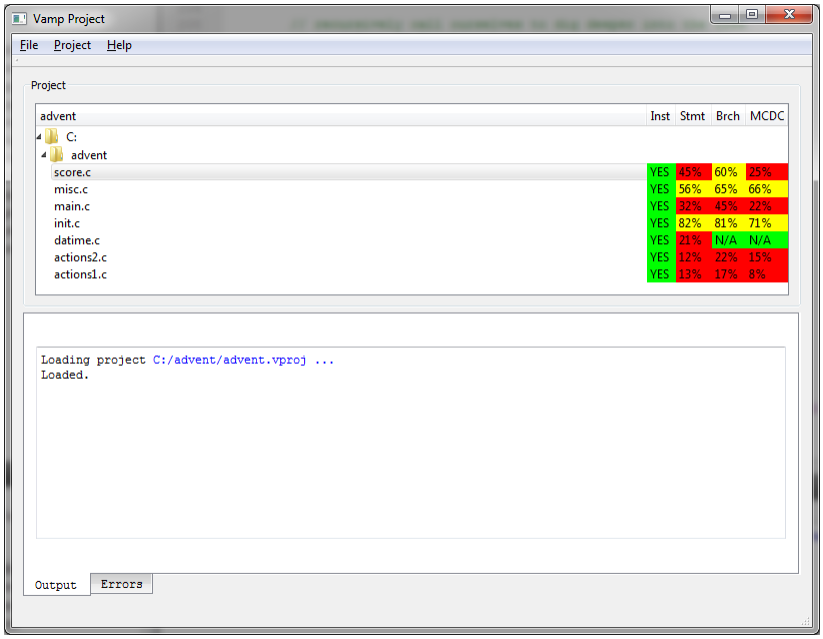
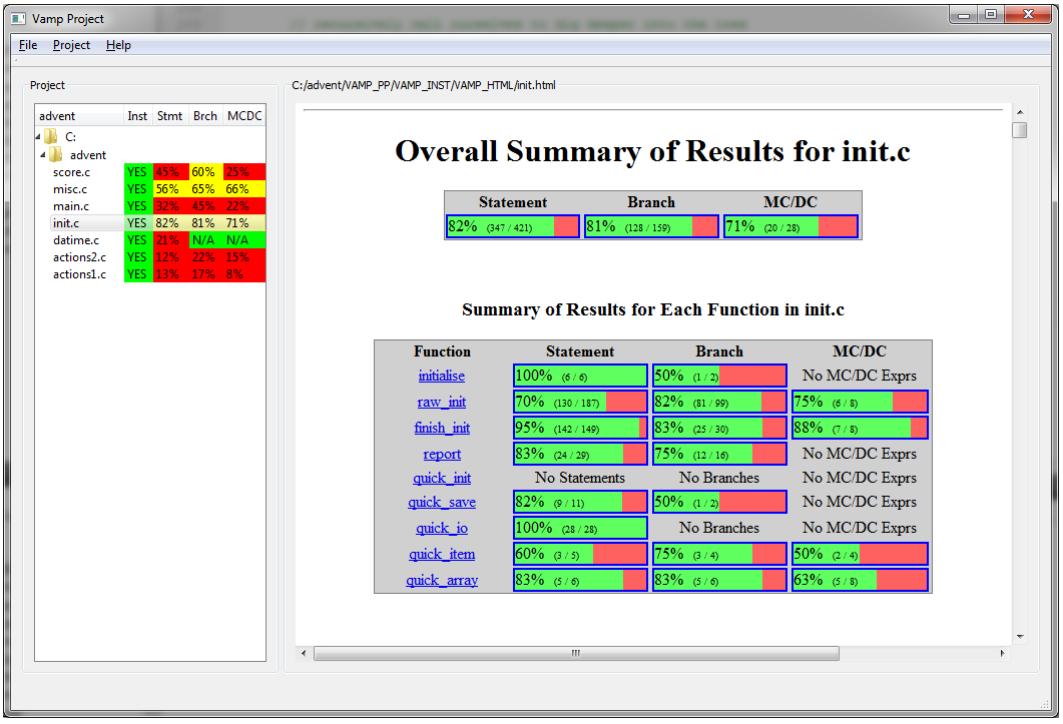
The VAMP Graphical User Interface (VampGUI)

As **Vamp** simplifies the process of Software Coverage Analysis (SCA), the **Vamp Graphical User Interface** (**VampGUI**) simplifies the management of SCA for an entire project. **VampGUI** provides an at-a-glance view of the current SCA status of a project.



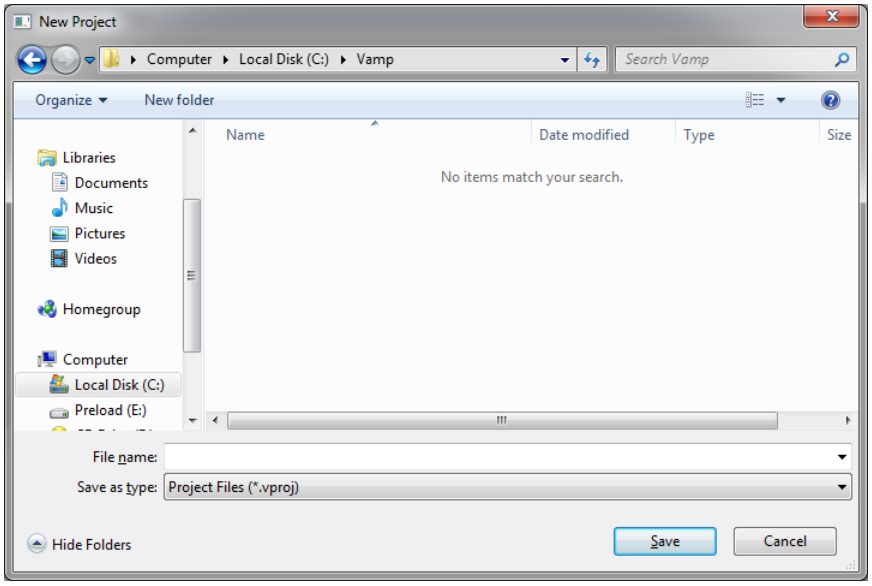
Coverage reports can be viewed by double-clicking on a file in the project list.



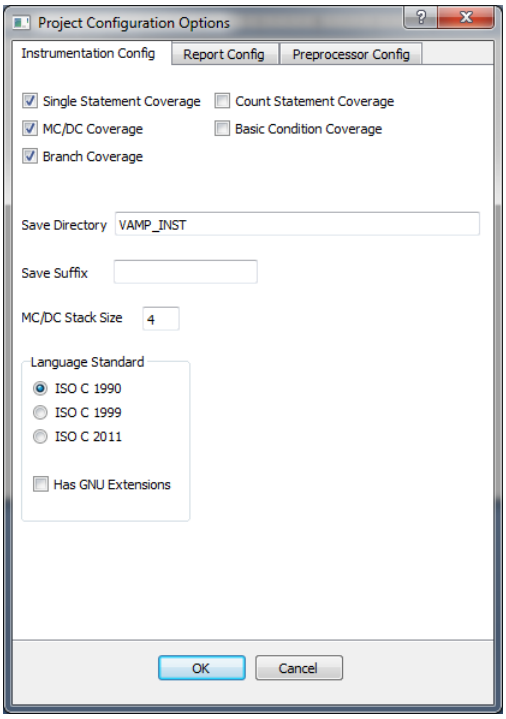
Creating a New Project

The following steps allow for the creation of a new project.

Open the VampGUI executable. From the *Project* menu, select New Project. The following display appears:



Navigate to the desired project directory and enter a project name in the *File Name:* box. A dialog opens allowing project configuration options to be selected:



The first configuration tab, *Instrumentation Config*, controls the instrumentation of the source code. The types of instrumentation supported are:

1. *Single Statement Coverage* – Perform basic statement coverage. The report will show if a statement has been covered.
2. *Count Statement Coverage* – Another form of statement coverage. The report will show if a statement has been covered, and the number of times the statement was reached during the current execution run (mutually exclusive with option 1).
3. *MC/DC Coverage* – Modified Condition Decision Coverage is performed.
4. *Basic Condition Coverage* – Condition Coverage (sometimes called Predicate Coverage) is performed (mutually exclusive with option 3).
5. *Branch Coverage* – Branch Coverage is performed.

By default, Single Statement Coverage, MC/DC Coverage and Branch Coverage are selected.

Additional instrumentation options include:

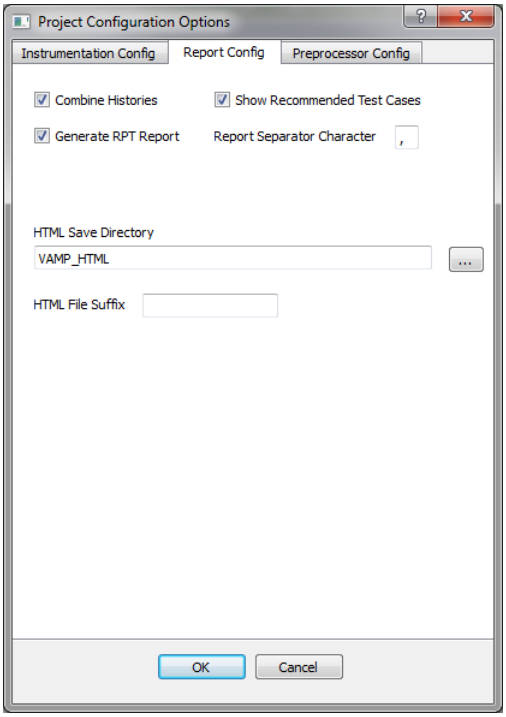
Save Directory – The name of the directory where the instrumented source file and database description files are to be placed. This directory can be either a fixed path or a path relative to where the preprocessed source files are kept. By default, this directory is called **VAMP_INST**.

Save Suffix – The suffix to add to the end of the source filename. For example, with a suffix of **_inst**, instrumenting the file **test.c**, will generate the files **test_inst.c** and **test_inst.json**. By default, no suffix is added.

MC/DC Stack Size – This defines the maximum stack size for MC/DC expressions. Whenever a MC/DC condition contains a function call, all accumulated MC/DC expression collection information is placed on a stack until the function returns. If the function called is in the same source file as the condition, and the function itself has a MC/DC expression containing a function call, the stack depth is increased. This is a relatively rare occurrence, so normally a low value for the stack size is required. There may be an occasion, such as in a recursive function, where the stack size needs to be increased. A warning message is displayed in the report if the stack overflows, in which case data collection for that file ceases for the remainder of the current execution run.

Language Standard – Determines the ISO C Language standard to use (C 1990, C 1999 or C 2011). A checkbox is available for allowing GNU C extensions. By default ISO C 1990 is selected with no GNU extensions.

The *Report Config* tab allows for the selection of instrumentation report options.



These options are:

Combine Histories – Combine current execution history with previous execution histories.

Show Recommended Test Cases – Display a list of recommended MC/DC test cases that can be used to generate full MC/DC coverage.

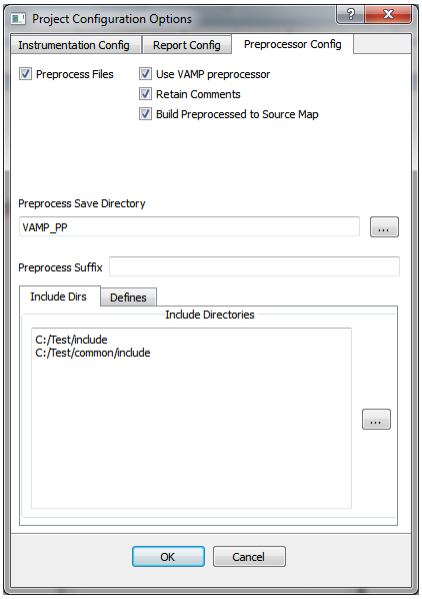
Generate RPT Report – Generates a text-based report summary (.rpt) file for each instrumented file.

Report Separator Character – Character to use for separating fields in the report summary.

HTML Save Directory – Directory in which the **.html** and **.rpt** report files are saved. This directory can be either a fixed path or a path relative to where the instrumented source files are kept. By default, this directory is called **VAMP_HTML**.

HTML File Suffix – Suffix to add to filename when generating **.html** and **.rpt** report files. For example, with the suffix "**_vamp**", processing **test.json** will generate the files **test_vamp.html** and **test_vamp.rpt**.

The *Preprocessor Config* tab allows for selection of preprocessor options.



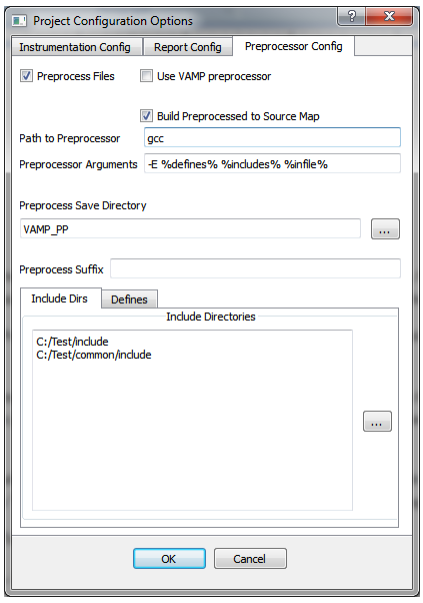
Vamp requires that all source code be preprocessed. As part of the instrumentation process, the source code is rewritten to capture what code is executed, which branches are taken, and the results of conditionals. Thus macros and include files containing such constructs must first be expanded. Three preprocessor approaches are available:

1. The files used in the project are already preprocessed. In this case, the *Preprocess Files* checkbox should be unchecked. The remaining information in this tab will be hidden and may be ignored.
2. Use the internal Vamp preprocessor to handle all code preprocessing. In this case, both the *Preprocess Files* checkbox and *Use VAMP Preprocessor* checkbox should be checked. With the Vamp preprocessor it is possible retain all comments from the original file by checking the *Retain Comments* checkbox. This makes it easier to read the preprocessed source code when necessary. When the *Build Preprocessed to Source Map* checkbox is checked, a database file is built that maps the original source code statement line numbers to the corresponding preprocessed line numbers. When a report is generated, both sets of line numbers are displayed. The database file is generated in the same directory as the preprocessed file, with a suffix of ".ppmap".

   The *Include Dirs* tab contains a list of all include directories to search for **#include** directives.

   The *Defines* tab contains a list of all macro definitions to define as part of the preprocessing.

   See the next section for definitions of both the *Preprocess Save Directory* box and the *Preprocess Suffix* box.



3. Use an external preprocessor to handle all code preprocessing. In this case, the *Preprocess Files* checkbox should be checked while the *Use VAMP Preprocessor* checkbox should be unchecked.

   The *Path to Preprocessor* box should be completed with the path to the desired preprocessor executable.

   The *Preprocessor Arguments* box should be completed with any arguments that need to be passed to the preprocessor. The following macro abbreviations will be replaced with their equivalents:

   %includes% will be replaced with each of the directories listed in the *Include Directories* tab prefixed with "-I". For example in the example above, %includes% will be replaced with:

   -IC:/Test/include -IC:/Test/common/include

   Click on the … button to browse for an include path to add.

   %defines% will be replaced with each of the defines listed in the *Defines* tab prefixed with "-D".

   %infile% will be replaced with the name of the source file to be preprocessed.

   The *Preprocess Save Directory* box should contain the path to where the preprocessed output file will be placed. This may be a path relative to the source file directory or an absolute path. By default, this is the relative path "VAMP_PP".

   The *Preprocess Suffix* box may contain a suffix to add to the end of the source filename. For example, with a suffix of **_pp**, preprocessing the file **test.c**, will generate the file **test_pp.c**. By default, no suffix is added.

When the *Build Preprocessed to Source Map* checkbox is checked and the preprocessor has generated linemarkers, a database file will be built that maps the original source code statement line numbers to the corresponding preprocessed line numbers. When a report is generated, both sets of line numbers are displayed. The database file is generated in the same directory as the preprocessed file, with a suffix of ".ppmap". For a compiler such as GCC, passing the -E directive will cause the specified source file to be preprocessed and also to have linemarkers generated. Passing the directives -E -P will preprocess without linemarkers. In this case, the *Build Preprocessed to Source Map* checkbox should be unchecked.

Now that the project configuration pages are complete, click on OK. You can edit the configuration from the *Project->Edit Configuration* menu.

The next step is to add files to the project which will later be instrumented. Go to the *Project->Add* menu or right-click on the Project pane and select *Add Files*. Select one or more files to add to the project then click on *OK*. The selected files will be added to the project. You can click on the first file, shift-click on the last, and control-click to add or remove other files within the current directory. Do this for each directory, and you will soon have a tree structure built with the desired files to instrument.

Save the project with *Project->Save Project*. This will create a **<*project*>.vproj** file (where <*project*> is the project name you selected). The files **vamp.cfg** and **vamp_process.cfg** files will be created in the same directory. These files contain the instrumentation configuration and report configuration information. Details on these files are found in a later section.

Choosing a Directory Structure

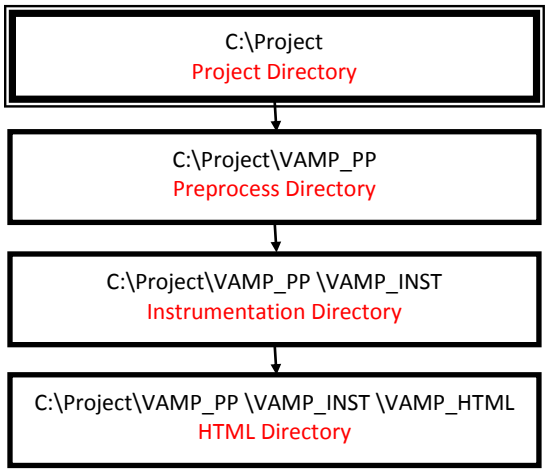
When creating a project, it is important to decide where the generated files will be placed. VAMP provides the option of using either relative paths or absolute paths. If your source files all exist in a single directory, using relative paths may be the best choice. If your source files exist across multiple directories, you may prefer to use absolute paths.

The following paths from the configuration menu may be selected:

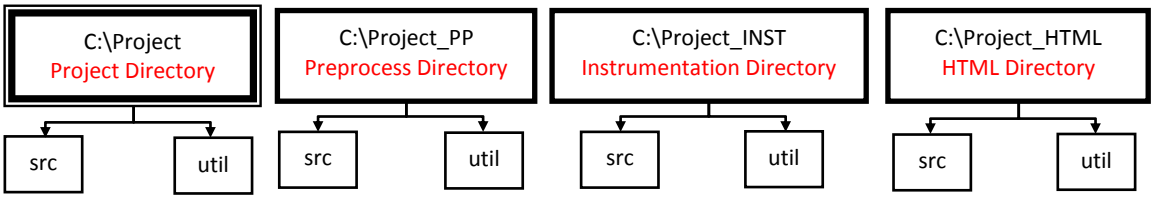
- Instrumentation Directory – Shown as *Save Directory* on the *Instrumentation Config* tab, this is the directory where instrumented files will be placed.
- Preprocess Save Directory – Shown on the Preprocessor Config tab when *Preprocess Files* is selected, this is where preprocessed files will be placed.
- HTML Save Directory – Shown on the History/Report Config tab, all report files are saved here.
- History Directory - Shown on the History/Report Config tab, history files can be found here.

When working with relative paths, the following directory relationships exist:

- Instrumentation Directory – relative to the Preprocess Save Directory when *Preprocess Files* is checked, otherwise relative to the project directory.
- Preprocess Save Directory – relative to the project directory.
- HTML Save Directory – relative to the Instrumentation Directory.
- History Directory - relative to the Instrumentation Directory.

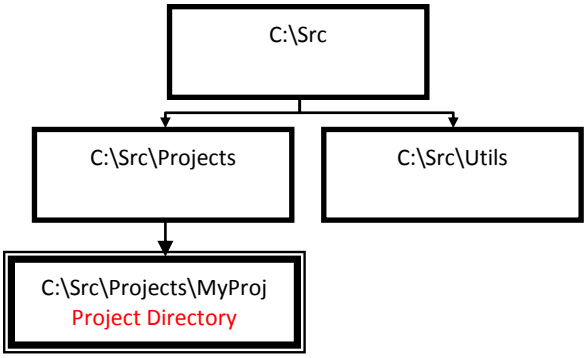


Consider absolute paths as parallel directories. For example, if absolute paths are selected for each type of directory:

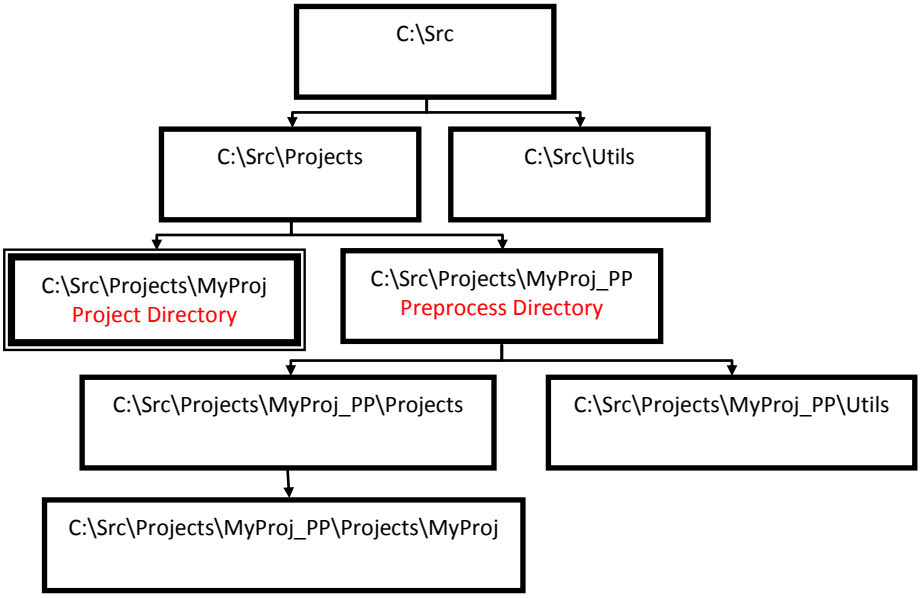


The advantage here is that the build configuration can be copied to the C:\Project_INST directory and an instrumentation build can be created separate from the source build.

The above example assumes the project directory is above all source directories. Consider the following structure:

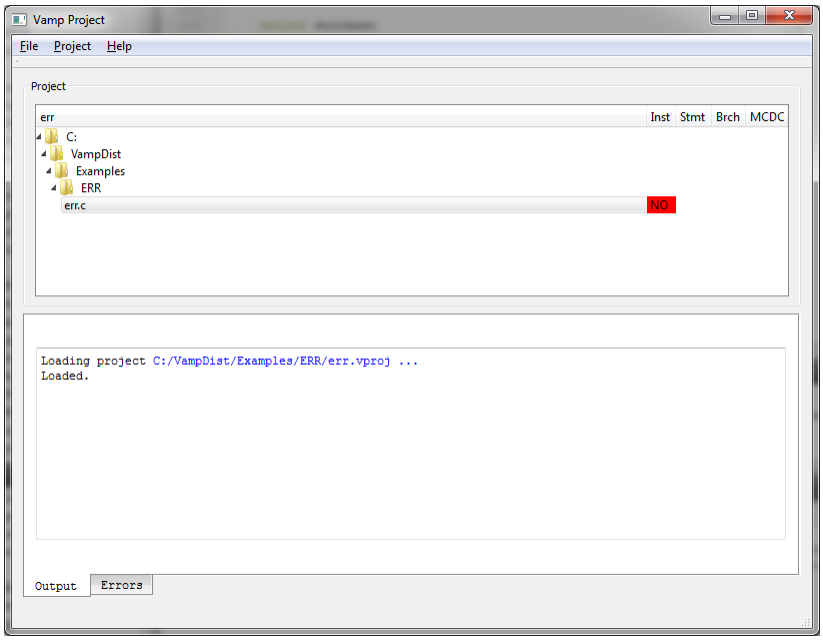


Source files are found in both C:\Src\Projects\MyProj and C:\Src\Utils. If we create an absolute directory path for preprocessing at C:\Src\Projects\MyProj_PP, the following structure will be generated:



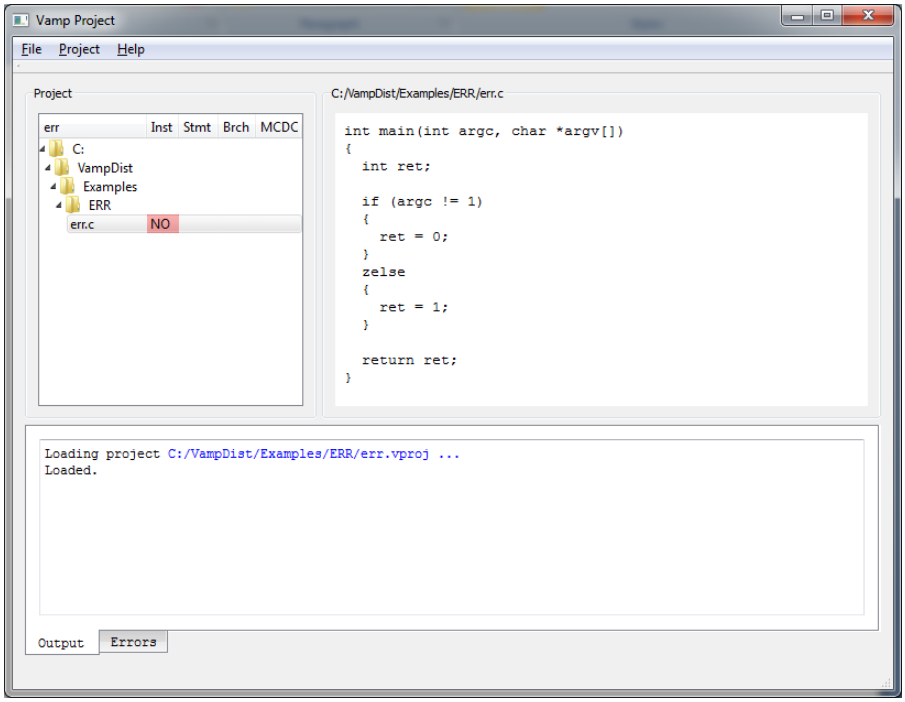
When all project files are at or beyond the project directory, Vamp chooses the project directory as the base directory, and mirrors the directory structure following for each absolute path selected. When any project file exists before the project directory, Vamp chooses the lowest common directory as the base directory.

Let's walk through some examples of the process of instrumentation and results collection. The Vamp distribution contains an Examples directory. The first example we will look at is in the **Examples/ERR** directory. Run the VampGUI executable, and select *Project->Open Project*. Navigate to the **Examples/ERR** directory and select **err.vproj**. The display will look like:

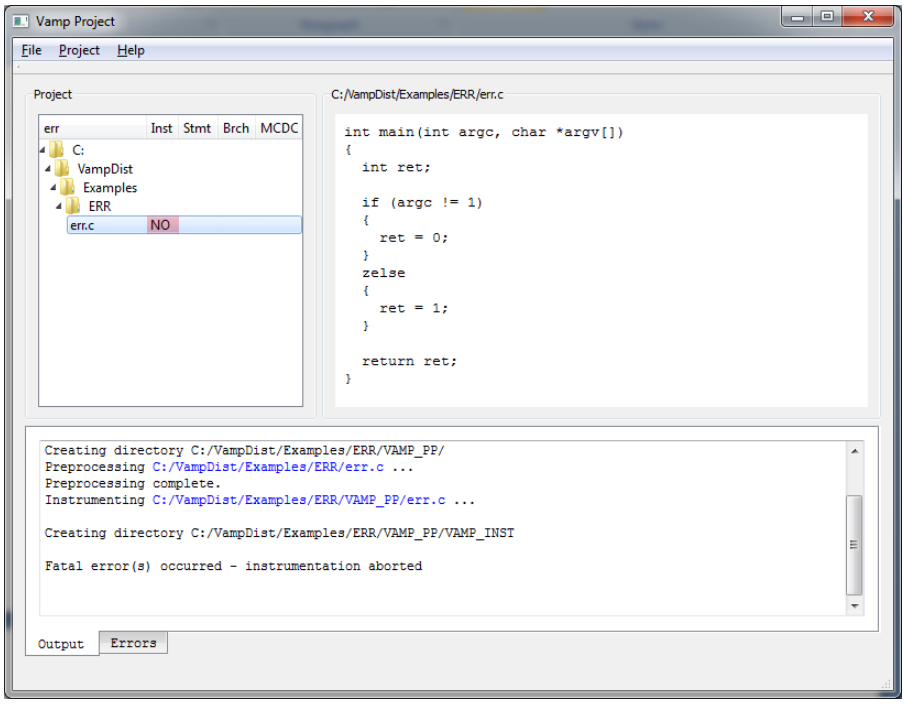


Note the file *err.c* is shown and under the *Inst* tab shows "NO" highlighted in red. This shows that no instrumentation results have been collected for err.c.

Right click on *err.c* and select *Show source file*. The display will now look like:



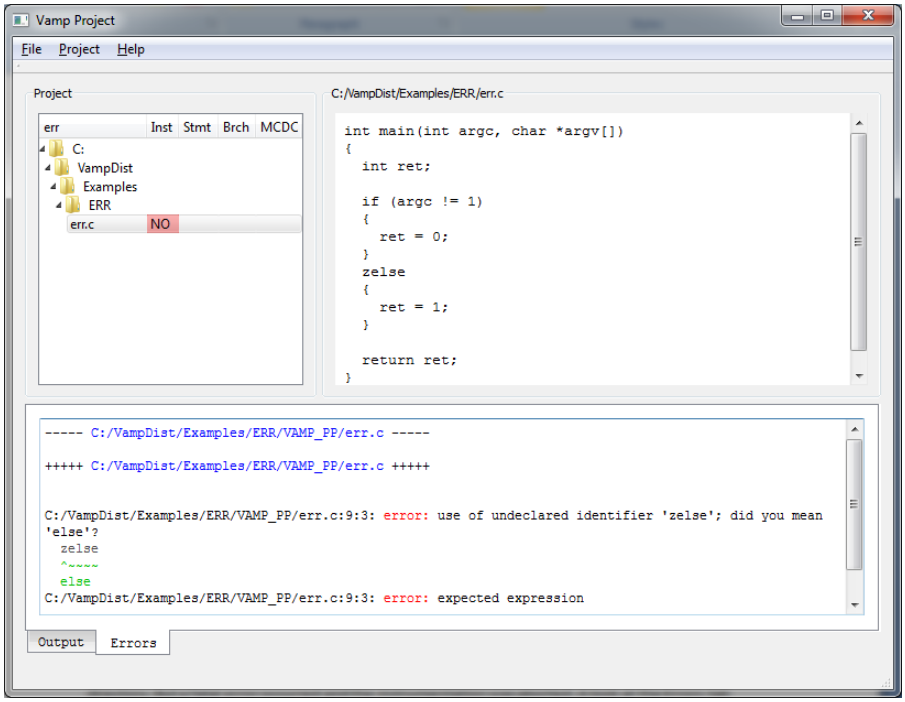
Note the syntax error in function main(), with "**else**" misspelled as "**zelse**". Right click on err.c again and select *Instrument file*. The display will now look like:



Note the contents of the Output tab:

> Creating directory C:/VampDist/Examples/ERR/VAMP_PP/
> Preprocessing C:/VampDist/Examples/ERR/err.c ...
> Preprocessing complete.
> Instrumenting C:/VampDist/Examples/ERR/VAMP_PP/err.c ...
>
> Creating directory C:/VampDist/Examples/ERR/VAMP_PP/VAMP_INST
>
> Fatal error(s) occurred - instrumentation aborted

The directory VAMP_PP is created to hold a preprocessed version of err.c. The file is then preprocessed. Instrumentation begins with the creation of the VAMP_INST directory. But a fatal error occurred and the instrumentation was aborted. A look at the Errors tab shows:



The first line:

----- C:/VampDist/Examples/ERR/VAMP_PP/err.c -----

shows any errors that occurred during preprocessing. No errors occurred.

The next line:

+++++ C:/VampDist/Examples/ERR/VAMP_PP/err.c +++++

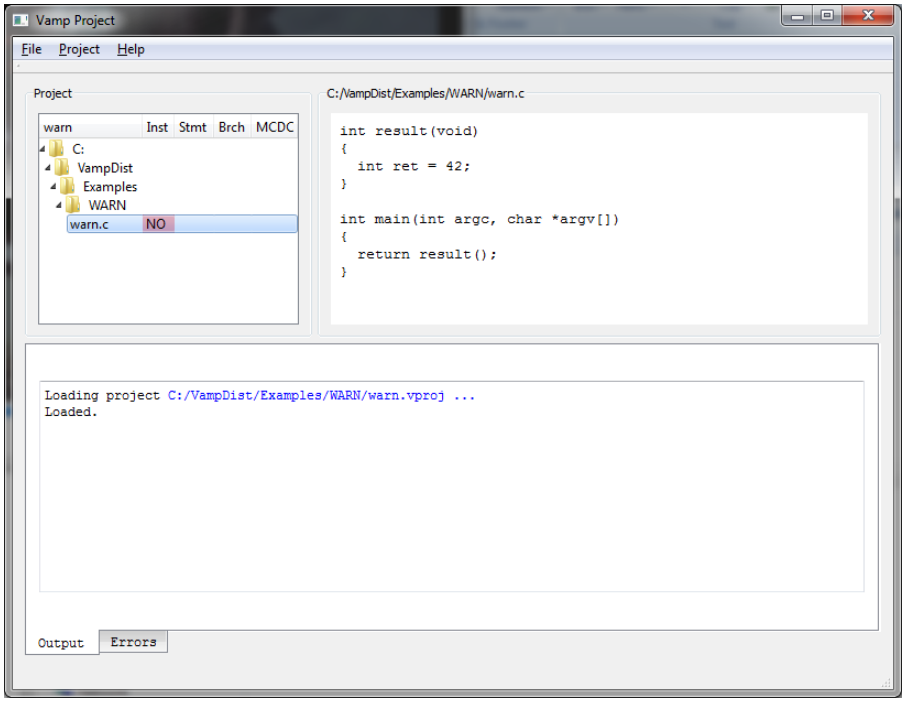
shows any errors that occurred during instrumentation. The following error lines are displayed:

C:/VampDist/Examples/ERR/VAMP_PP/err.c:9:3: error: use of undeclared identifier 'zelse'; did you mean 'else'?
zelse
^~~~~
else
C:/VampDist/Examples/ERR/VAMP_PP/err.c:9:3: error: expected expression

Vamp detected the syntax error and reported the error, then exited as the C source must be error-free in order to perform the instrumentation. Note the error message provides a hint as to how to correct the syntax errors. Vamp diagnostics strive to be user-friendly.

Log information on either the Output tab or Errors Tab may be saved by right-clicking on the area and selecting Save As. The information will be saved in HTML format. Optionally, you can right-click and choose Select All, then copy the information by pressing Crtl-C, and paste the text into a document. Text color will be lost.

Vamp will also report warnings. Now select *Project->Open Project*. Navigate to the **Examples/WARN** directory and select **warn.vproj**. Right click on **warn.c** and select *Show source file*. The display will look like:



The file **warn.c** looks like:

```
int result(void)
{
  int ret = 42;
}

int main(int argc, char *argv[])
{
  return result();
}
```

Note the lack of a return statement in the function result(). Right click on warn.c and select *Instrument file*. The Errors tab shows:

> C:/VampDist/Examples/WARN/VAMP_PP/warn.c:4:1: warning: control reaches end of non-void function
> }
> ^

As only a warning was generated, the file was successfully instrumented. Vamp may provide warnings not generated by other compilers. Vamp syntax checking strives to be very thorough and informative.

Generating a Vamp Report

Now select *Project->Open Project*. Navigate to the **Examples/TEST** directory and select **test.vproj**. Right-click on **test.c** and select *Instrument file*. Now from the menu select *Project->Generate vamp_output.c*. This will create the test harness necessary to build a complete instrumented project. The file is created in the instrumented file directory (in this case **Examples/TEST/VAMP_PP/VAMP_INST).** The file *vamp_output.h* is also created in this directory. This file is included by each instrumented file and should be in the *Include* path for a build. Copy the file **vamp_send.c** found in the Vamp distribution directory (at the same level as **Examples**) to the **Examples/TEST/VAMP_PP/VAMP_INST** directory.

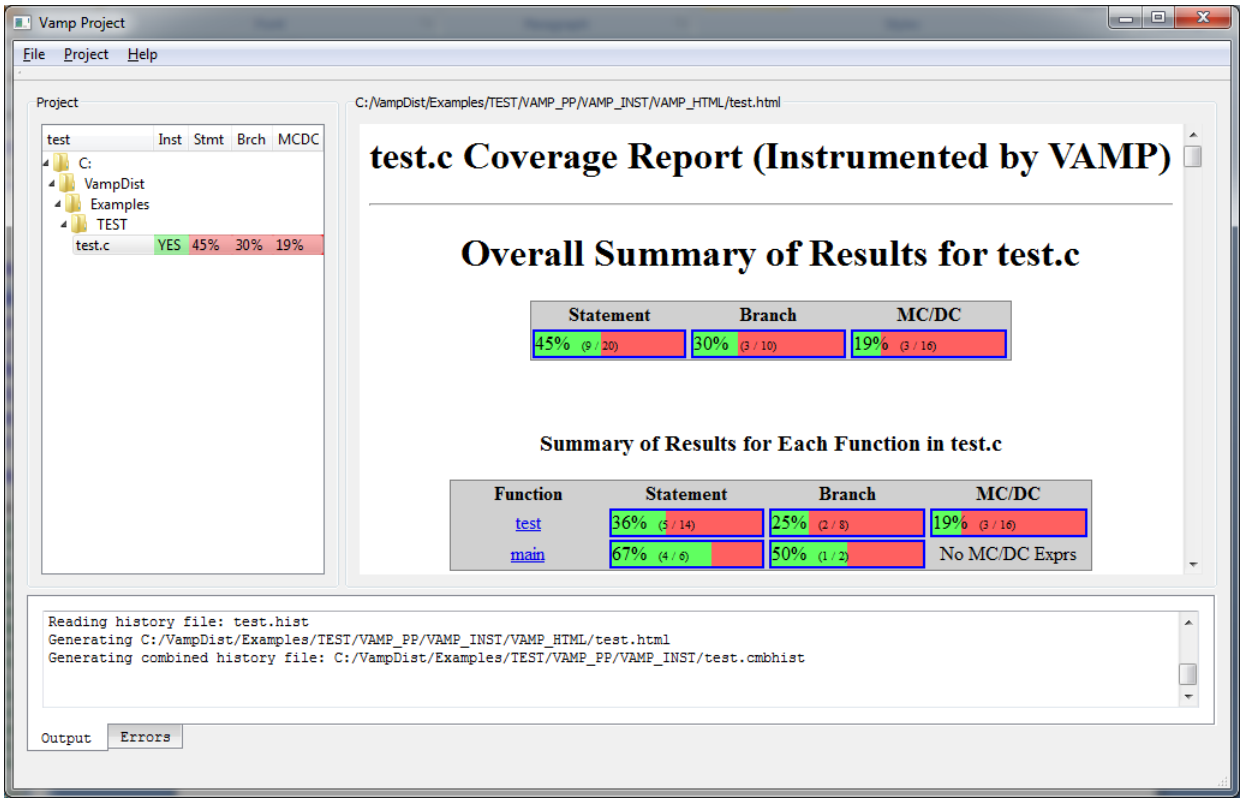
We now have three source files in the **Examples/TEST/VAMP_PP/VAMP_INST** directory, **test.c**, **vamp_output.c, vamp_send.c**. Using a C compiler, compile these three files into an executable. For example, using the GNU C compiler:

```
gcc –o test –I. test.c vamp_output.c vamp_send.c
```

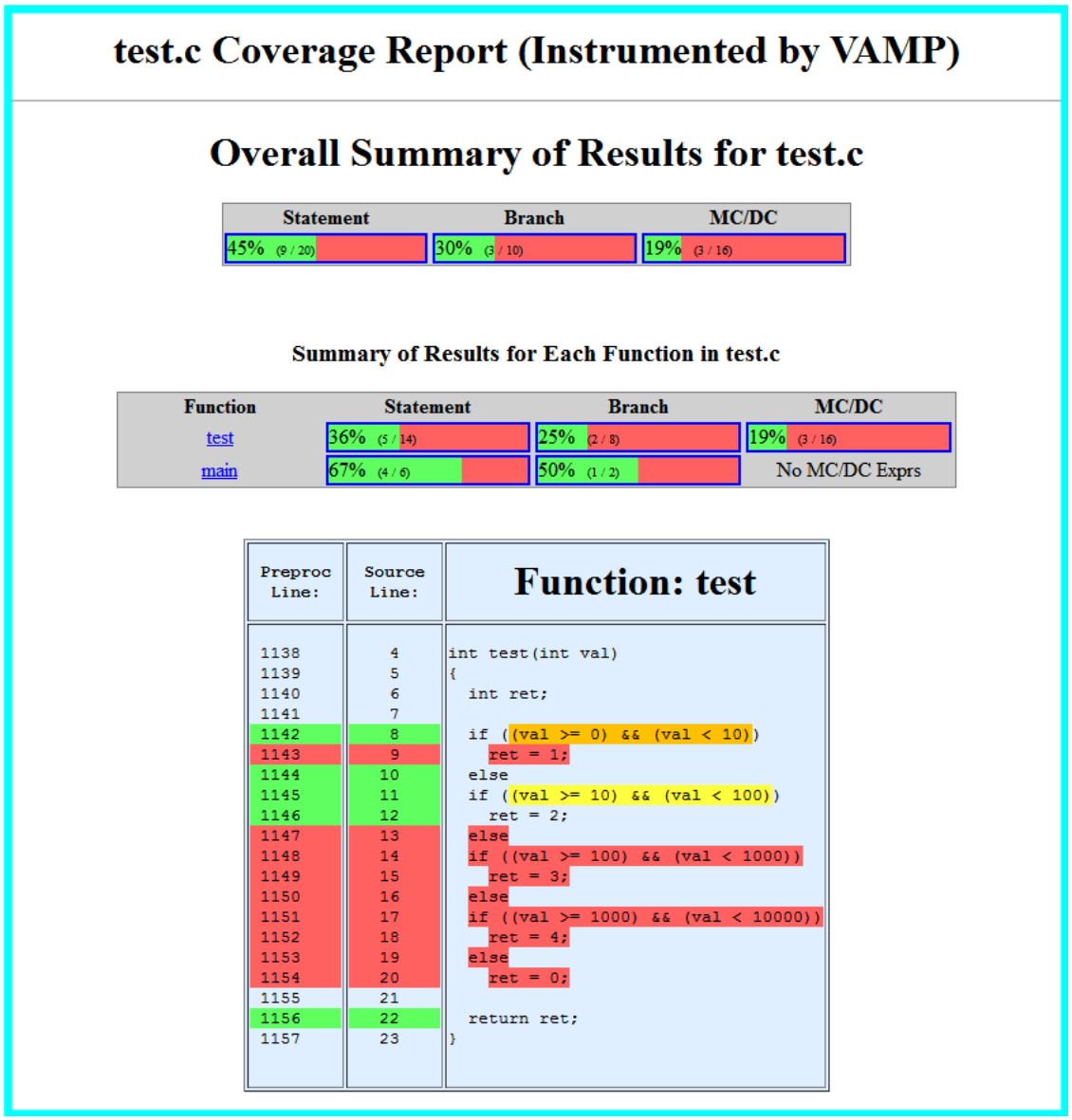
Now we run the executable with a test value of 42:

```
./test 42
Result is 2
```

Upon completion, the history file **test.hist** is created in the **Examples/TEST/VAMP_PP/VAMP_INST /VAMP_HTML** directory. Now we process the results by right-clicking on **test.c** and selecting *Process* results. Now double-click on **test.c** and the instrumentation results are displayed.



We can load the report **test.html** into a web browser. The top of the report shows:



test.c Coverage Report (Instrumented by VAMP)

Overall Summary of Results for test.c

Statement	Branch	MC/DC
45% (9 / 20)	30% (3 / 10)	19% (3 / 16)

Summary of Results for Each Function in test.c

Function	Statement	Branch	MC/DC
test	36% (5 / 14)	25% (2 / 8)	19% (3 / 16)
main	67% (4 / 6)	50% (1 / 2)	No MC/DC Exprs

Function: test

Preproc Line:	Source Line:	
1138	4	int test(int val)
1139	5	{
1140	6	int ret;
1141	7	
1142	8	if ((val >= 0) && (val < 10))
1143	9	ret = 1;
1144	10	else
1145	11	if ((val >= 10) && (val < 100))
1146	12	ret = 2;
1147	13	else
1148	14	if ((val >= 100) && (val < 1000))
1149	15	ret = 3;
1150	16	else
1151	17	if ((val >= 1000) && (val < 10000))
1152	18	ret = 4;
1153	19	else
1154	20	ret = 0;
1155	21	
1156	22	return ret;
1157	23	}

The first line shows the source file (**test.c**) used for this report. This is followed by an overall summary of results for the entire file, taking into account all functions in **test.c**. The results show 9 of 20 statements are covered for 45% statement coverage, 3 of 10 branches for 30% branch coverage, and 3 of 16 MC/DC conditions for 19% coverage. This is further broken down into percentages for each function in **test.c** (**test()** and **main()**). The amount of green in each bar shows the percentage achieved, while the amount of red shows the percentage remaining.

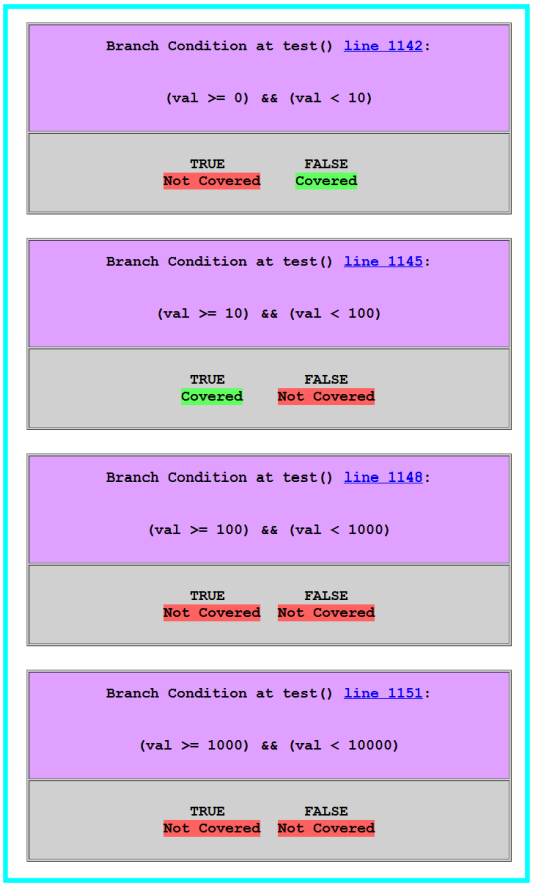
The source code for the function **test()** is shown next. The left-hand column shows the line numbers for the function statements in the preprocessed version of **test.c**. The next column shows the corresponding line numbers for the function statements in the original source version of **test.c**. The right-hand column shows the actual lines of source code. Note the first line number shown is 1138.

Since the file has been preprocessed, the lines above 1138 are from the include files **stdio.h** and **stdlib.h**. Line numbers highlighted in green are statements that have achieved statement coverage. Line numbers highlighted in red are statements that have not achieved statement coverage. Line numbers with no highlighted color are not considered statements for the purposes of statement coverage. Note also that the source code for lines that received no coverage is also highlighted in red. This helps to indicate areas where additional testing is required.

Lines 1142 and 1145 have branch expressions highlighted. In general, all branch expressions on lines with statement coverage will be highlighted with a color showing the extent of coverage achieved:

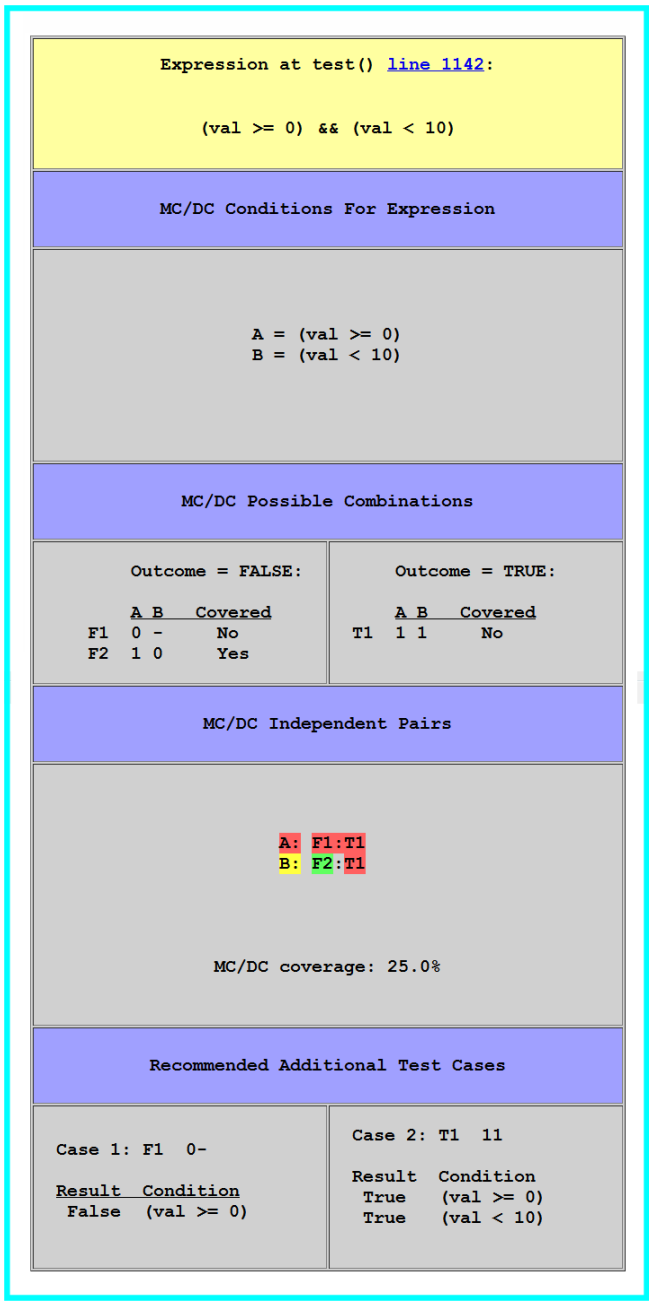
- Green implies full branch coverage was achieved (i.e. both the **true** and **false** conditions were met).
- Yellow implies only the **true** condition of branch coverage was achieved.
- Orange implies only the **false** condition of branch coverage was achieved.
- Red implies neither the **true** nor **false** condition of branch coverage was achieved.

Following the source code for function **test()**, we see:



This shows the two branch conditions at lines 1142 and 1145 we previously discussed, followed by two other branch conditions that did not receive statement coverage. Click on any of the highlighted line numbers to navigate back to the function source code displayed in the report.

The report next shows MC/DC expression coverage. The first MC/DC expression in the block is:



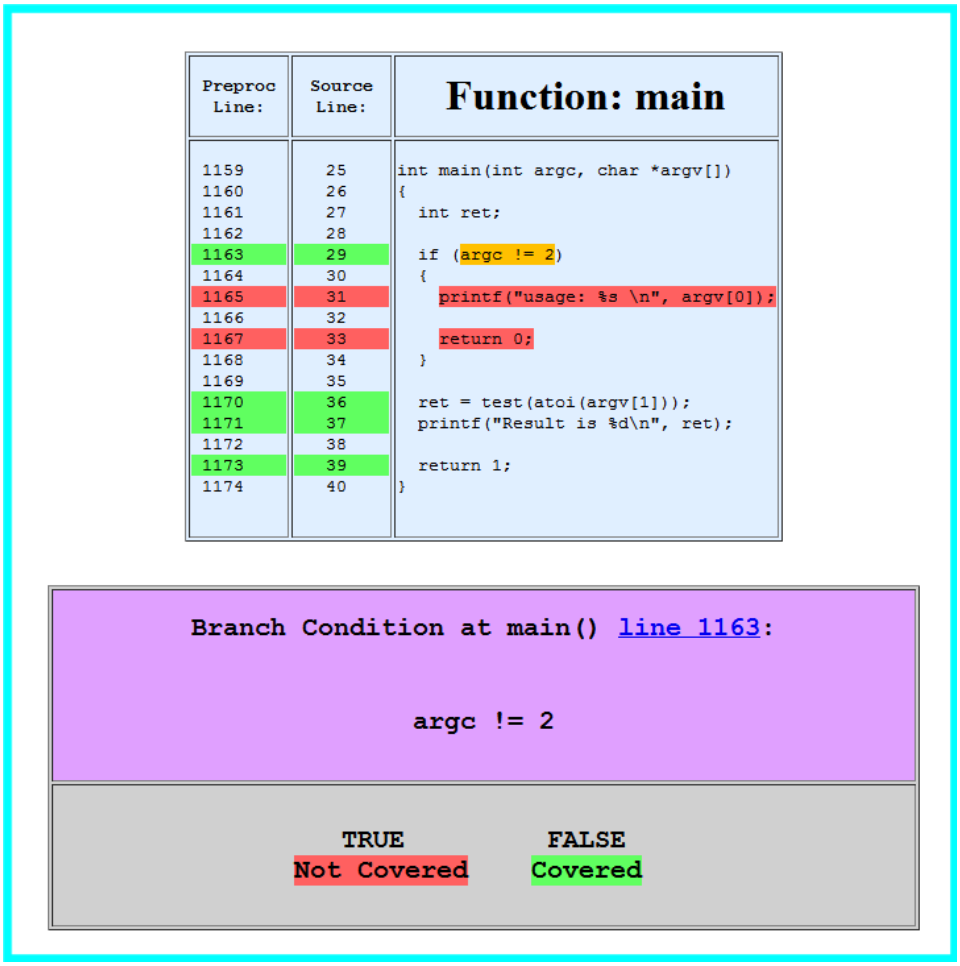
The MC/DC expression at line 1142 is shown: **(val >= 0) && (val < 10)**. This is broken down into the two conditions, **A = (val >= 0)** and **B = (val < 10)**.

This is followed by the possible MC/DC combinations that may occur for this expression. Rather than supply a truth table for the four possible combinations of **A** and **B**, the table is simplified due to the short-circuit evaluation of Boolean expressions found in the C language. For the expression **A && B**, if **A** evaluates to **false**, the entire expression must evaluate to **false**, so **B** is not evaluated and is treated as a "don't care" condition (see **F1** in the table). Likewise with the expression **A || B**, if **A** evaluates to true,

the entire expression must evaluate to **true**, so **B** is not evaluated. For MC/DC expressions with several conditions, this simplifies the possible combinations that may occur.
To achieve MC/DC coverage, each condition must show it can independently affect the outcome of the expression. All possible sets of independent pairs are calculated and displayed; in this instance there is only one independent pair for each condition. To demonstrate the independence of condition **A**, one must show both **F1** and **T1** have occurred. To demonstrate the independence of condition **B**, one must show both **F2** and **T1** have occurred. In the expression above, only **F2** was demonstrated, in which **A** was **true** and **B** was **false**, or **(val >= 0)** was **true** (**A**) and **(val < 10)** was **false** (**B**). Recall the input value was 42, so we see that this indeed is the case (**(42 >= 0)** is **true** and **(42 < 10)** is **false**).

In order to achieve complete MC/DC coverage for this expression, we must also show **F1** with an outcome of **false**, and **T1** with an outcome of **true**. This is shown in the recommended test cases. For MC/DC expressions with several conditions, the recommended test cases shows a minimal number of cases needed to achieve full coverage.

For the sake of simplicity, the remaining MC/DC expressions in the report will be omitted. The last function is then shown:



Preproc Line:	Source Line:	Function: main
1159	25	int main(int argc, char *argv[])
1160	26	{
1161	27	int ret;
1162	28	
1163	29	if (argc != 2)
1164	30	{
1165	31	printf("usage: %s \n", argv[0]);
1166	32	
1167	33	return 0;
1168	34	}
1169	35	
1170	36	ret = test(atoi(argv[1]));
1171	37	printf("Result is %d\n", ret);
1172	38	
1173	39	return 1;
1174	40	}

The previous description should be sufficient to explain this portion of the report.

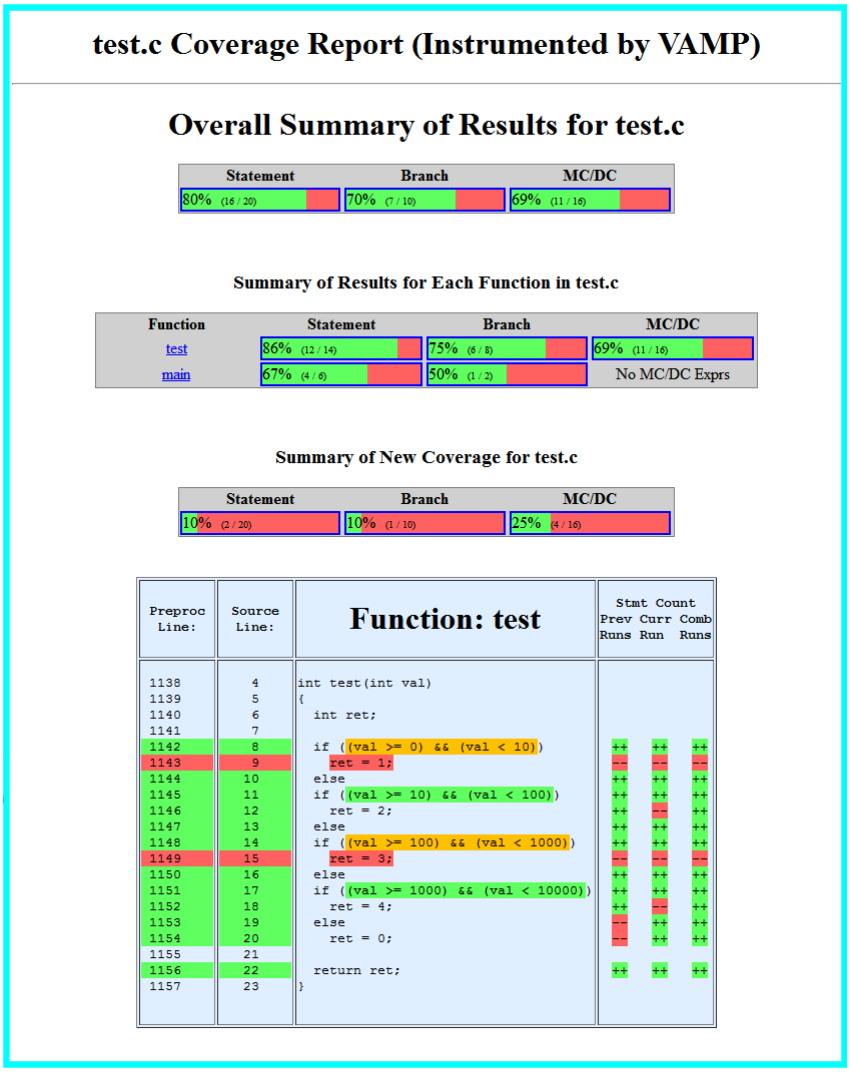
Combining Coverage

Now that we have achieved partial coverage for our test program, we will supplement that coverage with some additional tests. Each time *Process results* is called to generate a report, it combines the new **test.hist** coverage information into a combined **.cmbhist** file. Thus the first call to *Process results* copies the **test.hist** file to **test.cmbhist**. Each subsequent call combines the **test.hist** information into **test.cmbhist**. The generated report reflects the combined coverage, the current run information and all new coverage. For example we will add 2 new test cases then examine the second report:

./test 1234
Result is 4
Right-click on **test.c** and select *Process results*.

./test -1
Result is 0

Right-click again on **test.c** and select *Process results*. The new report looks like:



test.c Coverage Report (Instrumented by VAMP)

Overall Summary of Results for test.c

Statement	Branch	MC/DC
80% (16 / 20)	70% (7 / 10)	69% (11 / 16)

Summary of Results for Each Function in test.c

Function	Statement	Branch	MC/DC
test	86% (12 / 14)	75% (6 / 8)	69% (11 / 16)
main	67% (4 / 6)	50% (1 / 2)	No MC/DC Exprs

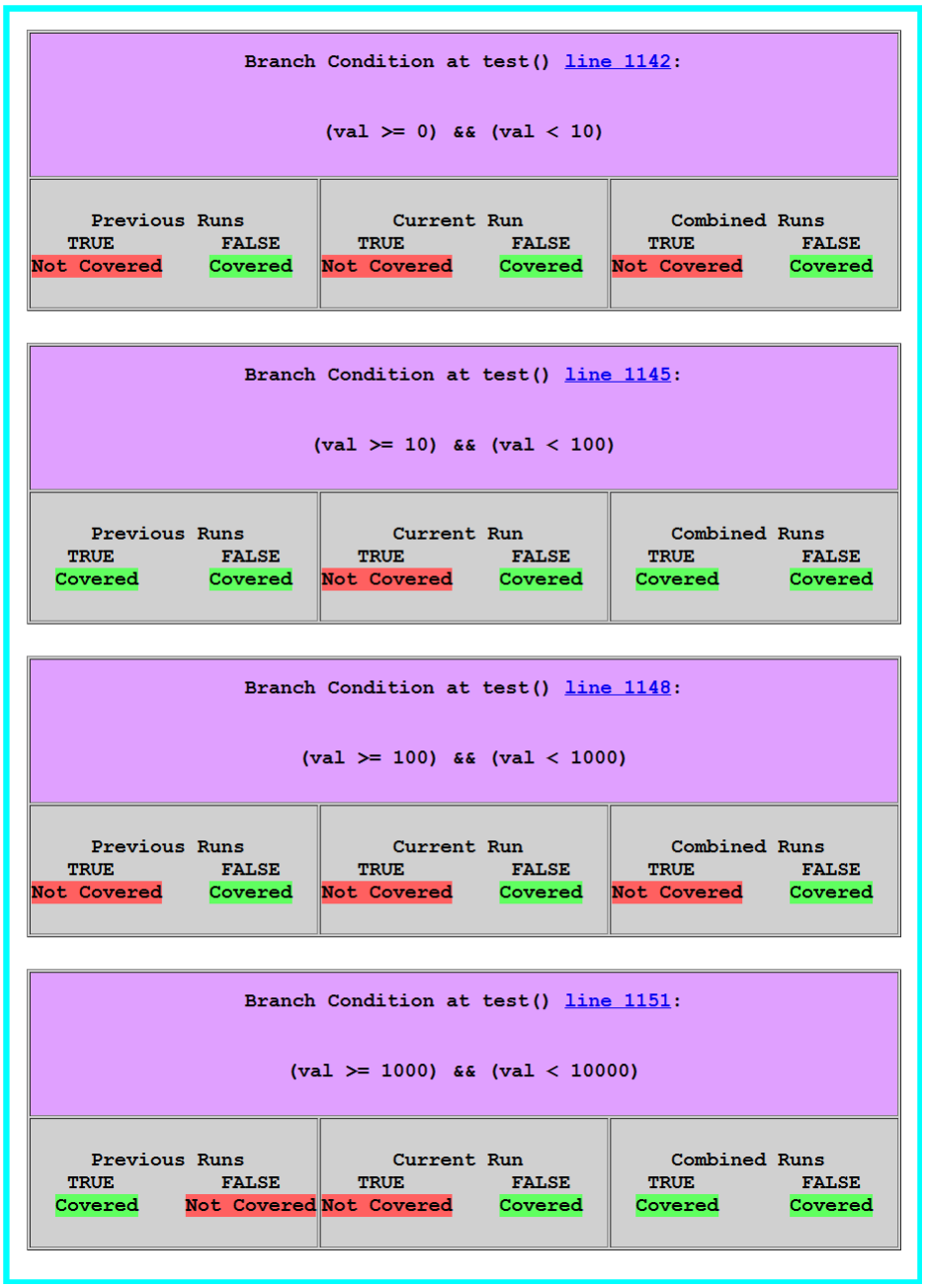
Summary of New Coverage for test.c

Statement	Branch	MC/DC
10% (2 / 20)	10% (1 / 10)	25% (4 / 16)

Preproc Line:	Source Line:	Function: test	Stmt Count Prev Runs	Curr Run	Comb Runs
1138	4	int test(int val)			
1139	5	{			
1140	6	int ret;			
1141	7				
1142	8	if ((val >= 0) && (val < 10))	++	++	++
1143	9	ret = 1;	--	--	--
1144	10	else	++	++	++
1145	11	if ((val >= 10) && (val < 100))	++	++	++
1146	12	ret = 2;	++	--	++
1147	13	else	++	++	++
1148	14	if ((val >= 100) && (val < 1000))	++	++	++
1149	15	ret = 3;	--	--	--
1150	16	else	++	++	++
1151	17	if ((val >= 1000) && (val < 10000))	++	++	++
1152	18	ret = 4;	++	--	++
1153	19	else	--	++	++
1154	20	ret = 0;	--	++	++
1155	21				
1156	22	return ret;	++	++	++
1157	23	}			

You can see the overall percentages of coverage have improved. There is a new **Summary of New Coverage** line showing both additional percentage and new statement, branch and MC/DC coverage. In addition, there are new columns in the source pane showing statement coverage. The first column (**Prev Runs**) shows all coverage for prior runs. The middle (**Curr Run**) shows statement coverage for only the current run (**./test -1**), and the last column (**Comb Runs**) shows the combined coverage from all runs. Note lines 1153-1154 show new coverage from this run, as the input value of -1 results in the final else clause being executed.

Branch coverage shows:



Branch Condition at test() line 1142: (val >= 0) && (val < 10)					
Previous Runs		Current Run		Combined Runs	
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Not Covered	Covered	Not Covered	Covered	Not Covered	Covered

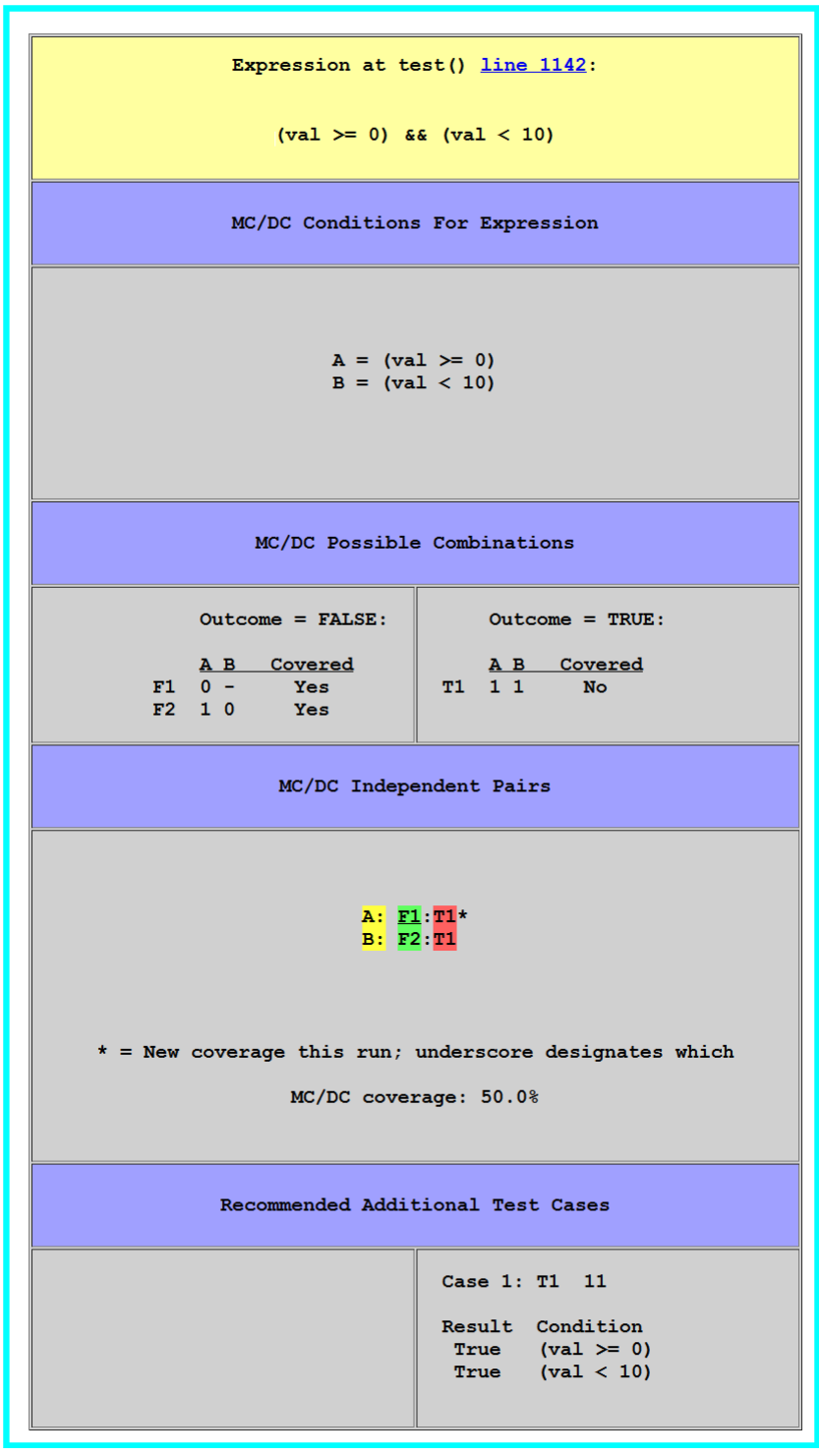
Branch Condition at test() line 1145: (val >= 10) && (val < 100)					
Previous Runs		Current Run		Combined Runs	
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Covered	Covered	Not Covered	Covered	Covered	Covered

Branch Condition at test() line 1148: (val >= 100) && (val < 1000)					
Previous Runs		Current Run		Combined Runs	
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Not Covered	Covered	Not Covered	Covered	Not Covered	Covered

Branch Condition at test() line 1151: (val >= 1000) && (val < 10000)					
Previous Runs		Current Run		Combined Runs	
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
Covered	Not Covered	Not Covered	Covered	Covered	Covered

For each branch condition, a column is present showing branch coverage for all **Previous Runs**, the **Current Run**, and the **Combined Runs** (current run combined with all previous runs).

MC/DC coverage shows:



Expression at test() line 1142:

(val >= 0) && (val < 10)

MC/DC Conditions For Expression

A = (val >= 0)
B = (val < 10)

MC/DC Possible Combinations

Outcome = FALSE:

	A	B	Covered
F1	0	-	Yes
F2	1	0	Yes

Outcome = TRUE:

	A	B	Covered
T1	1	1	No

MC/DC Independent Pairs

A: F1:T1*
B: F2:T1

* = New coverage this run; underscore designates which

MC/DC coverage: 50.0%

Recommended Additional Test Cases

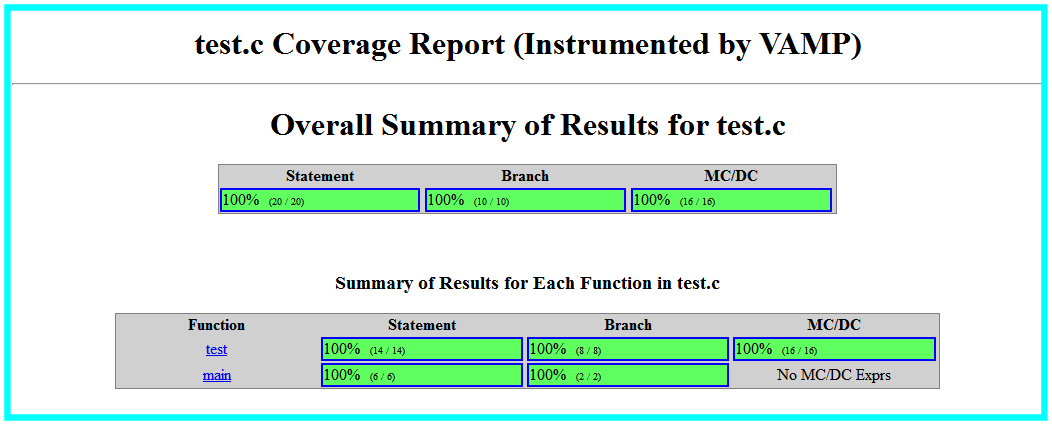
Case 1: T1 11

Result	Condition
True	(val >= 0)
True	(val < 10)

The new coverage for this run is **F1**, where **(val >= 0)** evaluates to **false**. **F1** is shown underscored to designate this run as new coverage.

To achieve complete coverage we will add 4 new test cases, then examine the final report:

./test 3
Result is 1
Right-click on **test.c** and select *Process results*.
...
./test 123
Result is 3
Right-click on **test.c** and select *Process results*.
...
./test 12345
Result is 0
Right-click on **test.c** and select *Process results*.
...
./test 1 2
usage: ./test <val>
Right-click on **test.c** and select *Process results*.



test.c Coverage Report (Instrumented by VAMP)

Overall Summary of Results for test.c

Statement	Branch	MC/DC
100% (20 / 20)	100% (10 / 10)	100% (16 / 16)

Summary of Results for Each Function in test.c

Function	Statement	Branch	MC/DC
test	100% (14 / 14)	100% (8 / 8)	100% (16 / 16)
main	100% (6 / 6)	100% (2 / 2)	No MC/DC Exprs

We now have a full set of test cases.

Understanding the Text-Based Report

The text-based report generated by *Process results* contains a summary of coverage for the specified file. After each run of *Process results*, the file **test.rpt** is generated. The contents after the third run contain:

```
File Coverage Summary,Statement Covered,Statement Total,Branch
Covered,Branch Total,MCDC Covered,MCDC Total
test.c,16,20,7,10,11,16
Function Coverage Summary,Statement Covered,Statement Total,Branch
Covered,Branch Total,MCDC Covered,MCDC Total
test,12,14,6,8,11,16
main,4,6,1,2,0,0
File Coverage New,Statement Covered,Statement Total,Branch
Covered,Branch Total,MCDC Covered,MCDC Total
test.c,2,20,1,10,4,16
```

This is a comma delimited file showing the amount covered followed by total amount for each type of coverage. This file can be imported into a spreadsheet with a typical result appearing as:

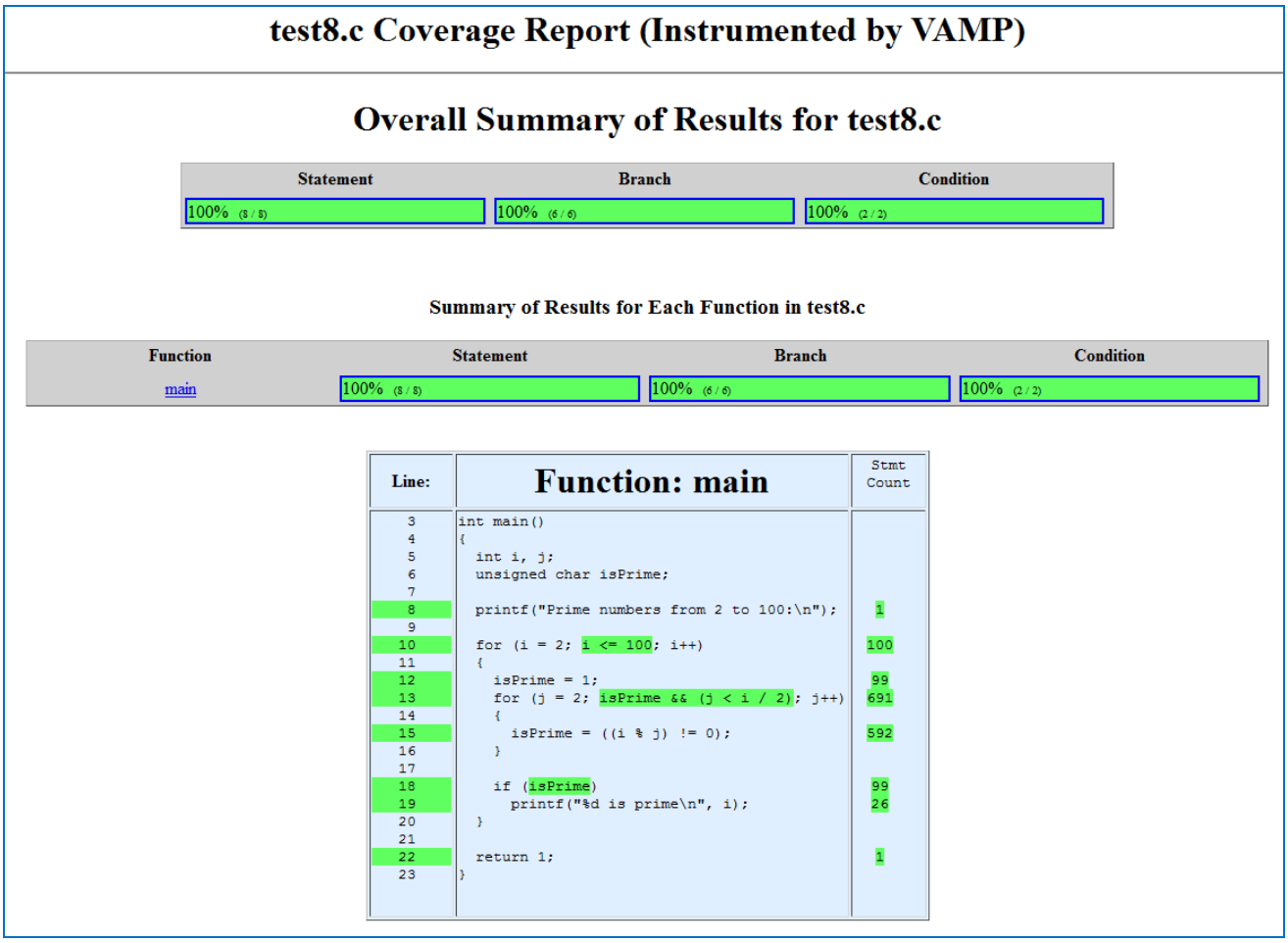
File Coverage Summary	Statement Covered	Statement Total	Branch Covered	Branch Total	MCDC Covered	MCDC Total
test.c	16	20	7	10	11	16
Function Coverage Summary	Statement Covered	Statement Total	Branch Covered	Branch Total	MCDC Covered	MCDC Total
test	12	14	6	8	11	16
main	4	6	1	2	0	0
File Coverage New	Statement Covered	Statement Total	Branch Covered	Branch Total	MCDC Covered	MCDC Total
test.c	2	20	1	10	4	16

The percentage may be calculated with the formula:

int percent = (1000 * covered / total + 5) / 10;

Additional Examples

When the *Instrumentation config* options **Count Statement Coverage, Basic Condition Coverage** and **Branch Coverage** are checked and the options **Single Statement Coverage** and **MC/DC Coverage** are unchecked, a sample report appears as:



test8.c Coverage Report (Instrumented by VAMP)

Overall Summary of Results for test8.c

Statement	Branch	Condition
100% (8 / 8)	100% (6 / 6)	100% (2 / 2)

Summary of Results for Each Function in test8.c

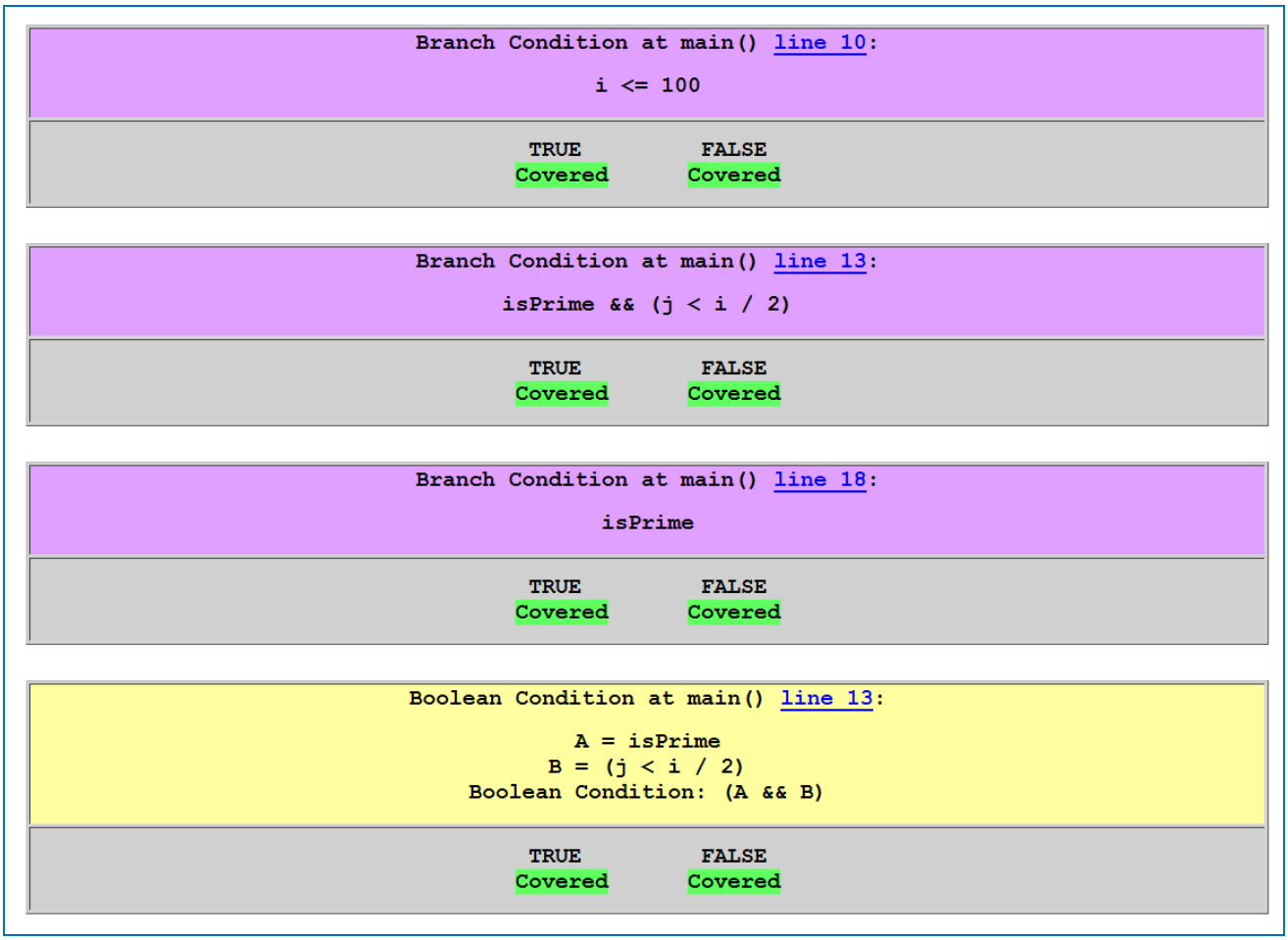
Function	Statement	Branch	Condition
main	100% (8 / 8)	100% (6 / 6)	100% (2 / 2)

Function: main

Line:		Stmt Count
3	int main()	
4	{	
5	int i, j;	
6	unsigned char isPrime;	
7		
8	printf("Prime numbers from 2 to 100:\n");	1
9		
10	for (i = 2; i <= 100; i++)	100
11	{	
12	isPrime = 1;	99
13	for (j = 2; isPrime && (j < i / 2); j++)	691
14	{	
15	isPrime = ((i % j) != 0);	592
16	}	
17		
18	if (isPrime)	99
19	printf("%d is prime\n", i);	26
20	}	
21		
22	return 1;	1
23	}	

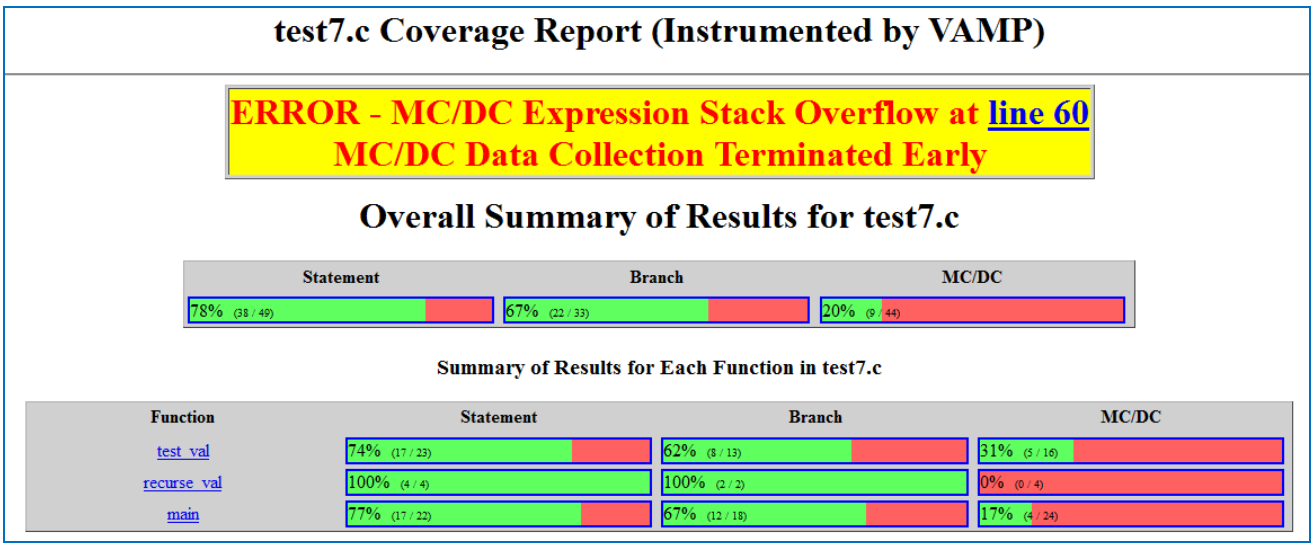
Note the statement count column in the source pane. The count information can be useful for profiling or other verification activities. The use of the **Count Statement Coverage** configuration option has a memory penalty when used, as instead of a single bit being required for each statement when **Single Statement Coverage** is used, 32 bits are required for each instrumented statement.

The report continues:



Note that instead of MC/DC information, line 13 now has Boolean Condition information and shows both the **true** and **false** conditions were met for the entire Boolean condition.

In the following example, a MC/DC stack overflow occurred. Note that results were captured up to the point of the stack overflow, and the line number where the overflow occurred is shown.



test7.c Coverage Report (Instrumented by VAMP)

ERROR - MC/DC Expression Stack Overflow at line 60
MC/DC Data Collection Terminated Early

Overall Summary of Results for test7.c

Statement	Branch	MC/DC
78% (38 / 49)	67% (22 / 33)	20% (9 / 44)

Summary of Results for Each Function in test7.c

Function	Statement	Branch	MC/DC
test_val	74% (17 / 23)	62% (8 / 13)	31% (5 / 16)
recurse_val	100% (4 / 4)	100% (2 / 2)	0% (0 / 4)
main	77% (17 / 22)	67% (12 / 18)	17% (4 / 24)

Configuration Options

Vamp has two configuration files, one for instrumentation options and the other for report generation options. Instrumentation options are handled by the file **vamp.cfg**. A typical copy of this file is:

```
{
  /* Perform basic statement coverage */
  "do_statement_single": true,
  /* Perform statement coverage, counting number of times line hit */
  "do_statement_count": false,
  /* Perform branch coverage */
  "do_branch": true,
  /* Perform MC/DC coverage (supercedes condition coverage) */
  "do_MCDC": true,
  /* Perform condition coverage (predicate coverage) */
  "do_condition": false,
  /* Location to save instrumented and database files */
  "save_directory": "VAMP_INST",
  /* Suffix to add to <file>.c to become <file><suffix>.c */
  "save_suffix": "",
  /* MC/DC stack size */
  "mcdc_stack_size": 4,
  /* C language standard */
  "lang_standard": "lang_c89"
}
```

The configuration options in this file include:

- **do_statement_single** – Perform basic statement coverage. The report will show if a statement has been covered.
- **do_statement_count** – Another form of statement coverage. The report will show if a statement has been covered, and the number of times the statement was reached during the current execution run. If **true**, **do_statement_single** must be **false**.
- **do_branch** – Perform branch coverage.
- **do_MCDC** – Perform MC/DC coverage.
- **do_condition** – Perform condition (predicate) coverage. If **true**, **do_MCDC** must be **false**.
- **save_directory** – Directory where instrumented source file and database description files are saved.
- **save_suffix** – Suffix to add to source filename when generating instrumented file and database description file. For example, with the suffix "**_vamp**", instrumenting **test.c** will generate the files **test_vamp.c** and **test_vamp.json**.
- **mcdc_stack_size** – This defines the maximum stack size for MC/DC expressions. Whenever a MC/DC condition contains a function call, all accumulated MC/DC expression collection information is placed on a stack until the function returns. If the function called is in the same source file as the condition, and the function itself has a MC/DC expression containing a function call, the stack depth is increased. This is a relatively rare occurrence, so normally a low value for the stack size is required. There may be occasion, such as in a recursive function, where the stack size needs to be increased. A warning message is displayed in the report if the stack

overflows, in which case data collection for that file ceases for the remainder of the current execution run.

- **lang_standard** – The ISO C language standard to use. Options are:

C Language Standard Identifier	C Standard	Has GNU C Extensions
lang_c89	ISO C 1990	No
lang_gnu89	ISO C 1990	Yes
lang_c99	ISO C 1999	No
lang_gnu99	ISO C 1999	Yes
lang_c11	ISO C 2011	No
lang_gnu11	ISO C 2011	Yes

If no **vamp.cfg** file exists, the following default values shall be used:

```
do_statement_single: true
do_statement_count: false
do_branch: true
do_condition: false
do_MCDC: true
save_directory: "VAMP_INST"
save_suffix: ""
mcdc_stack_size: 4
language_standard: lang_c89
```

The configuration options do_statement_single and do_statement_count shall be mutually exclusive.

The configuration options do_condition and do_MCDC shall be mutually exclusive.

Configuration options for report generation are kept in the file vamp_process.cfg. A typical copy of this file is:

```
{
  /* Combine current history with previous histories */
  "combine_history": true,
  /* Show recommended test cases */
  "show_test_cases": true,
  /* Generate report summary */
  "do_report": true,
  /* Report field separator */
  "report_separator": ",",
  /* Location to save instrumented and database files */
  "html_directory": "VAMP_HTML",
  /* Suffix to add to <file>.c to become <file><suffix>.c */
  "html_suffix": ""
}
```

The configuration options in this file include:

- **combine_history** – Combine current execution history with previous execution histories.
- **show_test_cases** – Show a set of recommended MC/DC test cases to complete coverage.
- **do_report** – Generate the report summary (**.rpt**) file.
- **report_separator** – The character to use for separating fields in the report summary.
- **html_directory** – Directory in which the .html and .rpt report files are saved.
- **html_suffix** – Suffix to add to database filename when generating **.html** and **.rpt** report files. For example, with the suffix "**_vamp**", processing the results for **test.c** will generate the files **test_vamp.html** and **test_vamp.rpt**.

If no vamp_process.cfg exists, the following default values are used:

```
combine_history: true
show_test_cases: true
do_report: true
report_separator: ","
html_directory: "VAMP_HTML"
html_suffix: ""
```

Definitions

Statement – The definition of a C statement, as used in the context of statement coverage, varies and is open to interpretation. Vamp considers a C statement as follows:

- Any line of C code within a function that is not a declaration and ends with a "**;**".
- Any non-static variable declaration within a function that has an initializer.
- For an **if** statement, the keyword "**if**" and it's associated condition.
- The keyword "**else**".
- For a **while** statement or a **do/while** statement, the keyword "**while**" and it's associated condition.
- For a **switch** statement, the keyword "**switch**" and it's associated condition.
- Any **case** or **default** statement within a **switch** statement.

Statement Coverage – A statement is considered covered when any portion of that statement has been executed. **Statement Coverage** is also known as **Line Coverage**.

Branch Coverage – A branch is a decision point within an "**if**", "**while**" or "**switch**" statement. Complete coverage occurs when each possible branch has been taken (both **true** and **false** branches for **if** and **while** statements, and each possible **case** or **default** in a **switch** statement). **Branch Coverage** is also known as **Decision Coverage**.

Decision – A decision is a Boolean expression containing two conditions coupled by either a logical AND or logical OR (**&&** or **||**). Based on this definition of **Decision**, we avoid the use of the term "**Decision Coverage**" and instead use the term "**Branch Coverage**".

Condition Coverage – A condition is any evaluation of a Boolean **Decision**. Such conditions need not be part of a branch decision. Coverage is considered complete when the Boolean expression has taken on both **true** and **false** outcomes. **Condition Coverage** is also known as **Predicate Coverage**.

Modified Condition/Decision Coverage or **MC/DC** – As with **Condition Coverage**, a **MC/DC** expression is the evaluation of a Boolean expression. An MC/DC expression contains any number of logical ANDs and logical ORs (**&&** or **||**). In addition, each condition in the **MC/DC** expression must be shown to independently affect the outcome of the decision. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.