Annales Mathematicae et Informaticae
42 (2013) pp. 29–44
http://ami.ektf.hu

Complexity metric based source code transformation of Erlang programs[*]

Roland Király

Eszterházy Károly Collage
Institute of Mathematics and Informatics
kiraly.roland@aries.ektf.hu

Submitted December 4, 2013 — Accepted December 11, 2013

Abstract

In this paper we are going to present how to use an analyzer, which is a part of the *RefactorErl* [10, 12, 13], that reveals inadequate programming style or overcomplicated erlang [14, 15] program constructs during the whole lifecycle of the code using complexity measures describing the program. The algorithm [13], which we present here is also based upon the analysis of the semantic graph built from the source code, but at this stage we can define default complexity measures, and these defaults are compared to the actual measured values of the code, and so the differences can be indicated. On the other hand we show the algorithm measuring code complexity in Erlang programs, that provides automatic code transformations based on these measures. We created a script language that can calculate the structural complexity of Erlang source codes, and based on the resulting outcome providing the descriptions of transformational steps. With the help of this language we can describe automatic code transformations based on code complexity measurements. We define the syntax [11] of the language that can describe those series of steps in these automatic code refactoring that are complexity measurement [7, 9] based, and present the principle of operation of the analyzer and run-time providing algorithm. Besides the introduction of the syntax and use cases, We present the results we can achieve using this language.

Keywords: software metrics, complexity, source code, refactorerl

[*]This research was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4. A/2-11-1-2012-0001 'National Excellence Program'.

1. Introduction

Functional programming languages, thus Erlang as well, contain several special program constructs, that are unheard of in the realm of object-oriented and imperative languages.

The special syntactic elements make functional languages different, these attributes contribute to those being interesting or extraordinary, but also due to these, some of the known complexity measures are not, or only through modifications usable to measure code.

This does not mean that complexity measures are not developed to these languages, but very few of the existing ones are generic enough to be used with any functional language [3, 4, 8] language-independently, therefore with Erlang as well, because most of these only work well with one specific language, thus have low efficiency with Erlang codes.

For all of this I needed to define the measures of complexity that can be utilized with this paradigm, and create new ones as necessary.

There are tools for measuring software complexity, like Eclipse [6], or the software created by Simon, Steinbrückner and Lewerentz, that implements several complexity measures that help the users in measurement.

The aim of the *Crocodile* [5] project is to create a program that helps to efficiently analyze source code, therefore it can be used quite well to makes measurements after code transformations. *Tidier* [17, 18] is an automatic source code analyzer, and transformer tool, that is capable of automatically correcting source code, eliminating the syntactic errors static analysis can find, but neither software/method uses complexity measures for source code analysis and transformation.

This environment raised the demand for a complex and versatile tool, that is capable of measuring the complexity of Erlang codes, and based on these measurements localize as well as automatically or semi-automatically correct unmanageably complex parts.

We have developed a tool *RefactorErl* [12, 10, 13] which helps to performing refactoring steps. In the new version of the tool we implemented the algorithm and the transformation script language, which enables to write automatic metric based source code transformations.

Problem 1.1 (Automated program transformations)**.** *In this article we examined the feasibility of automated transformation of (functional) Erlang [14, 15] programs' source codes based on some software complexity measures, and if it is possible to develop transformation scheme to improve code quality based on the results of these measurements.*

In order to address the problem we have created an algorithm that can measure the structural complexity of Erlang programs, and can provide automatic code transformations based on the results, we have also defined a script language that offers the description of the transformation steps for the conversion of different program designs.

In our opinion the analysis of complexity measures on the syntax tree created from the source code and the graph including semantic information built from this [12, 16] allows automatic improvement of the quality of the source code.

To confirm this statement, we attempt to make a script that improves a known *McCabe* complexity measure, namely the cyclomatic number (defined in Chapter 2.), in the language described in the first section of Chapter 4., and run this on known software components integrated in Erlang distributions.

We chose *McCabe*'s cyclomatic number for testing, because this measure is well enough known to provide sufficient information on the complexity of the program's source code not only to programmers that are familiar with Erlang or other functional languages.

With the examination of the results of measurements performed in order to validate our hypothesis, and with the analysis of the impact of the transformations we addressed the following questions:

- The modules' cyclomatic number is characterized by the sum of the cyclomatic numbers of the functions. This model cannot take into account the function's call graph, which distorts the resulting value. Is it worthwhile to examine this attribute during the measurements, and to add it to the result?

- Also in relation with the modules, the question arises as to which module is more complex: one that contains ten functions, all of whose *McCabe* value is 1, or one that has a function bearing a *McCabe* value of 10?

- The cyclomatic number for each function is at least one, because it contains a minimum of one path. Then if we extract the more deeply embedded selection terms from within the function, in a way that we create a new function from the selected expression (see Chapter 2.) the cyclomatic number that characterizes the module increases unreasonably (each new function increases it by one). Therefore, each new transformation step is increasingly distorting the results. The question in this case is that this increase should or should not be removed from the end result?

- Taking all these into account, what is the relationship between the cyclomatic number of the entire module, and the sum of the cyclomatic numbers of the functions measured individually?

- How can we best improve the cyclomatic number of Erlang programs, also what modifications should be carried out to improve the lexical structure, the programming style, of the program?

- If a function contains more consecutive selections and another one embeds these into each other, should the cyclomatic numbers of the two functions be regarded as equivalent?

2. Used complexity metrics

In this chapter, for the sake of clarity, we define the complexity metrics that are used during the application of the scripts that manage the transformations. Out of the applicable metrics of the analytical algorithm we have made, in the present writing we only use the *McCabe* cyclomatic number, the *case* statements' maximum embeddedness metric, and the measuring of the number of functions, therefore we only describe these in detail.

The *McCabe* McCabe complexity measure is equivalent to the number of basic routes defined in the control graph [1] constructed by *Thomas J. McCabe*, namely how many types of outputs can a function have not counting the number of the traversal paths of the additionally included functions.

The *McCabe* cyclomatic number was originally developed for the measurement of subprograms in procedural languages. This metric is also suitable for the measurement of functions implemented within modules in functional languages, such as Erlang [14]. *Thomas J. McCabe* defines the cyclomatic number of programs as follows:

Definition 2.1 (McCabe cyclomatic number)**.** The $G = (v, e)$ control flow graph's $V(G)$ cyclomatic number is $V(G) = e - v + 2p$, where p represents the number of the graph's components, which corresponds to the number of linearly connected loops that are located in the strongly connected graph [9].

The *McCabe* number to measure the functions of Erlang programs can be specified as follows:

Definition 2.2 (McCabe in Erlang)**.** Let f_i be the branches (overload versions) of the $fc(f_i)$, function, let $if_{cl}(f_i)$, and $case_{cl}(f_i)$ denote the branches of the if's, and $case$'s within the branches. Then the result of the *McCabe* cyclomatic number measured for functions is $MCB(f_i) = |fc(f_i)| + |case_{cl}(f_i)| + |if_{cl}(f_i)|$.

The measure can be applied to a group of functions:

$$MCB(f_1, ..., f_k) = \sum_{j=1}^{k} MCB(f_j).$$

The results measured on the module's functions $m_i \in M$ are equal to the sum of the results measured on all the function from within the module:

$$MCB(m_i) = MCB(F(m_i))$$

The next measure of complexity we use measures the maximum embedding of the case statements within the functions.

$$MCB(M) = \sum_{m \in M} MCB(F(m))$$

The next measure of complexity we use measures the maximum embedding of the *case* statements within the functions.

```
$c_0$:
  case $e$ of
    $p_1$ [when $g_1$] $\rightarrow e_1^1, \dots, e_{l_1}^1$;
    $\vdots$
    $p_n$ [when $g_n$] $\rightarrow e_1^n, \dots, e_{l_n}^n$
  end
```

denotes an Erlang *case* case statement, where e, ands $e_i \in E$ are Erlang expressions, $p \in P$ are patterns, $g_i \in G$ are guards in the branches. The e_j^i expressions in branches of the *case* statements may contain nested control structures, including further *case* expressions.

Definition 2.3 (Max depth of cases)**.** In order to measure the embeddedness $T(f_i)$ be the set of all *case* expression located in the f_i function. Let $t(c_1, c_2)$ denote any branch of the *case* expression c_1 that contains *case* expression c_2 and $\nexists c_3$ *case* expression that $t(c_1, c_3) \wedge t(c_3, c_2)$. Let $t_s(c, c_x)$ denote the case that *case* expression c contains in one of its branches, at some depth *case* expression c_x that is $\exists$ $c_1, ..., c_n$ *case* expressions so that

$$t(c, c_1), t(c_1, c_2), ..., t(c_{n-1}, c_n), t(c_n, c_x).$$

The $|t_s(c, c_x)|$'s embeddedness depth in this case is $n + 1$. Let $T_0(f_i)$ be the set of those *case* expressions which are not contained in any of the $T(f_i)$ set's *case* expressions (top-level *case* statement). Then the

$$MDC(f_i) = max\{|t_s(c, c_x)| \ \ |c \in T_0(f_i), c_x \in T(f_i)\}.$$

After defining the embeddedness depth let us inspect the third metric we have applied, which measures the number of functions in the modules. This measure is particularly relevant in the characterization of functional programs, since these contain a large number of function-constructions, so in addition to the number of rows, by using this metric we can infer the size of the modules. The general definition of the functions in Erlang can be described by the following formula:

```
$f_0$:
  $\mathsf{f}_1^c(p_1)$ when $g_1$ $\rightarrow$
    $e_1^1, \dots, e_{l_1}^1$;
    $\vdots$
  $\mathsf{f}_n^c(p_n)$ when $g_n$ $\rightarrow$
    $e_1^n, \dots, e_{l_n}^n$.
```

where f_i^c is the i^{th} function's branch, $e_i \in E$ are Erlang expressions $g_i \in G$ are guards belonging to the branches, and $p_i \in P$ are patterns that form the function's formal parameter list.

Definition 2.4 (Number of functions)**.** The result of the measurement for all the modules in the semantic graph [12, 16] used to store the source code is $NOF(M) = |F(M)|$, where $F(M)$ denotes all functions in all modules.

In addition to the metrics presented here the analytical algorithms is capable of assessing several other complexity measures, and can apply these measurement results in the construction of various transformations.

3. Transformations used to improve code quality

This chapter describes the operation of the transformation steps from the scripts used to improve the quality of the source code. The scripts automatically transform the program constructions located in the source code, based on the complexity measures presented in Chapter 2.

To improve the *McCabe* cyclomatic number and the programming style we apply the extraction of deeply embedded *case* statements, and in some cases, where, as the effect of the transformations, the number of functions becomes too high, the transformation steps carrying out the movement of functions.

3.1. Conversion of a *case* expression into a function

This transformation step converts the *case* statement designated for extraction into a function, then places a call to the new function in place of the original expression in the way that the bound variables in the expression are converted into parameters (see: Figure 1.).

$$\begin{array}{|l|}\hline \textit{case}\text{ statement}\\ \hline e_0: \\ \quad \mathsf{case}\ e\ \mathsf{of}\\ \qquad p_1 \text{ when } g_1 \to e^1_1,\dots,e^1_{l_1};\\ \qquad \vdots \\ \qquad p_n \text{ when } g_n \to e^n_1,\dots,e^n_{l_n}\\ \quad \mathsf{end}\\ \hline\end{array} \quad \to \quad \begin{array}{|l|}\hline \text{function statement}\\ \hline e_0: \\ \quad \mathsf{m} : \mathsf{f}(e) \text{ or}\\ \quad \mathsf{f}(e)\\ \\ \text{where the definition of the}\\ \text{function is:}\\ \\ \mathsf{f}(p_1) \text{ when } g_1 \ \to\\ \quad e^1_1,\dots,e^1_{l_1};\\ \qquad \vdots\\ \mathsf{f}(p_n) \text{ when } g_n \ \to\\ \quad e^n_1,\dots,e^n_{l_n}.\\ \hline\end{array}$$

Figure 1: The extraction of a case expression

The transformation in terms of impact affects the complexity of the transformed function, and its modules. The number of functions, of rows, and of characters may increase in the module, but along with this decreases in the function. The transformation is local to the module.

It has a beneficial effect on the rates of embeddedness. As long as when applying, the extractions are kept at bay by limiting the number of functions, good results can be achieved regarding the *McCabe* number and the rate of embeddedness.

3.2. Movement of functions between modules

The moving of functions transformation transfers the selected functions to another module. Of course, these transformation steps (in compliance with pre- and well-defined rules) perform the necessary compensatory measures such as ensuring availability of related records, and macros, and managing or replacing the calls from the function. The transformation is complex, so the structural complexity levels are also markedly changed.

The original code

```
-module(movefun).
-export([f/1, g/1]).

 f(X)->
   X + 1.

 g(X) ->
   X -1.
```

$\rightarrow$

The result code

```
 -module(movefun).
 -export([f/1]).

  f(X)->
    X + 1.

----movefun_new.erl-----

 -module(movefun_new).
 -export([g/1]).

  g(X)->
    X - 1.
```

Figure 2: Move function to another module

It has an impact on the participating function, but only in the event if it calls other functions or it contains qualified function calls. It affects the function's module, the functions and modules that are linked via function calls, and of course the module that is designated as the intended destination of the move. There is a change in the number of measured values of the relationships between modules, and the inbound and outbound function calls.

4. Transformation scripts

In this chapter, we present a language [19, 20, 21] suitable for automatic program transformations that we have developed and implemented to create scripts aimed to improve the code complexity measurements.

We present the syntax of the language, and also present the operation of the algorithm that was prepared to run it. We show the ways in which the quality of the program code can be improved based on the complexity metrics.

Based on measurements, and taking into account the predefined conditions the transformation language is suited to automatically convert source code stored in the semantic graph [12, 16] constructed from the program code and afterwards restore the code from the modified graph. By using the optimizer scripts and taking into account the changes in the complexity measures, the quality of the source code can be automatically transformed.

$$\begin{array}{rcl}
\textbf{Query} & \rightarrow & \textbf{MetricQuery} \mid \textbf{OptQuery} \\
\textbf{OptQuery} & \rightarrow & \textbf{Opti Where Limit} \\
\textbf{Opti} & \rightarrow & \textbf{optimize Transformation} \\
\textbf{Transformation} & \rightarrow & \mathit{TransformationName} \\
 & \mid & \mathit{TransformationName}\ \textbf{Params} \\
\textbf{Params} & \rightarrow & \textbf{(}\mathit{Attr}\textbf{,}\ \mathit{ValueList}\textbf{)} \\
\textbf{Where} & \rightarrow & \textbf{where Cond} \\
\textbf{Cond} & \rightarrow & \mathit{Metric\ Rel\ CondValue} \\
 & \mid & \textbf{Cond}\ \mathit{LogCon}\ \textbf{Cond} \\
\textbf{Limit} & \rightarrow & \textbf{limit}\ \mathit{Int}
\end{array}$$

Figure 3: Language of the transformation scripts

In the specification of the syntax *Transformation* denotes a transformation (e.g. `move_fun`), *Rel* stands for a relation or other operator (e.g. $<, <=, >=, >, like$), *LogCon* denotes a logical operator (e.g. "and", "or"), the *CondValue* can be an integer or a designated lexical item (for example, using a modules name with *like*). *Int* in the *limit* section a can be substituted with a positive integer. (The non-terminal elements are with capital initials whereas the keywords of the language begin with small letters.)

In the *optimize* section the applied conversion's transformation steps and its parameters can be specified. The complex condition that can be defined after the *where* keyword, is the one that initiates the measurements, and controls the execution of transformations, namely under what conditions a given transformation step should be re-execute, and also which nodes of the semantic graph should be transformed.

Therefore the "basic condition" that can be specified, which is a logical expression, must contain at least one partial expression, which includes a measurable level of complexity on the modules or functions designated for transformation, an arithmetic operator, as well as a constant value.

The selected elements, which are the subject of the transformation, are not directly defined software constructions or expressions, but program slices that are selected automatically based on the given terms. With this method, the designation of the program parts that need to be transformed is transferred from the lexical level to the level of semantic analysis [11].

During the execution of the script written in the transformation language the analyzer searches for the program parts that fit the conditions, then performs the transformation given in the *optimize* section after which it measures the values of the complexity metrics specified in the criteria for all semantic graph node. The nodes that do not need to be included in further transformations based on the operator, and the constant, are drop out from the scope of the script. If there are no nodes on which the transformation must be re-executed, the script stops running.

Using the transformations we do not always reach the set objective, that is, by executing the script over and over again it always finds graph nodes awaiting another transformation (sometimes the script itself creates these with the application of the transformations).

Under these conditions, there may be cases when the execution does not terminate. To avoid this problem, the maximum number of executable iterations can be defined with the constant given after the *limit* keyword. Therefore if the transformation step does not produce the desired results, the constant of *limit* will definitely stop the execution after the given number of steps.

5. Measurement results

The measured software is the *Dialyzer*, that is part of the Erlang language; it is complex enough to produce results for each of the analyzed measures.

Overall, it consists 19 modules, and the modules contain 1023 functions in total. The number of function's branches is 1453. The most functions within a module is 163, and the highest number of function's branches in one module is 238.

The sum of the measured cyclomatic numbers on the modules was 3024, and with the same measurement the highest value for an individual module was 704, which is an outstanding result. (The source code will not be shared, since it is included in the Erlang distributions, and thus freely available.

The results apply to the release available at time of writing of the article). These figures make the software suitable to test the transformation algorithm on it. In the first experiment, we measured the number of functions, from a module and the *McCabe* number, and then we took the ratio of the two values:

$$\frac{mcCabe(src)}{number\_of\_fun(src)},$$

where $mcCabe(src)$ is *McCabe*'s cyclomatic number measured in the source code, src the measured source code, while nof is the number of all the functions in the module.

```
                    Mc Cabe number

optimize
   extract_function (exprtype, case_expr)
where
   f_mcCabe > 6
    and
   f_max_depth_of_structures > 2
    and
   f_max_depth_of_cases > 1
limit
   7;
```

Figure 4: The code quality improving script

The result is: $x_1 = \frac{704}{165} = 4.26666666667$. This value was taken as the base and we tried to improve it with the help of the script; that is we tried to improve the module's cyclomatic number in some way.

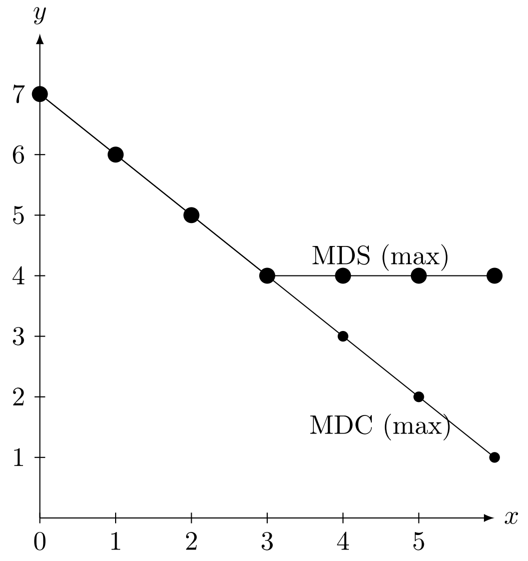

Figure 5: The maximum embeddedness of structures (MDS) and of *case* statements (MDC) (y-axis) during the transformation steps (x-axis)

We divided the cyclomatic number by the number of functions during the test, because the distorting effect that developed due to the increase in the number of functions had to be eliminated.

After running the script on the source code, whose exact task is to extract *case* expressions nested deeper than a given depth and insert in their place a call to functions generated by it, the following results were obtained:

$$x_2 = \frac{mcCabe(src')}{number\_of\_fun(src')} = \frac{794}{255} = 3,1137254901960785$$

$$\frac{x_2}{x_1} = 0.8473122889758293 = 84\% = 16\% \uparrow^{(limit=1)}$$

$$\frac{x_2}{x_1} = 0.729779411764706 = 72\% = 28\% \uparrow^{(limit=2)}$$

To obtain better results, we measured the maximum embeddedness levels of the *case* expressions in the module ($max\_depth\_of\_cases$). The measurement result indicated seven levels, that is the value that we should specify in the script's *limit* section, as this instructs the script to perform the extraction at least seven times.

$$x_3 = \frac{mcCabe(src'')}{number\_of\_fun(src'')} = \frac{868}{329} = 2.6382978723404$$

$$\frac{x_3}{x_1} = 0.6183510638297872 = 61\% = 39\% \uparrow^{(limit=7)}$$

By examining the new results we can draw some important conclusions. The first of which is that the measured values of the modules' cyclomatic number have increased because of the new functions.

Comparing the number of functions, and the cyclomatic number before and after the transformation, it is clear that $mcCabe(src) = mcCbae(src') - (nof(src) - nof(src'))$, so with the extractions the cyclomatic number of modules does not change, in the case the degree of embeddedness and the number of functions are not included in the calculated value.

This is so because the "decisions" from the expressions of the functions remain in the module, that is, whether or not a decision inside a function is extracted to a new function it does not disappear from the module (hence earlier we have divided the value by the number of functions).

In addition to the measured values of the modules we have to consider the cyclomatic number of each function in the module measured individually, as well as the maximum and the minimum from these values. If the changes of these results are compared with the values before and after the transformation, only then can we get a clear picture of the impact of the transformation. (Otherwise the average values of the module cannot be called accurate and the number of functions in the original module can greatly influence the results, as each new function adds at least one to this value...)

Analyzing the performed measurements we can see that the sum of cyclomatic numbers measured before the transformations is 704, and 794 after, if it is not divided by the number of functions; also prior to the transformation, the number of functions is 165, and 225 thereafter. Since $794 - 704 = 255 - 165$, it is clear that the newly created functions bring the increase in value.

In light of these we can make the suggestion that when measuring *McCabe*)'s cyclomatic number the values measured in the module should not be taken into account, but rather the highest reading obtained from the module's functions should be compared before the execution of the transformation, and thereafter.

$$max(mcCabe_f(src)) > max(mcCabe_f(src'))$$

We should consider the extent of the nestedness of different control structures, and so we should calculate according to the following formula, also we need to develop the appropriate transformation scripts based on this. Calculation of the result for the initial source code is as follows:

$$mcCabe(src) + sum(max\_dept\_of\_struct(src)) - z$$

$$+number\_of\_exceptions(src)$$

From the maximum of the embeddedness value the number of those functions where the degree of embeddedness is one (or the value we optimized the script to) can be subtracted as they also distort the value (in the formula this value is denoted by z). The $+(number\_of\_exceptions)$ section, which accounts for choices brought in by the exception handlers is optional, but if we use it for the initial condition, we cannot omit it from the calculation of the post-transformation state. (We would have even more accurate results if we would also included the branches of the exception handlers, that is the possible outcomes of exceptions, in the results. At this point, we have introduced a new measurement, but only in order to achieve better results. This metric returns the number of exception handlers located in programs in the module and function type nodes. To implement this measurement the function realising the measuring of the function expressions was converted so that it does not only return the number of expressions (fun_expr), but also the number of exception handlers(try_expr).

In the Erlang language, the exception handling *try* block can contain branches based on pattern matching that are customary for *case* control structures, also in a catch block the program's control can branch in multiple directions. So the solution does not find the decisions in the exception handlers, but rather it only returns the number of exception handlers, therefore it is not entirely accurate, but it is still convenient.) For the transformed text the result can be calculated with the following formula:

$$mcCabe(src') + sum(max\_dept\_of\_struct(src')) - z$$

$$+number\_of\_exceptions(src')$$

Thereafter from the measured maximum value of the functions and from the values calculated with the formula it can be decided with a high degree of certainty whether the result is better or not than the initial value. The calculation method already takes into account the depth of embeddedness, by increasing the cyclomatic number of a given function or module with each level.

Unfortunately this method together with the additional elements is still not perfect, because with regard to the measured values of the module it does not take into account the relationships between functions, and the resulting decision-making situations, which can be mapped to the call graph, but it is definitely better than the previous ones.

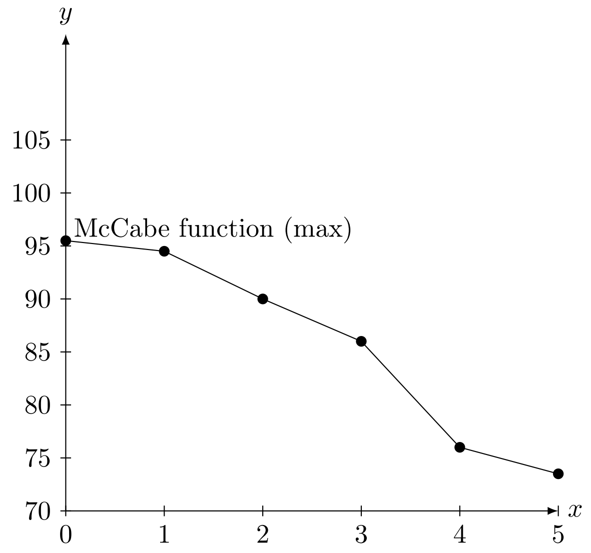

Figure 6: The maximum *McCabe* number of functions (y-axis) after each transformation step (x-axis)

In order to obtain more representative results than the previous one we had to analyze the complete source code of the Dialyzer software, with the source code scanning algorithm and then we transformed it. To perform the sequence of steps the previously used script was applied however, we took into account the proposed changes, so the embeddedness is added to the result, and the minimums and the maximums measured for the functions are also examined when the conclusions are deducted. The measured maximum value of the cyclomatic number of the functions before the transformation $max(mcCabe_f(src)) = 96$, and after restructuring $max(mcCabe_f(src')) = 73$, that is $max(mcCabe_f(src)) > max(mcCabe_f(src'))$.

The results show an improvement, but the script performs the extraction on all the function of each module that has an embeddedness greater than one. This embeddedness depth is not necessarily bad. As far as possible extractions should only be applied to areas where this is absolutely necessary, that is, in those modules in which the measured maximum cyclomatic numbers of the functions is high.

6. Conclusion

We introduced the language we have developed and the operation of the analysis algorithm. The language enables us to write automated program transformation scripts based on the measuring of complexity rates. In Chapter 2 we presented those structural complexity measures that were used to measure the complexity of Erlang source codes.

In Chapters 3 and 4 we examined the possibility of implementing an automated program for transformations based on the measurement and analysis of the complexity levels.

We defined the syntax of the language suitable for describing and executing the sequence of automated transformation steps based on software complexity measurements and described the operating principle of the analyzing and execution conducting algorithm that was constructed for the language.

In Chapter 5 using example programs, and their execution results we demonstrated the operability of automatic code quality improvement.

Beside the syntax and the descriptions of use cases we showed what results can be achieved by using a simple script made up of only a couple of lines.

In summary, the analyzing and the optimizing algorithm based on complexity measurements, which can be used to automatically or semi-automatically improve the source code of software written in Erlang language as well as previously published programs that are awaiting conversion, operated properly during the transformation of large-scale software.

The sequences of transformational steps improved the complexity rates which were designated for optimization. During the transformation the meaning of the source code did not change, and the program worked as expected following the re-translation.

In the following by using the results presented here we would like to test the parser and the transformational language constructed for it, on working client-server based software and programs from the industrial environment, for analyzing and also improving the quality of the source code. In addition we attempt to prove that following the execution of the transformation script, the modified source code's meaning conservation properties and correctness by using mathematical methods.

References

[1] McCabe T. J. A Complexity Measure, *IEE Trans. Software Engineering, SE-2(4), pp.308–320 (1976)*

[2] Frank Simon, Frank Steinbrückner, Claus Lewerentz *Metrics based refactoring* IEEE Computer Society Press 2001 30-38,

[3] Klaas van den Berg.: Software Measurement and Functional Programming, *PhD Thesis University of Twente (1995)*

[4] Ryder, C. Software Measurement for Functional Programming, *PhD thesis, Computing Lab, University of Kent, Canterbury, UK 2004)*

[5] Claus Lewerentz, Frank Simon *A Product Metrics Tool Integrated into a Software Development Environment* Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543 Springer-Verlag 256–257

[6] Eclipse Foundation
http://www.eclipse.org/

[7] Zoltán Porkoláb, Ádám Sipos, Norbert Pataki, Structural Complexity Metrics on SDL Programs. *Computer Science, CSCS 2006, Volume of extended abstracts, (2006)*

[8] Ryder, C., Thompson, S. *Software Metrics: Measuring Haskell*, In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming (September 2005)

[9] Zoltán Porkoláb Programok Strukturális Bonyolultsági Mérőszámai. *PhD thesis Dr Töke Pál, ELTE Hungary, (2002)*

[10] Zoltán Horváth, Zoltán Csörnyei, Roland Király, Róbert Kitlei, Tamás Kozsik, László Lövei, Tamás Nagy, Melinda Tóth, and Anikó Víg.: Use cases *for refactoring in Erlang, To appear in Lecture Notes in Computer Science, (2008)*

[11] Csörnyei Zoltán *Fordítóprogramok* Typotex Kiadó, Budapest, 2006. 3

[12] R. Kitlei, L. Lövei, M Tóth, Z. Horváth, T. Kozsik, T. Kozsik, R. Király, I. Bozó, Cs. Hoch, D. Horpácsi.: Automated Syntax Manipulation in RefactorErl. *14th International Erlang/OTP User Conference. Stockholm, (2008)*

[13] Lövei, L., Hoch, C., Köllö, H., Nagy, T., Nagyné-Víg, A., Kitlei, R., and Király, R.: Refactoring Module Structure *In 7th ACM SIGPLAN Erlang Workshop, (2008)*

[14] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy, T.: Refactoring in Erlang, a Dynamic Functional Language *In Proceedings of the 1st Workshop on Refactoring Tools, pages 45–46, Berlin, Germany, extended abstract, poster (2007)*

[15] Erlang - Dynamic Functional Language
http://www.erlang.org

[16] T. Kozsik, Z. Horváth, L. Lövei, T. Nagy, Z. Csörnyei, A. Víg, R. Király, M. Tóth, R. Kitlei.. Refactoring Erlang programs. *CEFP'07, Kolozsvár (2007)*

[17] Thanassis Avgerinos, Konstantinos F. Sagonas *Cleaning up Erlang code is a dirty job but somebody's gotta do it.* Erlang Workshop 2009: 1–10

[18] Konstantinos F. Sagonas, Thanassis Avgerinos *Automatic refactoring of Erlang programs.* PPDP '09 Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming 2009: 13–24

[19] Király, R., Kitlei R.: *Complexity measurments for functional code* 8th Joint Conference on Mathematics and Computer Science (MaCS 2010) refereed, and the proceedings will have ISBN classification July 14–17, 2010

[20] Király, R., Kitlei R.: *Implementing structural complexity metrics in Erlang.* '10 ICAI 2010 – 8th International Conference on Applied Informatics to be held in Eger, Hungary January 27-30, 2010

[21] Roland Király, Róbert Kitlei: *Metrics based optimization of functional source code* a research paper in Annales Mathematicae et Informaticae 38 (2011) Pages: 59–74