MLcov 1.2 – User Guide

Thomas Moniot

November 24, 2010

1 Introduction

1.1 Purpose

MLcov 1.2 is a code coverage analyzer for programs written in the Objective Caml language. MLcov helps you identify which parts of the code were not fully exercised during the testing phase. It proceeds by instrumentation, *i*.e. by adding probes to your program.

MLcov is able to measure both Structural Coverage and Modified Condition/Decision Coverage. Recall that MC/DC is fulfilled when every program point has been reached at least once, every condition in a decision has taken on all possible outcomes at least once, and each condition has been shown to affect that decision's outcome independently. This measure is required in particular for DO-178B Level A certification.

For further details on the underlying instrumentation algorithm, you are invited to refer to the [PADL'08] paper:

- B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaco, T. Moniot, P. Wang
  Certified development tools implementation in Objective Caml
  in Practical Aspects of Declarative Languages 2008
  Lecture Notes in Computer Science, vol. 4902, pp. 2-17.

1.2 Overview

Basically, the various steps involved in using MLcov are:

1. Instrument the source code.
2. Build the instrumented code.
3. Run the instrumented program one or more times.
4. Generate and view coverage reports.
5. Repeat steps 3 and 4 until you reach a satisfying coverage of the code.

Sections 2, 3 and 4 of this document explain how to perform those steps. Section 5 explains the justification mechanism. MLcov 1.2 actually applies to a subset of the Objective Caml language: see section 6 for further explanations.

2 Instrumenting for Coverage Measurement

To instrument an Objective Caml source code, you just have to pass it as a parameter of MLcov:

```
mlcov [-keep-instr] <file>.ml
```

The resulting instrumented file, which you will have to build and execute (see next section), is then dumped to the standard output. You may use the `-keep-instr` option in order to save the instrumented file to *<file>*`_instr.ml`. The file is then stored in the same directory as *<file>*`.ml`.

As a shortcut, you may possibly instrument several source codes at the same time. For instance,

```
mlcov <file1>.ml <file2>.ml <file3>.ml
```

is strictly equivalent to: `mlcov` *<file1>*`.ml && mlcov` *<file2>*`.ml && mlcov` *<file3>*`.ml`

Pay attention that the instrumentation stage of MLcov makes use of Objective Caml's typing information. Hence, before you can instrument a file `A.ml` whose implementation depends on another file `B.ml`, you have to make sure that the compiled interface of the second one, namely `B.cmi`, is available. If the file `B.cmi` is not in the same directory as `A.ml`, you can provide its path, as you would do with the Objective Caml compiler:

```
mlcov -I <directory> A.ml
```

3 Building and Running the Instrumented Program

To build the instrumented program, you must link the instrumented code to the `Unix` library[1] and to the `MlcovTrace` trace handler. You will probably have to provide the path of the directory where the files `mlcovTrace.cmi`, `mlcovTrace.cmx` and `mlcovTrace.o` are stored. Using the Objective Caml native compiler, you should write:

```
ocamlopt unix.cmxa -I <directory> <directory>/mlcovTrace.cmx <file>_instr.ml
```

It is still possible to use the Objective Caml bytecode compiler. In this case, you should write:

```
ocamlc unix.cma -I <directory> <directory>/mlcovTrace.cmo <file>_instr.ml
```

It may be more convenient, in a Makefile typically, to combine the instrumentation and build steps in one single command line, by using the `-pp` option of the Objective Caml compiler. This way you would write:

```
ocamlopt unix.cmxa -I <directory> <directory>/mlcovTrace.cmx -pp "mlcov" <file>.ml
```

As usual, you can use the `-o` option to specify the output name of the instrumented program, or `-c` to skip the linking stage.

Then run the resulting program one or more times. The first time you run it, a trace file *<file>*`.mlcov` is created; the other times, this trace file is just consequently updated. By default, trace files are created in the current working directory, but you can use the `MLCOV_OUTPUT_DIR` environment variable to specify the directory where they shall be stored.

[1] Trace files are indeed accessed using files lock from the `Unix` library, to ensure MLcov's re-entrance.

4 Viewing Results

Text reports
To create coverage reports, just pass the trace file as a parameter of MLcov:

```
mlcov <file>.mlcov
```

MLcov then generates two files in text format:

- an annotated version of the source code *<file>*`_annot.txt`, showing the local index associated to each program point or MC/DC decision;
- a file *<file>*`_info.txt`, consisting of both a summary of coverage statistics and a mapping between local indexes and coverage information.

As a shortcut, you may generate several reports at the same time. For instance,

```
mlcov <file1>.mlcov <file2>.mlcov <file3>.mlcov
```

is equivalent to: `mlcov` *<file1>*`.mlcov && mlcov` *<file2>*`.mlcov && mlcov` *<file3>*`.mlcov`
Note that, in addition, a global report `index.txt` is created, gathering the statistics of all the trace files.

Paragraphs 4.1 and 4.2 are related to MLcov's text outputs.

HTML reports
Alternatively, MLcov is able to generate reports in HTML format, thanks to the `-html` option:

```
mlcov -html <file>.mlcov
```

This latter command line generates two files:

- a summary of coverage measurements *<file>*`_summary.html`;
- a source code report *<file>*`_annot.html`.

A global report `index.html` is also created, gathering the statistics of all the trace files. In this report, modules that are not 100% covered are highlighted in red.

Paragraphs 4.3 and 4.4 describe MLcov's HTML outputs, and you can refer to appendix A.8 for an example of HTML global report.

Other options
• By default, coverage reports are created in the current working directory, but you can use the `-d` option to specify the output directory for the reports:

```
mlcov -d <directory> <file>.mlcov
```

• Use the `-g` option to include traces specified in a given file:

```
mlcov -g <global_file>
```

The file *<global_file>* shall contain paths (one on each line) to several trace files, for example:

```
A.mlcov
B.mlcov
C.mlcov
```

As a result, it behaves as if you had written: `mlcov A.mlcov B.mlcov C.mlcov`

• The `-src-base-dir` option allows you to provide the base directory where original source files are stored:

`mlcov -src-base-dir <directory> <file>.mlcov`

Typically, this can be very useful when the instrumentation and reporting steps are not executed on the same machine; otherwise the annotated report could not be produced (*cf.* appendix B). Pay attention that this option works only if relative paths were given to MLcov during instrumentation.

• Finally, the `-line-numbers` option may be used in order to display line numbers in the annotated source code:

`mlcov -line-numbers <file>.mlcov`

All the options mentionned above apply to both HTML and text outputs, and may be combined with each other.

4.1 Source Code annotated with local indexes (text)

In the text version, the annotated source code corresponds to the original code, but additional ML comments have been inserted in order to show the index associated to each program point or MC/DC decision. Indexes are positive integers starting at 1. They are guaranteed to be unique within the scope of a given toplevel function, provided that the name of this function is unique in the current module. If there exist several functions having the same name in a given module, then their local indexes will not overlap. See appendix A.5 for an example of annotated code.

Those local indexes will be involved in the justification mechanism of MLcov (see section 5), since they will allow the user to consider a given program point or MC/DC decision as being covered, regardless of its actual coverage information.

4.2 Mapping between local indexes and coverage information (text)

As pointed out before, this file is intended to provide a mapping between each program point (or MC/DC decision) and its matching coverage information, with the following conventions:

- Program points and MC/DC decisions are tagged as 'OK' if they have been covered.
- Uncovered program points are tagged as 'NOT COVERED'.
- Uncovered MC/DC decisions are tagged as 'NOT REACHED'.

For each MC/DC decision, MLcov also indicates the set of test vectors corresponding to the current test base. Moreover, every condition in a decision is tagged as either 'OK' or 'NOT REACHED', and when it is 'OK' MLcov exhibits a pair of test vectors satisfying the independence criterion.

For sake of clarity, the various items are displayed in the same order as they appear in the annotated source code. Please refer to appendix A.6 for an example.

In addition to this, a summary of coverage statistics is provided at the beginning of the file. It shows the ratio and percentage of program points (and MC/DC decisions) that have been covered by the tests. This information is exactly similar to the HTML Summary of Coverage Measurements describing in paragraph 4.3.

4.3 Summary of Coverage Measurements (HTML)

The HTML summary of coverage measurements provides Structural Coverage and MC/DC statistics on the execution of instrumented code. See appendix A.3 for an example of such a coverage report.

With regard to Structural Coverage of expressions, it shows the total ratio and percentage of program points that have been covered by the testing phase (*i.e.*, that have been reached at least once while executing the instrumented code). In addition to this main result, a ratio and a percentage is given for every toplevel function in the program; functions that are not 100% covered are highlighted in red[2].

Concerning Modified Condition/Decision Coverage of Boolean expressions, MLcov shows the total ratio and percentage of covered conditions. It also gives a ratio and a percentage for every decision in the program; decisions for which MC/DC is not reached are highlighted in red.

Note that you can access to the source code report by clicking on either the name of a toplevel function or the number of an MC/DC decision.

4.4 Source Code Report (HTML)

The HTML source code report is an annotated version of the source code where every structural program point, every MC/DC decision and every condition has been marked.

Program points are highlighted in green as soon as they have been covered, in red when they are not covered. There is a special case for expressions whose evaluation never terminates (such as `failwith "error"`), as you may read in the [PADL'08] paper. That's why we decided to choose a special color for those expressions: they are highlighted in gray when covered (but still in red when they are not covered).

Conditions are highlighted in blue when MC/DC has been reached, in purple if MC/DC is not reached. Moreover, a system of colored boxes has been designed in order to help interpreting the coverage of conditions: a green box indicates that the condition always evaluated to true, a red box that it always evaluated to false, a black box that it was never evaluated yet. Finally, the whole MC/DC decision is always underlined, for sake of clarity.

Additional information is provided through tooltips[3]: *e.g.*, for each one of the covered conditions, MLcov exhibits a pair of test vectors satisfying the independence criterion. The local index (as explained in paragraph 4.1) of each program point or MC/DC decision is also displayed in this tooltip. Finally, MLcov displays a string identifying the last run(s) thanks to which a given counter or MC/DC condition has been reached: it is either the value of the `TEST_ID` environment variable (if it was set during execution), or the command line used for that run.

See appendix A.4 for an example of source code report.

[2]In this document, we describe the default color code used for HTML reports. However, you may define your own color code by overwriting the `mlcov.css` style sheet. Similarly, MLcov provides a hook to integrate JavaScript code (*e.g.* for syntax highlighting), as it includes a file `mlcov.js` from which functions `mlcov_summary()` and `mlcov_annot()` are invoked when the HTML Coverage and Source Code reports (respectively) have finished loading.

[3]For versions of Firefox older than 3.x, long tooltips are broken with "...", which is especially annoying when they contain useful informations. However, the *Long Titles* add-on, available at https://addons.mozilla.org, allows to overcome this issue.

5 Code Justification

It may occur that some parts of your code cannot be covered, in other words, will never be exercised whatever the testing phase is. You should probably delete this dead code, but for some reason (defensive programming, technical issue, *etc.*) you may want to keep it.

For example, it is quite common in ML programming to use catch-all patterns (_) to identify branches that are never reached, like in the following function, returning the sum of the first two elements of a list of integers:

```
let f l =
  assert (List.length l >= 2); (* precondition *)
  match l with
  | x1::x2::rem -> x1 + x2
  | _ -> failwith "empty or singleton list should not occur!" (* unreachable code *)
```

MLcov allows you to justify this uncovered code, thanks to the `-j` option:

```
mlcov -j <justif_file> <file>.mlcov
```

You may use a single justification file for several source files. For instance:

```
mlcov -j <justif_file> <file1>.mlcov <file2>.mlcov <file3>.mlcov
```

On the contrary, you may also split the justification base into several files:

```
mlcov -j <justif_file1> ...  -j <justif_fileN> <file>.mlcov
```

5.1 Format of the justification file

The file *<justif_file>* (see appendix A.9 for an example) shall be written in text format. It is parsed line by line. Blank lines and lines starting with the `'#'` character are ignored, which allows you to write some comments or structure the document. Other lines are either justification entries or definitions of labeled messages (`$<label> = <text>`).

Here is the BNF for a justification entry:

justif_entry ::= *<file>* ; *<toplevel_path>* ; *justif_pattern* ; *justif_message* ; *justif_author*
justif_pattern ::= `'cov'` ; *idx_list* | `'mcdc'` ; *idx_list* | `'*'`
idx_list ::= *<local_index>* | *<local_index>* , *idx_list*

where *justif_message* and *justif_author* can be either raw text or a reference (`$<label>`) to a labeled message defined elsewhere, which allows a better factorization of the justifications.

For instance, you should write:

```
myFile.ml ; my_fun ; cov ; 3,4,5 ; Justification ; Author
```

to justify the structural program points number 3, 4 and 5 in file `myFile.ml` and toplevel function `my_fun` (write `M1.M2.my_fun` if the function belongs to module `M2`, which is itself a submodule of `M1`).

Likewise, write:

```
myFile.ml ; my_fun ; mcdc ; 2 ; Justification ; Author
```

to justify the MC/DC decision number 2 in file `myFile.ml` at path `my_fun`.

The star character `'*'` is a wildcard expression:

```
myFile.ml ; my_fun ; * ; Justification ; Author
```

allows you to justify at once all the program points and all the MC/DC decisions in file `myFile.ml` at path `my_fun`.

Be careful that justification files are parsed sequentially, following the order of the command line's arguments, and that MLcov does not detect if several justification entries are bound to the same program point of MC/DC decision: in this case, the last entry overwrites the previous one. The same remark applies to the definition of labels. Possible parse errors are described in appendix B.

5.2 Computation of the statistics

MLcov considers a justified program point or MC/DC decision as being covered. Thus, if a whole toplevel function has been justified by the user, the statistics are computed as if this function was 100 % covered by the tests.

The name of a toplevel function should be unique within a given module, because there is no way to refer to the n-th function whose path is *<toplevel_path>*. If such a case appears, all the functions named *<toplevel_path>* are handled simultaneously by the justification mechanism.

In the text version of the coverage report, a justified program point or MC/DC decision is tagged as 'JUSTIFIED' and the matching justification is given. In the HTML source code report, a justified item is highlighted in yellow and its justification is provided as a tooltip.

During the generation of the annotated source report, a warning is displayed when a program point is both covered and justified (see appendix B). Moreover, in the HTML source code report, the justified but yet covered point is written in yellow on a black background.

6 Current Limitations

Supported subset of Objective Caml
As written before, MLcov 1.2 only applies to a subset of the Objective Caml language. Indeed, the tool was originally designed to be compliant with the coding standard used at Esterel Technologies[4].

In particular, the following constructs are currently not supported:

- objects
- polymorphic variants
- local, recursive or first-class modules
- streams
- multi-threading extensions

[4]Esterel Technologies has been using MLcov as a verification tool for the certification of a DO-178B level A code generator written in Objective Caml. For stability and traceability reasons, the subset of Objective Caml used by Esterel Technologies basically matches that of Caml Special Light, *i.e.* is restricted to the core and module languages.

Programs containing at least one unsupported construct cannot be instrumented by MLcov to measure Structural Coverage of expressions and MC/DC of Boolean expressions. For convenience, the original (non-instrumented) source file is still dumped to the standard output, so that typical build procedures can continue to work, especially when MLcov is passed through the `-pp` option of the Objective Caml compiler.

As a new (experimental) feature of MLcov 1.2, labels and optional arguments are now taken into account.

Instrumentation of tail-recursive functions
As explained in [PADL'08], MLcov's instrumentation algorithm systematically turns tail-recursive functions into non-tail-recursive ones, which may lead to unexpected stack overflows during the execution of the instrumented program.

Pretty-print of constrained signatures
Technically speaking, MLcov is built upon the frontend of the Objective Caml compiler. In particular, it relies on the type pretty-printer used by the `-i` option, which infers an interface definition from a given implementation file. Sometimes, when using constrained signatures such as `S with type t = t`, the pretty-printer may fail at producing a well-typed output (*cf.* bug #778 on http://caml.inria.fr/mantis). It means that, in those cases, the instrumented file cannot be compiled, so you need to rewrite the source file to circumvent the problem.

A Example

This appendix gives an example of use of MLcov on a bunch of ML files (`myFile1.ml`, `myFile2.ml` and `myFile3.ml`).

For sake of clarity, sections A.1 (source code), A.2 (test base), A.3 (HTML coverage report), A.4 (HTML source code report), A.5 (source code annotated with local indexes), A.6 (mapping between local indexes and coverage information) and A.7 (explanations) are exclusively related to `myFile1.ml`.

Section A.8 proposes an example of HTML global report, while section A.9 is an example of justification file.

A.1 Source Code

This simplistic ML program defines the evaluation of three logical formulae `formula1`, `formula2` and `formula3`, each one depending on a positive integer `n`. Here, a formula returns 1 when it is true (applied to a given integer), 0 otherwise.

```
let eval_and a b =
  if a && b then 1 else 0

let eval_or a b =
  if a || b then 1 else 0

let formula1 n =
  if n < 0 then assert false; (* precondition *)
  eval_and (n < 100) (n mod 2 = 0)

let formula1_bad n =
  if n > 0 then assert false; (* mistake in the precondition!  *)
  eval_and (n < 100) (n mod 2 = 0)

let formula2 n =
  eval_and (n >= 10) (n < 20)

let formula3 n =
  eval_and (n mod 2 = 0) (eval_and (n >= 10) (n < 20) = 1)
```

A.2 Test Base

This small test base will be used to measure the coverage of `myFile1.ml` (see previous section).

```
let _ =  MyFile1.formula1 3
let _ =  MyFile1.formula1 101
let _ =  MyFile1.formula1 102

let _ =  try MyFile1.formula1_bad 3 with Assert_failure _ -> 0

let _ =  MyFile1.formula2 5
let _ =  MyFile1.formula2 15
```

A.3 HTML Coverage Report

MLcov – Coverage Report (file `myFile1.ml`)

generated by MLcov, version 1.2

Structural coverage statistics

Function name	Covered points	Total points	Percentage
eval_and	2	2	100.0 %
eval_or	0	2	0.0 %
formula1	1	2	50.0 %
formula1_bad	1	2	50.0 %
formula2	1	1	100.0 %
formula3	1	1	100.0 %
TOTAL	**6**	**10**	**60.0 %**

MC/DC statistics

Decision number	Covered conditions	Total conditions	Percentage
#1 (eval_and)	2	2	100.0 %
#2 (eval_or)	0	2	0.0 %
#3 (formula1)	1	1	100.0 %
#4 (formula1)	1	1	100.0 %
#5 (formula1)	0	1	0.0 %
#6 (formula1_bad)	0	1	0.0 %
#7 (formula1_bad)	0	1	0.0 %
#8 (formula1)	0	1	0.0 %
#9 (formula2)	1	1	100.0 %
#10 (formula2)	0	1	0.0 %
#11 (formula3)	1	1	100.0 %
#12 (formula3)	1	1	100.0 %
TOTAL	**7**	**14**	**50.0 %**

A.4 HTML Source Code Report

MLcov – Source Code Report (file `myFile1.ml`)

generated by MLcov, version 1.2

```
let eval_and a b =
   if a && b then 1 else 0

let eval_or a b =
   if a || b then 1 else 0

let formula1 n =
   if n < 0 then assert false; (* precondition *)
   eval_and (n < 100) (n mod 2 = 0)
```

```
let formula1_bad n =
   if  n > 0  then assert false; (* mistake in the precondition!  *)
   eval_and  (n < 100)  (n mod 2 = 0)

let formula2 n =
   eval_and (n >= 10)  (n < 20)

let formula3 n =
   eval_and (n mod 2 = 0)  (eval_and (n >= 10) (n < 20) = 1)
```

A.5 Source Code annotated with local indexes

```
(*
Source Code Report (file myFile1.ml)
generated by MLcov, version 1.2
*)

let eval_and a b =
   if (*mcdc:1.1*) a && (*mcdc:1.2*) b then (*cov:1*) 1 else (*cov:2*) 0

let eval_or a b =
   if (*mcdc:1.1*) a || (*mcdc:1.2*) b then (*cov:1*) 1 else (*cov:2*) 0

let formula1 n =
   if (*mcdc:1.1*) n < 0 then (*cov:1*) assert false; (* precondition *)
   (*cov:2*) eval_and (*mcdc:2.1*) (n < 100) (*mcdc:3.1*) (n mod 2 = 0)

let formula1_bad n =
   if (*mcdc:1.1*) n > 0 then (*cov:1*) assert false; (* mistake in the precondition!  *)
   (*cov:2*) eval_and (*mcdc:2.1*) (n < 100) (*mcdc:3.1*) (n mod 2 = 0)

let formula2 n =
   (*cov:1*) eval_and (*mcdc:1.1*) (n >= 10) (*mcdc:2.1*) (n < 20)

let formula3 n =
   (*cov:1*) eval_and (*mcdc:1.1*) (n mod 2 = 0) (*mcdc:2.1*) (eval_and (n >= 10) (n < 20) =
1)
```

A.6 Coverage information

```
Coverage Report (file myFile1.ml)
generated by MLcov, version 1.2

--------------------------------
 Structural coverage statistics
--------------------------------
```

```
Function name ; Covered points ; Total points ; Percentage
     eval_and ;              2 ;            2 ;    100.0 %
      eval_or ;              0 ;            2 ;      0.0 %
     formula1 ;              1 ;            2 ;     50.0 %
 formula1_bad ;              1 ;            2 ;     50.0 %
     formula2 ;              1 ;            1 ;    100.0 %
     formula3 ;              1 ;            1 ;    100.0 %
        TOTAL ;              6 ;           10 ;     60.0 %

------------------
 MC/DC statistics
------------------

Decision number ; Covered conditions ; Total conditions ; Percentage
    1 (eval_and) ;                  2 ;                2 ;    100.0 %
     2 (eval_or) ;                  0 ;                2 ;      0.0 %
    5 (formula1) ;                  1 ;                1 ;    100.0 %
    3 (formula1) ;                  1 ;                1 ;    100.0 %
    4 (formula1) ;                  0 ;                1 ;      0.0 %
8 (formula1_bad) ;                  0 ;                1 ;      0.0 %
6 (formula1_bad) ;                  0 ;                1 ;      0.0 %
7 (formula1_bad) ;                  0 ;                1 ;      0.0 %
    9 (formula2) ;                  1 ;                1 ;    100.0 %
   10 (formula2) ;                  0 ;                1 ;      0.0 %
   11 (formula3) ;                  1 ;                1 ;    100.0 %
   12 (formula3) ;                  1 ;                1 ;    100.0 %
           TOTAL ;                  7 ;               14 ;     50.0 %

---------------------------------
 Structural coverage information
---------------------------------

eval_and:cov:1 = OK
eval_and:cov:2 = OK

eval_or:cov:1 = NOT COVERED
eval_or:cov:2 = NOT COVERED

formula1:cov:1 = NOT COVERED
formula1:cov:2 = OK

formula1_bad:cov:1 = OK
formula1_bad:cov:2 = NOT COVERED

formula2:cov:1 = OK

formula3:cov:1 = JUSTIFIED
   Justified by TM:
   not yet tested (structural cov)
```

```
-------------------
 MC/DC information
-------------------

eval_and:mcdc:1 = OK
   test vectors: {[1:T, 2:T]=T; [1:T, 2:F]=F; [1:F, 2:*]=F}
   cond:1.1 = OK ([1:F, 2:*]=F; [1:T, 2:T]=T)
   cond:1.2 = OK ([1:T, 2:F]=F; [1:T, 2:T]=T)

eval_or:mcdc:1 = NOT REACHED
   test vectors: {}
   cond:1.1 = NOT REACHED
   cond:1.2 = NOT REACHED

formula1:mcdc:1 = NOT REACHED
  test vectors: {[1:F]=F}
  cond:1.1 = NOT REACHED
formula1:mcdc:2 = OK
  test vectors: {[1:T]=T; [1:F]=F}
  cond:2.1 = OK ([1:F]=F; [1:T]=T)
formula1:mcdc:3 = OK
  test vectors: {[1:T]=T; [1:F]=F}
  cond:3.1 = OK ([1:F]=F; [1:T]=T)

formula1_bad:mcdc:1 = NOT REACHED
  test vectors: {[1:T]=T}
  cond:1.1 = NOT REACHED
formula1_bad:mcdc:2 = NOT REACHED
  test vectors: {}
  cond:2.1 = NOT REACHED
formula1_bad:mcdc:3 = NOT REACHED
  test vectors: {}
  cond:3.1 = NOT REACHED

formula2:mcdc:1 = OK
   test vectors: {[1:T]=T; [1:F]=F}
   cond:1.1 = OK ([1:F]=F; [1:T]=T)
formula2:mcdc:2 = NOT REACHED
   test vectors: {[1:T]=T}
   cond:2.1 = NOT REACHED

formula3:mcdc:1 = JUSTIFIED
   Justified by TM:
   not yet tested (MC/DC)
formula3:mcdc:2 = JUSTIFIED
   Justified by TM:
   not yet tested (MC/DC)
```

A.7 Explanations

From the latter reports, we can conclude that:

- Function `eval_and` has been fully covered, with respect to both Structural Coverage (2 points covered out of 2, highlighted in green) and MC/DC (reached for the 2 conditions, highlighted in blue).
- On the contrary, function `eval_or` is not covered at all. Indeed, this function is obviously dead code in `myFile1.ml`. Remember that uncovered points are highlighted in red wrt. Structural Coverage, and purple enclosed in a black box wrt. MC/DC.
- We can see that the precondition of function `formula1` has always been satisfied: the `n < 0` condition always evaluated to false (enclosed in a red box) and the `assert false` expression was never reached (highlighted in red). The following expression has been fully covered, wrt. both Structural Coverage (highlighted in green) and MC/DC (highlighted in blue).
- On the contrary, the precondition of function `formula1_bad` has been violated: the `n > 0` condition always evaluated to true (enclosed in a green box) and the `assert false` expression was reached (highlighted in gray, since it is an expression whose evaluation never terminates).
- Function `formula2` is covered wrt. Structural Coverage (1 point out of 1), but not wrt. MC/DC because the second condition has not been sufficiently tested. For instance, add `let _ = MyFile1.formula2 25` to the test base if you want to achieve full MC/DC coverage.
- Finally, it is straightforward that function `formula3` cannot be covered since it has never been invoked in the test base. We will consider justified (highlighted in yellow) its one and only structural point, and both its MC/DC decisions (*cf.* section A.9).

A.8 Global Report

MLcov – Global Report

generated by MLcov, version 1.2

File name	Cov. points	Total	Percentage	Cov. conditions	Total	Percentage
myFile1.ml	6	10	60.0 %	7	14	50.0 %
myFile2.ml	5	5	100.0 %	0	3	0.0 %
myFile3.ml	0	10	0.0 %	0	0	–
TOTAL	**11**	**25**	**44.0 %**	**7**	**17**	**41.17 %**

A.9 Justification File

```
##################
# JUSTIFICATIONS #
##################

# Common justifications:
$ASSERT = Defensive programming. Assertions are used for checking invariants at runtime\
or identifying unreachable branches.
```

```
# Justifications for myFile1:
myFile1.ml ; formula3    ; cov  ; 1   ; not yet tested (structural cov) ; TM
myFile1.ml ; formula3    ; mcdc ; 1,2 ; not yet tested (MC/DC)          ; TM

# Justifications for myFile2:
myFile2.ml ; compute_old ;      *     ; to be deleted (deprecated code) ; Bill
myFile2.ml ; _           ;      *     ; for testing purpose...          ; Bill
myFile2.ml ; f           ; cov  ; 1   ; $ASSERT                         ; Bill

# Justifications for myFile3:
myFile3.ml ; M1.M2.print ;      *     ; debugging information           ; Thomas
myFile3.ml ; g           ; cov  ; 2   ; $ASSERT                         ; Thomas
```

B Warnings and Errors

This appendix provides the list of warnings and errors that may be raised by MLcov 1.2. Note that all the warning and error messages are printed to the standard error.

Updating the trace

- A trace file `<trace>` cannot be used by MLcov if it does not correspond to a suitable version of the `MlcovTrace` trace handler. This typically occurs if you try to read or update a trace resulting from the instrumentation of any source code with an older version of MLcov. The existing trace file is then replaced with a new one, and the following warning is then raised:

```
>> Warning:  trace file "<trace>" ignored!  (wrong magic number)
```

- A trace file `<trace>` cannot be used by MLcov if it does not correspond to the suitable source code. The problem occurs when you try to update a trace resulting from the instrumentation of an older version of the same source code. The existing trace file is then replaced with a new one, and the following warning is raised:

```
>> Warning:  trace file "<trace>" ignored!  (bad source file checksum)
```

- Finally, a trace file `<trace>` cannot be read if it contains corrupted data. The existing trace file is then replaced with a new one, and the following warning is raised:

```
>> Warning:  trace file "<trace>" ignored!  (deserialization failure:  <msg>)
```

  where `msg` is a technical explanation of why the deserialization function failed.

Generating the coverage reports

- The following warning is raised by MLcov if the trace file `<trace>` you specified cannot be found:

```
>> Warning:  trace file "<trace>" ignored!  (does not exist)
```

- An entry of the global file (the one passed to the `-g` option) is ignored if its extension is not `.mlcov`, and the following warning is raised:

```
>> Warning:  file "<filename>" ignored, not a trace file!
```

- MLcov is not able to generate the annotated source report `<annotated_report>` (neither in text nor in HTML format) if the original source code `<source_code>` cannot be found; the following warning is then displayed:

```
>> Warning:  file "<annotated_report>" could not be generated!
                ("<source_code>" not found)
```

- MLcov is not able to generate the annotated source report `<annotated_report>` (neither in text nor in HTML format) if the trace file you specified does not correspond to the suitable source code `<source_code>`; the following warning is then displayed:

```
>> Warning:  file "<source_code>" changed since instrumentation,
             "<annotated_report>" not generated!
```

- A warning is emitted during the generation of the annotated source report if a program point is covered and justified at the same time:

```
>> Warning:  file "<source_code>", line <l>, program point #<i>
            has been reached but is also justified!
```

  This works with both text and HTML formats, but only regarding Structural Coverage. Be careful that MLcov does not detect justified decisions or conditions for which MC/DC has yet been reached.

Parsing the justification file

- A parse error is raised if the tag provided in a justification entry is neither `'cov'` nor `'mcdc'`:

```
>> Fatal error:  unknown tag '<tag>' in justification file "<filename>", line <l>
```

- A parse error is raised if one of the indexes provided in a justification entry is not a valid integer:

```
>> Fatal error:  invalid index '<idx>' in justification file "<filename>", line <l>
```

- If another error occurs while parsing a justification entry, the following message is displayed:

```
>> Fatal error:  bad format in justification file "<filename>", line <l>
```

- The following message is displayed if an error occurs while parsing a label definition:

```
>> Fatal error:  invalid labeled message in justification file "<filename>", line
                 <l>
```