Complexity Measures in 4GL Environment

Csaba Nagy[1], László Vidács[2], Rudolf Ferenc[1], Tibor Gyimóthy[1], Ferenc Kocsis[3], and István Kovács[3]

[1] Department of Software Engineering, University of Szeged
[2] Research Group on Artificial Intelligence, University of Szeged & HAS
[3] SZEGED Software Zrt.

Abstract. Nowadays, the most popular programming languages are so-called third generation languages, such as Java, C# and C++, but higher level languages are also widely used for application development. Our work was motivated by the need for a quality assurance solution for a fourth generation language (4GL) called Magic. We realized that these very high level languages lie outside the main scope of recent static analysis techniques and researches, even though there is an increasing need for solutions in 4GL environment.
During the development of our quality assurance framework we faced many challenges in adapting metrics from popular 3GLs and defining new ones in 4GL context. Here we present our results and experiments focusing on the complexity of a 4GL system. We found that popular 3GL metrics can be easily adapted based on syntactic structure of a language, however it requires more complex solutions to define complexity metrics that are closer to developers' opinion. The research was conducted in co-operation with a company where developers have been programming in Magic for more than a decade. As an outcome, the resulting metrics are used in a novel quality assurance framework based on the Columbus methodology.

Keywords: 4GL, Magic, software metrics, software complexity, software quality assurance

1 Introduction

Programming languages are usually categorized into five levels or "generations" [1]. Solely binary numbers, the machine languages are the first generation languages (1GLs). Lower level programming languages (e.g. assembly) are the second generation languages (2GLs) and currently popular procedural and object-oriented languages are the third generation languages (3GLs). The higher level languages are all closer to human thinking and spoken languages. Using fourth generation languages (4GLs) a programmer does not need to write source code, but he can program his application at a higher level of abstraction, usually with the help of an application development environment. Finally, fifth generation languages (5GLs), would involve a computer which responds directly to spoken or written instructions, for instance English language commands.

The main motivation of this work was to provide a quality assurance solution for a 4GL called Magic. Quality assurance tools are built heavily on software metrics, which reflect various properties of the analyzed system. Although several product metrics are already defined for mainstream programming languages, these metrics reflect the specialties of third generation programming languages. We faced the lack of software quality metrics defined for 4GLs. As we revealed the inner structure of Magic programs, we identified key points in defining new metrics and adapting some 3GL metrics to Magic. Our work was carried out together with a software company, where experts helped us in choosing the right definitions. The greatest challenge we faced was the definition of complexity metrics, where experienced developers found our first suggestions inappropriate and counterintuitive. Enhancing our measures we involved several developers in experiments to evaluate different approaches to complexity metrics.

In this paper we present our experiences in defining complexity metrics in 4GL environment, particularly in the application development environment called Magic, which was recently renamed to uniPaaS. Our contributions are:

- we adapted two most widespread 3GL complexity metrics to Magic 4GL (McCabe complexity, Halstead);
- we carried out experiments to evaluate our approaches (we found no significant correlation between developers ranking and our first adapted McCabe complexity, but we found strong correlation between a modified McCabe complexity, Halstead's complexity and between the developers ranking);
- as an outcome of the experiments we defined new, easily understandable and applicable complexity measures for Magic developers.

Supporting the relevance of the adapted metrics our experiment was designed to address the following research questions:

RQ1: Is there a significant correlation between adapted metrics of Magic programs?

RQ2: Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?

The paper is organized as follows. First, in Section 2 we introduce the reader to the world of Magic and then in Section 3 we define our complexity metrics that were adapted to 4GL environment. Validating these metrics we carried out experiments which we describe in Section 4 and evaluate in Section 5. We discuss related work in Section 6 and finally we conclude in Section 7.

2 Specialties of 4GLs and the Magic Programming Language

It is important to understand the specialties of a fourth generation language before discussing its quality attributes. Hence, in this section we give an introduction into Magic as a fourth generation language. We present the basic structure of a typical Magic application and we discuss potential quality attributes of a Magic application.

Magic 4GL was introduced by Magic Software Enterprises (MSE) in the early 80's. It was an innovative technology to move from code generation to the use of an underlying meta model within an application generator.

2.1 The Structure of a Magic Application

Magic was invented to develop business applications for data manipulating and reporting, so it comes with many GUI screens and report editors. All the logic that is defined by the programmer, the layout of the screens, the pull down menus, reports, on-line help, security system, reside inside tables called Repositories. The most important elements of the meta model language are the various entity types of business logic, namely the Data Tables. A Table has its Columns and a number of Programs (consisting of subtasks) that manipulate it. The Programs or Tasks are linked to Forms, Menus, Help screens and they may also implement business logic using logic statements (e.g. for selecting variables, updating variables, conditional statements).

Focusing on the quality – especially on the complexity – of a Magic software, the most important language elements are those elements that directly implement the logic of the application. Figure 1 shows these most important language entities. A Magic *Application* consists of *Projects*, the largest entities dividing an application into separate logical modules. A Project has Data *Tables* and *Programs* (a top-level Task is called a Program) for implementing the main functionalities. A Program can be called by a *Menu* entry or by other Programs during the execution of the application. When the application starts up, a special program, the *Main Program* is executed. A *Task* is the basic unit for constructing a program. A Program can be constructed of a main task and subtasks in tree-structured task hierarchy. The Task represents the control layer of the application and its Forms represent the view layer. It typically iterates over a Table and this iteration cycle defines so-called *Logic Units*. For instance, a Task has a Prefix and a Suffix which represent the beginning and the ending of a Task, respectively. A record of the iteration is handled by the Record Main logic unit, and before or after its invocation the Record Prefix or Suffix is executed. A Logic Unit is the smallest unit which performs lower level operations (a series of *Logic Lines*) during the execution of the application. These operations can be simple operations, e.g. calling an other Task or Program, selecting a variable, updating a variable, input a data from a Form, output the data to a Form Entry.

Programming in Magic requires a special way of thinking. Basically, the whole concept is built on the manipulation of data tables which results in some special designs of the language. It can be seen that a Task belongs to an iteration over a data table so when a Task is executed it already represents a loop. Hence, the language was designed in a way that loops cannot be specified explicitly at statement level. It is also interesting that the expressions of a Task are handled separately so an expression can be reused more than once simply by referring to its identifier. For example, each Logic Line has a condition expression which

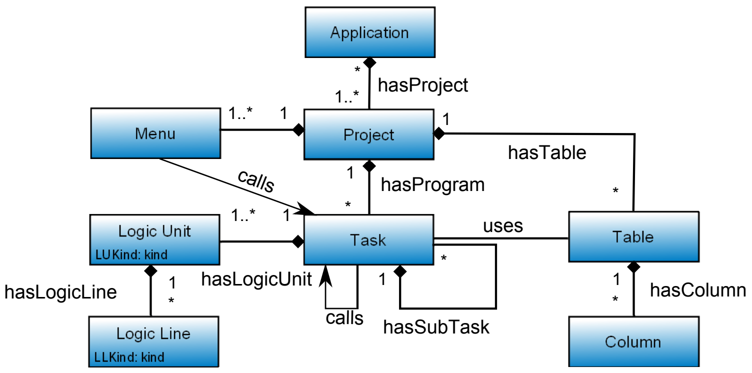

Fig. 1. Most important Magic schema entities.

determines whether the operation should be executed or not. This condition can be easily maintained through the application development environment and the same expression may be easily used for more statements. So the developers are more comfortable in using conditional branches in the logic of an application. Consequently, they can easily see when the execution of statements belongs to the same condition even if the statements do not directly follow each other.

2.2 Measuring the Quality of a Magic Application

In previous projects [13], [14] we re-used and adapted elements of the Columbus methodology in the Magic environment. This methodology was successfully applied on object-oriented languages before [8] and today it covers the most influential areas of the software life cycle including the following goals [3]: decrease the number of post-release bugs, increase maintainability, decrease development/test efforts, assure sustainability though continuous measurement and assessment. Goals are targeted with continuous monitoring: scheduled analysis, data processing, storing and querying, visualization and evaluation. To accomplish these goals it is important to measure the characteristics of the software under question. For more details about Columbus methodology, please refer to our previous paper [3].

In case of third level languages, usually the best description of the software under question is its source code. It is obvious that the analysis of the source code is important to specify certain quality attributes. In case of fourth generation languages, developers do not necessarily write source code in the traditional way. In Magic, developers simply edit tables, use form editors, expression editors, etc. In such a language, the meta model of an application serves as a “source code” that can be analyzed for quality assurance purposes. Using this model we can describe the main characteristics of an application and we can locate potential coding problems or structures which may indicate bugs or bad design. We determined a number of product metrics for Magic and categorized them in *size*, *coupling*, and *complexity* groups. Most of them are based on popular and well-known product metrics such as the *Lines of Code*, *Number of Classes*, *Number of Attributes*, *Coupling Between Object classes* [4]. We realized that some metrics can be easily adapted from third generation languages, but their meaning and benefits for the developers may be completely different, compared to 3GL counterparts.

In case of size metrics, for instance, there is a possibility to identify a series of "Number of" metrics (e.g. *Number of Programs*, *Menus*, *Helps*), but they are considered less useful and interesting for the developers. The reason for that is that these numbers can be easily queried through the application development environment. The Lines of Code (*LOC*) metric can be easily adapted by taking into account that the *Logical Line* language entity of Magic can be corresponded to a "Line of Code" in a third generation language. However, the adapted metric should be used with caution because it carries a different meaning compared to the original *LOC* metric. In 3GLs *LOC* typically measures the size of the whole system and it is used to estimate the programming effort in different effort models (e.g. *COCOMO* [5]). In case of Magic, a project is built on many repositories (Menus, Help screens, Data Tables, etc.) and *LOC* measures just one size attribute of the software (the Program repository). Hence, *LOC* is not the sole size attribute of an application so it cannot be used alone for estimating the total size of the full system. It is interesting to note that when 4GLs became popular, many studies were published in favor of their use. These studies tried to predict the size of a 4GL project and its development effort, for instance by calculating function points [16],[17] or by combining 4GL metrics with metrics for database systems [10].

Coupling is also interesting in a 4GL environment. In object-oriented languages a typical metric for coupling is the *Coupling Between Object classes* (*CBO*) metric which provides the number of classes to which a given class is coupled. A class is coupled to another one if it uses its member functions and/or instance variables. 4GLs usually do not have language elements representing objects and classes. For instance in Magic, there are no entities to encapsulate data and related functionalities, however there are separated data entities (Tables) and their related functionalities are specified in certain Tasks or Programs. Therefore it makes sense measuring the *Coupling Between Tasks and Data Tables*, not unlike the *Coupling Between Tasks and Tasks.*

3 Measuring the Complexity of Magic Applications

We identified different quality attributes and defined a bunch of metrics for Magic applications. Simple size and coupling metrics reflected well the opinion of the developers, but this was not the case for complexity metrics. It was our biggest challenge to measure the complexity of a 4GL system. There are many different approaches for third generation languages [6]. At source code level, well known approaches were developed by McCabe [11] and Halstead [9], which are widely used by software engineers, e.g., for software quality measurement purposes and for testing purposes.

We adapted McCabe's cyclomatic complexity and Halstead's complexity metrics in 4GL environment, but when we showed the results to developers, their feedback was that all the programs that we identified as most complex programs in their system are not that much complex according to their experience. We note here that all the programmers have been programming in Magic for more than 3 years (some of them for more than a decade) and most of them were well aware of the definition of structural complexity [1], but none of them have heard before about cyclomatic or Halstead complexity.

3.1 McCabe's Cyclomatic Complexity Metric

In this section we present our adaptations of complexity metrics and a modified cyclomatic complexity measure.

First, we adapted McCabe's complexity metric [11] to Magic. McCabe used a graph-theory measure, the *cyclomatic number* to measure the complexity of the control flow of a program. It was shown that of any structured program with only one entrance and one exit point, the value of McCabe's cyclomatic complexity is equal to the number of decision points (i.e., the number of "if" statements and conditional loops) contained in that program plus one.

McCabe's complexity is usually measured on method or function level. For object-oriented languages it is possible to aggregate complexities of methods to class level. The idea of *Weighted Methods per Class* (*WMC*) [7] is to give weights to the methods and sum up the weighted values. As a complexity measure this metric is the sum of cyclomatic complexities of methods defined in a class. Therefore *WMC* represents the complexity of a class as a whole.

In case of Magic, the basic operations are executed at Logic Unit level. A Logic Unit has its well-defined entry and exit point too. Likewise, a Task has predefined Logic Units. That is, a Task has a Task Prefix, Task Suffix, Record Prefix, Record Main, Record Suffix, etc. This structure is similar to the construction of a Class where a Class has some predefined methods, e.g., constructors and destructors. Hence, we defined McCabe's complexity at Logic Unit level with the same definition as it is defined for methods (see definition of $McCC(LU)$ in Definition 1). So it can be simply calculated by counting the statements with preconditions (i.e., the branches in the control flow) in a Logic Unit. Likewise, the complexity of a Task can be measured by summing up the complexity values of its Logic Units. We call this complexity measure as the *Weighted Logic Units per Task* (see $WLUT(T)$ in Definition 2).

$McCC(LU) = Number\ of\ decision\ points\ in\ LU + 1.$
LU: a Logic Unit of a Task

Def. 1: The definition of McCabe's cyclomatic complexity for Logic Units.

$WLUT(T) = \sum_{LU \in T} McCC(LU)$
T: a Task in the Project
LU: a Logic Unit of T

Def. 2: The definition of Weighted Logic Units per Task (WLUT).

The $McCC(LU)$ and $WLUT(T)$ metrics were adapted directly from the 3GL definitions simply based on the syntactic structure of the language. When we first showed the definitions to the developers they agreed with them and they were interested in the complexity measures of their system. However, the results did not convince them. Those Tasks that we identified as the most complex tasks of their system were not complex according to the developers, not unlike, those tasks that were identified complex by the developers had lower $WLUT$ values.

Developers suggested us, that in addition to the syntactic structure of the language, we should add the semantic information that a Task is basically a loop which iterates over a table and when it calls a subtask it is rather similar to an embedded loop. This semantic information makes a Task completely different from a Class. Considering their suggestion we modified the McCabe complexity as follows ($McCC_2$). For a Logic Unit we simply count the number decision points, but when we find a call for a subtask it is handled as a loop and it increases the complexity of the Logic Unit by the complexity of the called subtask. That is, the complexity of a Task is the sum of the complexity of its Logic Units. For the formalized definition see Definition 3.

$McCC_2(LU) = \textit{Number of decision points in } LU + \sum_{TC \in LU} McCC_2(TC) + 1.$

$McCC_2(T) = \sum_{LU \in T} McCC_2(LU)$

LU: a Task of the Project

LU: a Logic Unit of T

TC: a called Task in LU

Def. 3: The definition of the modified McCabe's cyclomatic complexity ($McCC_2$).

The main difference between $WLUT(T)$ and $McCC_2(T)$ is that $McCC_2(T)$ takes into account the complexity of the called subtasks too in a recursive way. A recursive complexity measure would be similar for procedural languages when a function call would increase the complexity of the callee function by the complexity of the called function. (Loops in the call graph should be handled.)

Developers found the idea of the new metric more intuitive as it takes into account the semantics too. Later, in our experiments we found that the new metric correlates well with the complexity ranking of the developers (see Section 4).

3.2 Halstead's Complexity Metrics

Some of the developers also complained that our metrics do not reflect the complexity of the expressions in their programs. It should be noted here that Magic handles the expressions of a Task separately. An expression has a unique identifier and can be used many times inside different statements simply by referring to its identifier. The application development environment has an expression editor for editing and handling expressions separately. This results in a coding style where developers pay more attention on the expressions they use. They see the list of their expressions and large, complex ones may be easily spotted out.

Halstead's complexity metrics [9] measure the complexity of a program based on the lexical counts of symbols used. The base idea is that complexity is affected by the used operators and their operands. Halstead defines four base values for measuring the number of distinct and total operands and operators in a program (see Definition 4). The base values are constituents of higher level metrics, namely, *Program Length* (*HPL*), *Vocabulary size* (*HV*), *Program Volume* (*HPV*), *Difficulty level* (*HD*), *Effort to implement* (*HE*). For the formalized definitions see Definition 5.

n_1: the number of distinct operators
n_2: the number of distinct operands
N_1: the total number of operators
N_2: the total number of operands

Def. 4: Base values for measuring the number of distinct and total operands and operators in a program.

$$HPL = N_1 + N_2$$
$$HV = n_1 + n_2$$
$$HPV = HPL * log_2(HV)$$
$$HD = (\frac{n_1}{2}) * (\frac{N_2}{n_2})$$
$$HE = HV * HD$$

Def. 5: Halstead's complexity measures.

In case of Magic, symbols may appear inside expressions so the choice of Halstead's metrics seemed appropriate for measuring the complexity of expressions. Operands can be interpreted as the symbols like in a 3GL language (e.g. variable names, task identifiers, table identifiers) and operators are the operators (plus, minus, etc.) inside expressions.

Later, in our experiments we found that the Halstead's metrics correlate with the complexity ranking of the developers (see Section 4), but the modified McCabe's complexity is closer to the opinion of the developers.

4 Experiments with Complexity Metrics

Although the classic complexity metrics are successfully adapted to the Magic language, there are no empirical data available on how they relate to each other and on their applicability in software development processes. We observed that, except the McCabe metric, complexity metrics generally do not have a justified conceptual foundation. Rather, they are defined based on experience [18]. We plan to fill in the gap first, by calculating and evaluating the adapted metrics on industrial size programs to see their relations; second, by surveying experts at a Magic developer company to see the usability of the definitions. We emphasize the importance of feedback given by Magic experts. There is no extensive research literature on the quality of Magic programs. Hence, the knowledge accumulated during many years of development is essential to justify our metrics.

Thus, to evaluate our metrics, metrical values were computed on a large-scale Magic application, and a questionnaire was prepared for experienced Magic developers to see their thoughts on complexity. We sought for answers for the following research questions:

RQ1: *Is there a significant correlation between adapted metrics of Magic programs?*
RQ2: *Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?*

We performed static analysis and computed metrics on a large-scale application using the ***MAGISTER*** system [13] (see Table 1). There are more than 2,700 programs in the whole application, which is a huge number in the world of Magic. The total number of non-Remark Logic Lines of this application is more than 300,000. The application uses more than 700 tables.

Metric	Value
Number of Programs	2 761
Number of non-Remark Logic Lines	305 064
Total Number of Tasks	14 501
Total Number of Data Tables	786

Table 1. Main characteristics of the system under question.

There were 7 volunteer developers taking part in the survey at the software developer company. The questionnaire consisted of the following parts:

1. Expertise:
   (a) Current role in development.
   (b) Developer experience in years.
2. Complexity in Magic:
   (a) At which level of program elements should the complexity be measured?
   (b) How important are the following properties in determining the complexity of Magic applications? (List of properties is given.)
   (c) Which additional attributes affect the complexity?
3. Complexity of concrete Magic programs developed by the company.
   (a) Rank the following 10 Magic programs (most complex ones first).

The most important part of the questionnaire is the ranking of the concrete programs. This makes possible comparing what is in the developers' mind to the computed metrics. Subject programs for ranking were selected by an expert of the application. He was asked to select a set of programs which *a)* is representative to the whole application, *b)* contains programs of various size, *c)* developers are familiar with. He was not aware of the purpose of selection. The selected programs and their main size measures can be seen in Table 2. The number of programs is small as we expected a solid, established opinion of participants in a reasonable time. In the table the Total Number of Logic Lines (containing task hierarchy) (*TNLL*), the Total Number of Tasks (*TNT*), Weighted Logic Units per Task (*WLUT*) and the cyclomatic complexity ($McCC_2$) are shown.

5 Results

We first discuss our findings about complexity measurements gathered via static analysis of the whole application. Later, we narrow down the set of observed programs to those taking part in the questionnaire, and finally we compare them to the opinion of the developers.

Id	Name	*TNLL*	*TNT*	*WLUT*	*McCC*$_2$
69	Engedmény számítás egy tétel	1352	24	10	214
128	TESZT:Engedmény/rabatt/formany	701	16	14	63
278	TÖRZS:Vevő karbantartó	3701	129	47	338
281	TÖRZS:Árutörzs összes adata	3386	91	564	616
291	Ügyfél zoom	930	29	8	27
372	FOK:Főkönyv	1036	31	113	203
377	Előleg bekérő levél képzése	335	6	5	20
449	HALMOZO:Havi forgalom	900	22	3	117
452	HALMOZO:Karton rend/vissz	304	9	4	34
2469	Export_New	7867	380	382	761

Table 2. Selected programs with their size and complexity values.

5.1 RQ1: *Is there a significant correlation between adapted metrics of Magic programs?*

Here we investigate the correlation between the previously defined metrics.The McCabe and Halstead metrics are basically different approaches, so first we investigate them separately.

Halstead metrics Within the group of Halstead metrics significant correlation is expected, because – by definition – they depend on the same base measures. In spite of that, different Halstead measures capture different aspects of computational complexity. We performed a Pearson correlation test to see their relation in Magic. Correlation values are shown in Table 3. Among the high expected correlation values, *HD* and *HE* metrics correlate slightly lower with the other metrics. We justified Halstead metrics using the *Total Number of Expressions* (*TNE*), which can be computed in a natural way as expressions are separately identified language elements. The relatively high correlation between *TNE* and other Halstead metrics shows that the *TNE* metric is a further candidate for a complexity metric. This reflects suggestions of the developers too. For the sake of simplicity, in the rest of this paper we use the *HPV* metric to represent all five metrics of the group.

	HPL	*HPV*	*HV*	*HD*	*HE*	*TNE*
HPL	1.000	0.906	0.990	0.642	0.861	0.769
HPV	0.906	1.000	0.869	0.733	0.663	0.733
HV	0.990	0.869	1.000	0.561	0.914	0.773
HD	0.642	0.733	0.561	1.000	0.389	0.442
HE	0.861	0.663	0.914	0.389	1.000	0.661

Table 3. Pearson correlation coefficients (R^2) of Halstead metrics and the Total Number of Expressions (*TNE*) (all correlations are significant at 0.01 level).

Comparison of adapted complexity metrics Table 4 contains correlation data on McCabe-based complexity (*WLUT*, *McCC*$_2$), *HPV* and two size metrics. The three complexity measures has significant, but only a slight correlation, which indicates that they show different aspects of the program complexity.

We already presented the differences between *WLUT* and $McCC_2$ before. The similar definitions imply high correlation between them. Surprisingly, based on the measured 2700 programs their correlation is the weakest (0.007) compared to other metrics so they are almost independent. $McCC_2$ is measured on the subtasks too, which in fact affects the results. Our expectation was that, for this reason, $McCC_2$ has a stronger correlation with *TNT* than *WLUT*. However, the $McCC_2$ metric only slightly correlates with *TNT*. This confirms that developers use many conditional statements inside one task, and the number of conditional branches has a higher impact on the $McCC_2$ value.

	WLUT	$McCC_2$	*HPV*	*NLL*	*TNT*
WLUT	1.000	0.007	0.208	0.676	0.166
$McCC_2$	0.007	1.000	0.065	0.020	0.028
HPV	0.208	0.065	1.000	0.393	0.213

Table 4. Pearson correlation coefficients (R^2) of various complexity metrics (all correlations are significant at 0.01 level).

Rank-based correlation From this point on, we analyze the rank-based correlation of metrics. The aim is to facilitate the comparison of results to the ranks given by the developers. The number of considered programs is now narrowed down to the 10 programs mentioned before in Section 4. Ranking given by a certain metric is obtained in the following way: metric values for the 10 programs are computed, programs with higher metric values are ranked lower (e.g. the program with highest metric value has a rank no. 1). The selection of 10 programs is justified by the fact, that the previously mentioned properties (e.g. different sizes, characteristics) can be observed here as well. In Figure 2, the ranking of Halstead metrics is presented. On the x axis the programs are shown (program Id), while their ranking value is shown on the y axis (1-10). Each line represents a separate metric. Strong correlation can be observed as the values are close to each other. Furthermore, the HD and HE metrics can also be visually identified as a little bit outliers. (Note: Spearman's rank correlation values are also computed.) The ranking determined by the three main complexity metrics can be seen in Figure 3. The x axis is ordered by the $McCC_2$ complexity, so programs with lower $McCC_2$ rank (and higher complexity) are on the left side. The similar trend of the three metrics can be observed, but they behave in a controversial way locally.

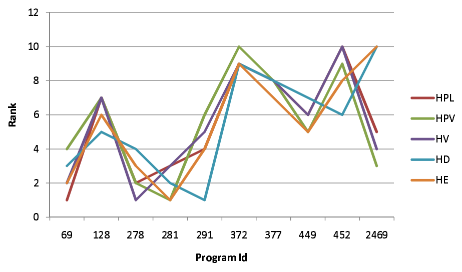

Fig. 2. Ranking of Halstead complexity metrics (ordered by program ID).

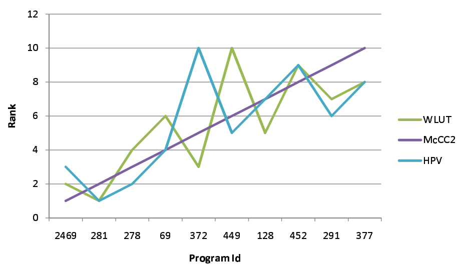

Fig. 3. Ranking of main complexity metrics (ordered by $McCC_2$).

Answering our research question, we found that some of the investigated complexity measures are in strong correlation, but some of them are independent measures. We found strong correlation between the Halstead metrics and we also found that these metrics correlate to the Total Number of Expressions. We found that our first adaptation of cyclomatic complexity ($WLUT$) has only a very weak correlation to our new version ($McCC_2$), which correlates well with other measures. This also confirms that the new measure might be a better representation of the developers opinion about complexity.

5.2 RQ2: *Is there a significant correlation between the complexity ranking given by developers and the ranking given by the adapted metrics?*

In the third part of the questionnaire developers were asked to give an order of the 10 programs which represents their complexity order. Previously, developers were given a short hint on common complexity measures, but they were asked to express their subjective opinion too. Most of the selected programs were probably familiar to the developers since the application is developed by their company. Furthermore they could check the programs using the development environment during the ranking process.

Ranks given by the 7 developers are shown in Figure 4, where each line represents the opinion of one person. It can be seen that developers set up different ranks. There are diverse ranks especially in the middle of the ranking, while the top 3 complex programs are similarly selected. Accordingly, developers agree in the least complex program, which is 2469. Correlations of developers' ranks were also computed. Significant correlation is rare among the developers, only ranks of P4, P5 and P6 are similar (Pi denotes a programmer in Figure 4).

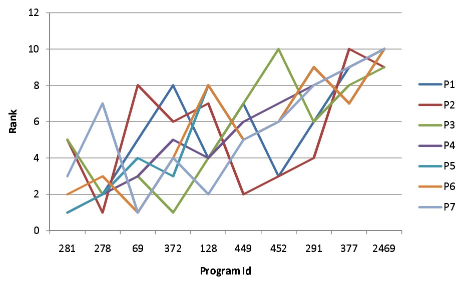

Fig. 4. Ranks given by Magic experts.

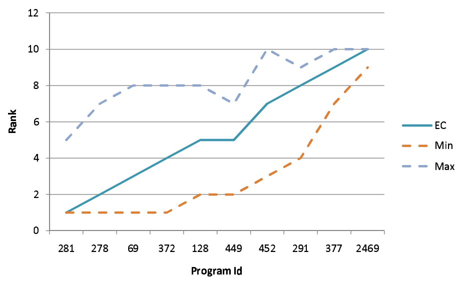

Fig. 5. The EC value, min and max ranks.

We defined the EC value (*Experiment Complexity*) for each selected program as the rank based on the average rank given by developers. In Figure 5 the EC value is shown together with min and max ranks of the developers. We note that summarizing the developers' opinion in one metric may result in loosing information since developers may had different aspects in their minds. We elaborate on this later in the Threats to Validity section. We treat this value as the opinion of the developer community.

We compared the EC value to the previously defined complexity metrics. Table 5 contains correlation values of main metrics. The EC value shows significant correlation only with the HE measure.

	$WLUT$	$McCC_2$	HPV	HE	EC
$WLUT$	1.000	0.575	0.218	0.004	0.133
$McCC_2$	0.575	1.000	0.520	0.027	0.203
HPV	0.218	0.520	1.000	0.389	0.166
HE	0.004	0.027	0.389	1.000	0.497
EC	0.133	0.203	0.166	0.497	1.000

Table 5. Correlation of Magic complexity metrics and developers‘ view (Spearman's ρ^2 correlation coefficients, marked values are significant at the 0.05 level).

Besides statistical information, complexity ranks are visualized as well. We found that the rank based correlation obscures an interesting relation between $McCC_2$ and the EC value. Ranks for each program are shown in Figure 6. The order of programs follows the $McCC_2$ metric. Despite that Spearman's ρ^2 values show no significant correlation, it can be clearly seen that developers and $McCC_2$ metric gives the same ranking, except for program 2469. This program is judged in an opposite way. The program contains many decision points, however developers say that it is not complex since its logic is easy to understand. According to the HE metric, this program is also ranked as the least complex.

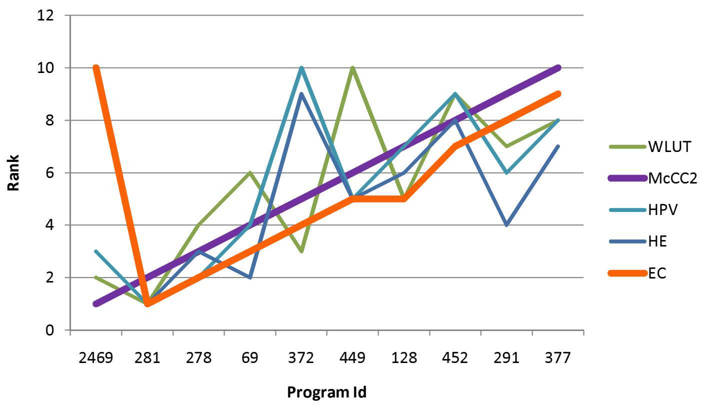

Fig. 6. The EC value compared to the main complexity metrics

Answering our research question we found that the rankings given by adapted metrics have significant and sometimes surprisingly strong relation to the ranking given by developers, except for the $WLUT$ metric. Halstead's metrics have a significant correlation here, especially the HE metric. However, the strongest relation was discovered in case of the $McCC_2$ metric.

5.3 Discussion of the Limitations

Although we carefully designed our experiments, there are some points which may affect our results and observations. Complexity metrics were computed on a large-scale and data-intensive application, but the results may be affected by coding style and conventions of a single company. Measurements of Magic applications from other domains and developer companies are needed. This applies to the questionnaire as well. The number of participants and selected programs should be increased to draw general conclusions. Programs were selected by a person, not randomly based on a specific distribution, which may also affect our results. Evaluation of developers' view is done by means of ranking, which results in loss of information in transforming measured values into ranks. The *EC* value is an average rank given by the developers. It would be more realistic to formalize their viewpoints during the ranking process.

6 Related work

We cited related papers before when we elaborated on our metrics and experiments. We note here, that there are many different approaches for measuring the complexity of a software at source code level. First, and still popular complexity measures (McCabe [11], Halstead [9], Lines of Code [2]) was surveyed by Navlakha [15]. A recent survey which sums up todays complexity measures was published by Sheng Yu et al. [18]. In 4GL environment, to our best knowledge, there were no previous researches to measure structural complexity attributes of a Magic application. Even though, for other 4GLs there are some attempts to define metrics to measure the size of a project [16], [17], [10]. There are also some industrial solutions to measure metrics in 4GL environment. For instance *Rain-Code Roadmap*[4] for Informix 4GL provides a set of predefined metrics about code complexity (number of statements, cyclomatic complexity, nesting level), about SQLs (number of SQL statements, SQL tables, etc.), and about lines (number of blank lines, code lines, etc.). In the world of Magic, there is a tool for optimization purposes too called Magic Optimizer[5] which can be used to perform static analysis of Magic applications. It does not measure metrics, but it is able to locate potential coding problems which also relates to software quality.

In 3GL context there are also papers available to analyze the correlation between certain complexity metrics. For instance, Meulen et al. analyzed about 71,917 programs from 59 fields written in C/C++ [12]. Their result showed that there are very strong connections between LOC and HCM, LOC and CCM. Our work found also similar results, but our research was performed in a 4GL context with newly adapted complexity metrics. We additionally show, that in our context traditional metrics have totally different meanings for the developers.

7 Conclusions and Future Work

The main scope of our paper was to adapt most widespread 3GL structural complexity metrics (McCabe's cyclomatic complexity and Halstead's complexity measures) to a popular 4GL environment, the Magic language. We introduced

[4] `http://www.raincode.com/fglroadmap.html`

[5] `http://www.magic-optimizer.com/`

the specialties of Magic and we presented formal definitions of our metrics in 4GL environment. Besides the simple adaptation of the metrics, we presented a modified version of McCabe's cyclomatic complexity ($McCC_2$), which measured the complexity of a task by aggregating the complexity values of its called subtasks too. We addressed research questions about our new metrics whether they are in relation with developers' complexity ranking or not. We designed and carried out an experiment to answer our questions and we found that:

- there is significant correlation among all the investigated metrics, and there is strong correlation between the Halstead measures which also correlate to the Total Number of Expressions;
- the rankings given by adapted metrics have significant and very strong relation to the ranking given by developers (especially in case of the $McCC_2$, but except for the $WLUT$ metric).

As an outcome, we found also that our modified measure has a strong correlation with developers' ranking.

To sum up the conclusions of our work, we make the following remarks:

- We made advancement in a research area where no established metrics (previous similar measurements and experience reports) were available.
- We successfully adapted 3GL metrics in a popular 4GL environment, in the Magic language.
- We evaluated our metrics by the developers in a designed experiment and metrics were found easily understandable and useful.
- A modified version of the McCabe's cyclometic complexity was found to reflect surprisingly well the ranking given by the developer community.

Besides gathering all the previously mentioned experiences, the defined metrics are implemented as part of a software quality assurance framework, namely the ***MAGISTER***[6] system which was designed to support the development processes of an industrial Magic application.

About our future plans, as we offer quality assurance services, we expect to gain data from other application domains to extend our investigations. Most importantly we plan to set up appropriate baselines for our new metrics in order to better incorporate them into the quality monitoring process of the company and into the daily use.

Acknowledgements

We thank István Siket, Dániel Fritsi, Ferenc Smohai and the volunteer developers for their contributions. This research was supported by the Hungarian national grant GOP-1.1.2-07/1-2008-0007, TECH 08-A2/2-2008-0089, GOP-1.1.1-07/1-2008-0081, OTKA K-73688 and by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

[6] `http://www.szegedsw.hu/magister`

References

1. IEEE Standard Glossary of Software Engineering Terminology. Tech. rep. (1990)
2. Albrecht, A.J., Gaffney, J.E.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transaction on Software Engineering 9, 639–648 (November 1983)
3. Bakota, T., Beszédes, Á., Ferenc, R., Gyimóthy, T.: Continuous software quality supervision using SourceInventory and Columbus. In: ICSE Companion. pp. 931–932 (2008)
4. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Transaction on Software Engineering 22, 751–761 (October 1996)
5. Boehm, B.W.: Software Engineering Economics. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. (1981)
6. Burgin, M., Debnath, N.: Complexity measures for software engineering. J. Comp. Methods in Sci. and Eng. 5, 127–143 (January 2005)
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transaction on Software Engineering 20, 476–493 (June 1994)
8. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: Proceedings of the 18th International Conference on Software Maintenance (ICSM'02). pp. 172–181. IEEE Computer Society (Oct 2002)
9. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
10. MacDonell, S.: Metrics for database systems: An empirical study. In: Proceedings of the 4th International Symposium on Software Metrics. pp. 99–107. IEEE Computer Society (1997)
11. McCabe, T.: A complexity measure. IEEE Transaction on Software Engineering SE-2(4) (dec 1976)
12. van der Meulen, M., Revilla, M.: Correlations between internal software metrics and software dependability in a large population of small C/C++ programs. In: Proceedings of ISSRE 2007, The 18th IEEE International Symposium on Software Reliability. pp. 203–208 (Nov 2007)
13. Nagy, C., Vidács, L., Ferenc, R., Gyimóthy, T., Kocsis, F., Kovács, I.: MAGISTER: Quality assurance of magic applications for software developers and end users. In: Proceedings of ICSM 2010, 26th IEEE International Conference on Software Maintenance. pp. 1–6. IEEE Computer Society (Sep 2010)
14. Nagy, C., Vidács, L., Ferenc, R., Gyimóthy, T., Kocsis, F., Kovács, I.: Solutions for reverse engineering 4GL applications, recovering the design of a logistical wholesale system. In: Proceedings of CSMR 2011, 15th European Conference on Software Maintenance and Reengineering. IEEE Computer Society (Mar 2011)
15. Navlakha, J.K.: A survey of system complexity metrics. The Computer Journal 30, 233–238 (June 1987)
16. Verner, J., Tate, G.: Estimating size and effort in fourth-generation development. IEEE Software 5, 15–22 (1988)
17. Witting, G., Finnie, G.: Using artificial neural networks and function points to estimate 4GL software development effort. Australasian Journal of Information Systems 1(2) (1994)
18. Yu, S., Zhou, S.: A survey on metric of software complexity. In: Proceedings of ICIME 2010, The 2nd IEEE International Conference on Information Management and Engineering. pp. 352–356 (April 2010)