Software Analyzers

The Metrics plug-in

For Frama-C 29.0 beta (Copper)

Richard Bonichon & Boris Yakobowski

Work licensed under Creative Commons BY-SA licence
https://creativecommons.org/licenses/by-sa/4.0/

© 2011-2024 CEA-List

Contents

1 **Quick overview** 3

1.1 Description . . . 3

1.1.1 Command-line options . . . 3

1.1.2 Metrics and abstract syntax trees . . . 3

1.2 Metrics on the normalized abstract syntax tree . . . 4

1.2.1 Syntactic metrics . . . 4

1.2.2 Graphical User interface . . . 4

1.2.3 Reachability coverage . . . 4

1.2.4 Eva coverage . . . 4

1.3 Metrics on the original abstract syntax tree . . . 7

2 **Practical notes and comments** 13

2.1 Cyclomatic complexity . . . 13

2.1.1 Calculation . . . 13

2.1.2 Practical notes . . . 13

2.2 Halstead complexity . . . 14

2.2.1 Calculation . . . 14

2.2.2 Practical notes . . . 14

2.3 Eva coverage . . . 14

Quick overview 1

The Metrics plug-in computes complexity measures on C source code. It can operate either on Frama-C's normalized abstract syntax tree (AST) or on the original source code.

1.1 Description

1.1.1 Command-line options

The complete list of command-line options can be obtained by:

```
% frama-c -metrics-help
```

Let us now detail some of them:

-metrics is a necessary switch to activate the plug-in. It also triggers the computation of syntactic metrics (slocs, number of statements of certain types, ...) on the normalized AST.

-metrics-by-function also computes (and displays) the above metrics, but this time on a per-function basis.

-metrics-ast is used to choose the AST the metrics should be computed on. It can be either Frama-C's normalized AST or the original AST. Section 1.1.2 covers this topic in some more details. How this affects the availability of metrics is discussed in Section 1.2 and 1.3.

-metrics-output redirects metrics' calculations to a file. The selection of the output is automatically detected from the extension of the output file. As of now, only .html or .txt outputs are supported.

-metrics-cover specifies a set of functions from which to compute a coverage estimate. This item is detailed in Section 1.2.4.

-metrics-eva-cover activates the Eva coverage estimation. This item is detailed in Section 1.2.4.

-metrics-libc controls whether functions from Frama-C's standard library are shown within the results. By default, this option is not set, and those functions are hidden.

1.1.2 Metrics and abstract syntax trees

Frama-C analyses usually operate on its own internal *normalized* abstract syntax tree[1]. The normalization process adds for example missing return statements, transforms all loops into while(1) { ... } statements, introduces some temporary variables... Although this normalization process does not affect the semantic

[1] Note that the parsing machinery and the production of the abstract syntax trees originally come from CIL (http://cil.sourceforge.net/).

contents of the code, its syntactic counterpart is indeed changed. It can therefore affect metrics computed from the source code.

However, Frama-C also keeps the original abstract syntax tree of the source as is. Even though Frama-C's analyses are usually centered around the normalized representation, some facilities do exist to manipulate the other original AST.

Users of the Metrics plug-in can specify the nature of the AST from which the metrics should be computed. Some metrics are available only for one AST representation.

The default behavior, as enabled by the `-metrics` switch, is to calculate syntactic metrics on the normalized AST. Metrics available for each AST are the objects of Sections 1.2 and 1.3.

1.2 Metrics on the normalized abstract syntax tree

1.2.1 Syntactic metrics

Only cyclomatic numbers are available for the normalized AST. These are also available through Frama-C's graphical user interface (GUI) (see Section 1.2.2).

Let us calculate the cyclomatic complexity of the program of Figure 1.1 using the following command [2].

```
% frama-c -metrics -metrics-by-function reach.c
```

The results are detailed in Figures 1.2 and 1.3. The output contains a summary, for each function, of the number of assignments, function calls, exit points, declared functions (it should always be 1), goto instructions, if statements, pointer dereferencings and lines of code. The cyclomatic number of the function is also computed. Per-function results are available only if the option -metrics-by-function is specified; otherwise only the "Syntactic metrics" part of the output is shown (Figure 1.3). Note that these results can be printed to a text or html file using the -metrics-output option.

1.2.2 Graphical User interface

Metrics on the normalized AST are also accessible through Frama-C's GUI. Cyclomatic complexity numbers are shown either globally on the left-hand side pane of the GUI, after left-clicking the Measure button (see Figure 1.4), or for a chosen function (see Figure 1.5). To access this functionality, you must right-click on the line where the function is defined to make a menu appear, then left-click on Metrics as shown in the figure.

1.2.3 Reachability coverage

Given a function f, the reachability coverage analysis over-approximates the functions of the program that can be called from f. On our example, to activate it on the functions main and foo, one can use:

```
% frama-c -metrics -metrics-cover main,foo reach.c
```

The results are displayed in Figure 1.6. The reachability coverage analysis is conservative. For example, it considers that all function whose addresses are referenced within a reachable function may be called. This explains why it considers that baz and foo are reachable from the main function.

1.2.4 Eva coverage

The -metrics-eva-cover option can be used to compare the code effectively analyzed by Eva with what Metrics considers reachable from the main function (wrt. the criterion described in Section 1.2.3). The

[2] frama-c -metrics -metrics-by-function -metrics-ast cil reach.c is also a valid command for this purpose.

```c
void (*bar) (int); void (*t[2])(int);

void baz (int j) { return; }

void (*t[2])(int)= {
  baz,
  0};

void foo (int k) {
  int i = 0;
  return;
}

/* foo is unreachable since j is always 0; baz is not called */
int main() {
  int j = 0;
  void (*(*pt)[2])(int) = &t;
  if (!j) {
    return 1;
  }
  else {
    bar = foo;
    bar (1);
    return 0;
  }
}
```

Figure 1.1: Source code for reach.c

```
  Stats for function <tests/metrics/reach.c/baz>
  ==============================================
  Sloc = 1
  Decision point = 0
  Global variables = 0
  If = 0
  Loop = 0
  Goto = 0
  Assignment = 0
  Exit point = 1
  Function = 1
  Function call = 0
  Pointer dereferencing = 0
  Cyclomatic complexity = 1

  Stats for function <tests/metrics/reach.c/foo>
  ==============================================
  Sloc = 2
  Decision point = 0
  Global variables = 0
  If = 0
  Loop = 0
  Goto = 0
  Assignment = 1
  Exit point = 1
  Function = 1
  Function call = 0
  Pointer dereferencing = 0
  Cyclomatic complexity = 1

  Stats for function <tests/metrics/reach.c/main>
  ===============================================
  Sloc = 12
  Decision point = 1
  Global variables = 0
  If = 1
  Loop = 0
  Goto = 2
  Assignment = 5
  Exit point = 1
  Function = 1
  Function call = 1
  Pointer dereferencing = 1
  Cyclomatic complexity = 2
[metrics] Defined functions (3)
```

Figure 1.2: Output of by-function syntactic metrics for the normalized AST of reach.c

results of this option on our example are given in Figure 1.7. This particular feature is activated by the following command:

```
% frama-c -metrics -metrics-eva-cover reach.c
```

Syntactic reachability is an over-approximation of what is actually reachable by Eva. Thus, the coverage estimation will always be equal to or less than 100%, especially if the source code relies a lot

```
====================
 baz (address taken) (0 call); foo (address taken) (0 call); main (0 call);

Specified-only functions (0)
============================


Undefined and unspecified functions (0)
=======================================


'Extern' global variables (0)
=============================


Potential entry points (1)
==========================
 main;

Global metrics
==============
Sloc = 15
Decision point = 1
Global variables = 2
If = 1
Loop = 0
Goto = 2
Assignment = 6
Exit point = 3
Function = 3
Function call = 1
Pointer dereferencing = 1
Cyclomatic complexity = 4
```

Figure 1.3: Output of global syntactic metrics for the normalized AST of reach.c

on the use of function pointers.

For all functions that are considered syntactically reachable, but that are not analyzed by Eva, the plug-in indicates the locations in the code where a call *might* have been analyzed. In our example, this consists in the call to foo at ligne 24. Also, since the address of baz is contained in the initializer of the array t (itself referenced in main), baz is considered as callable; thus the plug-in signals the initializer of t as a possible calling point. Finally, the plug-in displays the percentage of statements analyzed by Eva for each function.

1.3 Metrics on the original abstract syntax tree

Only syntactic metrics are available on the original AST. Both Halstead and cyclomatic complexity are computed. Note that this part of the plug-in cannot be used through the GUI.

```
% frama-c -metrics -metrics-by-function -metrics-ast cabs reach.c
```

The effect is to calculate both cyclomatic and Halstead complexities for the argument files, as shown in Figures 1.8 and 1.9 . The results for Halstead measures are only global while cyclomatic numbers can

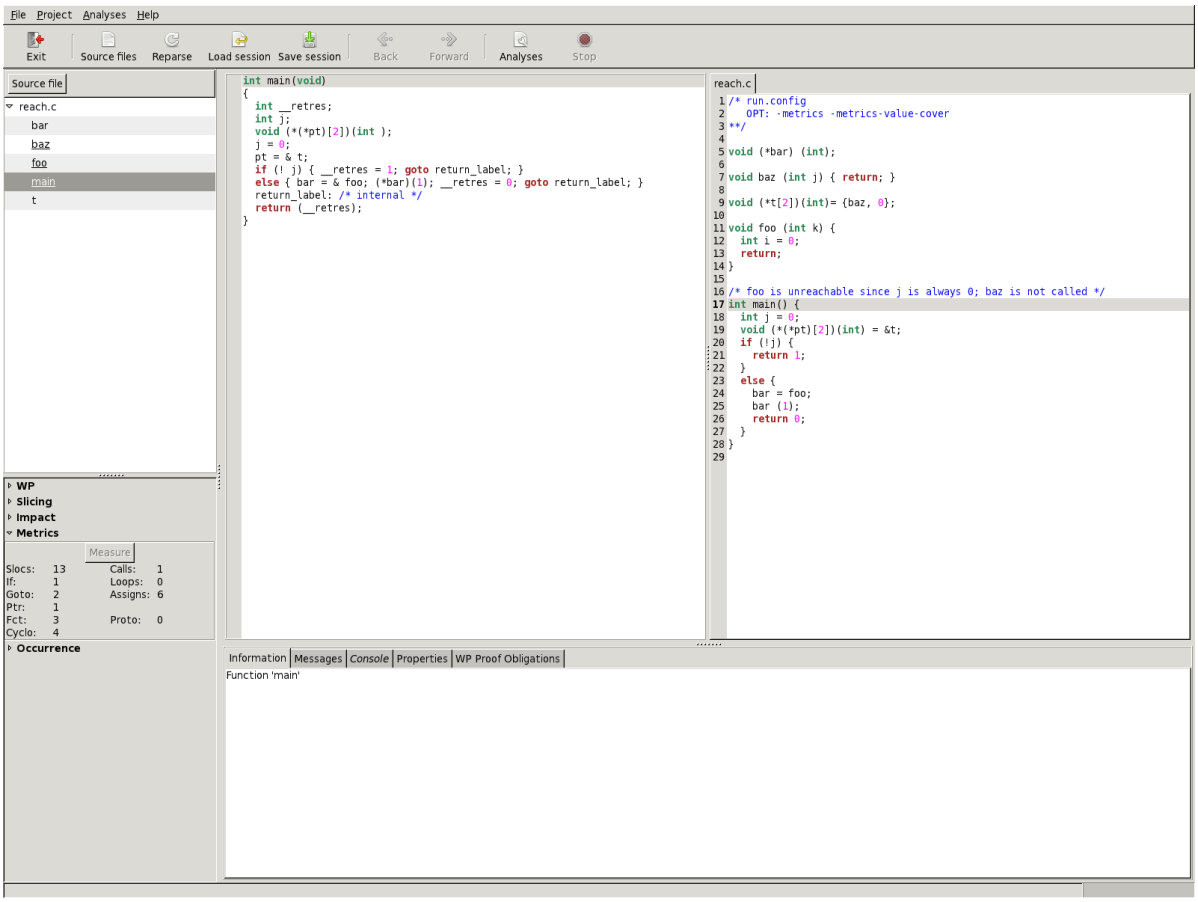

Figure 1.4: Metrics GUI: calculate global metrics

be done on a per-function basis. Halstead measures also produce a detailed account of the syntactic elements of the files.

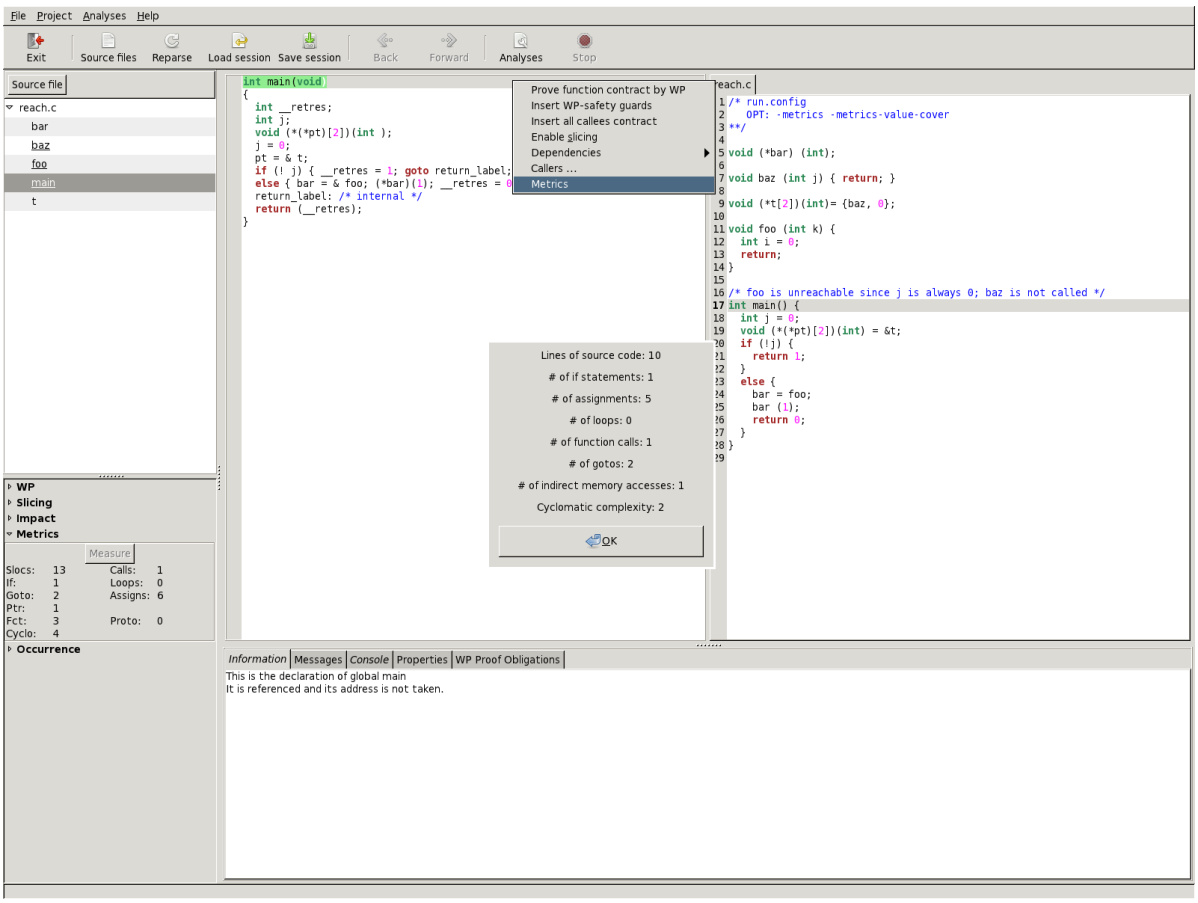

Figure 1.5: Metrics GUI: calculate metrics for a function

```
  Function call = 1
  Pointer dereferencing = 1
  Cyclomatic complexity = 4
[metrics] Functions syntactically reachable from foo: 1
  ------------------------------------------------
  <tests/metrics/reach.c>: foo;

  Functions syntactically unreachable from foo: 2
  ------------------------------------------------
  <tests/metrics/reach.c>: baz; main;
[metrics] Functions syntactically reachable from main: 3
  ------------------------------------------------
  <tests/metrics/reach.c>: baz; foo; main;

  Functions syntactically unreachable from main: 0
  ------------------------------------------------
```

Figure 1.6: Reachability coverage for reach.c

```
 ------------------------------------------------------------------------------
 1 function analyzed (out of 3): 33% coverage.
 In this function, 7 statements reached (out of 12): 58% coverage.
 ------------------------------------------------------------------------------
 No errors or warnings raised during the analysis.
 ------------------------------------------------------------------------------
 0 alarms generated by the analysis.
 ------------------------------------------------------------------------------
 No logical properties have been reached by the analysis.
 ------------------------------------------------------------------------------
[metrics] Eva coverage statistics
 ======================
 Syntactically reachable functions = 3 (out of 3)
 Semantically reached functions = 1
 Coverage estimation = 33.3%

 Unreached functions (2) =
   <tests/metrics/reach.c>: baz; foo;
[metrics] References to non-analyzed functions
 ------------------------------------
 Function main references foo (at tests/metrics/reach.c:27)
 Initializer of t references baz (at tests/metrics/reach.c:11)
[metrics] Statements analyzed by Eva
 -------------------------
 12 stmts in analyzed functions, 7 stmts analyzed (58.3%)
 main: 7 stmts out of 12 (58.3%)
```

Figure 1.7: Eva coverage estimate for reach.c

```
[metrics] Cabs:

 Stats for function <tests/metrics/reach.c/baz>
 ================================================
 Sloc = 1
 Decision point = 0
 Global variables = 0
 If = 0
 Loop = 0
 Goto = 0
 Assignment = 0
 Exit point = 1
 Function = 1
 Function call = 0
 Pointer dereferencing = 0
 Cyclomatic complexity = 1

 Stats for function <tests/metrics/reach.c/foo>
 ================================================
 Sloc = 2
 Decision point = 0
 Global variables = 0
 If = 0
 Loop = 0
 Goto = 0
 Assignment = 0
 Exit point = 1
 Function = 1
 Function call = 0
 Pointer dereferencing = 0
 Cyclomatic complexity = 1

 Stats for function <tests/metrics/reach.c/main>
 ================================================
 Sloc = 9
 Decision point = 1
 Global variables = 0
 If = 1
 Loop = 0
 Goto = 0
 Assignment = 1
 Exit point = 2
 Function = 1
```

Figure 1.8: Cyclomatic metrics on the original AST for reach.c

```
================
Total operators: 32
Distinct operators: 13
Total_operands: 18
Distinct operands: 9
Program length: 50
Vocabulary size: 22
Program volume: 222.97
Effort: 2898.63
Program level: 0.08
Difficulty level: 13
Time to implement: 161.04
Bugs delivered: 0.07


Global statistics (Halstead)
============================
Operators
---------
  if: 1
  return: 4
  ): 1
  ,: 2
  {: 2
  (: 1
  }: 2
  ;: 10
  =: 1
  !: 1
  &: 1
  int: 2
  void: 4

Operands
--------
  foo: 1
  pt: 1
  baz: 1
  t: 3
  j: 2
  bar: 3
  i: 1
  1: 2
  0: 4
```

Figure 1.9: Halstead metrics on the original AST for reach.c

Practical notes and comments 2

2.1 Cyclomatic complexity

Cyclomatic complexity, also called conditional complexity was introduced by Thomas McCabe [2] in 1976. It is a measure of the number of paths through a source code and represent the complexity of the control-flow of the program.

The cyclomatic number of a source code has been shown to be weakly correlated to its number of defects.

2.1.1 Calculation

Cyclomatic complexity is a notion defined on a directed graph. It can therefore be defined on a program taken as its control-flow graph. For a directed graph, the cyclomatic complexity C is defined as

$$C = E - N - 2P$$

where E is the number of edges of the graph, N the number of nodes and P the number of (strongly) connected components.

This notion of complexity is extended to deal with programs using the following formula

$$C = \pi - s + 2$$

where π is the number of decision points in the program and s the number of exit points.

2.1.2 Practical notes

Cyclomatic complexity can be computed on all abstract syntax trees in Frama-C. The resulting complexity will nonetheless stay the same in both AST representations, as Frama-C's normalized AST does not add control-flow directives to the source code.

Prior to the computation of cyclomatic complexity, the plug-in gathers the following syntactic information from the source code:

- Number of lines of code (assuming one C statement equals one line of code);
- Number of if statements;
- Number of loops;
- Number of function calls;
- Number of gotos;
- Number of assignments;
- Numbers of exit points (return statements);
- Number of functions declared;
- Number of pointer derefencings.

– Number of decision points (conditional statements (if) and expressions (`? :`), switch cases, lazy logical operators, loops).

These informations are computed both for the complete source code, and on a per-function basis – except for the number of functions declared.

Cyclomatic complexity is then derived from these informations, both for the full code and for each defined function.

2.2 Halstead complexity

Halstead complexity is as set of software metrics introduce by Maurice Halstead [1] in 1977. The goal is to identify measurable properties of the code and to go beyond pure complexity measures.

2.2.1 Calculation

Halstead complexity measures first need the following informations from the source code:

– η_1 is the number of distinct operators;
– η_2 is the number of distinct operands;
– N_1 is the total number of operators;
– N_2 is the total number of operands;

From the above informations, Halstead defines the following measures:

Program vocabulary	η	$\eta_1 + \eta_2$
Program length	N	$N_1 + N_2$
Calculated program length	$\hat{N}$	$n_1 \times \log_2 \eta_1 + \eta_2 \times \log_2 \eta_2$
Volume	V	$N \times \log_2 \eta$
Difficulty	D	$(\eta_1/2) \times N_2/\eta_2$
Effort	E	$D \times V$
Time required to program	T	$E/18$
Bugs	B	$E^{2/3}/3000$

Note that “Time required to program” is an estimate given in seconds.

2.2.2 Practical notes

To implement the measures defined in Section 2.2.1, it is necessary to define what the operands and operators of the language are. For Frama-C, the target language is C and we define its operands and operators as follows:

Distinct operands Identifiers and constants are operands, as well as type names and type specification keywords.

Distinct operators Storage class specifiers, type qualifiers, reserved keywords of C and other operators $(+, ++, +=, ...)$ are considered as operators.

It is important to note that the measure of bugs delivered is considered under-approximated for programs written in C.

2.3 Eva coverage

This part of the Metrics plug-in is thought of as a help for new code exploration with Frama-C’s Eva plug-in. The first steps into a new code can be quite complicated and, more often than not, Eva stops due

to too much imprecision. This imprecision can have a lot of different causes (body of library functions missing, imprecisions of reads in memory, ...).

The graphical user interface helps visualizing where Eva has stopped. The penetration estimate of Metrics aims to complement that by giving a rough approximate of the percentage of code that Eva has seen. This is especially interesting when comparing two runs of Eva on the same code, after performing some tweaks and improvements to help Eva: one can thusly quantify the added penetration due to the tweaks.

Bibliography

[1] Maurice H. Halstead. *Elements of Software Science*. Elsevie North-Holland, 1977.

[2] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.