it over. For example, on `machine1`:

     ```
     cov-manage-emit --dir idir query-build-id > my-build-id.txt
     ```

   - The instrumented binaries.

     The binaries can be copied over from `machine1`.

3. Run your tests, but send coverage to `machine1`. From `machine2`, run the following:

   ```
   cov-build --test-capture --emit-server machine1 --c-coverage gcov     \
     --build-id-file my-build-id.txt make test
   ```

4. Stop the server on `machine1`.

```
cov-manage-emit --dir idir stop-server
```

At this point, the coverage data is complete in the intermediate directory (`idir` on `machine1`) and you can continue with the typical Test Advisor workflow of gathering SCM data, running analysis, and so forth.

1.7.2.3. Using the intermediate directory library

The above examples demonstrate the emit server operating on a single intermediate directory. It has an additional mode of operation where it can operate on an intermediate directory library. The work flow is similar, with the following main differences:

- You need to add your intermediate directory to the intermediate directory library.
- You need to start the server differently.

For example:

1. Run the build.

   ```
   cov-build --dir idir --c-coverage gcov make build
   ```

2. Get the build ID for the intermediate directory.

   ```
   cov-manage-emit --dir idir query-build-id > my-build-id.txt
   ```

3. Start the emit server on an intermediate directory library.

   ```
   cov-manage-emit --idir-library idir-lib start-server
   ```

4. Add the intermediate directory to the intermediate directory library.

   ```
   cov-manage-emit --idir-library idir-lib add-to-library --dir idir
   ```

5. Execute the tests.

   ```
   cov-build --test-capture --emit-server localhost --c-coverage gcov     \
   --build-id-file my-build-id.txt make test
   ```

6. Stop the emit server.

   ```
   cov-manage-emit --idir-library idir-lib stop-server
   ```

For more information about the use of an intermediate directory library see coverage merging.

1.7.2.4. Error recovery

Because a given network is not always 100% reliable, you might encounter some errors when trying to connect to the emit server, or sending coverage data to the emit server. `cov-build` will attempt to gracefully handle these errors. The default behavior when an error occurs is as follows:

1. Log a message to standard error

2. Sleep

3. Retry

The amount that Test Advisor sleeps before retrying changes with the number of retries. The sleep schedule is:

```
sleep  1s
sleep  5s
sleep 30s
sleep 60s
```

This means that the first retry will sleep for 1 second, then for 5 seconds, then for 30 seconds, and then every time thereafter for 60 seconds. To prevent the retry and sleep mechanism from continuously sleeping and retrying, the maximum wait time is 300 seconds (5 minutes).

In the event that Test Advisor exhausts the retries, the process will exit with a default code of 19.

1.7.2.4.1. User control for error recovery

As mentioned above, there are reasonable defaults for the error recovery, but they can be customized. The options are all controlled through command line arguments to `cov-build`:

--telemetry-network-error-log `<filename>`
: By default Test Advisor logs errors to standard error, however this can interfere with some test processes. If this argument is specified, errors will be logged to `<filename>` instead of standard error.

--telemetry-network-error-max-wait `<# of seconds>`
: By default, the maximum wait time is 500s. This option can be used to customize the maximum wait time. For example if you want to fail fast, you could specify a maximum wait time of 0s.

--telemetry-network-error-code `<exit code>`
: By default, if Test Advisor cannot recover from an error, the process will exit with a default return code of 19. This option can be used to customize that value.

1.7.2.5. Current limitations with network flush

Please note the following important limitations for network flush:

- Build and test must be separate when using the emit server on a remote machine.

  When using network flush on a remote machine, your build and test must be separate. That is, if you run `make test`, the test should not compile anything because the intermediate directory on which the emit server is running will not know about source files that are compiled on the remote machine, so it will not be possible to add coverage.

  A locally run emit server does not have this limitation.

- Currently supports IPv4 only.

The emit server does not support IPv6 at this time.

1.7.3. Merging coverage from multiple intermediate directories

Coverage contained in an intermediate directory library can be merged together into a single target intermediate directory. This is useful for combining coverage from different builds for the purpose of Test Advisor analysis. For example, tests might be run on multiple platforms (thus multiple builds as they relate to intermediate directories), and you want to combine coverage across all platforms for Test Advisor. Coverage merging can combine coverage from different sources with small changes between them. This enables you to combine coverage from different revisions of the same codebase. Coverage merging can also be used to copy coverage from a single intermediate directory to a target intermediate directory. A coverage selection option can be used to specify what coverage should be copied and what should be ignored.

1.7.3.1. Merging coverage from an intermediate directory library

Merging coverage from multiple intermediate directories is accomplished by using the `add-coverage` sub-command to `cov-manage-emit`. For example:

```
cov-manage-emit --dir idir-to-merge-into add-coverage      \
  --from-idir-library idir-lib-to-merge-from --coverage-selection all
```

This command will merge all coverage from the intermediate directory library `idir-lib-to-merge-from` into the intermediate directory `idir-to-merge-into`.

The coverage that is selected to merge is controlled by the `--coverage-selection` argument. For more about this argument and its options, see Section 1.7.3.4, “Coverage selection ”.

1.7.3.2. Merging from an individual intermediate directory

The example above for coverage merging from an intermediate directory library shows the most common use case of selecting coverage from an intermediate directory library. However, `cov-manage-emit` can also merge coverage from a single intermediate directory. This is also done using the `cov-manage-emit add-coverage`:

```
cov-manage-emit --dir idir-to-merge-into add-coverage  \
  --from-dir idir-to-merge-from --coverage-selection all
```

Note that you can only specify one of `--from-dir` or `--from-idir-library` to `cov-manage-emit`.

1.7.3.3. How coverage is merged

Coverage is copied from the source intermediate directory library (or single intermediate directory) on a file-by-file basis. Files contained in the source intermediate directory are matched to files contained in the target intermediate directory for this purpose, with the matching done by filename comparison.

As the builds may have been performed in different directories, the "build root" (the directory in which `cov-build` was invoked to create the intermediate directory) is stripped off the beginning of filenames before the filename comparison is made. Additional directories to strip from filenames before comparison can be specified using the `--strip-path` option to `cov-manage-emit add-coverage`. Note that this is similar to the behavior of the `--strip-path` option to `cov-analyze`. For example:

```
cov-manage-emit --dir idir-to-merge-into     \
  add-coverage --from-idir-library idir-lib-to-merge-from \
  --coverage-selection all --strip-path /addtl/path/to/strip
```

When the contents of a file differ between the source and the target, the coverage from the source is remapped according to the textual differences between the source and target files. Due to the difficulty of this operation, you should expect some inaccuracies in the resulting coverage when file contents differ. However. the procedure implemented for coverage merging works well for small code differences.

Note

If you receive unexpected coverage results after merging, ensure that the test capture run status is set appropriately for the items being merged. Any test capture runs with an unset status will not be merged.

1.7.3.4. Coverage selection

By default, all coverage from the source is merged into the target intermediate directory. However, the `--coverage-selection` option can be used to restrict what coverage is merged. The syntax for the `coverage-selection` option is as follows:

```
SELECTION := SELECTION_ITEM
            | SELECTION_ITEM "," SELECTION
 SELECTION_ITEM := (FILTER+ "from")? GROUPING ("with" PATTERN+)?
 FILTER := "latest"
         | "successful"
 GROUPING := "all"
           | "each" TCRDATA
 TCRDATA := "capturetag"
           | "suitename"
           | "testname"
 PATTERN := TCRDATA "not"? "regex" REGEX
 REGEX := '"' STRING '"'
 | "'" STRING "'"
```

A `SELECTION` specifies the subset of the tests present in the source which will be merged into the target intermediate directory. A selection is formed by taking the union of a list of `SELECTION_ITEM`s.

A `SELECTION_ITEM` specifies a subset of the tests from the source. A `SELECTION_ITEM` is specified by a grouping, an optional list of filters, and an optional list of patterns. The `SELECTION_ITEM` is formed by taking the tests whose TCRDATA satisfy the given patterns, grouping the results into elements, then applying the filters each of the resulting elements. TCRDATA refers to data attributed to a test capture run.

1.7.3.4.1. Groupings

A grouping specifies a set of elements, where each element is a subset of tests. The grouping determines the granularity at which the filters are to be applied.

Valid groupings are:

`all`
: The grouping contains a single element; that element contains all tests.

`each TCRDATA`
: The grouping contains one element for each unique TCRDATA component (`capturetag`, `suitename`, or `testname`) present in the source. Each element contains all of the tests that have the associated TCRDATA.

1.7.3.4.2. Filters

A filter specifies which tests within each element of a grouping should be included in a selection item. The determination of which tests satisfy the filter is performed by examining and comparing the tests within each element of a grouping. This can be thought of as extracting a subset tests from of each element of a grouping. Valid filters are:

`latest`
: The test with the latest (largest) test capture run timestamp in each element is used. If more than one test share the same timestamp, all such tests are included in the result. Note that since typically the user's entire suite of tests is run for each test capture run, this filter will usually yield more than one test.

`successful`
: Only the tests which have their `TestCaptureRun` successful field set to true are used.

Filters are evaluated from right-to-left. That is, only those tests which satisfy the rightmost filter are evaluated by the next filter, and so on. **Note that there is a difference between the results of the filter lists "latest successful" and "successful latest"**.

1.7.3.4.3. Patterns

A pattern contains a regex that is matched against a specified item of TCRDATA (`capturetag`, `suitename`, or testname) for each test. A test satisfies a pattern if the regex match succeeds. Optionally, a pattern can include the `not` keyword, which inverts its meaning so that tests satisfy that pattern if the regex match fails. Tests that satisfy all patterns for a selection item are included for grouping.

1.7.3.4.4. Coverage selection examples

Note that in the following descriptions, the possibility for the "latest" filter to yield multiple tests is not considered.

`all`
: Use all tests.

`latest from all`
: Use only the latest test.

`all with capturetag regex "foo"`
: Use only the tests for which `capturetag` matches "foo".

`latest from all with capture tag regex "foo"`
: Use only the latest test among all those for which `capturetag` matches "foo".

`latest from each capturetag with capture tag regex "foo"`
: Use only the latest test for each unique `capturetag` that matches "foo".

`latest successful from all`
: Use only the latest test including those which were successful.

`successful latest from all`
: Use only the latest test from those which were successful.

`latest from each capturetag`
: Use only the latest test for each unique `capturetag`.

`latest successful from each capturetag`
: Use only the latest successful test for each unique `capturetag`.

`latest from each suitename`
: Use only the latest test for each unique `suitename`.

`latest from each testname`
: Use only the latest test for each unique `testname`.

`latest from each testname with capturetag regex "Linux", latest from each testname with capturetag regex "Windows"`
: Take the latest test for each unique testname from those with `capturetag` matching "Linux", together with the latest test for each unique testname from those with `capturetag` matching "Windows"

For example, consider the following set of test capture runs:

```
TestCaptureRun A: { BuildID: "1", captureTag: "foo1",  timestamp: 1234, successful:
 true }
TestCaptureRun B: { BuildID: "1", captureTag: "foo2",  timestamp: 1235, successful:
 true }
TestCaptureRun C: { BuildID: "1", captureTag: "foo3",  timestamp: 1236, successful:
 false }
TestCaptureRun D: { BuildID: "2", captureTag: "foo1",  timestamp: 9944, successful:
 true }
TestCaptureRun E: { BuildID: "2", captureTag: "foo2",  timestamp: 9945, successful:
 false }
TestCaptureRun F: { BuildID: "1", captureTag: "other", timestamp: 1239, successful:
 true }
```

The following table indicates the `TestCaptureRuns` whose tests are included for merging (marked with "Y") given the specified patterns:

Pattern	**A**	**B**	**C**	**D**	**E**	**F**
all	Y	Y	Y	Y	Y	Y

Pattern	A	B	C	D	E	F
latest from all					Y	
all with capturetag regex "foo"	Y	Y	Y	Y	Y	
latest from all with `capturetag` regex "foo"					Y	
latest from each `capturetag` with `capturetag` regex "foo"			Y	Y	Y	
latest successful from all				Y		
successful latest from all						
latest from each `capturetag`			Y	Y	Y	Y
latest successful from each `capturetag`		Y		Y		Y

1.7.3.4.5. Debugging coverage selections

When you try to define a `--coverage-selection` value that gives you the coverage that you want, there is a convenient way for `cov-manage-emit` to display this without actually having to merge the coverage. For example:

```
cov-manage-emit --dir idir-to-merge-into add-coverage    \
  --from-idir-library idir-lib-to-merge-from \
  --dry-run-list-tests --coverage-selection all
```

Note in this example that there is the additional `--dry-run-list-tests`. This command will report the tests that would have been merged, without actually merging them. The output will look similar to the following:

```
Intermediate Directory: /idir-lib-to-merge-from/dir.0
captureTag suitename testname timestamp
capturetag0 suite0 test0 2014-05-02T00:00:00+00:00
capturetag0 suite0 test1 2014-05-02T00:00:00+00:00
capturetag0 suite0 test2 2014-05-03T00:00:00+00:00
capturetag0 suite1 test0 2014-05-02T00:00:00+00:00
capturetag0 suite1 test1 2014-05-02T00:00:00+00:00

Intermediate Directory: /idir-lib-to-merge-from/dir.1
captureTag suitename testname timestamp
capturetag1 suite0 test0 2014-05-02T00:00:00+00:00
capturetag1 suite0 test1 2014-05-02T00:00:00+00:00
capturetag1 suite1 test0 2014-05-02T00:00:00+00:00
capturetag1 suite1 test1 2014-05-02T00:00:00+00:00
```

1.7.4. Function Coverage Instrumentation

In addition to coverage tools like gcov, Test Advisor now supports Function Coverage Instrumentation for C/C++ on 32-bit and 64-bit Linux and Windows systems.

Function coverage tracks which functions have been called; it does not track which lines of code are executed. Function coverage might be useful when you do not require the higher detail offered by line coverage. Instrumentation at the function level might provide faster runtimes than line-level instrumentation, and can be tried out as an alternative when throughput becomes an issue. Because Function Coverage Instrumentation is integrated into Test Advisor, management of the coverage data is straightforward and users do not require third-party tools, licenses, or documentation.

Important

To use Function Coverage Instrumentation, you must install the "Extend SDK" when installing the Coverity Analysis package.

Please observe the following restrictions for the given product:

- **Linux systems**

  Compiling with Function Coverage Instrumentation requires a build machine running Linux with an x86 or x86_64 microprocessor. Function coverage instrumentation supports C and C++ only, using the gcc/g++ compiler only. Programs instrumented for function coverage must target x86, x86_64, or ARM, and must be run on a Linux machine with an x86, x86_64, or ARM microprocessor.

  Code instrumentation on Linux requires that glibc be installed on the machine that is doing the build capture. If you are building 32-bit binaries on a 64-bit machine, you will need to ensure that the 32-bit version of glibc is installed.

- **Linux ARM systems**

  The Coverity Function Coverage run-time library must be recompiled for ARM and deployed to the target device in order to collect coverage data. See the Coverity Runtime Library Development Guide for more information.

- **Windows**

  Compiling with Function Coverage Instrumentation requires a build machine running Windows with an x86 or x86_64 microprocessor. Function coverage instrumentation supports C and C++, using the Visual C++ compiler only. Programs instrumented for function coverage must target x86, or x86_64, and must be run on a Windows machine with an x86 or x86_64 microprocessor.

1.7.4.1. Function Coverage Instrumentation workflow with Network Flush

On the server that holds the emit database for your project, start an emit server:

```
cov-manage-emit --dir idir start-server --port 15772
```

On the test machine, provide the emit server address and timestamps to the tests.

```
PATH="<install_dir>/lib:$PATH" \
  COVERITY_EMIT_SERVER_ADDR=remote-host-name:15772 \
  COVERITY_TEST_CAPTURE_RUN_TIMESTAMP='date +%s' \
  make test
```

- The option `--port <portnumber>` selects the specified port for the server. The default port number is 15772.

- When using Network Flush, the COVERITY_COVERAGE_FILE environment variable must not be set.

When tests have completed, stop the emit server on the server machine:

```
cov-manage-emit --dir idir stop-server
```

1.7.4.2. Function Coverage Instrumentation workflow with file output

Use the environment variable COVERITY_COVERAGE_FILE to provide the file name for the coverage data, and then run your tests. Once tests have completed, use `cov-manage-emit` to add the coverage to your intermediate directory.

Example for Linux:

```
PATH="<install_dir>/lib:$PATH" \
  COVERITY_COVERAGE_FILE=/tmp/coverage-data  \
  COVERITY_TEST_CAPTURE_RUN_TIMESTAMP=`date +%s` \
  make test

cov-manage-emit --dir idir add-coverage \
  --coverage-file=/tmp/coverage-data
```

Example for Windows:

```
set PATH=<install_dir>\lib;%PATH%
set COVERITY_COVERAGE_FILE=c:\temp\coverage-data
set COVERITY_TEST_CAPTURE_RUN_TIMESTAMP=1

REM run an instrumented program
.\test.exe

cov-manage-emit --dir idir add-coverage ^
  --coverage-file=%COVERITY_COVERAGE_FILE%
```

When capturing data to a file, the COVERITY_EMIT_SERVER_ADDR environment variable must not be set.

1.7.4.3. Function Coverage Instrumentation environment variables

The following environment variables can be used to configure a program built and run with Function Coverage Instrumentation.

COVERITY_TEST_CAPTURE_RUN_TIMESTAMP
: Sets the test capture run timestamp. See Test capture runs for more information. Required when not using `cov-build`.

COVERITY_EMIT_SERVER_ADDR
: Network name and port address of a server when using network flush to capture data.

COVERITY_COVERAGE_FILE
: Name of a file used by the instrumented program to write coverage data.

COVERITY_LOG_FILE

Optional log file written by the instrumented program, useful for gathering debugging and diagnostics information. If environment variable is undefined, the name of the log file varies with the command used to run the test:

- If the test is run under `cov-build`, log messages will be written out to the `cov-build` log file in the intermediate directory, named `build-log.txt`.
- When you run the `cov-build --test-capture` command, the log file is named `capture-log.txt.`

COVERITY_DISABLE_OPTION_CHECK

Setting this environment variable to a non-empty value allows instrumented programs to execute without requiring COVERITY_EMIT_SERVER_ADDR or COVERITY_COVERAGE_FILE. In this scenario, no coverage output will be produced. If the program is run with COVERITY_EMIT_SERVER_ADDR or COVERITY_COVERAGE_FILE set, COVERITY_TEST_CAPTURE_RUN_TIMESTAMP does not need to be set, the timestamp will default to the current system time, and normal coverage output will be produced.

Either COVERITY_EMIT_SERVER_ADDR or COVERITY_COVERAGE_FILE must be provided when running without `cov-build`. It is an error to define both.

Test boundary environment variables are supported outside of `cov-build` by Function Coverage Instrumentation to provide test separation for both data files and network flushing.

1.7.4.4. Function Coverage Instrumentation configuration file

An alternative to configuring the instrumented program with environment variables is to do so with a configuration file. On Linux, instrumented programs will look for this file at `/tmp/synopsys-ci-runtime.conf` and `/etc/synopsys-ci-runtime.conf`; on Windows, they will look for it at `C:\synopsys-ci-runtime.conf` and `C:\Temp\synopsys-ci-runtime.conf`. The format of the file is a sequence of `KEY=VALUE` lines. Any of the environment variables used by the function coverage instrumentation may be used as a key; whatever follows the '=' will be treated as the associated value. Extraneous whitespace is not permitted, and no characters should be escaped in any way. An example of what such a file might look like is:

```
COVERITY_EMIT_SERVER_ADDR=localhost:1234
COVERITY_LOG_FILE=/tmp/function-coverage-log
```

An instrumented program with an empty environment would, if it finds such a file, behave as if the environment variable `COVERITY_EMIT_SERVER_ADDR` was set to the value `localhost:1234` and the environment variable `COVERITY_LOG_FILE` was set to the value `/tmp/function-coverage-log`. If a value is specified in both this configuration file and the environment, the instrumented program uses the value in the environment. Thus, if an instrumented program with the environment `COVERITY_LOG_FILE=/dev/null` found the above configuration file, it would proceed with `COVERITY_EMIT_SERVER_ADDR` having the value `localhost:1234`, and `COVERITY_LOG_FILE` having the value `/dev/null`.

On Linux, this file is expected to be encoded with ASCII or UTF-8 without a byte order mark. On Windows, this file may be encoded with any of ASCII, UTF-16 (little-endian), UTF-16 (big-endian), or

UTF-8, with the requirement that a byte order mark must be present if one of the Unicode encodings is used. As a rule, the commonly-available tools on a platform (e.g. echo, vi, emacs on Linux; Notepad on Windows) will create files with the appropriate encoding and, if necessary, byte order mark.

1.7.4.5. Building with Function Coverage Instrumentation

If you are performing a build on a single machine, you will have a `cov-build` command which looks something like:

```
cov-build <cov-build args> $BUILD_COMMAND
```

Function Coverage Instrumentation can be enabled in such a build by changing this to:

```
cov-build <cov-build args> --c-coverage function $BUILD_COMMAND
```

If you are performing a distributed build, you will be using `cov-translate` directly instead of `cov-build`. You must first initialize any intermediate directories you will be using:

```
cov-build -c path/to/config.xml --dir path/to/idir --initialize
cov-manage-emit --dir path/to/idir set-build-id --build-id "example-build-id"
```

Then, given a `cov-translate` invocation like:

```
cov-translate <cov-translate args> $COMPILER_INVOCATION
```

Function Coverage Instrumentation can be enabled as follows:

```
cov-translate <cov-translate args> \
   --c-coverage function --c-coverage-log-file /path/to/log-file \
   --build-id "example-build-id" $COMPILER_INVOCATION
```

Further information about our support for distributed builds can be found in the Coverity Analysis 2020.12 User and Administrator Guide, section 3.3.4.3, "Multiple builds on multiple machines".

1.7.4.6. Controlling which files are instrumented for function coverage

By default, "`cov-build --c-coverage function`" will instrument all source files for function coverage, including system header files. There are several options available for instrumenting only a subset of those files.

```
  --include-instrumentation-path <path-or-file>, -ip <path-or-file>
  --exclude-instrumentation-path <path-or-file>, -ep <path-or-file>
  --default-instrumentation-mode (include|exclude), -im (include|exclude)
```

Examples:

1. Instrument only your source repository (and exclude system header files):

   ```
   cov-build ... --include-instrumentation-path /path/to/repo ... make
   ```

   ```
   cov-build ... -ip . -ip /other/repo ... make
   ```

2. Instrument only a few directories or files:

```
cov-build ... -ip dir1 -ip dir2/file.c ... make
```

3. Instrument everything except specific files or paths:

   ```
   cov-build ... --exclude-instrumentation-path ./performance/sensitive ... make
   ```

   ```
   cov-build ... -ep ./path/to/badfile.cpp ... make
   ```

4. A combination that excludes both system header files and certain files:

   ```
   cov-build ... -ip . -ep ./dontcare.cpp -ep ./performance/sensitive ... make
   ```

Some things to remember:

- The order of the command line arguments doesn't matter. When `-ep` and `-ip` overlap, as in #4 above, `-ep` always overrides `-ip`, no matter which one occurs first on the command line.
- The default behavior is to instrument all source files, including system headers files. However, if `--include-instrumentation-path` is used, then the default changes to exclude all source files from instrumentation unless explicitly included.
- `--default-instrumentation-mode` is optional, and can be used to confirm that source files not referenced by `-ip` and `-ep` are instrumented or not. An error is given when this argument conflicts with the other arguments. For example:

  ```
  cov-build ... --default-instrumentation-mode exclude -ep badfile.cpp ... make
  ```

  This gives an error because no files will be instrumented.

Chapter 1.8. Managing test results in Coverity Connect and Coverity Policy Manager

Table of Contents

After your source code and tests are captured, analyzed, and committed to the Coverity Connect database, the associated test coverage information and policy violations discovered through the analysis are displayed in the Coverity Connect UI. Additionally, metric data can be collected and displayed in Coverity Policy Manager charts.

This chapter describes Coverity Connect and Coverity Policy Manager functionality specifically related to Test Advisor. It assumes that you have a working knowledge of Coverity Platform components. For general Coverity Connect information, such as non-Test Advisor filter definitions, advanced GUI usage and navigation, how to triage defects, and so on, see the *Coverity Platform 2020.12 User and Administrator Guide*.

Coverity Connect provides a number of tools to view and/or manage the following:

- Test policy violations
- A list of the tests that exist in a given project
- Test coverage data
- SCM change data
- Coverage exclusions
- Change impact
- Policy violations metrics (in Coverity Policy Manager)

These tools help you to make important decisions about effectiveness of your developer test data and what action to take based on the results.

1.8.1. Examining and managing test policy violation data in Coverity Connect

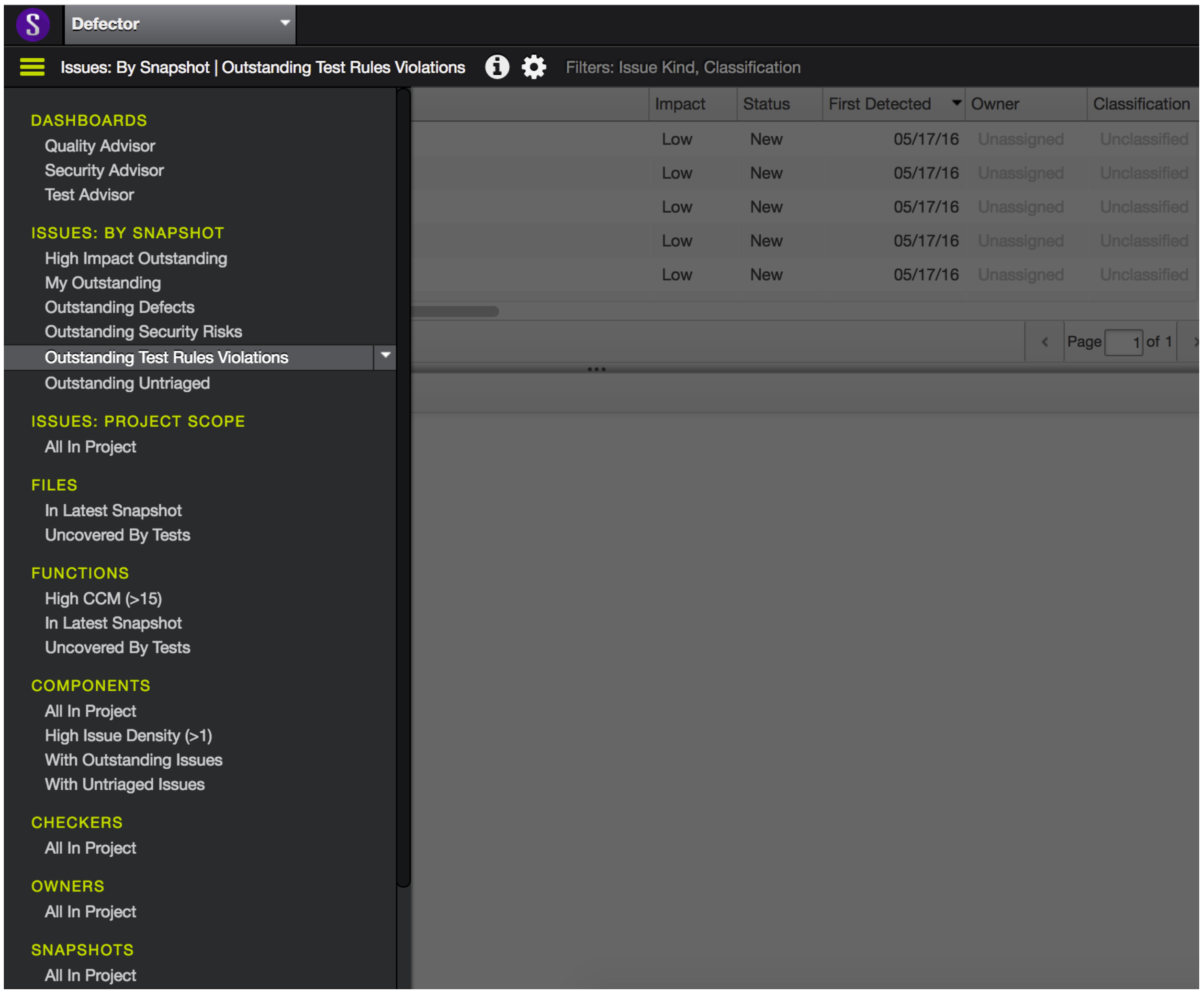
Policy violations are identified by the analysis through rule and filter definitions in the specified test policy file (see Chapter 1.5, *Creating Test Advisor policies*) and then ultimately displayed in Coverity Connect as issues.

In Coverity Connect, you can view and manage test policy violations, and you can view issue-related data (file, function, or test data) that is specific to Test Advisor. The following procedure introduces you to parts of the Coverity Connect UI that pertain to Test Advisor. For details about common Coverity Connect UI elements, which Test Advisor issues share with quality, security, and other issues, see *Coverity Platform 2020.12 User and Administrator Guide*.

1. Navigate to a view that contains a test policy violation (for example, the default *Outstanding Test Rules Violations* view in the Issues: By Snapshot view type).

   a. To see the available views and view types, open the View panel (by clicking the icon with the three horizontal bars that is located near the top-left corner of the Coverity Connect window).

Figure 1.8.1. Example: View panel in Coverity Connect



Impact	Status	First Detected	Owner	Classification
Low	New	05/17/16	Unassigned	Unclassified
Low	New	05/17/16	Unassigned	Unclassified
Low	New	05/17/16	Unassigned	Unclassified
Low	New	05/17/16	Unassigned	Unclassified
Low	New	05/17/16	Unassigned	Unclassified

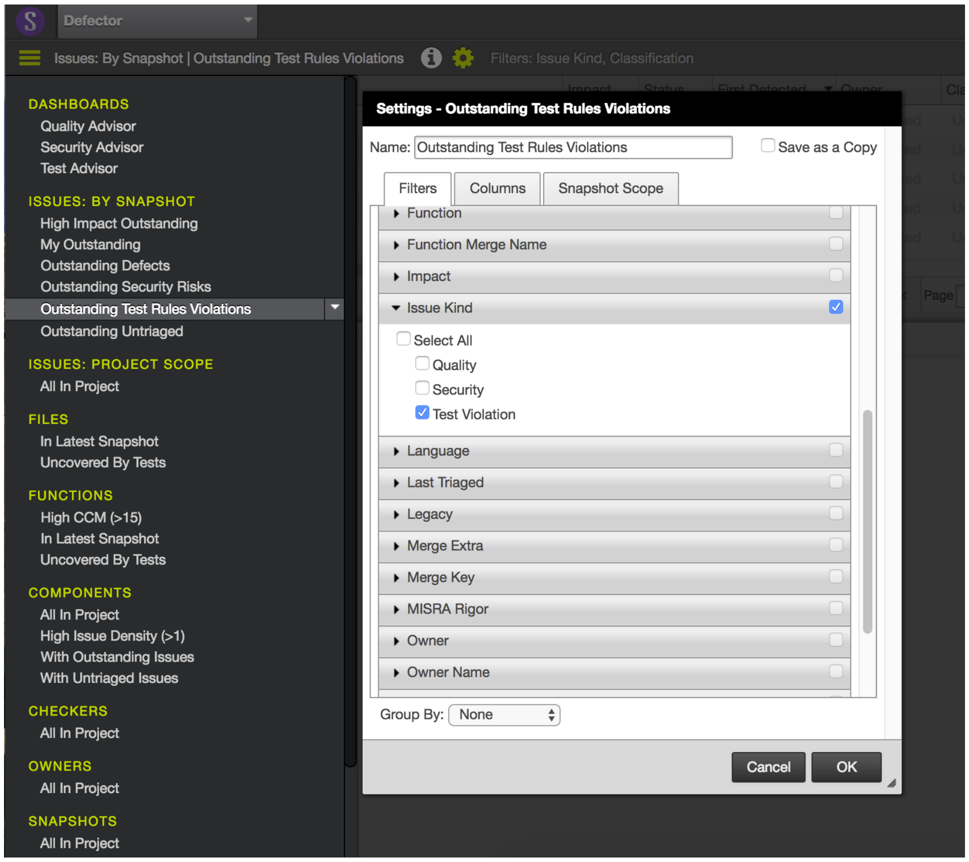
Default views and view types for Test Advisor

- DASHBOARDS (Dashboards view type) - The *Test Advisor* view is dedicated to Test Advisor charts.
- ISSUES: BY SNAPSHOT (Issues: By Snapshot view type) - The *Outstanding Test Rules Violations* view lists test violations for a given project. See Figure 1.8.3, “Example: Outstanding Test Rules Violations in Coverity Connect”.
- FILES (Files view type) - The *Uncovered By Tests* view lists the files that are not covered by tests. See Section 1.8.4, “Viewing test coverage in files and functions”.
- FUNCTIONS (Functions view type) - The *Uncovered By Tests* view lists the functions that are not covered by tests. See Section 1.8.4, “Viewing test coverage in files and functions”.
- TESTS (Tests view type) - The *All Tests* and *Currently Failing* views list the current status of your tests. See Section 1.8.5, “Viewing test status”.

Note

You can use view filters in the Settings menu (available from the Edit Settings menu for a view and from the gear icon in the header) to specify the scope of the issues and issue-related data that you want to display in the View pane for a selected view. For example, you might want to display only one kind of issue: Test Violations.

Figure 1.8.2. Example: View filter in Coverity Connect



Settings - Outstanding Test Rules Violations

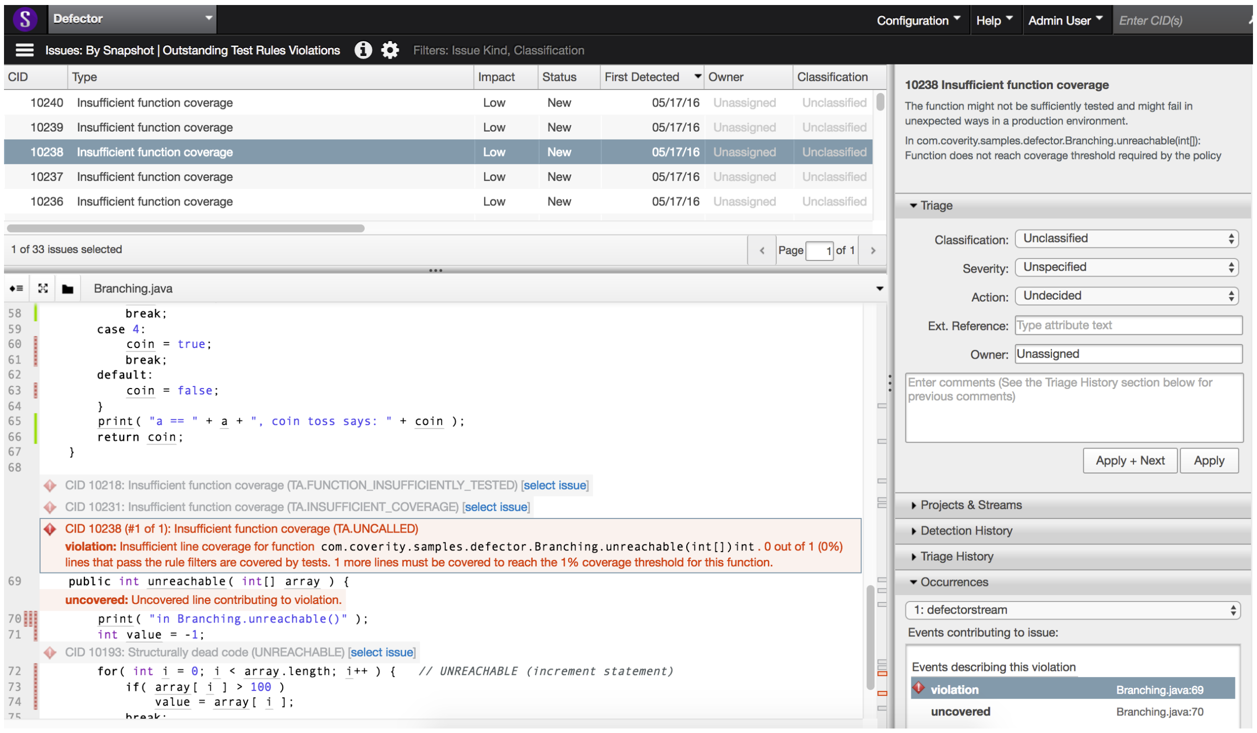
Name: Outstanding Test Rules Violations ☐ Save as a Copy

Issue Kind ☑
- ☐ Select All
- ☐ Quality
- ☐ Security
- ☑ Test Violation

Group By: None

b. Click a test policy violation in a view to display data for the violation in the Source browser and triage pane.

Figure 1.8.3. Example: Outstanding Test Rules Violations in Coverity Connect



Issues: By Snapshot | Outstanding Test Rules Violations   Filters: Issue Kind, Classification

CID	Type	Impact	Status	First Detected	Owner	Classification
10240	Insufficient function coverage	Low	New	05/17/16	Unassigned	Unclassified
10239	Insufficient function coverage	Low	New	05/17/16	Unassigned	Unclassified
10238	Insufficient function coverage	Low	New	05/17/16	Unassigned	Unclassified
10237	Insufficient function coverage	Low	New	05/17/16	Unassigned	Unclassified
10236	Insufficient function coverage	Low	New	05/17/16	Unassigned	Unclassified

1 of 33 issues selected   Page 1 of 1

Branching.java

```
            break;
        case 4:
            coin = true;
            break;
        default:
            coin = false;
        }
        print( "a == " + a + ", coin toss says: " + coin );
        return coin;
    }

```

CID 10218: Insufficient function coverage (TA.FUNCTION_INSUFFICIENTLY_TESTED) [select issue]

CID 10231: Insufficient function coverage (TA.INSUFFICIENT_COVERAGE) [select issue]

CID 10238 (#1 of 1): Insufficient function coverage (TA.UNCALLED)
violation: Insufficient line coverage for function `com.coverity.samples.defector.Branching.unreachable(int[])int`. 0 out of 1 (0%) lines that pass the rule filters are covered by tests. 1 more lines must be covered to reach the 1% coverage threshold for this function.

```
    public int unreachable( int[] array ) {
```

uncovered: Uncovered line contributing to violation.

```
        print( "in Branching.unreachable()" );
        int value = -1;
```

CID 10193: Structurally dead code (UNREACHABLE) [select issue]

```
        for( int i = 0; i < array.length; i++ ) {    // UNREACHABLE (increment statement)
            if( array[ i ] > 100 )
                value = array[ i ];
```

10238 Insufficient function coverage

The function might not be sufficiently tested and might fail in unexpected ways in a production environment.

In com.coverity.samples.defector.Branching.unreachable(int[]): Function does not reach coverage threshold required by the policy

Triage

Classification:	Unclassified
Severity:	Unspecified
Action:	Undecided
Ext. Reference:	Type attribute text
Owner:	Unassigned

Enter comments (See the Triage History section below for previous comments)

1: defectorstream

Events contributing to issue:

Events describing this violation

violation	Branching.java:69
uncovered	Branching.java:70

- Issue list (top): Coverity Connect lists test policy violations that the analysis uncovered. You can click a test policy violation to display the associated source code and Triage pane for the issue.

  To distinguish test policy violations from quality, security, and other issues, the Coverity Connect UI uses the *Test advisor issues* label in the Category column. For Test Advisor checker names, Coverity Connect uses the value of the violation_name field (for example, TA.INSUFFICIENT_COVERAGE) that is specified in the Test Advisor policy file. Test Advisor checker names appear alongside other checker names in the Coverity Connect Checker column.

- Source browser (bottom): This section of the UI shows the source code and can include descriptions of events and CIDs that pertain to Test Advisor (and possibly to Coverity Analysis components that are related to quality and/or security).

  In addition to showing the location of the violation in the source and the events that led to the violation, the Source browser can show many different types of data, including SCM and coverage data. For details, see Section 1.8.2, “Displaying SCM data, line numbers, events, and Test Advisor coverage data” and Section 1.8.3, “Displaying line-level data (test coverage, test policy rules, exclusions, change impact)”.

- Triage pane (right): Allows you to update the state of the test policy violation, including the classification, the severity, the action to take, and the owner of the violation.

  The triage fields for policy violations are identical to the other kinds of issue, with the exception of Classification, which by default contains attributes that are directly related to Test Advisor:

  - `Untested` - Reflects a determination that a section of the code analyzed by Test Advisor requires that a test be added to the code.
  - `No Test Needed` - Reflects a determination that even though the Test Advisor analysis found a section of code that is not covered by a test, you are aware of the violation and that there is an accepted reason that the code is not covered.
  - `Tested Elsewhere` - Indicates that a section of code is tested by a test outside of the set of tests that are specified in the Test Advisor analysis process.

2. Use Files and Functions views (see Default views and view types for Test Advisor) to get issue counts, metrics, and coverage details on files and functions.

   See Section 1.8.4, "Viewing test coverage in files and functions".

3. Use Test view (see Default views and view types for Test Advisor) to get test metrics.

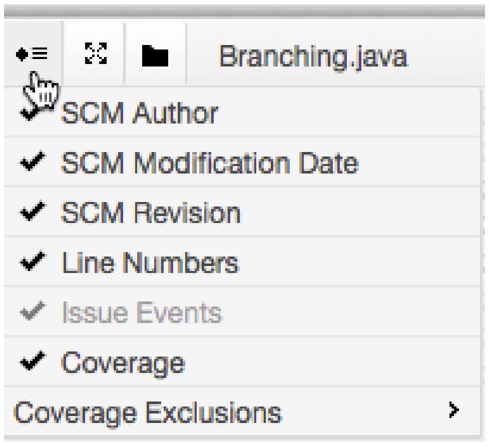
   See Section 1.8.5, "Viewing test status".

For more information about triage states, see the *Coverity Platform 2020.12 User and Administrator Guide*.

1.8.2. Displaying SCM data, line numbers, events, and Test Advisor coverage data

You can use this Source Gutter menu in Coverity Connect to display data for individual lines in the source code. This information includes SCM data, line numbers, events, Test Advisor code coverage, age, and coverage exclusion bars.

Figure 1.8.4. Example: Source Gutter menu



1.8.2.1. Displaying SCM change data

You can use tabs in the Source Gutter menu to display SCM change data for each line of code in the Source browser. Note that SCM change data is retrieved by running the `cov-import-scm` command *before the analysis and commit* process. You can display the following SCM information:

- *SCM Author* - A control used to display or hide the username of the user who checked in the code. You can use the author name as a guide to assigning an owner to the issue during the triage process.

- *SCM Modification Date* - A control used to display or hide the date that the changed code was checked into the SCM system.

- *SCM Revision* - A control used to display or hide the revision number corresponding the check-in of the changed code. Revision values depend on the SCM system. You might use the revision number to retrieve commit messages and other information from your SCM system, for example, to understand why the SCM author made a particular change to the code.

This data appears to the side of the source code in Coverity Connect. Note that SCM data can be useful for all types of analyses, including Test Advisor analyses.

1.8.2.2. Displaying line numbers

You can use the Line Numbers tab in the Source Gutter menu to display the line numbers in the Source browser.

- *Line Number* - A control used to display or hide line numbers in the source code. Line numbers can help you associate information in the source browser with particular lines of code.

  This data appears to the side of the source code in Coverity Connect.

1.8.2.3. Displaying events

You can use the Issue Events tab in the Source Gutter menu to display events found by the analysis.

- *Issue Events* - A control used to display or hide descriptions of events that were identified during the analysis. These descriptions appear in red font within the body of the displayed source code (see Figure 1.8.3, “Example: Outstanding Test Rules Violations in Coverity Connect”).

  Example:

  ```
  CID 10246: Insufficient function coverage (TA.UNCALLED) [select issue]
  ```

  You can click the *[select issue]* link to expand the event message:

  ```
  CID 10246 (#1 of 1): Insufficient function coverage (TA.UNCALLED)violation:
  Insufficient line coverage for function
  com.coverity.samples.defector.Defector.divByZero()void.
  0 out of 2 (0%) lines that pass the rule filters are covered by tests.
  1 more lines must be covered to reach the 1% coverage threshold for
  this function.
  ```

1.8.2.4. Displaying code coverage, exclusions, and age bars

You can use the Coverage tab in the Source Gutter menu to display vertical bars that indicate code age (based on the number of bars) and whether a given line of code or function is covered and/or marked for exclusion from tests by your test policy.

Coverage (and Age)

- *Coverage* - A control used to display or hide code coverage bars and indicate the age of the code.

  Coverage bars

  - Green bars (solid vertical bars) denote that the line is covered by a test.
  - Red bars (dotted vertical bars) denote that the line is not covered.

  Note

  For coverage information on a particular line of code, see Section 1.8.3, “Displaying line-level data (test coverage, test policy rules, exclusions, change impact)”.

  Code age

  - Code age is denoted in the Coverity Connect UI by the number of code coverage bars. More bars (three is the maximum) indicate newer code. Newer code is generally at higher risk to contain bugs, so this graphical code age representation allows you to easily identify the newer code for which you should concentrate your tests.

    The number of bars (code age) is determined by the boundaries set in the test policy file; specifically through the recent_date_cutoff and old_date_cutoff fields. Code that is newer than the value set for `recent_date_cutoff` is assigned three bars. Code that is older than the value set for `old_date_cutoff` is assigned one bar. Code that is in between these two values is assigned two bars.

Note

For more information about setting code age boundaries in the policy file, see Chapter 1.5, *Creating Test Advisor policies*.

Exclusions

- *Coverage Exclusions* - A menu used to display or hide coverage exclusion gutters (bars).

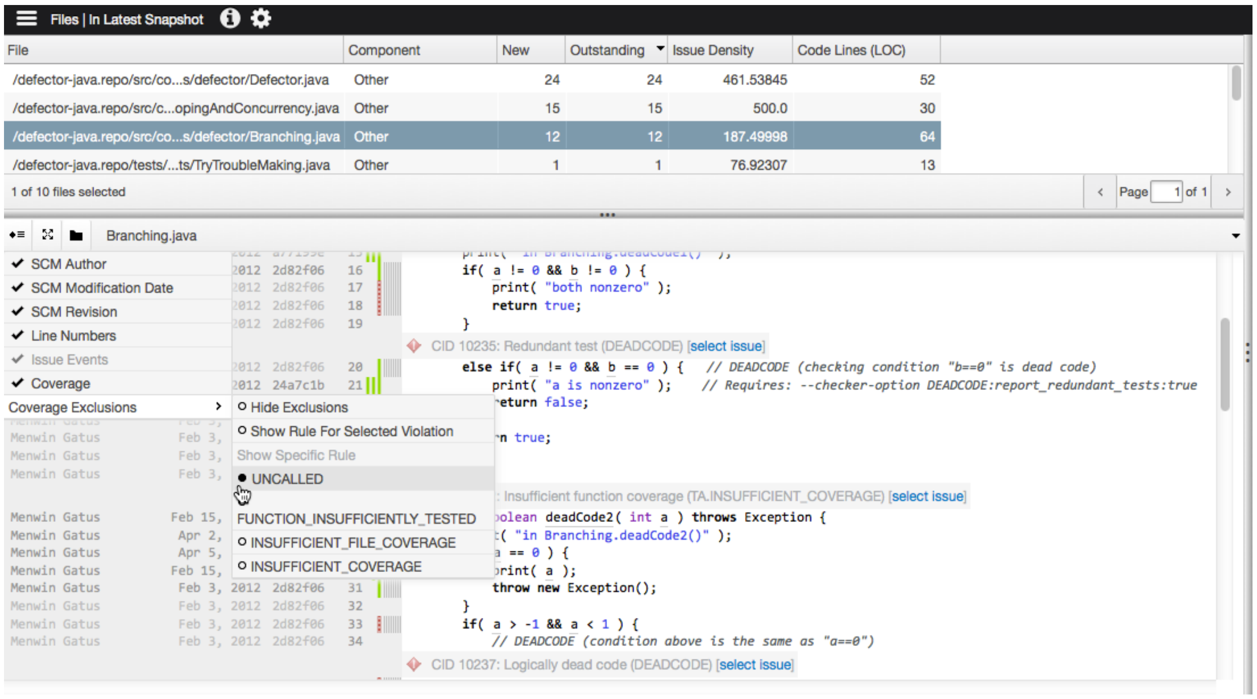
Exclusion bars

- Gray bars denote that the line was excluded. These are especially useful when you compare them with code coverage bars. For example, if you see the gray exclusion bar for the same lines that a red coverage bar covers, you know that the absence of coverage is not an issue. If you see the exclusion bars for lines where there is coverage, you might want to determine whether the coverage or exclusion is needed.

Coverage exclusions

Coverage exclusions come from filters in the policy file, which identifies lines in the source code that do not require tests. For details, see Section 1.5.4, “FileFilter”, Section 1.5.5, “FunctionFilter”, Section 1.5.6, “LineFilter”, and other subsections of Chapter 1.5, *Creating Test Advisor policies* that describe filters.

Figure 1.8.5. Example: Using the Coverage Exclusions menu



File	Component	New	Outstanding	Issue Density	Code Lines (LOC)
/defector-java.repo/src/co...s/defector/Defector.java	Other	24	24	461.53845	52
/defector-java.repo/src/c...opingAndConcurrency.java	Other	15	15	500.0	30
/defector-java.repo/src/co...s/defector/Branching.java	Other	12	12	187.49998	64
/defector-java.repo/tests/...ts/TryTroubleMaking.java	Other	1	1	76.92307	13

1 of 10 files selected — Page 1 of 1

```
    if( a != 0 && b != 0 ) {
        print( "both nonzero" );
        return true;
    }
CID 10235: Redundant test (DEADCODE) [select issue]
    else if( a != 0 && b == 0 ) {   // DEADCODE (checking condition "b==0" is dead code)
        print( "a is nonzero" );    // Requires: --checker-option DEADCODE:report_redundant_tests:true
        return false;

    return true;

: Insufficient function coverage (TA.INSUFFICIENT_COVERAGE) [select issue]
olean deadCode2( int a ) throws Exception {
t( "in Branching.deadCode2()" );
a == 0 ) {
print( a );
        throw new Exception();
    }
    if( a > -1 && a < 1 ) {
        // DEADCODE (condition above is the same as "a==0")
CID 10237: Logically dead code (DEADCODE) [select issue]
```

- Hide Exclusions:

  Hides the exclusion bars (gray lines) from the Source browser.

- Show Rule For Selected Violation:

  Displays the exclusion bar for the exclusion rule that is associated with the selected Test Advisor violation. For example, if the UNCALLED rule is associated with the selected violation, then gray exclusion bar will appear where the UNCALLED rule applies in the source file.

- Show Specific Rule:

  Allows you to switch between the exclusion rules that are specified for source code in a given project. This feature can be useful if you want to examine exclusions in a source code file independently of the selected test violation. For example, assume there are only two exclusion rules: UNCALLED and INSUFFICIENT COVERAGE. In this case, even if the selected test violation is associated with the UNCALLED exclusion rule, you can opt to view exclusions to which the INSUFFICENT COVERAGE rule applies. Note that selecting UNCALLED in this case is the same as selecting *Show Rule For Selected Violation*.

  Note

  For exclusion information on a particular line of code, see Section 1.8.3, “Displaying line-level data (test coverage, test policy rules, exclusions, change impact)”.

1.8.3. Displaying line-level data (test coverage, test policy rules, exclusions, change impact)

By clicking any line of code in the Source browser, you can open a set of tabs that contain Test Advisor data about that particular line.

Note

To help you decide which lines to examine, see the UI display options for the Source browser in Section 1.8.2, “Displaying SCM data, line numbers, events, and Test Advisor coverage data”.

1.8.3.1. Displaying test coverage data

You use the Test Coverage tab in the Source Gutter menu to find out if a line of source code is covered by a test and get information about the test:

- If the line is not executable, test coverage is not applicable. In this case, you will see the following message:

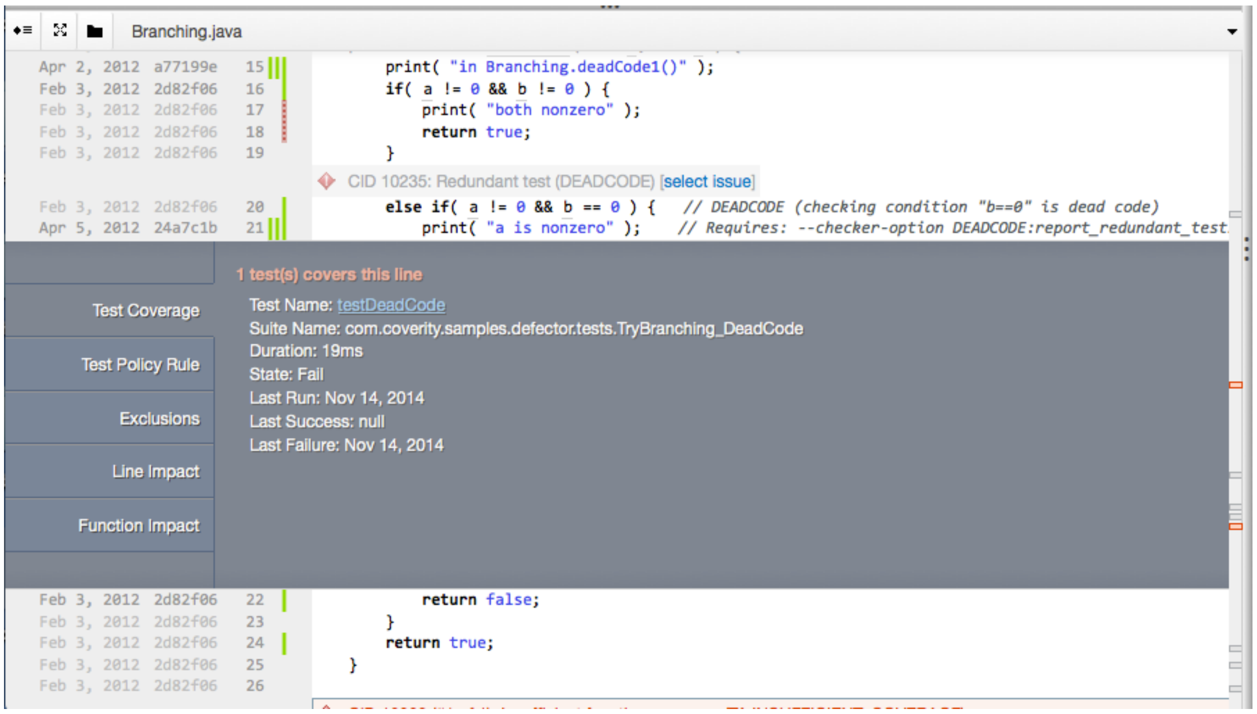
  ```
  This line is not executable so test coverage data is not applicable
  ```

- If the line is covered by one or more tests, the tab will provide the following information:

  - Test name (includes a link to the test)

- Suite name (test suite name)
- Duration of the test (time in milliseconds)
- State (pass or fail)
- Last run (date that the test last ran)
- Last sucesss (date that the test last succeeded)

Figure 1.8.6. Example: Test Coverage



```
Apr 2, 2012  a77199e        print( "in Branching.deadCode1()" );
Feb 3, 2012  2d82f06        if( a != 0 && b != 0 ) {
Feb 3, 2012  2d82f06            print( "both nonzero" );
Feb 3, 2012  2d82f06            return true;
Feb 3, 2012  2d82f06        }
CID 10235: Redundant test (DEADCODE) [select issue]
Feb 3, 2012  2d82f06        else if( a != 0 && b == 0 ) {   // DEADCODE (checking condition "b==0" is dead code)
Apr 5, 2012  24a7c1b            print( "a is nonzero" );    // Requires: --checker-option DEADCODE:report_redundant_test
```

1 test(s) covers this line

Test Name: testDeadCode
Suite Name: com.coverity.samples.defector.tests.TryBranching_DeadCode
Duration: 19ms
State: Fail
Last Run: Nov 14, 2014
Last Success: null
Last Failure: Nov 14, 2014

```
Feb 3, 2012  2d82f06            return false;
Feb 3, 2012  2d82f06        }
Feb 3, 2012  2d82f06        return true;
Feb 3, 2012  2d82f06    }
Feb 3, 2012  2d82f06
```

- If the line is not covered by a test, you will see the following message:

```
0 tests cover this line
```

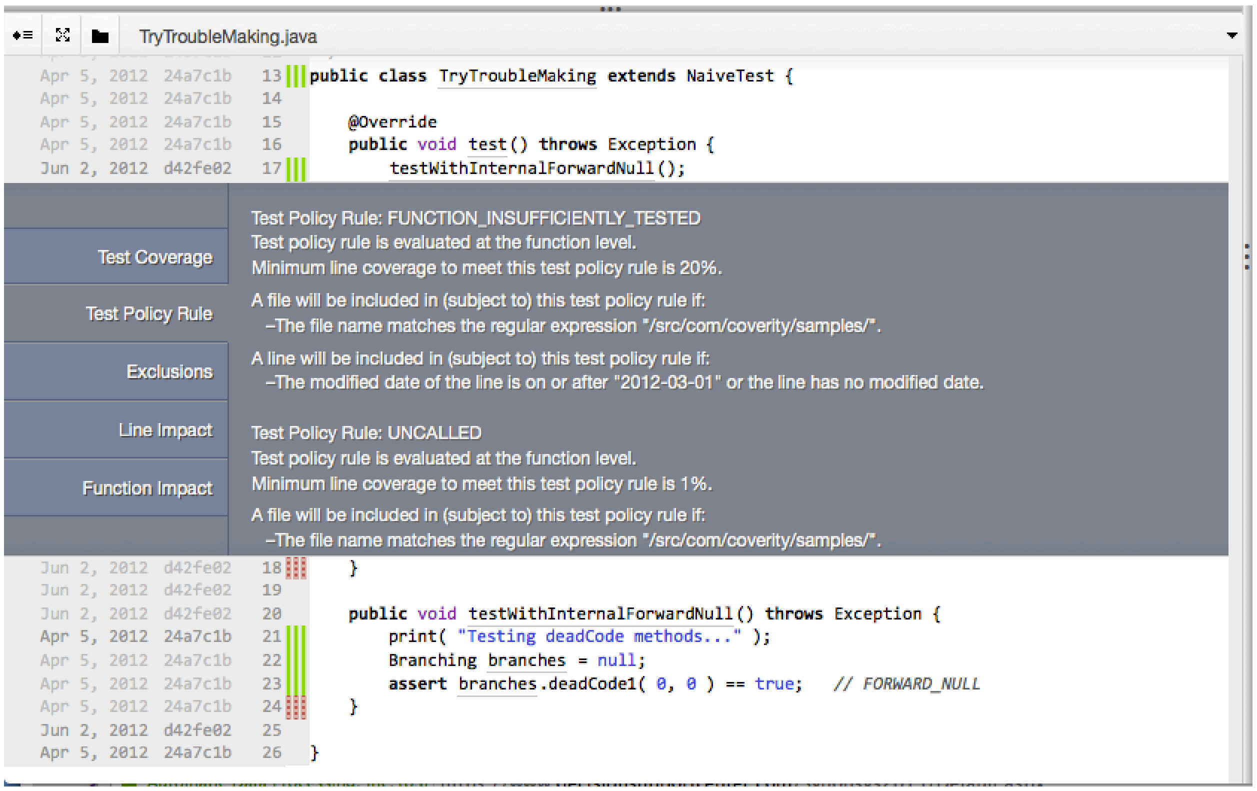
1.8.3.2. Test Policy Rule

You use this source gutter tab to find out what Test Policy rule was evaluated to determine this violation.

- If one or more Test Policy rules applies, the tab will provide the following information:
  - Name of the Test Policy rule (for example, UNCALLED)
  - Evaluation level (for example, the function level)

- Conditions for applying the rule to the file (for example, if the file name matches a specified regular expression)
- Conditions for applying the rule to the line (for example, if the line was modified on or after a particular date)

Figure 1.8.7. Example: Test Policy Rule



```
Apr 5, 2012  24a7c1b   public class TryTroubleMaking extends NaiveTest {
Apr 5, 2012  24a7c1b
Apr 5, 2012  24a7c1b       @Override
Apr 5, 2012  24a7c1b       public void test() throws Exception {
Jun 2, 2012  d42fe02           testWithInternalForwardNull();
```

Test Policy Rule: FUNCTION_INSUFFICIENTLY_TESTED
Test policy rule is evaluated at the function level.
Minimum line coverage to meet this test policy rule is 20%.

A file will be included in (subject to) this test policy rule if:
–The file name matches the regular expression "/src/com/coverity/samples/".

A line will be included in (subject to) this test policy rule if:
–The modified date of the line is on or after "2012-03-01" or the line has no modified date.

Test Policy Rule: UNCALLED
Test policy rule is evaluated at the function level.
Minimum line coverage to meet this test policy rule is 1%.

A file will be included in (subject to) this test policy rule if:
–The file name matches the regular expression "/src/com/coverity/samples/".

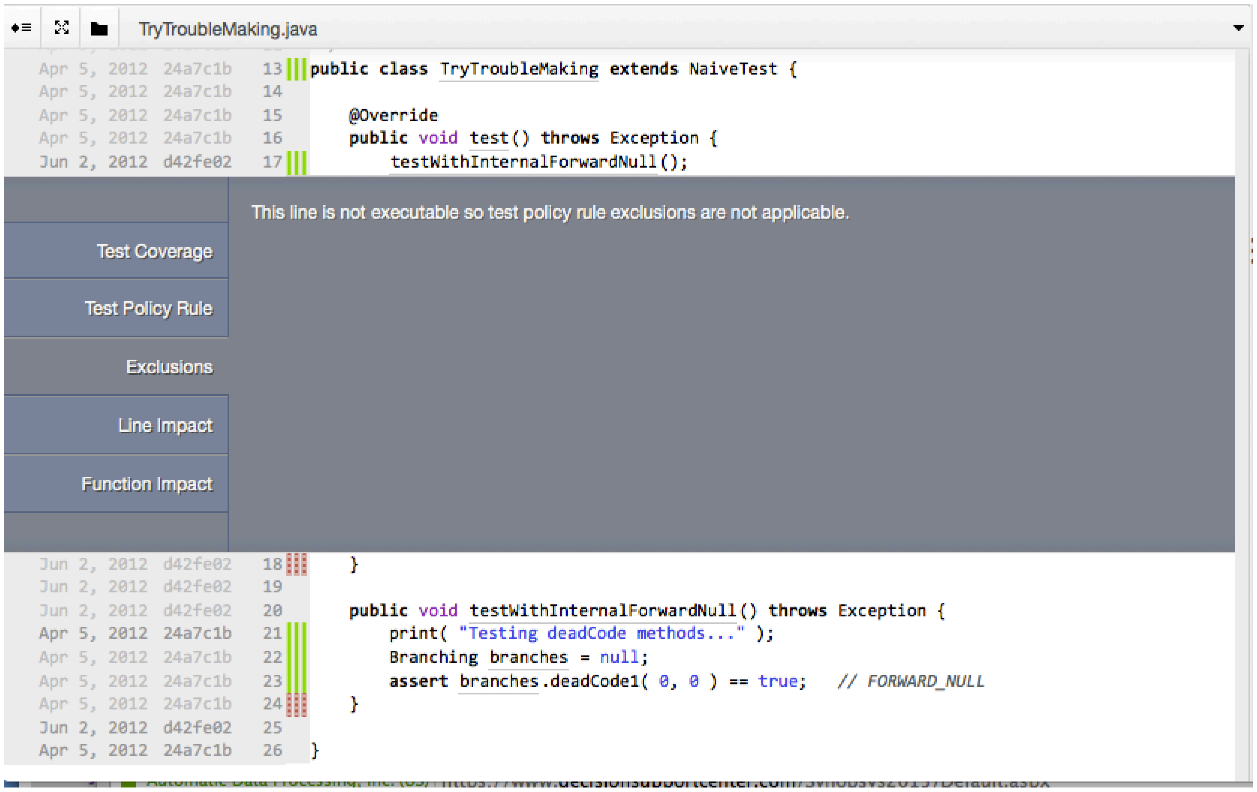
```
Jun 2, 2012  d42fe02       }
Jun 2, 2012  d42fe02
Jun 2, 2012  d42fe02       public void testWithInternalForwardNull() throws Exception {
Apr 5, 2012  24a7c1b           print( "Testing deadCode methods..." );
Apr 5, 2012  24a7c1b           Branching branches = null;
Apr 5, 2012  24a7c1b           assert branches.deadCode1( 0, 0 ) == true;    // FORWARD_NULL
Apr 5, 2012  24a7c1b       }
Jun 2, 2012  d42fe02
Apr 5, 2012  24a7c1b   }
```

1.8.3.3. Exclusions

You use this source gutter tab to determine whether a line was excluded from a Test Policy rule.

- If the line is is not executable, the exclusion is not applicable:

Figure 1.8.8. Example: Test Exclusions



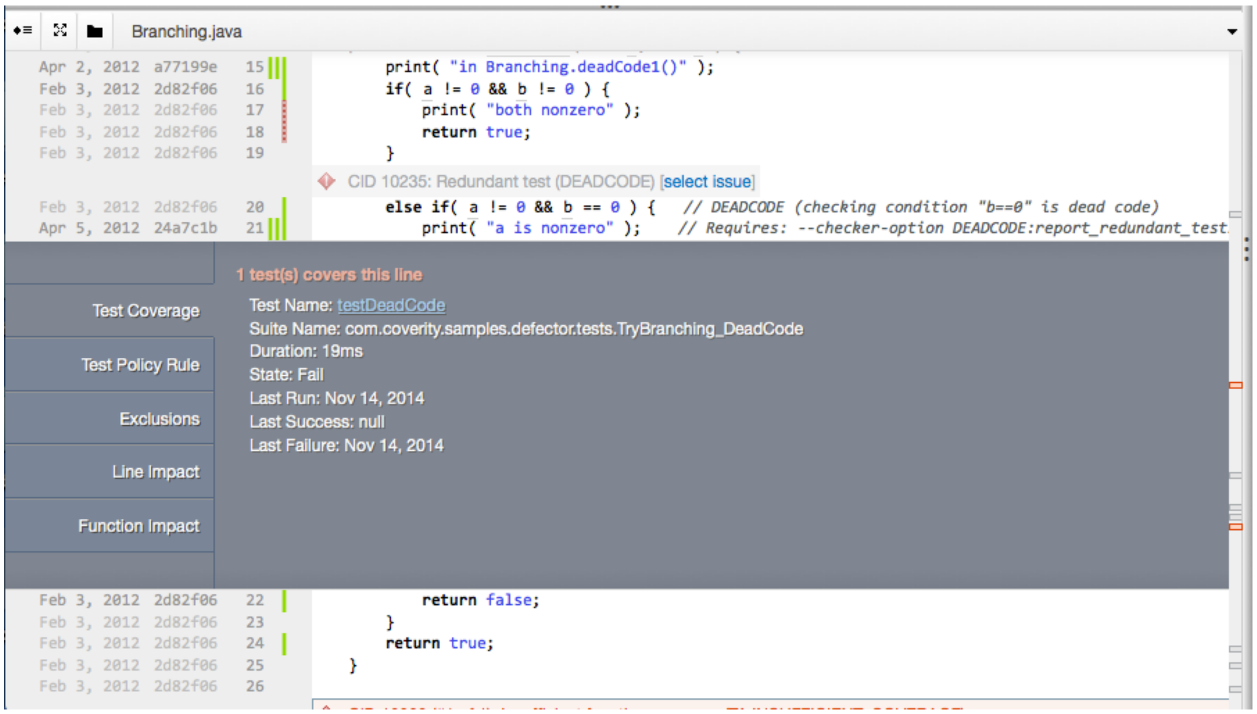
```
Apr 5, 2012  24a7c1b   public class TryTroubleMaking extends NaiveTest {
Apr 5, 2012  24a7c1b
Apr 5, 2012  24a7c1b       @Override
Apr 5, 2012  24a7c1b       public void test() throws Exception {
Jun 2, 2012  d42fe02           testWithInternalForwardNull();
```

This line is not executable so test policy rule exclusions are not applicable.

```
Jun 2, 2012  d42fe02       }
Jun 2, 2012  d42fe02
Jun 2, 2012  d42fe02       public void testWithInternalForwardNull() throws Exception {
Apr 5, 2012  24a7c1b           print( "Testing deadCode methods..." );
Apr 5, 2012  24a7c1b           Branching branches = null;
Apr 5, 2012  24a7c1b           assert branches.deadCode1( 0, 0 ) == true;    // FORWARD_NULL
Apr 5, 2012  24a7c1b       }
Jun 2, 2012  d42fe02
Apr 5, 2012  24a7c1b   }
```

- If the line is excluded due to the Test Policy rule, the tab identifies the rule and explains why the rule excludes the line:

Figure 1.8.9. Example: Test Exclusions



```
Apr 2, 2012 a77199e        print( "in Branching.deadCode1()" );
Feb 3, 2012 2d82f06        if( a != 0 && b != 0 ) {
Feb 3, 2012 2d82f06            print( "both nonzero" );
Feb 3, 2012 2d82f06            return true;
Feb 3, 2012 2d82f06        }
                       CID 10235: Redundant test (DEADCODE) [select issue]
Feb 3, 2012 2d82f06        else if( a != 0 && b == 0 ) {   // DEADCODE (checking condition "b==0" is dead code)
Apr 5, 2012 24a7c1b            print( "a is nonzero" );    // Requires: --checker-option DEADCODE:report_redundant_test
```

1 test(s) covers this line

Test Name: testDeadCode
Suite Name: com.coverity.samples.defector.tests.TryBranching_DeadCode
Duration: 19ms
State: Fail
Last Run: Nov 14, 2014
Last Success: null
Last Failure: Nov 14, 2014

```
Feb 3, 2012 2d82f06            return false;
Feb 3, 2012 2d82f06        }
Feb 3, 2012 2d82f06        return true;
Feb 3, 2012 2d82f06    }
Feb 3, 2012 2d82f06
```

- If the line is included (in other words, not excluded) and subject to the Test Policy rule, the tab will explain, for example:

```
This line is included and subject to the selected
test policy rule (A_TEST_POLICY_RULE_HERE)
```

1.8.3.4. Line Impact

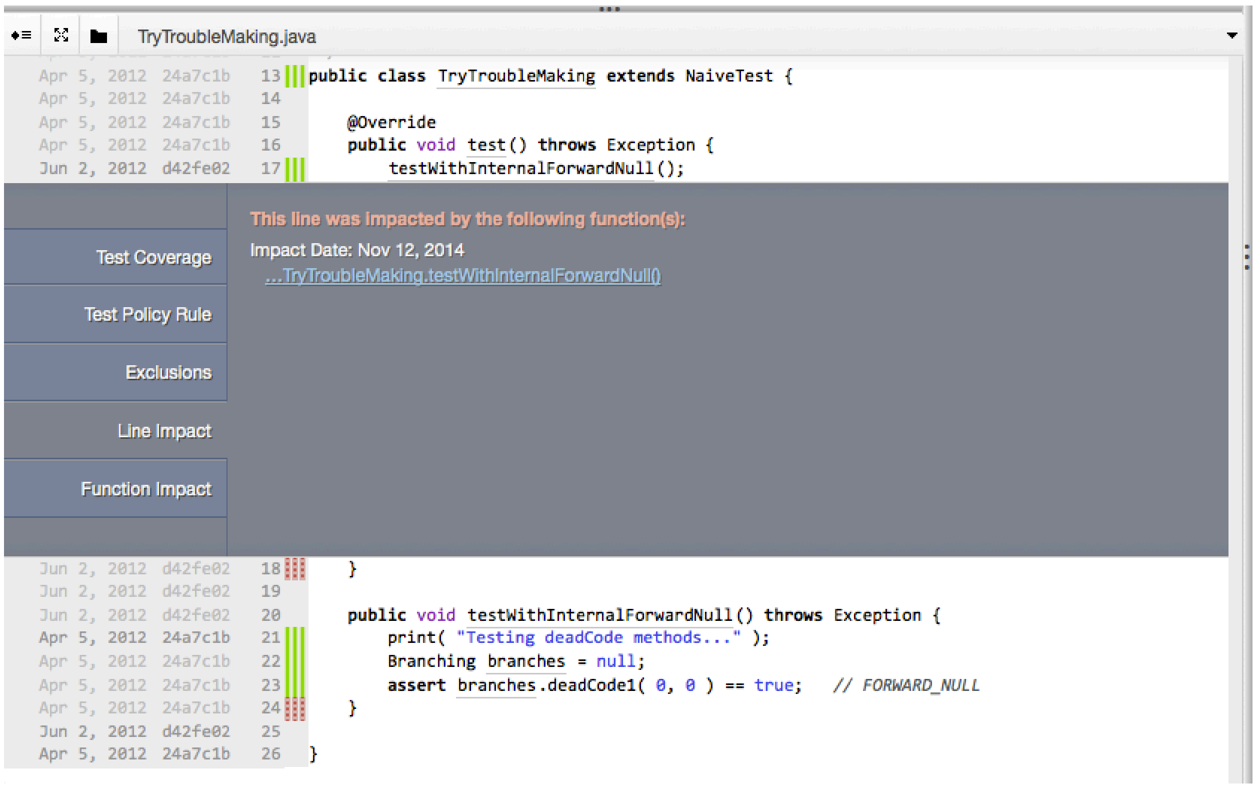
You use this source gutter tab to determine whether a line contains a function call to a function with a function summary that changed.

- If the line does not contain at least one function call, the tab will provide the following information:

```
No function calls exist on this line
so line impact data is not applicable.
```

- If the line contains one or more functions with function summaries that have changed, the tab will provide a link to each function and the date on which the line was impacted by the change.

Figure 1.8.10. Example: Line Impact



```
Apr 5, 2012  24a7c1b  13  public class TryTroubleMaking extends NaiveTest {
Apr 5, 2012  24a7c1b  14
Apr 5, 2012  24a7c1b  15      @Override
Apr 5, 2012  24a7c1b  16      public void test() throws Exception {
Jun 2, 2012  d42fe02  17          testWithInternalForwardNull();
```

This line was impacted by the following function(s):

Impact Date: Nov 12, 2014

...TryTroubleMaking.testWithInternalForwardNull()

```
Jun 2, 2012  d42fe02  18      }
Jun 2, 2012  d42fe02  19
Jun 2, 2012  d42fe02  20      public void testWithInternalForwardNull() throws Exception {
Apr 5, 2012  24a7c1b  21          print( "Testing deadCode methods..." );
Apr 5, 2012  24a7c1b  22          Branching branches = null;
Apr 5, 2012  24a7c1b  23          assert branches.deadCode1( 0, 0 ) == true;    // FORWARD_NULL
Apr 5, 2012  24a7c1b  24      }
Jun 2, 2012  d42fe02  25
Apr 5, 2012  24a7c1b  26  }
```

1.8.3.5. Function Impact

You use this source gutter tab to determine whether a function is directly or indirectly impacted.

- Direct impact occurs when the function changes, typically meaning that the Abstract Syntax Tree (AST) changed. Direct impact can also occur because of to changes in globals, types, and so on.
- Indirect impact occurs when the function directly calls a function whose function summary has changed.
- If the line has no function declarations, the tab provides the following information:

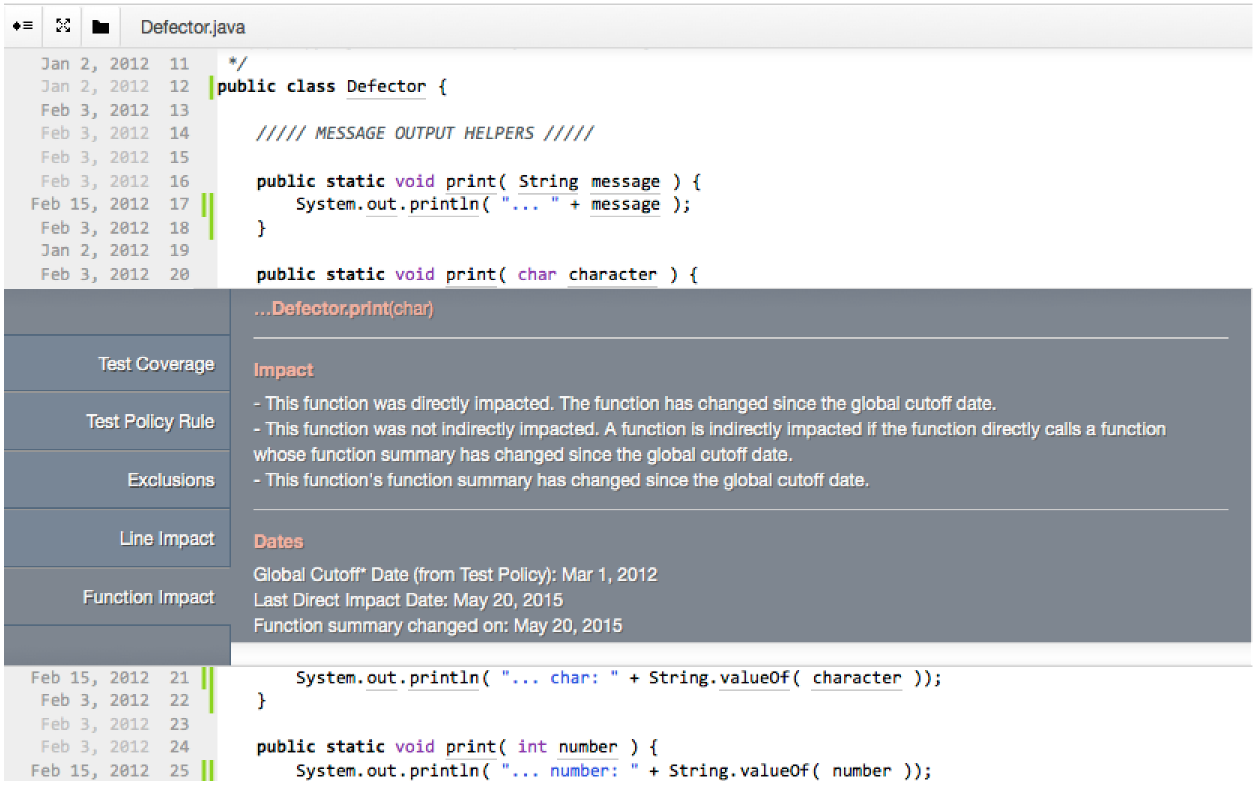
  ```
  This line has no function definitions so
  function impact data is not applicable.
  ```

- If the line has a function impact, the tab will indicate whether the impact is direct or indirect and will identify the following dates:
  - Global cutoff date (from Test Policy)

    If this rule has a rule-specific `recent_cutoff_date`, that will be used instead of the global cutoff date.

- Direct impact date
- Indirect impact date
- Function summary change date

Figure 1.8.11. Example: Test Coverage



```
Jan 2, 2012      */
Jan 2, 2012    public class Defector {
Feb 3, 2012
Feb 3, 2012        ///// MESSAGE OUTPUT HELPERS /////
Feb 3, 2012
Feb 3, 2012        public static void print( String message ) {
Feb 15, 2012           System.out.println( "... " + message );
Feb 3, 2012        }
Jan 2, 2012
Feb 3, 2012        public static void print( char character ) {
```

...Defector.print(char)

Impact

- This function was directly impacted. The function has changed since the global cutoff date.
- This function was not indirectly impacted. A function is indirectly impacted if the function directly calls a function whose function summary has changed since the global cutoff date.
- This function's function summary has changed since the global cutoff date.

Dates

Global Cutoff* Date (from Test Policy): Mar 1, 2012
Last Direct Impact Date: May 20, 2015
Function summary changed on: May 20, 2015

```
Feb 15, 2012           System.out.println( "... char: " + String.valueOf( character ));
Feb 3, 2012        }
Feb 3, 2012
Feb 3, 2012        public static void print( int number ) {
Feb 15, 2012           System.out.println( "... number: " + String.valueOf( number ));
```

1.8.4. Viewing test coverage in files and functions

Coverity Connect provides several different perspectives from which you can look up test policy coverage within a selected project. The Files view menu (see Figure 1.8.1, “Example: View panel in Coverity Connect”) contains views that list files with test coverage data, while the Function view menu contains views that list functions with test coverage data.

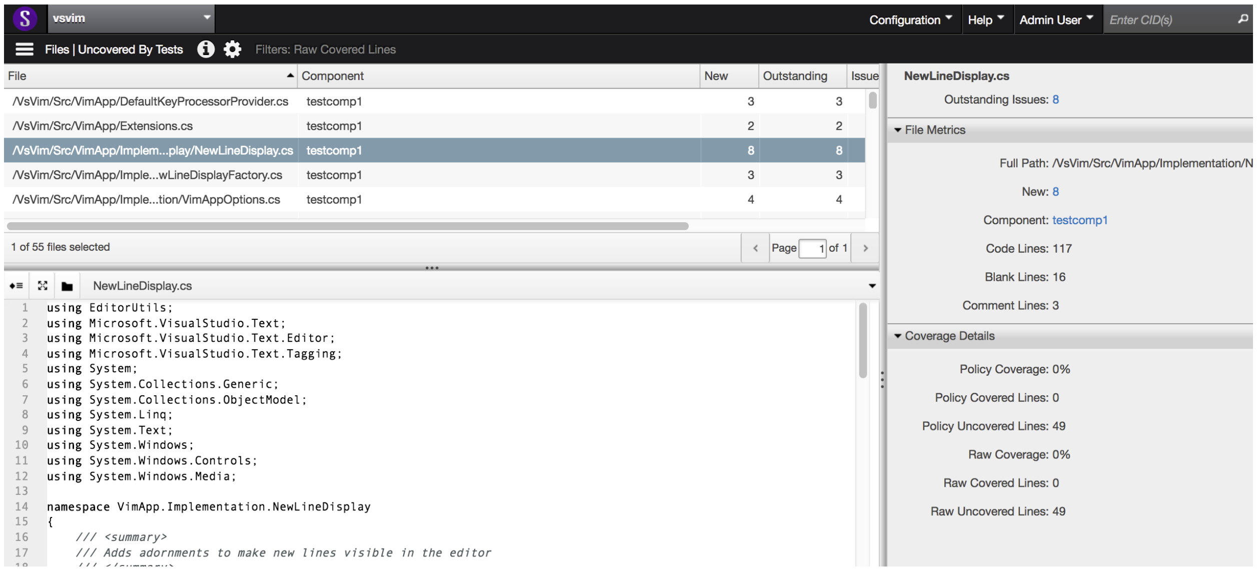
The default test coverage-related views for both files and functions:

- *Uncovered by tests* - Lists all of the files in the current project that have not been covered by a test.

Note

If you had views (such as Covered by Test) in a previous version of Coverity Connect and have upgraded to version 7.0, your views will be preserved in the upgraded version.

For example, if you click the *Uncovered by tests* view in the File menu, Coverity Connect displays the following screen:



File	Component	New	Outstanding
/VsVim/Src/VimApp/DefaultKeyProcessorProvider.cs	testcomp1	3	3
/VsVim/Src/VimApp/Extensions.cs	testcomp1	2	2
/VsVim/Src/VimApp/Implem...play/NewLineDisplay.cs	testcomp1	8	8
/VsVim/Src/VimApp/Imple...wLineDisplayFactory.cs	testcomp1	3	3
/VsVim/Src/VimApp/Imple...tion/VimAppOptions.cs	testcomp1	4	4

1 of 55 files selected   Page 1 of 1

NewLineDisplay.cs

```
using EditorUtils;
using Microsoft.VisualStudio.Text;
using Microsoft.VisualStudio.Text.Editor;
using Microsoft.VisualStudio.Text.Tagging;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace VimApp.Implementation.NewLineDisplay
{
    /// <summary>
    /// Adds adornments to make new lines visible in the editor
```

NewLineDisplay.cs
Outstanding Issues: 8

File Metrics
Full Path: /VsVim/Src/VimApp/Implementation/N
New: 8
Component: testcomp1
Code Lines: 117
Blank Lines: 16
Comment Lines: 3

Coverage Details
Policy Coverage: 0%
Policy Covered Lines: 0
Policy Uncovered Lines: 49
Raw Coverage: 0%
Raw Covered Lines: 0
Raw Uncovered Lines: 49

When you click a file or function, Coverity Connect opens the source code for that file in the Source browser, where you can obtain additional coverage data for that file.

You can define a more focused list of files or functions by using filters, and you can change the columns that appear in the list through tabs in the Settings window for the view. The Test Advisor-specific columns and filters are:

Table 1.8.1. Test Advisor filters

Filter	Description
Policy Coverage	Policy Coverage. See Understanding Policy Coverage versus Raw Coverage for more information about policy coverage and raw coverage.
Policy Covered Lines	Number of lines covered by tests according to your Test Advisor policy.
Policy Uncovered Lines	Number of lines not covered by tests according to your Test Advisor policy.
Raw Coverage	Raw coverage. See Understanding Policy Coverage versus Raw Coverage for more information about policy coverage and raw coverage.
Raw Covered Lines	Number of lines covered by tests as reported by the coverage tool.

Filter	Description
Raw Uncovered Lines	Number of lines not covered by tests as reported by the coverage tool.

Understanding Policy Coverage versus Raw Coverage

To understand the difference between policy coverage and raw coverage, assume that your Test Advisor policy states that all code must be 100% tested with the exception of setters/getters and IOExceptions.

It is possible for the code to meet this policy and have 100% coverage of lines that are not setters/getters or IOExeceptions. In this case, the policy coverage is 100%, as the policy excludes the setters/getters.

However, if you did not cover all setters/getters and IOExceptions, this would be shown in the raw coverage, which would be less than 100%.

While raw coverage will typically be lower than the policy coverage, it is possible for raw coverage to be higher than policy coverage if the only lines that are covered are excluded by the policy.

The following filters are specific to the Functions view:

Table 1.8.2. Test Advisor filters

Filter	Description
Last Impacted	Allows the you to filter the list of functions by the "last impacted" date of each function. Last impacted date is the date on which the function was either changed directly by the user, resulting in a modification of its syntactic structure (that is, the AST; whitespace/comment changes are ignored) or on which it was affected by a direct change elsewhere in the code base that affects this function's behavior. Examples are changes to the callees of the function or to the global variables used by the function.
Last Modified	Allows you to filter the list of functions by the last modification date of each function. Last modification date is computed from SCM data and is taken to be the most recent date on which any of the source code lines belonging to the function were modified.

1.8.5. Viewing test status

Coverity Connect provides a Tests view type (see Figure 1.8.1, “Example: View panel in Coverity Connect”) that contains views that help you quickly locate the actual tests in your project and examine the status (pass or fail) of the test.

- *All Tests* - Displays all tests within the project.
- *Currently Failing* - Displays only the tests that currently not passing.

These views have filters that you can apply to narrow the scope of your test list. The filters correspond to the columns that you choose to display in the data table. Columns and filters are activated through the View menu pull-down. The Test Advisor-specific columns and filters are:

Table 1.8.3. Test Advisor filters

Filter	Description
Duration	The time in milliseconds that it took to run the test.
Last Failure	Date when test last failed.
Last Run	Date when the test was last run.
Last Success	Date when the test last passed..
Name	Test name.
State	Test state. Pass, Fail, or Unknown
Suite	Test suite name.

If test boundary detection was specified during the capture (see Chapter 1.6, *Configuring test boundaries* , the test names provide links that you can click to display the source of the test in the Source browser.

1.8.6. Using Coverity Policy Manager to determine test effectiveness

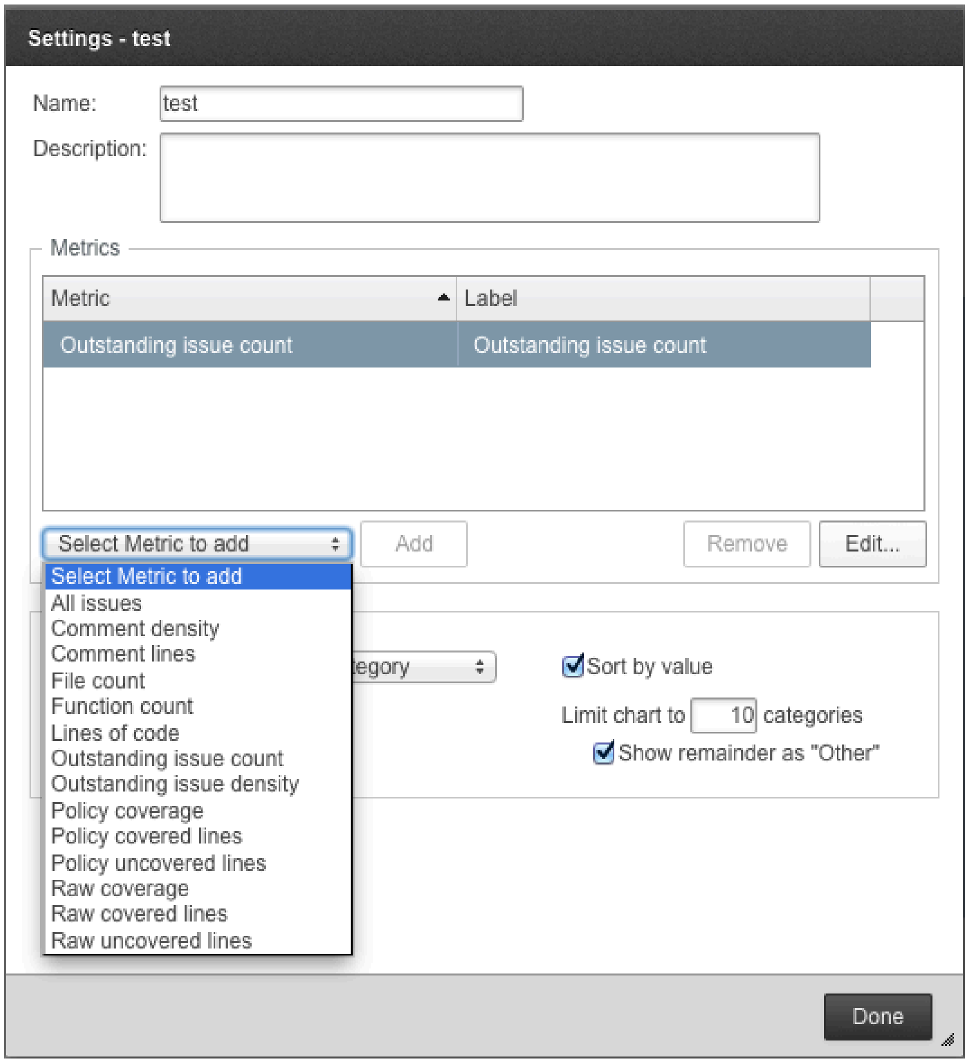
Coverity Policy Manager enables you to set policies for code quality and security, and policy test coverage. You can then manage, monitor, and report on these policies as code is tested. This section introduces only the Test Advisor features for Coverity Policy Manager. Furthermore, this section assumes that you have familiarity with using and/or configuring Coverity Policy Manager.

For more specific usage instructions, see the .

1.8.6.1. Defining test metrics in Coverity Policy Manager

Coverity Policy Manager metrics provide numeric information on data that is stored in the Coverity Connect database. Metrics are available for use in Coverity Policy Manager heatmaps, status reports, and trends. To add a metric, click the *Edit Settings...* option from the edit drop down menu.

The following metrics are measures in the "Data fields" drop-down to create or edit metrics:



Name: test

Metric	Label
Outstanding issue count	Outstanding issue count

Sort by value ✓

Limit chart to 10 categories

Show remainder as "Other" ✓

Policy Coverage - The percentage that is equal to the count of lines covered by tests divided by coverable lines.

- *Policy Covered Lines* - Filtered count of lines covered by tests.
- *Policy Uncovered Lines* - Number of lines covered by tests according to your Test Advisor policy.
- *Coverable Lines* - The sum of *Policy Covered Lines* and *Policy Uncovered Lines*.
- *Raw Coverage* - The percentage that is equal to the raw covered lines divided by raw coverable lines.
- *Policy Coverage Percent* - Policy covered lines, divided by coverable lines.
- *Raw Policy Coverage Percent* - Raw covered lines, divided by raw coverable lines.

Chapter 1.9. Using Test Advisor with Coverity Analysis

By default, `cov-analyze` will run the default set of quality checkers at the same time Test Advisor is run. This is the most efficient procedure to obtain both Coverity and Test Advisor defects on the same code base. All defects will be written to the same intermediate directory, and `cov-commit-defects` will commit all defects to the same stream.

If you want to only run Test Advisor, Coverity can be disabled by using the `--disable-default` option to `cov-analyze`.

It is also possible to run Test Advisor and Coverity separately on the same intermediate directory. In this case, you can use the `--append` option to `cov-analyze` to combine the generated defects. However, Coverity recommends that Test Advisor be the first `cov-analyze` command run on the intermediate directory. For more details, see the issue description for *46542 - Coverity Connect might not accurately display coverage bars* in the *Coverity Release Notes Archive*.

Part 2. Test Advisor impact analysis

This section provides detailed information on impact analysis with Test Advisor. For basic usage information, see Section 1.2.4, “Getting impact analysis results”. More advanced usage information is available in Chapter 2.4, *Using Impact analysis in Test Advisor*.

Chapter 2.1. Introduction to impact analysis

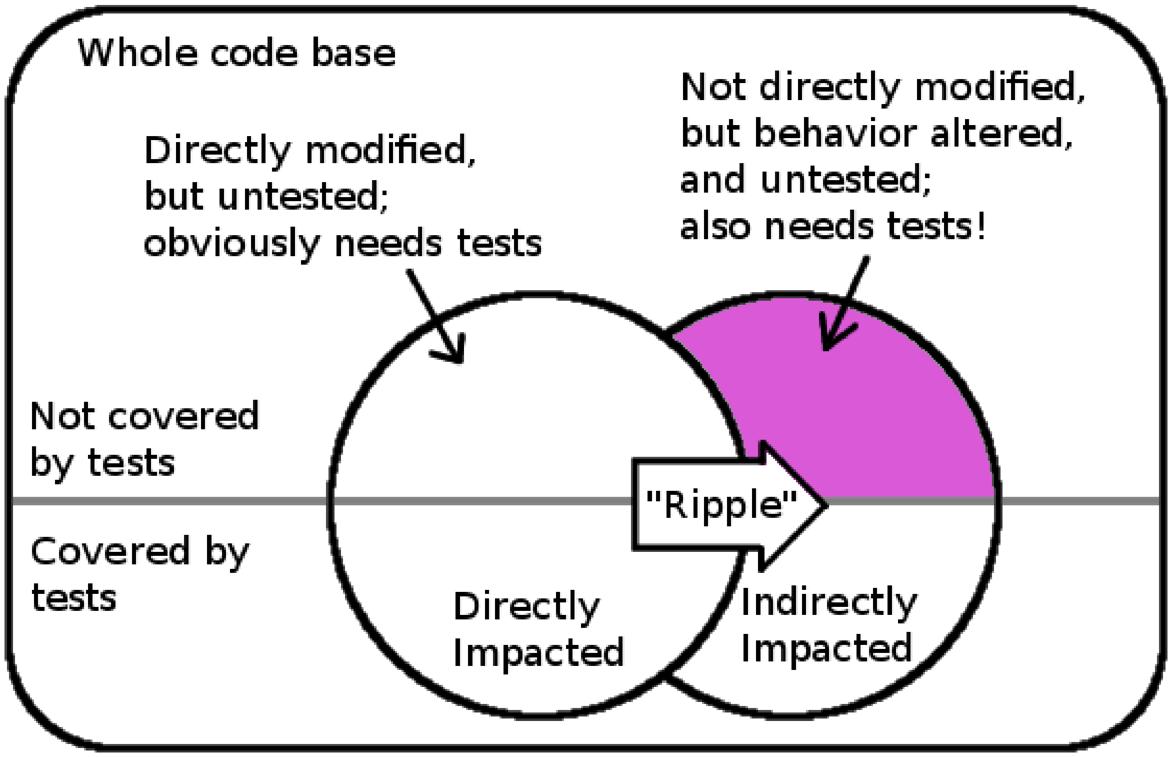
Impact analysis is an algorithm that computes a set of program entities (lines and functions) whose behavior is affected by a change to the program source code. That set includes both direct impact, meaning entities whose syntactic definition was changed, and indirect impact, meaning entities whose syntax did not change but whose behavior most likely changed due to changes made elsewhere.

Ripple
: When a change made in one area of the code base affects the behavior of code located in another area. Coverity refers to this indirect modification of code as a *ripple*.

Impact analysis is useful in a test policy that focuses on recent code changes. It is usually impractical to mandate a minimum level of code coverage across an entire code base at once, because even a project with high average coverage will almost always have a lot of uncovered code, and adding tests to cover that code is too large a task to complete in one step. Instead, a typical organization should define a date cutoff and start by focusing on code that has been modified since that date. However, a naive implementation of this approach leaves a blind spot, that is, code left untested because its behavior was modified by changes elsewhere in the code base. Impact analysis covers this blind spot by identifying such code and subjecting it to the test policy.

The following diagram illustrates the relationships.

Figure 2.1.1. Example: Ripple effect of changes to the code base


Whole code base
Directly modified, but untested; obviously needs tests
Not directly modified, but behavior altered, and untested; also needs tests!
Not covered by tests
Covered by tests
"Ripple"
Directly Impacted
Indirectly Impacted

In this figure, the circle on the left (labled *Directly impacted*) contains code that has been modified since a chosen cutoff date. The circle on the right (labled *Indirectly impacted*) is additional unmodified code that is impacted by the changes on the left. A divider separates code that has tests (labled *Covered by tests*) from that which does not (labled *Not covered by tests*). The purple shaded region contains code that is risky to leave untested. Test Advisor impact analysis is designed to identify such code.

Chapter 2.2. Impact concepts and definitions

Table of Contents

Suppose a programmer changes the code of a particular function in a code library. The behavior of the changed function might be modified due to that code change. However, the behavior of other functions that call the changed function might also be modified. Coverity uses the term *impacted* to refer to the functions whose behavior might have changed as a result of the code change. That is, a code change might cause several functions to be impacted.

Since impacted functions are those functions whose behavior might have changed, ensuring that such functions are well-tested is an important way to validate that the code change did not have any undesired effects. Chapter 2.4, *Using Impact analysis in Test Advisor* describes how to use this information in the Test Advisor policy to guide testing efforts towards these functions.

Impact is divided into two categories, direct impact and indirect impact. These are described in the following sections.

2.2.1. Direct impact

Direct impact occurs when the function itself changes. For example, in Figure 2.1.1, “Example: Ripple effect of changes to the code base”, the function modified by the programmer is directly impacted.

Direct impact

A function is directly impacted whenever a code change occurs within the body of that function. Thus, in the majority of cases, when a function is directly impacted, a code change has occurred within that function itself.

Consider the following C++ code example:

```
struct Coords {
    Coords() : x(0), y(0) { }
    int getX() {
        return x;
    }
    int getY() {
        return y;
    }
    void incrementX() {
        x++;
    }
    void incrementY() {
        y++;
    }
    void setX(int n) {
        x = n;
    }
    void setY(int n) {
        y = n;
    }
    int x, y;
};

void rewriteX(Coords &g, int v) {
    g.setX(v);
}

void rewriteY(Coords &g, int v) {
    g.setY(v);
}

void rewrite(Coords &g, int x, int y) {
    rewriteX(g, x);
    rewriteY(g, y);
}
```

Suppose a programmer decides to change the `incrementX()` and `incrementY()` functions to increment by 2 instead of 1:

```
// BEFORE                         // AFTER
...                               ...
                                  const int delta = 2;  // CHANGED
struct Coords {                   struct Coords {
   ...                               ...
   void incrementX() {                 void incrementX() {
       x++;                                x += delta;       // CHANGED
   }                                    }
   void incrementY() {                 void incrementY() {
       y++;                                y += delta;       // CHANGED
   }                                    }
   ...                                  ...
```

Impact analysis will indicate that `incrementX()` and `incrementY()` are directly impacted. All other functions are not directly impacted, because the code of those functions did not change.

The scope of direct impact is limited to changes to the abstract syntax tree of the function. Changes to comments or nonessential whitespace in the code are not treated as direct impacts to the code.

In certain cases, changes that occur outside of a function might cause direct impact to that function. For example, consider the impacts of the changes to the following code sample:

```
// BEFORE                         // AFTER
```

```
...                                 ...
const int delta = 2;                                      // CHANGED
struct Coords {                     struct Coords {
                                        const int delta = 3;  // CHANGED
    ...                                 ...
    void incrementX() {                 void incrementX() {
        x += delta;                         x += delta;
    }                                   }
    void incrementY() {                 void incrementY() {
        y += delta;                         y += delta;
    }                                   }
    ...                                ...
```

The code on the right has no change in the functions `incrementX()` and `incrementY()`. However, Test Advisor will determine that these functions are directly impacted. Direct impact occurs here because the version of code on the right no longer refers to the global variable `delta`. Instead, it refers to the member variable `Coords::delta`.

2.2.2. Indirect impact

Indirect impact is defined through the concept of a *function summary*. For the purpose of impact analysis, the summary of a function describes the "call contract", the set of obligations for both caller and callee for proper behavior. For example, it might require that the callee check a parameter for null on entry, or that the return value be eventually deallocated by the caller. Generally, the summary indicates when the function performs certain actions on its parameters, or when certain values are returned by the function. Details on what constitutes the function summary are described in Section 2.2.6, “Function summary details”.

A direct change to a function might (or might not) affect the function summary of that function. That is, a function that is directly impacted might cause the function summary to change. The example in Section 2.2.2, “Indirect impact” shows code changes that do not cause the function summary of `incrementX()` or `incrementY()` to change.

If a function calls a function whose function summary has changed, the caller's function summary might (or might not) change, as well. The Test Advisor analysis determines whether the caller's function summary changes based on the way that the function summary of the callee changed and how the callee is used by the caller.

Indirect impact
: Indirect impact occurs when a function directly calls a function whose function summary has changed. When the function summary changes for a callee, this indicates that the behavior of the callee has changed in a way that might affect the behavior of the caller. Thus, it becomes important to test the caller to ensure the change is correct.

Continuing with the code examaple in Chapter 2.2, *Impact concepts and definitions*, suppose a programmer makes the following change to `setY()`:

```
// BEFORE                     // AFTER
    ...                          ...
    void setY(int n) {           void setY(int n) {
        y = n;                       x = n;          // CHANGED
    }                            }
    ...                          ...
```

The change to `setY()` affects the function summary. In this case, the function now assigns to a different field of the implicit parameter `this`. Recall that there is a caller of `setY()`, namely `rewriteY()`:

```
void rewriteY(Coords &g, int v) {
    g.setY(v);
}
```

The function `rewriteY()` is not directly impacted, since the code change did not change the code of `rewriteY()`. However, `rewriteY()` is indirectly impacted by this code change, since the code change causes the function summary of `setY()` to be changed.

2.2.3. Relationship between Direct and Indirect impact

For Test Advisor, an *impacted function* is one that is directly impacted or indirectly impacted (or both).

Direct and indirect impact are determined separately. A function might be directly impacted only, indirectly impacted only, both directly and indirectly impacted, or neither, by any given code change. The code changes in Direct impact are examples where the changed functions are directly impacted but not indirectly impacted. The `incrementX()` and `incrementY()` functions do not call any other functions, so they cannot be indirectly impacted. In contrast, the code change in Section 2.2.2, “Indirect impact” is an example where there is a function (`rewriteY()`) that is indirectly impacted but not directly impacted. If that code change example included some change to the body of `rewriteY()` as well, then `rewriteY()` would be determined to be both directly and indirectly impacted.

With an exception described in Chapter 2.3, *Impact analysis limitations*, it is always possible to start from an indirectly impacted function and trace through the impacted callees to a directly impacted function. The directly impacted function can be thought of as a "source" of the impact through the call chain because it represents the code that was directly modified by that code change and is at the root of an indirect impact chain.

Recall that the code example had the following functions:

```
void rewriteY(Coords &g, int v) {
    g.setY(v);
}

void rewrite(Coords &g, int x, int y) {
    rewriteX(g, x);
    rewriteY(g, y);
}
```

As before, function `rewriteY()` is indirectly impacted by the code change and function summary change in `setY()`. However, `rewrite()` is also indirectly impacted because the function summary of `rewriteY()` also changes (`rewriteY()` now writes to a different field of its argument g). Thus, for this change, only `setY()` is directly impacted, but it is possible to follow the indirectly impacted function call stack from `rewrite()` to `rewriteY()` to `setY()`, to determine the source of indirect impact in `rewrite()`.

2.2.4. Propagation of indirect impact

The example in Section 2.2.3, “Relationship between Direct and Indirect impact ” shows a change in `setY()` that indirectly impacted both a caller `rewriteY()` and a caller of `rewriteY()`, namely `rewrite()`. This is because the code change also changed the function summary of `rewriteY()`. The result of callee `setY()` is used by `rewriteY()` in such a way that the function summary of `rewriteY()` is affected.

If impact analysis determines that a callee is not used in a way that can affect the function summary of the caller, then the function summary of the caller is not changed, and propagation of indirect impact stops. Now assume that the same code example also includes a function such as this one:

```
void createLocal() {
    Coords g;
    g.setX(6);
    g.setY(2);
    printToLog(g);
}
```

Here, the result of `setY()` is not used in any way that causes the function summary of `createLocal()` to change. That is, the obligations imposed by `createLocal()` on its callers do not change due to the code change in `setY()`. Thus, while `createLocal()` is indirectly impacted due to the code change in `setY()`, no callers of `createLocal()` will be indirectly impacted by that code change. So the propagation of indirect impact effectively stops at `createLocal()`.

2.2.5. Impact analysis and testing

Impact analysis does more than identify directly modified code for testing. Test Advisor uses impact analysis to identify code that is both directly and indirectly impacted by changes, which allows you to identify and test indirectly impacted code and to avoid testing gaps that arise from limiting your testing to directly modified code.

Suppose the code example from previous sections was modified in the following erroneous way (perhaps due to copy-and-paste errors):

```
// BEFORE                         // AFTER
struct Coords {                   struct Coords {
    ...                               ...
    int getY() {                      int getY() {
        return y;                         return x;       // CHANGED
    }                                 }
    ...                               ...
    void incrementY() {               void incrementY() {
        y += delta;                       x += delta;     // CHANGED
    }                                 }
    ...                               ...
    void setY(int n) {                void setY(int n) {
        y = n;                            x = n;          // CHANGED
    }                                 }
    ...                               ...
```

The code for the `struct Coords` now uses field `x` for operations that should instead use field `y`.

Suppose the code example includes a unit test:

```
void coordsTest() {
    Coords g;
    g.setX(4);
    assert(g.getX() == 4);
    g.incrementX();
    assert(g.getX() == 7);
    g.setY(2);
    assert(g.getY() == 2);
    g.incrementY();
    assert(g.getY() == 5);
}
```

This test executes all the changed code, but it fails to detect the incorrect implementation. A testing policy that only requires modified code to be tested would indicate that this test is sufficient, but this example shows that such a policy would not catch the testing deficiency.

Recall that the code example also includes the following functions:

```
void rewriteY(Coords &g, int v) {
    g.setY(v);
}

void rewrite(Coords &g, int x, int y) {
    rewriteX(g, x);
    rewriteY(g, y);
}
```

Even though these functions are not modified by the code change, they are indirectly impacted. A testing policy that requires impacted code to be tested would necessarily require such functions to be tested to make sure that the code change is correct. A programmer could then write a test similar to the following one:

```
void rewriteTest() {
    Coords g;
    rewrite(g, 4, 2);
    assert(g.getX() == 4);
    assert(g.getY() == 2);
}
```

This test would catch a testing deficiency that would be missed without the guidance of impact analysis.

2.2.6. Function summary details

If you are familiar with function models, it can help to know that the function summaries used by Test Advisor are a subset of the models used by the Coverity Analysis engine.

Function summary

The function summary of a function consists of a set of model actions. Each model action represents an operation that the function is found to perform on an interface (one of the parameters to the function, or the return value, or in some cases where the interface is unspecified but implicitly understood). The same action can appear multiple times in a function summary operating on different interfaces. A change in a function summary is caused by a change in the model actions that make up the function summary. Either actions are added, or actions are deleted, or both.

Function models are discussed in "Models, Annotations, and Primitives" in the *Coverity 2020.12 Checker Reference*.

Test Advisor assesses the following model actions when determining the function summary for a function:

- ACCESSDBUFFER: The function might access a parameter array or pointer at a particular index.
- ALLOC: The function might return allocated memory that the caller should eventually deallocate.
- BUFFERALLOC: The function might return an allocated array that the caller should eventually deallocate.
- DEREFERENCE: The function might dereference a parameter.
- ESCAPE: The function might store a copy of a parameter.
- FORMAT_STRING_SINK: The function might use a parameter as a `printf`-style format string.
- FREE: The function might free a parameter.
- IDENTITY: The function might return a parameter without modifying it.
- INDEXDYNAMIC: The function might use a parameter to index into an array.
- KILL: The function might terminate program execution.
- MAYREADNONPTRFIELDS: The function might read a non-pointer field of a parameter.
- NEGATIVE_RETURN: The function might return a negative value.
- NEGATIVE_SINK: The function might pass a parameter to a function that cannot accept a negative number.
- PTR_ARITH: The function might perform pointer arithmetic on a parameter.
- READ: The function might read a parameter value or field.
- RETURNSNULL: The function might return null.
- STRING_NULL_ARGUMENT: The function might fill an argument with a non-null-terminated string.
- STRING_NULL_RETURN: The function might return a non-null-terminated string.
- STRING_NULL_SINK: The function might pass an argument to a function that expects a null-terminated string.
- STRING_SIZE_RETURN: The function might return a string of unknown size.

- STRING_SIZE_SINK: The function might pass an argument to a callee whose behavior depends on its size.
- THROW: The function might throw an exception.
- UNCAUGHT_EXCEPT: The function might not catch an exception that is thrown during its execution.
- USE_HANDLE: The function might use a parameter as a handle.
- WRITE: The function might write a parameter value or field.
- ZERO_RETURN: The function might return zero.

For the purpose of impact analysis, an obligation is assumed to exist if there is at least one feasible path through the function that implies that obligation. (In contrast, when analyzing for static analysis defects, obligations are kept separately for each path.) For example:

```
// BEFORE             // AFTER
int foo(int x) {        int foo(int x) {
                            if(x > 0) { return 42; }  // CHANGED
    return 0;               return 0;
}                       }
```

In both cases, the function `foo()` will have a ZERO_RETURN action, since it might return 0 in both versions.

Due to the context insensitivity of function summaries, some changes to functions will not be interpreted as causing function summary changes. For the example above, even though the observable behavior of function `foo()` has changed due to this code change, this is not reflected in the function summary because of context insensitivity. Thus such a code change will not cause callers of `foo()` to be indirectly impacted.

Chapter 2.3. Impact analysis limitations

Table of Contents

This section describes some of the limitations of impact analysis that you might encounter in normal use.

2.3.1. Direct impact limitations

Because direct impact is determined by a change in the abstract syntax tree for a function, any code change that does not cause a change in the abstract syntax tree will not cause a direct impact (see an example in Section 2.3.3, “Traceability limitation for recursive functions”). However, note that code changes within the function body can generally be expected to cause a change in the abstract syntax tree.

Conversely, a code change that changes the abstract syntax tree but that does not change the behavior of the function at all will cause a direct impact where a user might not expect it. For example:

```
// BEFORE                   // AFTER
if(x == 0) {                if(x != 0) {    // CHANGED
    foo();                      bar();      // CHANGED
}                           }
else {                      else {
    bar();                      foo();      // CHANGED
}                           }
```

2.3.2. Indirect impact limitations

Because indirect impact is computed using the function summaries, limitations related to the function summaries also become limitations of indirect impact. As noted, function summaries ignore context (only a single path that exercises a model action is required for that action to be present in the function summary). Thus, a code change that only affects the path on which the model actions occur (and therefore do not affect the the model actions themselves), will not cause function summaries to change, so no indirect impact will be detected in such a case.

Note that the model actions used for function summaries are limited. Changes that affect code outside of these model actions will not be taken into account when computing indirect impact. For instance, the code example changes in Section 2.2.2, “Indirect impact” are examples of code changes that do not cause indirect impact because the function summaries do not change.

2.3.3. Traceability limitation for recursive functions

While it is usually possible to trace impact from indirectly impacted functions to directly impacted sources, in some cases this cannot be done. Specifically, such situations arise if all of the following conditions are true:

- The code change is not identified as causing direct impact.
- The code change causes function summaries to change.
- The function summary changes propagate through a cycle of recursive functions.

Consider the following pieces of a C++ code example:

```
// BEFORE                            // AFTER
struct q {                           struct q {
    char c;                              int c;           // CHANGED
};                                   };
int recur(int x) {                   int recur(int x) {
    if(sizeof(q) > 2) {                  if(sizeof(q) > 2) {
        return x ? recur(x-1) : 1;           return x ? recur(x-1) : 1;
    }                                    }
    return 0;                            return 0;
}                                    }
```

The code change will not be identified by Test Advisor as causing direct impact in the function `recur()`. However, the code change modifies the function summary of `recur()` (because it no longer has a ZERO_RETURN action). Because `recur()` now has a different function summary, it is identified as an indirectly impacted function. However, there is no way to trace from `recur()` to a directly-impacted function that is the source of the indirect impact observed in `recur()`.

Because this effect can only occur with recursive functions, it is fairly easy to identify when this exception to the traceability of impact occurs (one will trace through an entire cycle back to an already visited function).

Chapter 2.4. Using Impact analysis in Test Advisor

Table of Contents

Test Advisor allows you to include or exclude files, functions, and lines of code from the policy based on the result of impact analysis. For basic usage, see also Section 1.2.4, “Getting impact analysis results”.

2.4.1. Impact Dates

Test Advisor tracks the last time each function in the code base was observed to be directly impacted, indirectly impacted, or have its function summary changed. Each of these dates is tracked separately for every function. Thus, each function has a direct impact date, an indirect impact date, and a function summary change date. Because a function is impacted if it is either directly impacted or indirectly impacted, each function also has an impact date, which is the latest of the direct impact date and the indirect impact date for the function.

Each of these three dates associated with each function is defined as the code version date of the most recent analysis in which the corresponding property changed. The code version date is set using the `--code-version-date` option to cov-analyze. If this option is not specified when `cov-analyze` runs, the date of the cov-build invocation is used by default to construct the intermediate directory being analyzed.

Additionally, Test Advisor also computes an optional impact date for individual lines. The impact date for a line is the latest date that a change occurred to the function summary of a function called on that line. If the line contains no function calls, it does not have an impact date.

2.4.2. Impact Filters

The Test Advisor policy language provides three filters based on impact dates:

- RecentlyImpactedLineFilter: Includes/excludes lines based on the impact date for the line. For more detail, see Section 1.5.6.11, “RecentlyImpactedLineFilter”.
- RecentlyImpactedFunctionFilter: Includes/excludes functions based on the impact date for the function. For more detail, see Section 1.5.5.20, “RecentlyImpactedFunctionFilter”.
- RecentlyImpactedFileFilter: Includes/excludes files based on the impact date for the latest impacted function in the file. For more detail, see Section 1.5.4.10, “RecentlyImpactedFileFilter”.

These filters all use the attribute `recently_impacted`, which takes a value of `true` to indicate that only objects recently impacted should be included, or `false` to indicate that only objects not recently impacted should be included. The comparison is always performed using the `recent_date_cutoff` value in the policy file.

For example, the following policy specifies that all functions that are impacted on or after 2014-01-01 should be 100% covered. Such functions that are not 100% covered will result in a Test Advisor violation:

```
{
   type: "Coverity test policy definition",
   format_version: 1,
   recent_date_cutoff: "2014-01-01",
   old_date_cutoff: "2013-01-01",
   rules: [
     {
       violation_name: "NEEDS_TEST",
       aggregation_granularity: "function",
       minimum_line_coverage_pct: 100,
       function_filters: [
         { recently_impacted: true }
       ]
     }
   ]
}
```

This policy means that every function that was syntactically modified since the specified date (directly impacted), or whose behavior changed as a result of a change elsewhere since that date (indirectly impacted), is required to have all of its lines covered by tests. As noted in the introduction, it is insufficient to simply test the modified code because code changes can cause significant behavioral changes in unmodified code, which might be untested. Coverity recommends that you use policies such as this, instead of policies based on simple SCM modification dates, to avoid having blind spots (gaps) in your testsuite.

2.4.3. Direct Impact Filters

The Test Advisor policy language provides two filters based on direct impact dates:

- RecentlyDirectlyImpactedFunctionFilter: Includes/excludes functions based on the direct impact date for the function. For more detail, see Section 1.5.5.21, “RecentlyDirectlyImpactedFunctionFilter”.
- RecentlyDirectlyImpactedFileFilter: Includes/excludes files based on the direct impact date for the latest impacted function in the file. For more detail, see Section 1.5.4.11, “RecentlyDirectlyImpactedFileFilter”.

These filters both use the attribute `recently_directly_impacted`, which takes a value of `true` to indicate that only objects recently impacted should be included, or `false` to indicate that only objects not recently impacted should be included. The comparison is always performed using the `recent_date_cutoff` value in the policy file.

For example, the following policy specifies that all functions that are directly impacted on or after 2014-01-01 should be 100% covered. Such functions that are not 100% covered will result in a Test Advisor violation:

```
{
   type: "Coverity test policy definition",
   format_version: 1,
   recent_date_cutoff: "2014-01-01",
   old_date_cutoff: "2013-01-01",
   rules: [
     {
       violation_name: "NEEDS_TEST",
       aggregation_granularity: "function",
       minimum_line_coverage_pct: 100,
       function_filters: [
         { recently_directly_impacted: true }
       ]
     }
   ]
}
```

This policy means that every function that was syntactically modified since the specified date (directly impacted) is required to have all of its lines covered by tests. As noted in the introduction, it is insufficient to simply test the modified code because code changes can cause significant behavioral changes in unmodified code, which might be untested. Coverity recommends that you use policies such as this, instead of policies based on simple SCM modification dates, to avoid having blind spots (gaps) in your testsuite.

Chapter 2.5. Regeneration of impact history

Because impact analysis relies on keeping a history of the last time the functions in the codebase are directly impacted, indirectly impacted, or have their function summaries changed, any loss of or damage to this history may result in incorrect results from impact analysis.

For example, consider the basic impact analysis flow described in Section 1.2.4, “Getting impact analysis results”. Impact history is associated with a stream in Coverity Connect, and downloading history from that stream before each analysis is required. If the user mistakenly specifies the wrong stream name when performing this download, the result of subsequent analysis runs may be incorrect, since a portion of the impact history has been corrupted by the mistake. Functions may be indicated as impacted when they should not be, or vice versa, due to the incorrect impact history.

To recover from lost or damaged impact history, the history can be regenerated by following these steps:

1. Let D be the date one second prior to the `recent_date_cutoff` from the Test Advisor policy file used.

2. Checkout a revision of the codebase to be analyzed from date D.

3. Build/capture/analyze/commit as normal, including import of SCM data if needed, but do not download history from Coverity Connect. Analysis should use a `--code-version-date` of D; that is, the `cov-analyze` command line should look like:

   ```
   cov-analyze --dir <idir> --code-version-date <D> [other options] ...
   ```

4. Checkout the latest revision of the codebase.

5. Build/capture/analyze/commit as normal, including downloading the history that was committed in Step 3. The `--code-version-date` used for `cov-analyze` should be the date associated with the latest version of the codebase.

The above assumes that only the analysis results from the latest revision are required. If analysis results for several different codebase revisions are required, steps 4 and 5 should be repeated for each desired revision. These steps should be performed in order of increasing `--code-version-date` (i.e. from earliest to latest). All commits should use the same stream, so that the impact history downloaded in step 5 should come from the immediately preceding commit.

If multiple TA policy files are used, the entire flow (steps 1-5) should be repeated for each policy file.

Note that the procedure creates an impact history where all functions impacted before the `recent_date_cutoff` are identified as being impacted on date D. While the actual dates those functions were impacted on may be earlier, this does not matter for the purposes of the Test Advisor analysis, since all functions impacted before the `recent_date_cutoff` are treated the same by the impact filters in the policy. Likewise, all functions impacted after the `recent_date_cutoff` will be treated the same, regardless of the specific impacted date for those functions.

Part 3. Test Advisor commands

This section includes reference information for new Test Advisor commands and existing commands that have new Test Advisor options.

Chapter 3.1. cov-analyze

Analyze an intermediate directory for quality and security defects and policy test violations.

For more information, see `cov-analyze`

Chapter 3.2. cov-build

Intercepts all calls to the compiler invoked by the build system.

For more information, see `cov-build`.

Chapter 3.3. cov-emit-server

Allows for the collection of coverage from remote machines. This command can also be used on a local machine to improve performance for some tests.

For more information, see `cov-emit-server`.

Chapter 3.4. cov-extract-scm

Extracts the change data for each line from files in the SCM system.

For more information, see `cov-extract-scm`.

Chapter 3.5. cov-import-scm

Collects change data for source files from the SCM.

For more information, see `cov-import-scm`.

Chapter 3.6. cov-manage-emit

Manage an intermediate directory.

For more information, see `cov-manage-emit`.

Chapter 3.7. cov-manage-history

Manages historical data needed by Test Advisor.

For more information, see `cov-manage-history`.

Chapter 3.8. cov-patch-bullseye

Patch a Bullseye small runtime for use with Test Advisor.

For more information, see `cov-patch-bullseye`.

Appendix A. Additional Test Advisor usage notes

Table of Contents

A.1. Notes for coverage tools

This section provides miscellaneous notes for the specific coverage tools supported by Test Advisor.

A.1.1. Wind River VxWorks with Bullseye

The following notes list information that you should be aware of if you are using Bullseye with Wind River VxWorks:

- The Bullseye coverage tool supports Wind River VxWorks for Real Time Processes (RTPs), so Test Advisor can be used in this situation. However, VxWorks Kernel Modules are not supported by Bullseye. If you want to use VxWorks Kernel Modules, then you must use the Wind River coverage tool. For more information, see Section A.1.2, “Handling Wind River VxWorks coverage run files with CoverageScope”.

- As noted in the Bullseye documentation, VxWorks is supported by automatically compiling in the small footprint runtime. For more information, see http://www.bullseye.com/help/env-qnx.html.

  However for the purposes of building with `cov-build`, the small runtime should not be specified here. To build a VxWorks RTP project the cov-build command will look similar to the following:

  ```
  % cov-build --dir idir --c-coverage bullseye --bullseye-dir /path/to/bullseye make
  ```

- Because the tests are run on a simulator (like `VxSim`), `cov-build` cannot be used, so coverage must be manually added to the intermediate directory.

- When running the instrumented test binaries, `BullseyeCoverage.data-<pid>` files will be produced. When the test binaries are run from the IDE, these are usually created in the your temporary directory.

  See Bullseye small runtime: for specific instructions about how to manually add this coverage data to the intermediate directory.

A.1.2. Handling Wind River VxWorks coverage run files with CoverageScope

This section describes how you can use Test Advisor with Wind River VxWorks using the CoverageScope coverage tool.

This section describes two methods for running Test Advisor with VxWorks:

- Through the Wind River WorkBench IDE

- Using command line options

Both methods build off the following basic work flow for VxWorks test simulation:

Basic VxWorks Workflow:

1. Build application in WorkBench.

2. Launch the simulator (`vxsim`).

3. Test the application by running it on the simulator.

A.1.2.1. Using Test Advisor with VxWorks through WorkBench

Test Advisor can be added into the basic VxWorks work flow as follows:

1. Build the application under the `cov-build` command to produce an intermediate directory. The most effective way to do this is to build using the Makefiles generated by WorkBench:

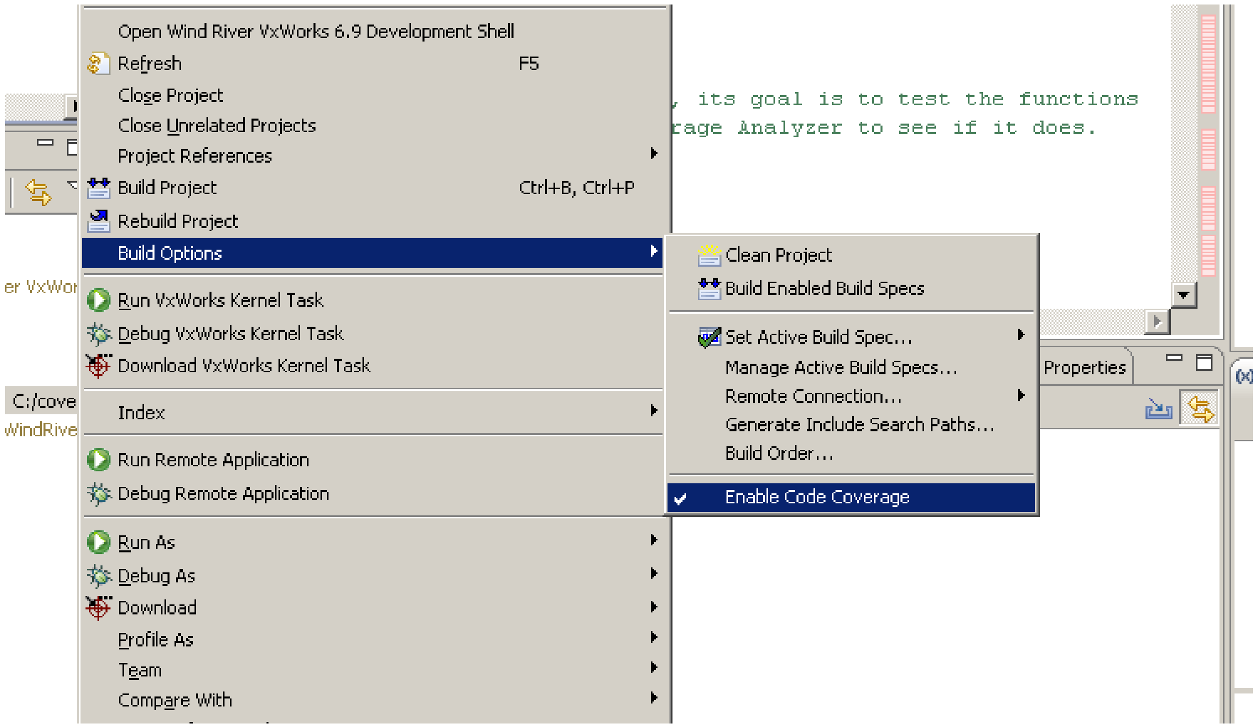
   ```
   % cov-build --dir idir make
   ```

   Note

   Make sure that this is a regular, non-coverage build.

2. Enable CoverageScope so that it can collect coverage data in WorkBench by selecting *Enable Code Coverage*.

   Figure A.1. Enable code coverage

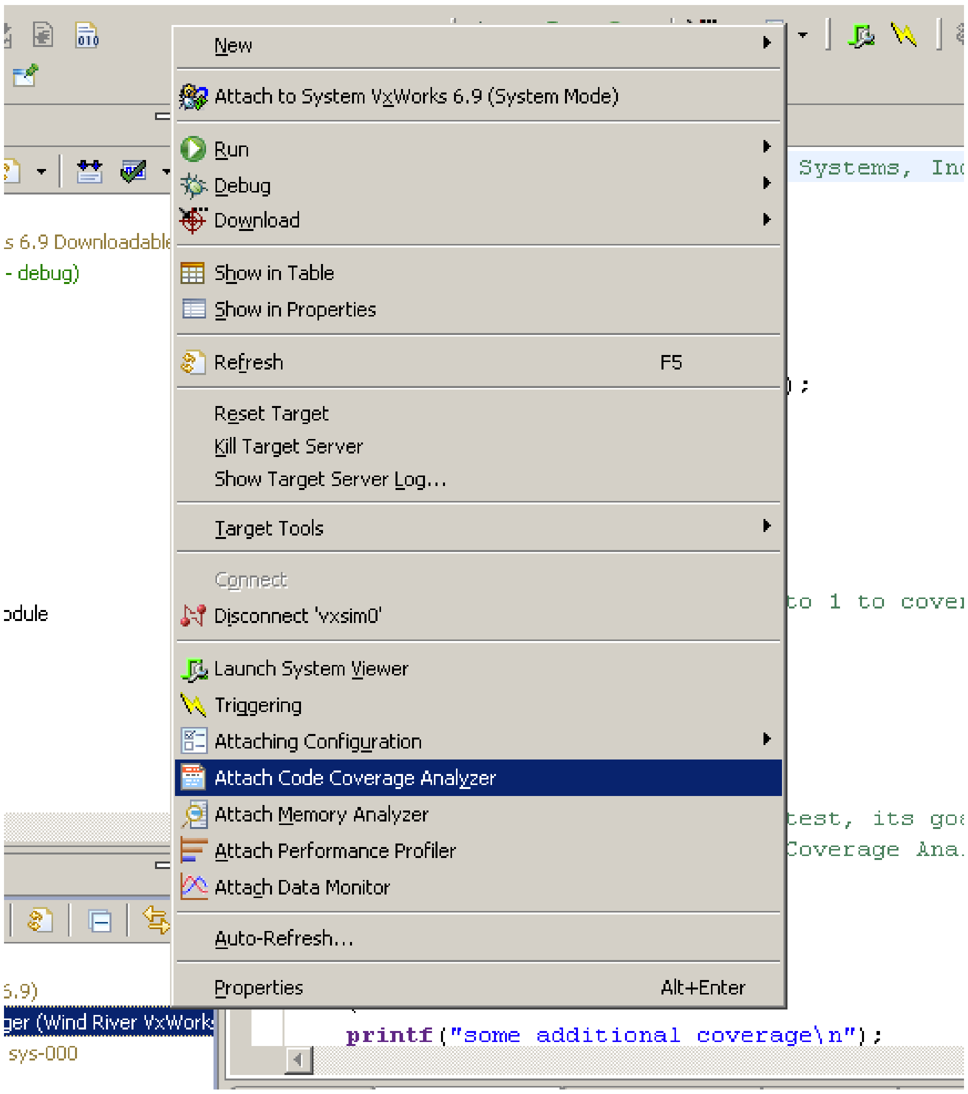


3. After you enable code coverage, be sure to rebuild your application.

   The application is now ready to be run to gather coverage.

4. Start the simulator in the same way that you would in your regular flow, but additionally attach the *Code Coverage Analyzer* to the simulator.

Figure A.2. Code coverage analyzer



```
printf("some additional coverage\n");
```

5. After the analyzer has been attached, run the test application.

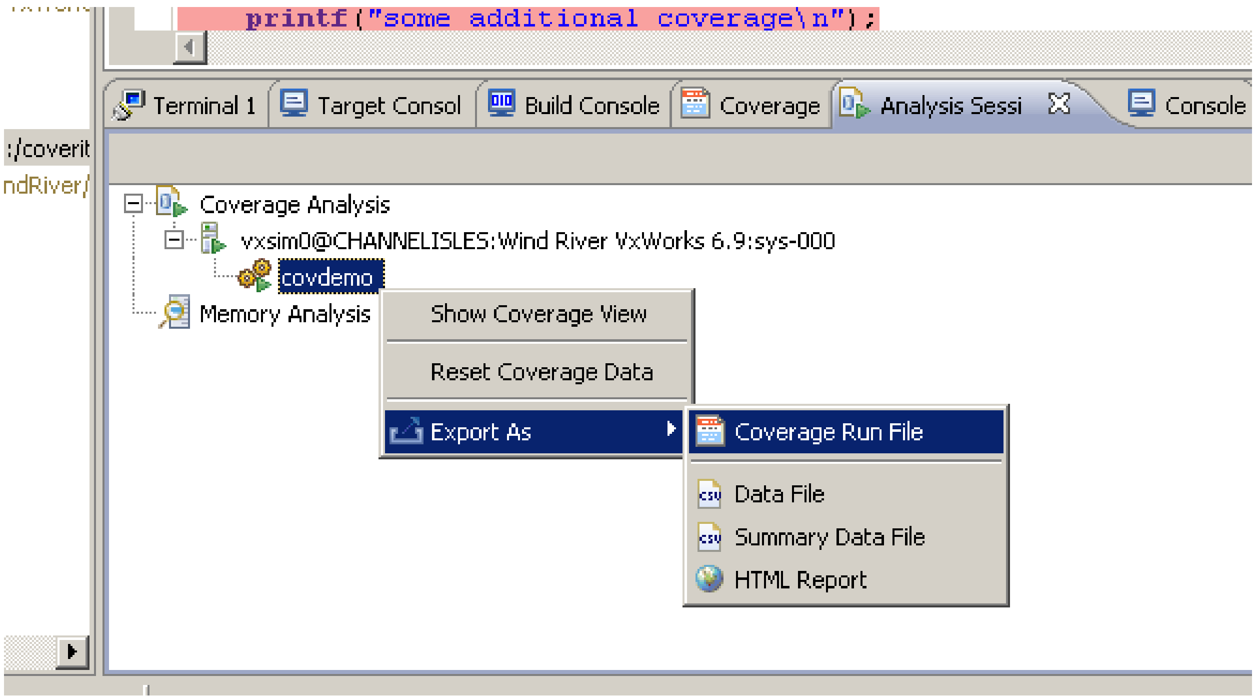
Note

The way in which tests are run most likely differ from user to user, so run them as you as you would normally as outlined in the VxWorks basic work flow.

6. After the test run is complete, export the coverage data into a format that the Coverity tools can recognize.

   To do so, export the coverage as a *Coverage Run*.

   Figure A.3. Coverage run



```
printf("some additional coverage\n");
```

   This creates a `.run` file with the required data.

7. Lastly, add the coverage data into the Coverity intermediate directory:

```
% cov-manage-emit --dir idir add-coverage --windriver-run <path-to.run>
```

   You can use the `--suitename`/`--testname` arguments to help connect the coverage data to a specific test name.

   At this point, the intermediate directory contains the coverage information for the test that was run. You can now continue to use the normal Test Advisor flow of gathering SCM data, running the analysis, and committing the results from this intermediate directory. For more information, see Chapter 1.2, *Getting Started with Test Advisor*.

A.1.2.2. Using Test Advisor with VxWorks through the command line

Additionally, it is also possible to run the process through the command line. Note that these steps do not include running the `cov-build` command to generate the intermediate directory. It is the same command line operation as described in Section A.1.2.1, “Using Test Advisor with VxWorks through WorkBench”. However, it is important to note that other commands might vary from project to project, so make sure of the following:

- Ensure that your build has *Code Coverage* enabled.
- Ensure that the *Code Coverage Analyzer* is attached to your simulator before running your tests.

The remaining steps to run your tests should not change. After the test runs are complete, you can download the `.run` file with the `coverageupload` tool. For example:

```
% coverageupload -t <target_server> -p <full_path_to_project.prj> -f <path-to.run>
```

After you retrieve the `.run` file, adding the coverage is the same `cov-manage-emit` command as mentioned in the previous section. For example:

```
% cov-manage-emit --dir idir add-coverage --windriver-run <path-to.run>
```

Note that you will need to generate an intermediate directory (built without code coverage enabled) before you can add the new coverage data.

A.1.3. Gcov and flushing coverage

By default Gcov will not flush its collected coverage to disk unless there is a clean exit (for example, a call to `exit()`). This can be a problem if you have a long running daemon that does not get shutdown cleanly (for example, by a forced shutdown with a KILL signal).

This can be solved by ensuring that the process flushes coverage before shutting down (for example, doing so in a signal handler). You can use the provided test separation API for this situation.

The code can be modified to call `__gcov_flush()` directly, however the benefit of using the COVERITY_FLUSH_COVERAGE() macro is that this will only be compiled in when it is built by `cov-build` with gcov coverage enabled and it will properly flush the coverage when this macro is called.

For more information on the test separation API see Section 1.6.1.2.1, “Using the test boundary API”.