Software Metrics for Human Developers

Pedro Faria, Pedro Silva, and Ulisses Costa

Department of Informatics, University of Minho

Abstract. Software Metrics can be applied to a piece of software or to it's specification. Our tool is focused on evaluating software quality, therefore, analysing a software piece. Software Metrics end purpose is, for example, to evaluate the quality of the software, estimate production cost and to do budget planning.
As a test case study we used a system for managing programming contests. The software submitted in the contest participation is used for quality evaluation. In this paper, we present our tool and it's features as the technology used to develop them.

Keywords: Software Metrics, Haskell, Traversal Strategies, RoR, Perl

1 Introduction

Quantitative measurements are essential in all sciences, and computer science is no exception. Although Software Metrics aren't often used in Software Development, there has been an effort on the computer science community to develop and improve this metrics. The goal is to make them valuable in many aspects, such as quality assurance.
We present a static code analyser, that by measuring a set of metrics over C code, will give a notion of quality to the user. The software is able to generate reports in LaTeX and in *XML* format.
A system for managing programming contests, was chosen as the case study. The competitors submit source code, which is their attempt to resolve a given problem. The goal is not only to check if the solution is correct, but evaluate the quality of the submited code.

2 Architecture and features

Our test case tool is a system for managing programming contests. It can be accessed trough a Web Interface which is a *RoR*[1] application, and some of it's features can also be used from the terminal, taking advantage of a *Perl* written script. In Figure 1, we present a diagram that shows a simplified version of the system architecture.

[1] Ruby on Rails - `http://rubyonrails.org/`

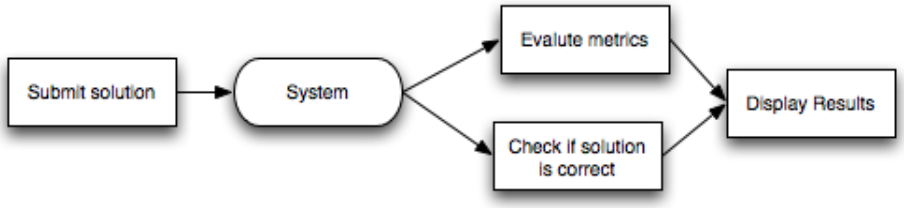

Fig. 1. Simplified System Architecture

2.1 *RoR* interface and *Perl* interface

As it would be expected, our system allows the creation of new contests, where among other things, we can stipulate the date in which the contest starts and ends, and also it's durations. Several exercises can be added to each contest, either manually, or by submitting one or more *XML* files. An exercise has a description and a set of input's and output's that the program will try to pass. Having the contest propely created and being available to the competitors, they can register to be a contest participant, and then start submitting their solutions to each of the contest exercises.

2.2 Parser and Traversal Strategies

All of the work related to C files parsing, metrics extraction from them and output generation is done using Haskell programming language. To extract some metrics we use a parser library[2] done by Benedikt Huber under a GSoC (Google summer of code). With this library we are able to construct a small parsing tree with the complete AST of C [Ker88] and some GNU C extensions.

To have an idea we only have 84 types of nodes (type constructors) in this tree, even so it is too big to traverse it by constructor, so it was mandatory to use strategies to traverse this tree. We decide to use Strafusnki library [LV03].

3 Metrics and Assessment

3.1 API design

To be able to manipulate and store the calculated metrics we have develop an API. Here are the relevent data types:

$$\begin{aligned}&\textbf{type}\ \mathit{Metrics} &&= \mathit{Map\ MetricName\ MetricValue}\\ &\textbf{type}\ \mathit{MetricName} &&= (\mathit{Name},\ \mathit{Maybe\ FileName},\ \mathit{Maybe\ FunctionName})\\ &\textbf{data}\ \mathit{MetricValue} &&= \mathit{Num\ Double}\end{aligned}$$

[2] Language.C - http://trac.sivity.net/language_c/

$$
\begin{array}{l}
\quad \mid \mathit{Clone}\ (\mathit{Map}\ \mathit{FileDst}\ [(\mathit{Ocurrency},\ \mathit{LineSrc},\ \mathit{LineDst})]) \\
\quad \mid \mathit{Includes}\ ([\mathit{SystemIncludes}], [\mathit{Includes}])
\end{array}
$$

We have used a Map to store our metrics, where the key is the triple $MetricName$. Some metrics just make sense regarding all the software package, others may only be relevant regarding files and other may be extracted from a function. By letting some of the $Maybe$ fields qith the value $Nothing$ we express this three categories of metrics. Regarding the $MetricValue$ we can have multiple types of metrics, most of them just return a numerary value, the cosntructor Num. And then we have more specific metrics related to clones found in the code, list of include and system includes.

Now we present one of the most important function to append metrics. The power of *Haskell* infix notation allow us to use comfortably this kind of functions.

$$
\begin{array}{l}
>\!\circ\!> :: \mathit{Metrics} \quad \rightarrow (\mathit{MetricName},\ \mathit{MetricValue}) \rightarrow \mathit{Metrics} \\
m\ >\!\circ\!>\ (mn, mv) = \textbf{case}\ \mathit{lookup}\ mn\ mv\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \textbf{Nothing} \quad \rightarrow m' \\
\qquad\qquad\qquad\qquad\qquad (\textbf{Just}\ mv') \rightarrow \textbf{if}\ mv' \equiv mv\ \textbf{then}\ m\ \textbf{else}\ m' \\
\quad \textbf{where} \\
\qquad m' = \mathit{insert}\ mn\ mv\ m
\end{array}
$$

The previous function can be used when we want create a new metric, if so we must use $emptyMetrics$ as first paramenter. The following function can be used when the user already have some metrics calculated and want to concat them into one.

$$
\begin{array}{l}
>\!+\!> \qquad\qquad :: \mathit{Metrics} \rightarrow \mathit{Metrics} \rightarrow \mathit{Metrics} \\
m_1\ >\!+\!>\ m_2 = \mathit{union}\ m_1\ m_2
\end{array}
$$

As an example we can use this function to take a list of $Metrics$ and concat all of them into just one $Metrics$ value. If we have a function

$$mccabeIndex\ ::\ FileContents \rightarrow IO\ Metrics$$

We can apply this function to a list of files lst and concat the results into one single $Metrics$ bag:

$$mapM\ mccabeIndex\ lst \gg\!= return \circ foldl\ (>\!+\!>)\ emptyMetrics$$

3.2 Metrics

Our system is able to calculate some well known metrics. We have divided this metrics in some groups: Metrics over comments, Complexity metrics, Metrics over functions, Metrics over includes and Metrics over lines quantity.

Comments metrics are used to understand the overall matureness of the software project, so we consider *lines of comments (lc)* as the non code comments number of lines and *comment lines density* as $cld = \frac{lc}{nrLines+lc} * 100$.

Complexity metrics for now our system is only capable of mesure one metric, Cyclomatic Complexity from one overall file or from a function. This metric is calculates giving the parsing tree of a C file (or C function) and for everyone condition in the code increment the value 1. So, for each If, $Switch$, For and $While$ statement we add 1 to our result.

Functions metrics and Includes metrics are not exactly quality metrics, but more a global view of the software project. We provide the following metrics: *get functions name* that returns a list of functions names from a parsing tree and *get functions signature* that returns a list of a signature node in the parsing tree (we can use pretty print functions provided by Language.C to print them out). For Includes metrics, we provide a function that receives the file name and returns a tuple containing two lists: the system includes and the library includes.

Metrics over lines where we extract some metrics related to lines, so for this metrics we do not use Language.C library. We implement *number of lines of code (ncloc)*, *number of physical lines* and two functions to exctract line clones and block of lines clones (one block could be 3 lines).

4 Conclusion and Future Work

The submission plataform, is already a very mature application, mainly when accessed by the Web Interface. In spite of not being the most important part of the tool, it does all that a normal submission plataform for programming contests should do. The metrics calculation application is now able to calculate a set of metrics. Several quality notions can be infered using this metrics. This interpretation is not yet available, being this one of the planed features to be implemented in our tool.

Acknowledgments

The authors wish to thank Professor João Saraiva for the inpiration and for having guided this work. The professor José João to help us doing the makefile and Jacomé Cunha for helping us installing Strafunski library.

References

Brian W. Kernighan. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988.

R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.