A Tool for Process and Product Assessment of C++ Applications

Fabrizio Fioravanti, Paolo Nesi, Sandro Perlini
Department of Systems and Informatics,
Faculty of Engineering, University of Florence, Italy
[fioravan@aguirre, nesi@ingfi1, perlini@aguirre].ing.unifi.it

Abstract

The present diffusion of the object-oriented paradigm and of the techniques for maintaining under control the development process must be supported by suitable tools. These tools should be based on confident and validated object-oriented metrics for assessing the several aspects of product and process development: effort, maintainability, re-usability, etc, as well as they should be capable of supporting the definition of specific metrics, profiles and histograms. These are useful to give the developers and managers an immediate representation of the system status. These tools must also provide metrics suitable for producing confident results since the early phases of the development life cycle. Mechanisms for metrics definition, validation and tuning must be available in order to establish a process of continuous improvement. This paper presents TAC++ [1] (Tool for Analyzing C++ Code) which supports all mentioned features and includes the most important metrics presented in the literature and many others.

Index terms: *object-oriented metrics, assessment tool, effort prediction, profiles and histograms, validation, tuning.*

1. Introduction

The Object-Oriented Paradigm (OOP) is currently considered the best technology for obtaining the return of the investment. In many cases, the OOP has been adopted hoping to reach high degrees of portability, re-usability, maintainability, etc., especially for the development of large systems. The adoption of the OOP implies a change in the whole development process. Hence, the introduction of the OOP is not immediate, since managers, developers, etc., must be involved.

In order to guarantee the control of the development process, quantitative metrics for evaluating and predicting

	$A/D/C$	OOP	$M/C/S$	V/C	C/Q	Rel	P/A	$F/B/S$	$T/C/P$	E/M/R
Ha [9]	C	N	M,C,S	V	-	-	A	F	T	E,M
LOC	C	N	M,C,S	S	-	-	A	F	T	E,M
Mc [10]	C	N	M,C,S	C	-	-	A	F,B	T	E,M
MC_m [6]	C	N	M	V,C	Q	-	A	F,B	T	E,M,R
CC_m [6]	A,D,C	Y	C	V,C	C,Q	P,I	P,A	F,B,S	T	E,M,R
CM_m [6]	C	Y/N	C	V,C	Q	-	P,A	F,B	T	E,M,R
$HSCC$ [7]	D,C	Y	C	V,C	C,Q	P,I	P,A	F,B,S	T	E,M,R
NAL	A	Y/N	C	V	Q	P	P,A	S	T	E,M,R
NAM [13]	A	Y	C	V	Q	P,I	P,A	S,B	T	E,M,R
$Size2$ [4]	A	Y	C	V	Q	P	P,A	S,B	T	E,M,R
$TJCC$ [11]	D,C	Y	C	V,C	C,Q	P	P,A	F,B,S	T	E,M
WMC [12]	C	Y	C	C	Q	-	A	F,B	T	E,M,R
$CCGI$ [13]	A,D	Y	C	C	C,Q	P,I	P,A	B	P,C	E,M,R
DIT [12]	D	Y	C	-	C,Q	I	P,A	S	P,T,C	M,R
$NSUP$ [13]	D	Y	C	-	C,Q	I	P,A	S	P,T,C	M,R
SC_m [6]	A,D,C	Y	S	V,C	-	P,I	P,A	F,B,S	T	E,M,R
T_m [13]	C	N	S	V,C	-	-	A	F,B	T	E,M

Table 1. Taxonomy of metrics in TAC++: $A/D/C$ Analysis, Design and Coding; OOP object-oriented suitability; $M/C/S$ Method, Class and System level; V/C Volume and/or Complexity metric; C/Q Conformity to OOP and/or Quality; Relationships: is-Part-of, Inheritance; P/A Predictive and/or *A Posteriori*; $F/B/S$ Functional, Behavioral, Structural aspect; $T/C/P$ Technical, Cognitive, Process-oriented metric; $E/M/R$ Effort, Maintenance, Reuse. $< metric >_m$ metrics can be based on McCabe, Halstead, or LOC.

[1]This work was partially supported by MURST Ex60% govern Ministry of University and Scientific and Technologic Research, and by CESVIT, High-Tech Agency, Center for Software Quality, Florence.

0-8186-8421-6/98 $10.00 © 1998 IEEE

product characteristics must be used. Product features are typically: quality (see ISO 9126), cost, reuse, conformance with the system requirements, conformance with market demands, etc. Obviously, each feature must be in some way measurable and suitable actions for its achievement must be identified. An ever growing attention to the software development process has created the need to get process-oriented information and to integrate metrics into the software development process and life-cycle; thus, both real data and measured features have to be continuously compared for controlling this process and, when needed, for adjusting the process model (continuous process identification and improvement). This means that it is important to adopt a unique method and approach for project measurement. This should be capable of being tuned to adapt its parameters to different life-cycle phases, types of projects, etc. This process of adaptation is usually performed by adjusting weights and thresholds [3]. Some studies with metrics and measurement frameworks for object-oriented systems have been presented in [4], [5], [6], [7], [8], where general concepts for the estimation of system size, complexity and reuse level have been proposed together with many other metrics. Unfortunately, the effort for defining new metrics has not been supported by the implementation of assessment tools. Therefore, an integrated framework for developing and maintaining under control the system under development must be supported by tools for: (i) defining and evaluating direct and indirect metrics, (ii) defining and showing suitable views of the system (profiles, diagrams, tables, graphs, histograms, etc.) as well as of its components/classes (views should be focussed on assessing quality, conformance to the OOP, etc.), (iii) tuning metrics by estimating weights and scale factors, (iv) controlling project evolution by using reference/threshold values, (v) collecting and comparing projects trends.

This paper describes TAC++ (Tool for Analyzing C++ code). This research tool has been developed in several years of work and is capable of estimating a huge number of different metrics. TAC++ is comprised of an integrated class browser/editor, which is capable of estimating the values of several direct metrics. On these bases, high-level indirect metrics can be obtained. TAC++ provides a set of instruments for: (i) defining new metrics by means of a visual interface; (ii) defining and visualizing specific views/profiles even considering typical, minimum, maximum, values; (iii) defining and visualizing statistic histograms about the characteristics of the system under analysis; (iv) identifying and tuning weights contained in complex indirect metrics by using a multi-linear regression tool.

2. Taxonomy of Object-Oriented Metrics

Metrics can be classified according to different criteria. A first classification can be performed on the basis of their capability in *predicting* and/or evaluating *a posteriori* system characteristics.

Metrics can also be divided in direct or indirect metrics. *Direct* metrics should produce a direct measure for the parameters under consideration; for example, the measure of the Lines of Code (LOC) for estimating the program length. *Indirect* metrics are usually related to high-level characteristics; for example, the number of system classes can be supposed to be related to the system complexity by means of a mathematical relationship, while LOC (as indirect metric) is typically related to effort of development.

According to the classical definition of functional metrics, these can be classified in *volume* (also called *size*) and *complexity* metrics. Metrics can be focused on considering functional, behavioral and/or structural aspects.

Another classification is based on a technical, cognitive, and process-oriented view of the system aspects. The *technical* view refers to the software engineering aspects of system specification (size, complexity, etc.); the *cognitive* view takes into account the external understandability and verifiability of the system; and, the *process-oriented* view refers to the system aspects that are influenced by or can influence the process of system development (productivity, reuse, cost of development, cost of maintenance, etc.).

Metrics are also frequently classified on the basis of the phase in which they can produce significant estimations; therefore, a distinction is made for *analysis*, *design* and *code* metrics.

Another very important classification, specifically used for object-oriented metrics, is based on the level of applicability; thus, method, class and system level metrics can be identified. *Class level* metrics are the most important, since according to OOP, all components (structural aspects) should be defined in terms of classes. In this case, functional, behavioral and structural aspects are considered.

As pointed out by many authors, traditional metrics for complexity/size estimation, often used for procedural languages, can be difficultly applied for evaluating object-oriented systems [3], [5], [4].

3. Short Overview of Metrics

By using Tab.1, it is possible to identify the most suitable metrics for assessing specific features of the system as described in the rest of this paper. In the next subsections several metrics that have been used for assessing object-oriented systems are briefly reviewed. These are organized in three main levels: method, class, and system.

In order to help the reader to understand metric formulation and discussion, the authors have prepared Tab.3 in which the metrics and their corresponding meaning are reported in alphabetic order.

3.1. Method Level Metrics

At the method level, traditional functional metrics can be used, such as the McCabe Ciclomatic Complexity, Mc, [10], [1], the Halstead measure, Ha [9], and the LOC. The method interface can be even effective for evaluating cognitive and data-flow aspects of method complexity. A generic method complexity can be defined as:

$$MC_m = w_{MIC_m} MIC_m + w_m m, \tag{1}$$

where MIC_m is the Method Interface Complexity/size, and m a complexity/size metric for *method evaluation*; w_{MIC_m}, and w_m are weights. These weights are determined by means of the validation process [6]. The presence of MIC_m makes MC_m usable as a predictive metric, since MIC_m can be estimated even if the method has not been coded. m can be Mc, LOC, Ha, etc. The above metrics are more complete than the simple complexity/size metrics on which they are based.

3.2. Class Level Metrics

Class metrics can be divided into complexity/size and class relationship metrics. The former are usually applied to estimate the effort of development or reuse, maintenance or documentation and the latter for assessing object-orientedness, quality, re-usability, mainteinability, etc. Thus, the former can be considered as technical metrics and the latter as cognitive metrics.

3.2.1 Complexity/Size Metrics

By using the above method level metrics, it is immediate to define corresponding class metrics. For example: CM_m is the complexity/size metric evaluated on all class methods by using metric m. Therefore, the following class metrics can be defined: CM_{Mc}, a class level metric based on McCabe metric (equivalent to WMC [12]); CM_{Ha}, a class level metric based on Halstead metric and CM_{LOC}, a class level metric based on the number of LOC (as used in [14]). In the literature, it has been often demonstrated that these metrics are not very suitable for evaluating object-oriented projects, since they are not capable of considering the object-oriented aspects [11], [7]. In fact, they neglect information about class specialization (*is-a*, that means code and structure reuse), and class association and aggregation (*is-part-of* and *is-referred-by*).

Thomas and Jacobson have suggested to estimate class complexity, $TJCC = w_{CACL}CACL + w_{CL}CL$, as the sum of attribute and method complexities [11], without considering the class external interface (i.e., method interface) and reuse (i.e., inheritance). Henderson-Sellers has added to the above metric a term for considering inheritance [7], $HSCC = w_{CACL}CACL + w_{CL}CL + w_{CI}CI$.

A fully object-oriented metric for evaluating class complexity/size has also to consider both locally defined and inherited attributes and methods [5], [6]. Therefore, the Class Complexity, CC_m, is regarded as the weighted sum of local and inherited class complexities:

$$\begin{aligned} CC_m &= w_{CACL_m} CACL_m + w_{CMICL_m} CMICL_m \\ &\quad + w_{CL_m} CL_m + w_{CACI_m} CACI_m \\ &\quad + w_{CMICI_m} CMICI_m + w_{CI_m} CI_m. \end{aligned} \tag{2}$$

where $CACL_m$ and $CACI_m$ are the Class Attribute Complexities Local and Inherited, $CMICL_m$ and $CMICI_m$ are the Class Method Interface Complexities of Local and Inherited methods, CL_m and CI_m are Class Complexities due to Local and Inherited methods. In this way, CC_m takes into account both structural and functional/behavioral aspects of class. CC_m metrics result to be a generalization of $TJCC$ and $HSCC$. The weights or the interpretation scale must be adjusted according to the phase of the system life-cycle in which they are evaluated as in [5], [6].

3.2.2 Prediction of Complexity/Size

Metric CC_m can also be used for predicting class complexity/size. The prediction is obtained on the basis of class definition, that is, attributes declarations and methods prototypes. This estimation can be performed during system analysis/early-design:

$$\begin{aligned} CC'_m &= w'_{CACL_m} CACL_m + w'_{CMICL_m} CMICL_m \\ &\quad + w'_{CACI_m} CACI_m + w'_{CMICI_m} CMICI_m. \end{aligned} \tag{3}$$

A cheaper approach can be simply based on counting the number of local attributes and methods (see metric $Size2 = NAL + NML$ defined by Li and Henry in [4]). On the other hand, the simple counting of class members (attributes and methods) could be in many cases too coarse. In order to improve the metric precision, a more general metric can be defined by considering the sum of the number of class attributes and methods, both locally defined and inherited. This is named NAM in our framework. As demonstrated by means of experiments (see Tab.2), this more complex metric does not increase too much the capability of complexity prediction with respect to $Size2$ [6], while CC'_m is better ranked.

3.2.3 Class Relationship Metrics

For a better evaluation of features related to cognitive aspects Class CoGnitive Index, $CCGI$, is introduced:

$$\begin{aligned} CCGI &= \frac{ECD}{CC} = \frac{ECD}{ECD + ICI} = \\ &\frac{CACI + CACL + CMICI + CMICL}{CACI + CACL + CMICI + CMICL + CI + CL} \end{aligned} \tag{4}$$

Metric $CCGI$ indicates to what extent the class can be understood by using the information contained in the external class description with respect to its global complexity. For example, if a class presents several small methods in its definition, then it is more understandable than a class that, having the same total complexity/size, presents a lower number of members.

According to ICI and ECD definitions, metrics $CCGIL$ and $CCGII$ can be easily derived. These can be useful for evaluating whether unsuitable values of $CCGI$ are due to the presence of local or inherited conditions.

The structure of the inheritance hierarchy impacts on system maintainability, re-usability, extensibility, etc., since a high number of superclasses can make classes hard to be understood and tested. In the literature, the so-called DIT, Depth of Inheritance Tree metric has been proposed. DIT estimates the number of direct superclasses until the root is reached [12]. A more useful metrics is the $NSUP$, Number of SUPerclasses till the roots are reached. Please note that the difference between DIT and $NSUP$ can be relevant even if the height of the inheritance tree is limited to 3 in the case of multiple inheritance.

3.3. System Level Metrics

According to the current C++ interpretation of the OOP, the *system level* is comprised of: (i) a set of classes which can be organized in one or several class trees, (ii) a set of C functions/procedures (even the main program can be considered as a function/procedure), (iii) a set of global definitions, and (iv) a set of global declarations of variables.

Therefore, System Complexity, SC_m, can be computed by using the complexity of all system classes added to the functional and data complexities due to non object-oriented parts. SC_m metric is capable of producing a more precise evaluation of system complexity since it considers functional, behavioral and structural aspects, differently from traditional pure functional metrics: Total McCabe Complexity (T_{Mc}) indirectly based on McCabe's metric, the Total Halstead T_{Ha}), and the Total LOC (T_{LOC}) defined as the sum of their respective metrics on all class methods of the system. These metrics can give a concise system evaluation, but in several occasions, they can lead to wrong deductions since the manager (the evaluator) is not capable of identifying the critical conditions.

4. Controlling Object-Oriented Development

The above-mentioned metrics, as many others, can be evaluated by using TAC++ (see Fig.1). TAC++ is a research prototype suitable for studying the metric behavior and development process, which includes a class browser with editing capabilities. TAC++ also allows to define new high-level metrics on the basis of the available metrics by composing them with simple operators. In addition, a dedicated visualizer can be used for defining specific graphical representations of views/profiles (by using Kiviat, line graphs, bars, pies, etc.). The defined views can be used for monitoring aspects of the system under assessment, at the level of method, class, and system. In these graphical view profiles, different weights, reference thresholds, minimum and maximum values for diagrams can be set according to the company/user goals and product profile.

By using TAC++ a direct control on the indicators for classes and methods is performed. This allows to maintain subsystem quality, effort, etc., within predefined ranges. On the other hand, the continuous metrication must be associated with a continuous re-validation of the adopted indicators. To this end, the measured values of class indicators have to be collected in a database. Non-automatically measurable data, such as the effort related to class are collected by filling forms or Java pages, which are stored into the project database.

As can be observed in Fig.1, in order to perform the above operations TAC++ is comprised of five main components addressing the problems of: navigating in the system classes, evaluating low-level metrics, defining and evaluating high-level metrics, defining and showing system profiles and histograms, and statistically system analyzing for metric validating and tuning.

4.1. Collecting Low-Level Measures

The process for collecting low-level measures is based on a first phase of preprocessing, such as those usually adopted in compilers. It allows definition solving, macro exploding, etc. The code obtained is processed by a lexical-grammatical analyzer. The relationships among classes are identified and stored in a suitable multi-linked structure (System Class Description). All relationships are established by iterating overall system classes.

To perform this process, the user can decide to assess or not class libraries (if any). In any case, the C++ headers files of the libraries used must be available. If the code of class library is available, it can be considered into the assessment process.

The above process is capable of producing Low-Level Metrics (LLM). These are direct metrics – such as LOC, Mc, Ha, NA, NM, MIC, etc. LLMs Evaluator saves on a file the multi-linked structure.

The results produced by the LLMs Estimator can be collected on the basis of the life-cycle phase in which they are estimated. The browser shows the list of classes with a synthetic description of their relationship: class hierarchy, list of methods, etc. By selecting a class and a method, the corresponding code is directly available in a separate editing window.

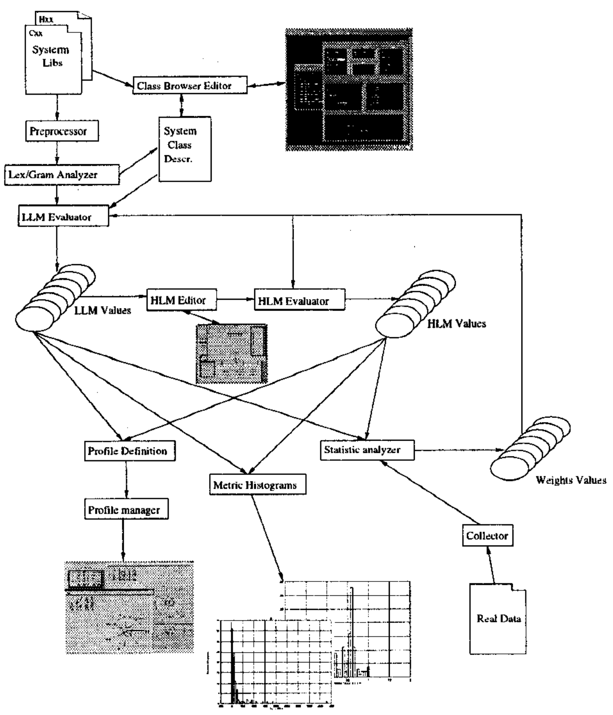


Figure 1. TAC++ Structure, Organization and Features

4.2. Defining and Collecting High-Level Metrics

High-Level Metrics (HLMs) can be defined by the user on the basis of LLMs. To this end, a specific interactive tool allows the definition of new metrics by means of a visual editor. HLMs can be defined according to the following structure:

$$NewMetric = \sum_i W_{M_i} M_i \frac{W_{U_i} \sum_x U_i}{W_{D_i} \sum_x D_i} \qquad (5)$$

where x is the context in which the sum is performed – for instance (i) on all system classes, (ii) on all class methods, (iii) on all class attributes, etc. Each sum on x can be set to operate on a single value, thus transforming the terms in a single term. Weights can be imposed on the basis of company experience and goals by using a set of reference projects with the aim of improving the related database of weights.

The HLMs Evaluator obtains the values of indirect metrics on the basis of the current definitions of HLMs and by using LLMs values.

4.3. Visualizing Results

In order to provide a fast and understandable view of the project status, the values obtained for LLMs and HLMs can be visualized in a set of specific views. Views can be histograms or profiles.

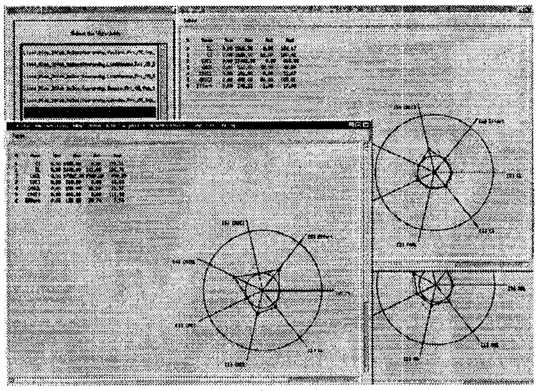


Figure 2. TAC++ Kiviat Diagram for a selected view.

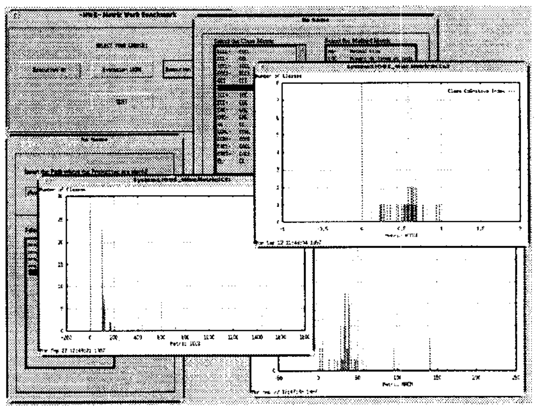


Figure 3. TAC++ generated histograms.

4.3.1 Profiles

A profile is a consumptive view which is capable of showing the values of several metrics with respect to their specific mean, maximum and minimum values. The minimum/maximum value(s) can be considered as the lowest/greatest value(s) under/over which a correction should be needed. By the use of normalized graphs shows these views: Kiviat, bar, pie, etc. by means of the Viewer (see Fig.2). For example, in order to monitor class quality a view reporting values of metrics: NA, NM, CC_m, CM_m/NM, $CCGI$, $NSUP$, etc., can be defined.

4.3.2 Histograms

By using TAC++ it is also possible to evaluate the statistic view of each metric for the system under assessment. For example: (i) the number of classes for the complexity of classes, (ii) the number of methods for the complexity of methods, (iii) the number of classes for their $CCGI$, etc. (see Fig.3). These histograms are useful for identifying which classes are outside of the bounds established or recommended by the company.

4.4. Validating and Tuning Metrics

Despite the high number of metrics, there exist in the literature only few papers reporting accurate validations of metrics. It mainly consists in identifying metric parameters (weights) on the basis of the knowledge of actual data, depending on the goals of the metric under validation.

The validation process can be used for (i) verifying which terms of each identified metric are relevant for its estimation (ii) evaluating the confidence of the measures obtained, (iii) tuning metric models according to different context and profiles (weights and scale identification), (iv) identifying metric parameters along the development life-cycle for evaluating the development progress with respect to reference trends. Usually, the validation can be performed by using mathematical and statistical techniques such as multilinear regression tools [15]. Real data reporting direct measures of the features that should be evaluated by metrics are needed – e.g., real effort, number of defects identified, etc. Using a Data Collector, which in our tool is developed in Java, to be portable in a wide number of platforms, collects this information. The Statistic Analyzer is capable of estimating all the metric weights used in a metric if the corresponding real value of the metric is available. Usually, a metric may present a high number of components, but not all the terms have the same importance. By using the Statistic tool, it is possible to verify not only the correlation of the whole metric with respect to the real data, but also the correlation of each term of the metric with respect to the collected effort, maintenance or other real data. In this way, a process of refinement can be performed in order to identify whose terms of the metric are more significant than others for obtaining the indirect measure.

Please note that the most important metrics of TAC++ have been analyzed and validated by using the above technique, during the development of several C++ projects.

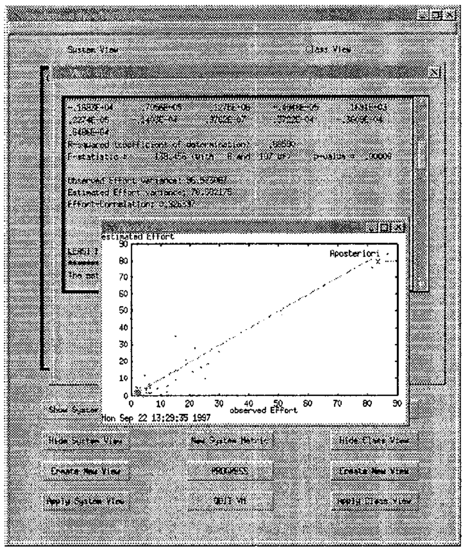
The engine of Statistic Analyzer is mainly based on multilinear regression techniques [15] (Progress). Since Progress tool presents a textual interface, a graphical user interface has been added to make it more user friendly. Moreover, the result of multilinear regressions can be easily interpreted since they can be graphically visualized as dot diagrams (see Fig.4).

4.5. Experiments on Effort Evaluation and Prediction

In Tab.2, the most diffuse metrics for effort estimation at the level of class are compared with the well known pure functional metrics. The comparison is made on the basis of correlation and variance values obtained by the TAC++ Statistic Analyzer (confidence values and values of weights have been omitted) (see [6] for the validation process).

Lower values of variance correspond to a less spread distribution. The analysis reported shows that traditional functional metrics can be profitably employed for evaluating object-oriented systems if they are used as a basis for more complete metrics (as in [14], [12]). The metrics proposed in [6] present both a higher value of correlation and a lower value of variance.

In order to estimate the trend of selected metrics and weights for subsequent evaluations, it is very important to record values during the early phases of the system lifecycle. Once the weights are identified, the adoption of predictive metrics is very useful (see Tab.2 in which the correlation values obtained for these metrics are very encouraging) because small errors can be accepted in the early phase of software development cycle. By using TAC++, a collection of development trends for some projects have been recorded by the workgroup of the authors in order to establish bounds for metrics and values for the weight depending on the application field.



```
-.1882E+04    .7068E+05    .1278E+06    -.4408E+05    .1591E+03
.2274E-05    -.1493E-04    .9762E-07    .3722E-04    -.3609E-04
.8496E-04
R-squared (coefficient of determination) = .8859
F-statistic =     138.456 (with  8 and 107 df)    p-value = .00000

Observed Effort variance: 86.523067
Estimated Effort variance: 76.682175
Effort-Correlation: 0.926337
```

Figure 4. TAC++ Statistical Tool

A Posteriori	Corr.	Variance
$WMC = CL_{Mc}$ [12]	0.90	245
CM_{LOC} [14]	0.91	186
CM_{Ha} [9]	0.82	423
CC_{Mc} [6]	0.93	149
CC_{Ha} [6]	0.94	216
CC_{LOC} [6]	0.94	145
$TJCC$ [11]	0.93	157
$HSCC$ [7]	0.93	146
Predictive	Corr.	Variance
CC'_{LOC} [6]	0.88	770
NAM [13]	0.73	1100
$Size2$ [4]	0.72	1700

Table 2. Comparison between metrics defined for evaluation and prediction of class effort.

5. Discussion and Conclusions

The adoption of the OOP has produced a great demand of specific metrics. Several direct and indirect metrics for the evaluating effort, maintenance and re-usability costs, have been defined. Therefore, an integrated tool for defining, showing and validating them is mandatory in order to manage this large number of metrics.

The tool presented in this paper offers developers, subsystem and project managers a highly configurable environment to control all the aspects of C++ projects since the early phases of system development. TAC++ provides many features for aiding project development and maintenance: (i) direct manipulation of code, (ii) low-level metric evaluation, (iii) high level metric definition and evaluation, (iv) graphical representation of metric profiles and histogram, (v) metrics validation and tuning by the means of a multilinear regression engine, and (vi) real data collector – i.e. real effort in person-hours, number of errors and faults, etc.

TAC++ has been profitably used for controlling the development and maintenance of several projects by the research group of the authors in order to validate its use in the application field. The results obtained by the above mentioned tests have permitted to the research group to establish bounds, reference profiles and histograms for a wide typology of applications, that will be used for developing present and future C++ projects.

Acknowledgments

The authors would like to thank all the members of TAC++ team. A special thank to B. Pages for an early version of the class browser (i.e., Xcoral), and to A. Borri of CESVIT, for his help in designing the first version of TAC++.

References

[1] B. Henderson-Sellers and J. M. Edwards, "The object oriented systems life cycle," *Comm. of the ACM*, vol. 33, 1990.

[2] P. Nesi, *Objective Software Quality, Proc. of Objective Quality 1995, 2nd Symp. on Soft. Quality Techniques and Acquisition Criteria.* Berlin: LNCS, N.926, Springer, 1995.

[3] B. Henderson-Sellers, D. Tegarden, and D. Monarchi, "Metrics and project management support for an object-oriented software development," in *TM2, TOOLS Europe'94*, (France), 1994.

[4] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *JSS*, vol. 23, 1993.

[5] P. Nesi and M. Campanai, "Metric framework for object-oriented real-time systems specification languages," *The JSS*, vol. 34, 1996.

[6] P. Nesi and T. Querci, "Effort estimation and prediction of object-oriented systems," *The JSS*, vol. in press, 1998.

[7] B. Henderson-Sellers, "Some metrics for object-oriented software engineering," in *Proc. of TOOLS 6 Pacific*, USA, 1991.

metric	comment
$CACI_m$[6]	Class Attribute Complexity/size Inherited
$CACL_m$[6]	Class Attribute Complexity/size Local
CC_m[6]	Class Complexity/size
CC'_m[6]	Class Complexity/size, predictive form
$CCGI$[13]	Class CoGnitive Index
$CCGII$[13]	Class CoGnitive Index Inherited
$CCGIL$[13]	Class CoGnitive Index Local
CI_m[6]	Class Method complexity/size Inherited
CL_m[6]	Class Method complexity/size Local, equivalent to CM_m
CM_m[6]	Class Method complexity/size equivalent to CL_m
$CMICI_m$[6]	Class Method Interface Complexity/size Inherited
$CMICL_m$[6]	Class Method Interface Complexity/size Local
DIT[12]	Deep Inheritance Tree
ECD[13]	External Class Description
Ha[9]	Halstead metric
$HSCC$[7]	Class Complexity by Henderson–Sellers
ICI[13]	Internal Class Implementation
LOC	number of Lines Of Code
Mc[10]	McCabe ciclomatic Complexity
MC_m[6]	Method Complexity/size
MIC_m[6]	Method Interface Complexity/size
NA	Number of Attributes of a class
NAI	Number of Attributes Inherited of a class
NAL	Number of Attributes Locally defined of a class
NAM	Number of Attributes and Methods of a class
NM	Number of Methods of a class
NMI	Number of Methods Inherited of a class
NML	Number of Methods Local of a class
NRC[6]	Number of Root Classes in the system class tree
$NSUP$ [13]	Number of SUPerclasses of a class
SC_m[6]	System Complexity/size
$Size2$[4]	Number of class attributes and methods
T_m	Total m-based functional Complexity
$TJCC$ [11]	Class Complexity
WMC [12]	Weighted Methods for Class, CL_{Mc} in our notation

Table 3. Glossary of the metrics mentioned in this paper. Metrics with $_m$ parameter are evaluated on the basis of a functional metric selected from: Mc, Ha or LOC; for example: CC_{Mc} Class Complexity/size based on McCabe ciclomatic Complexity.

[8] H. Zuse, "Quality measurement – validation of software metrics," in *Proc. of QW'94*, Software Research, 1994.

[9] H. M. Halstead, *Elements of Software Science.* Elsevier North Holland, 1977.

[10] T. J. McCabe, "A complexity measure," *IEEE Trans. on Soft. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[11] D. Thomas and I. Jacobson, "Managing object-oriented software engineering," *Tut. Note, TOOLS'89*, (France), 1989.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Soft. Eng.*, vol. 20, 1994.

[13] F. Fioravanti, P. Nesi, and S. Perlini, "Assessment of system evolution through characterisation," tech. rep., DSI, Facolta' di Ingegneria, Univ. Firenze, RT 22/97, Italy, 1998.

[14] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics, A Practical Guide.* New Jersey: PTR Prentice Hall, 1994.

[15] P. J. Rousseeuw and A. M. Leroy, *Robust Regression and Outlier Detection.* New York, J. Wiley & Sons, 1987.