Language Independent Software Complexity Measurements

Ádám Sillye
Eötvös Loránd University
Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/c
1117, Budapest, Hungary
madic@elte.hu

Zoltán Porkoláb
Eötvös Loránd University
Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/c
1117, Budapest, Hungary
gsd@elte.hu

ABSTRACT

Complexity metrics play an important role in software development; they are reducing the costs during almost the whole development process. There is a growing demand for measuring the complexity of large systems with keeping the consistency of the results regardless of the diversity of the programming languages. In this article we present a general software measurement process on .NET basis that fulfills the above criteria, namely the uniformity of calculation and the language independency. We present some popular metrics and their calculation from compiled .NET assemblies using Intermediate Language (IL) disassembling. Next, we examine the IL encoded data and prove its suitability for measurements. Finally, we show some concrete examples with simple Visual Basic and C# sources.

Keywords

.NET, software complexity, metrics, multilingual environment, IL, intermediate language

1. INTRODUCTION

Reducing costs of software development and maintenance is always a challenge, especially if we want to keep or even improve the software quality. Complexity metrics play an important role here; they help us decrease the cost from the development and testing phase and during the maintenance.

There is a growing demand for measuring the complexity of large multilingual systems for two reasons: in a large system, there is no way to estimate the cost of the test phase *manually* and having different languages in the same program demands a consistent measurement method.

Being able to trigger the critical system parts and give constructive advice with using an *automatic* measurement and evaluation system is a great advantage, since we save the verify efforts and gain the measurement result in the same time.

Having a generic – language independent – measurement tool in a multilingual environment is a great opportunity to produce consistent and reliable results. Creating dedicated measurement tools for each specific language is an excessive task, since we have to face all exercise that all compilers have. Usually there is no way to extend the compilers with measurement facilities, because either their source is not available or even if it is, their internal structure is highly optimized for the compilation. Therefore, it would be necessary to reimplement all programming language dependent exercise to have the input for the measurements. (Moreover, for every new programming language, we would have to repeat this task again.)

The Intermediate Language (IL) of .NET enables us to avoid the above language problems, provided that, the encoded information in .NET assemblies is enough to calculate metrics and the calculated results are not distorted. In this article we are going to show that these calculations are achievable by careful data extraction.

There are some good metric calculation packages in the Java world; unfortunately for .NET we only have a few. Our aim was more than to create a software measurement tool for a specific language. We created a generic multilingual measurement package by exploiting the diversity of the available .NET compilers.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-903100-4-4

In this paper we examine the IL from the point of view of measuring. First, we briefly present some currently popular metrics, including a multi paradigm metric. In order to calculate these metrics we look through the IL metadata and instructions. We will show that there is enough information to do the calculations and will also show that the encoded information is applicable for measurements, namely it is distorted and reflects the *original source*. Finally, we present some simple examples and measurement result.

2. SOFTWARE METRICS

In the following sections we give a brief overview of those metrics we used in our tool.

Size metrics

Most primitive metrics are based on the physical attributes of the program code. The *lines of code* (LOC) measures the actual size of code by counting the source lines, while *effective lines of code* (ELOC) ignores the possibly ineffective lines like comments, block commands, some preprocessor directives etc. Number of commands (NOC) works similarly, it counts the statements.

Great advantage of these metrics are the cheap computation and easy understanding of the results, moreover they have surprisingly high correlation with some other more sophisticated metrics. The critical weakness of physical basis is the semantics; namely the size is only one aspect of software complexity. In this article we are focusing on other – more advanced – metrics to achieve higher precision.

Structural metrics

The first well-known structural measure was developed by McCabe [McC76]. His cyclomatic complexity is based on the number of predicates (branches) in a program graph G: $V(G) = p + 1$. The aim of this metrics was to approximate the testing effort of FORTRAN programs; therefore the cyclomatic number reflects the independent testing passes of the program. The inadequacy of the measure becomes clear, if we realize that the complexity also depends on the nesting level of the predicate nodes. According to the McCabe cyclomatic complexity a sequence of ten loops is identically complex to ten loops nested into each other.

Harrison and Magel [HM81] proposed a metric which also takes into account the nesting depth of the predicate statements. Piwowarski [Piw82] got the same result independently. The more nested statements weighted higher in the complexity. This concept was improved by Howatt and Baker [HB89] whose definition for nesting level was applicable for non structured programs too.

Object oriented metrics

One of the most complete discussions about object oriented metrics was given by Chidamber and Kemerer [Chi94] and Henderson-Sellers [Hen96]. They defined the following important object oriented complexity metrics:

WMC (Weighted Methods per Class): This summarizes the structured complexity of methods within a class. McCabe metrics is the most common here.

DIT (Depth of Inheritance Tree): The length of the longest inheritance path in the inheritance tree. Understanding a class requires the understanding of all base classes, so deeper structures have higher psychological complexity.

NOC (Number of Child Classes): This measures the importance of the class rather than the complexity. Their reasoning was the following: by having more and more reuses of the same class, there is an increasing possibility of bad abstraction. (We think more or less the opposite: having more reuses of the same code decreases the overall complexity.)

CBO (Coupling Between Object Classes): The number of references pointing into the class (fan-in) and the number of references pointing out from the class (fan-out). Higher dependency from other classes raises the difficulties of reuse and maintenance.

RFC (Response for Class): Similar to CBO, but RFC takes into account the incoming and outgoing method calls. Similarly, the higher dependency and the increased intelligibility of the control structures raise the complexity.

LCOM (Lack of Cohesion in Methods): This metric is based on an empirical result: higher cohesion within a class reflects better abstraction and encapsulation.

Multi paradigm metrics

While object-orientation has become ubiquitously employed for design, implementation and even conceptualization, many practitioners recognize the simultaneous need for other programming paradigms according to problem domain [Mul]. Paradigm-independent software metrics are applicable for programs written in different paradigms or in mixed-paradigm environment. Such metrics are based on general programming language features which are paradigm- and language independent and the paradigm dependent attributes are derived from the above features:

Control structure of the programs: Most of the programs share the same control statements, like instructions, branches and loops.

Complexity of data types: The structure of data types are reflecting the complexity of information being manipulated.

Complexity of data access: The connection between the control structure and the program data relates to the complexity of information manipulations. The measurement analysis starts from the recognition of the data flow direction of and the nesting depth of data handling operations.

As a paradigm-independent measurement we applied the AV-graph that has been described in [PZ1]. In this metric, the three paradigm independent features are expressed in an extended control flow graph. The control structure of the programs is directly represented in the control flow with instruction and predicate (branch) nodes. Data nodes are added to represent the data types; their weighted sum reflects the data type complexity. Additionally the graph contains directed edges as a representation of data handling. Data access complexity is calculated based on these edges by taking into account the direction of data flow and the nesting depth of the data operation in the control flow. (We present an AV graph in Figure 1. in relationship with the IL code.)

Note that we are not calculating metrics for functional languages.

3. .NET – A MULTILINGUAL ENVIRONMENT

The .NET Framework is a development and execution environment that allows different programming languages and libraries seamlessly work together [DotNet]. It has two main components: the Common Language Runtime and the .NET Framework class library. The Common Language Runtime is the foundation of the .NET Framework; this executes the program code and provides additional services for the development.

The Common Intermediate Language

Programs written in different languages under the .NET development environment most cases are translated into Common Intermediate Language (IL). IL is the byte-code language of the Common Language Runtime and has been standardized by ECMA and ISO (ISO/IEC 23271:2003). It has many superior features comparing to the pure machine code. IL stores the actual program code with a special instruction set and high level meta information about the program. The latter one is not necessary to run the program, but enables reflection and code analysis easier. For example, it is possible to recover the class hierarchy, class and method names and variable declarations. The IL instruction set can be divided to the following major parts from the point of measuring:

Base instructions: These are a set of basic operations, like loading and storing data on the stack, arithmetic operations, branch instructions, method calls etc. (Some example mnemonics respectively: ldloc, ldarg, starg; add, sub, mul; beq, bge; call) These instructions closely correspond to what would be found on a real CPU.

Object model instructions: The object model instructions are less built-in than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system. These instructions are object instance creation, virtual method call, exception handling, memory allocation and type management. (Some example mnemonics: newobj; callvirt; throw, rethrow; castclass.)

With an appropriate utility, any .NET executable can be disassembled to IL (for example with *ildasm*) and can be decompiled to source level within moderate boundaries. Certainly, the decompiled code is only similar to the original source but the classes, methods and even the program flow structures are still recognizable. We utilized this fact for the measurements.

Now we will take a closer look to the IL from the point of software measuring.

4. THE IL AS THE SOURCE OF MEASUREMENTS

Being rigorous is very important requirement on the measurement process: precision enables us to have interpretable absolute values that can trigger critical parts of an implementation and also enables us to have the future chance of comparison different program results.

Now, let us examine the Intermediate Language as an input data for the measurement process. Clearly, it is key to have enough information encoded in the .NET executable to do precise measure offs. IL stands half way between the native code and the source code; obviously it is more native machine code and less than the source code. Now, we will prove that the IL is a better choice for measurements than the other two formats:

Native code is specific for measuring. The encoded information here is *only* for execution on a machine environment, so all irrelevant data are detached from the executables. Furthermore, native code relies on particular machine architecture, which leads to the similar problem with the different languages. To sum up, there is no way to have precise measurements on native executables.

Going along the opposite extreme, source code is too complex; it requires a lot of effort to extract the relatively simple measurement information. Dealing

with source code text requires literally a half compiler (lexical- and syntactical analysis plus some semantic analysis too) for each different programming language to build up the program structures. This would change provided the primary data format of program sources change from pure text representation to real structures; but until then we have to face with this problem.

Note that the above problems prevent the software industry to widely use metrics, namely it is not easy to create a reliable measure tool based on executables or source text.

Now, let us collect the measurement input information from a .NET assembly; two major piece of information are required: type and flow information.

We gain *type information* with the .NET reflection API can extract all information about the object hierarchy, data types, method signatures, class members etc.

Control and data flow information is extracted by looking directly into the assembly binary data. This was the only point were we need additional effort, since the API does not allow to read IL code. (We used the .NET 1.1 API)

In the following sections we match the above information with the need of each specific metric. We also point out some problems and propose solutions for them.

Size Metrics

The LOC and eLOC metrics are counting the lines of the *source* code. Obviously, we cannot deal with source lines since the information about the physical characteristics of the source is lost during the compilation. Nevertheless we can summarize the number of IL instructions which naturally correlates to LOC and eLOC.

Structural Metrics

We have to build up the flow graph of the programs to calculate structural metrics. Having the exact control flow graph is essential, but for some metrics, we need additional data information too. For precise measurements we have to reconstruct these graphs. In Figure 1 we present the relationship between the reconstructed flow graph and the IL code. The example is compiled from the following simple C# program:

```
for(int i=0; i<rgint.Length; i++)
  Console.WriteLine(rgint[i]);
```

The main question about these graphs is whether the reconstructed information from IL code faithfully

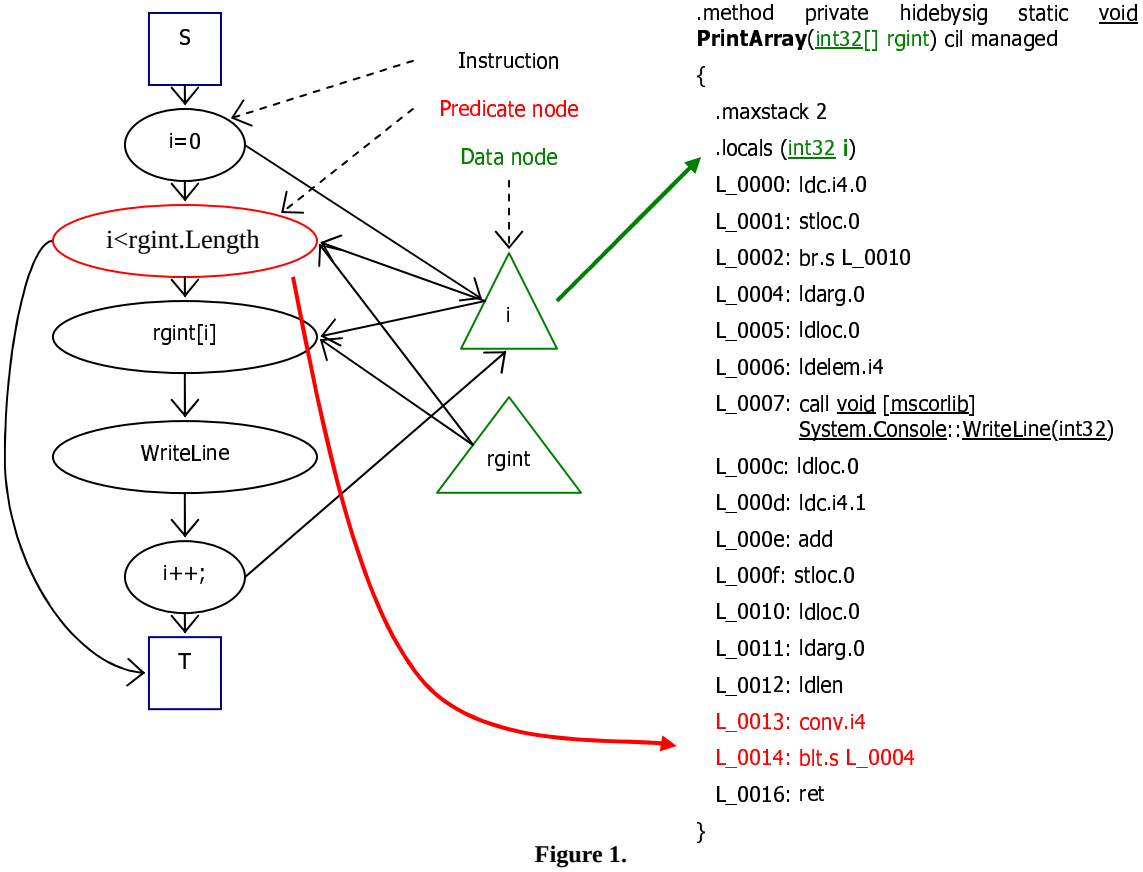

Figure 1.

reflects the original structure of the program. Here are some pro and cons on this question:

We examined the generated code of C# and the Visual Basic compilers and the results confirmed that the reconstructed flow graphs are very close to the ideal results. The essential nodes and edges of the graph were accurately rebuilt: there were no distortions on predicate nodes, control flow, data nodes and data accesses. We experienced slight differences in the number of simple instructions; in general this does not affect the final results radically.

Our goal is to have a measure tool that works with any (future) programming language, but there is no guarantee to have the above precise results. Obviously, it is possible to create a compiler that outputs such IL code that prevents us to extract the original flow information. In this case the compiler *probably* does not use exhaustively the IL features, generates inefficient or confused code. (Note that functional language compilers might generate code that is far from the source, but we do not consider them.)

For different programming languages the generated IL codes are sometimes not exactly the same, even if the language semantics are the same. For example, determining the length of an array is slightly different in the following two IL code compiled from C# and Visual Basic:

From C#:

```
ldlen
```

From Visual Basic:

```
callvirt instance int32 [mscorlib]
  System.Array::get_Length()
```

Code coming from Visual Basic has outside references to int32, mscorlib, System.Array, get_Length. This problem can be resolved by recognizing this special case and making them have the same weight.

Visual Basic generates special code for run time type checking, that may also distort the results: we will see an example on this in the next section.

Compiler optimizations could also cause slight differences in the measuring results. Microsoft has stated that they are not optimizing the generated IL code; their compilers only do dead code elimination and jump optimization. This helps us to create more precise flow graphs since optimizations are not simplifying the code. (However, small distortions may occur that encumber the process: for example, in the non optimized code there are superfluous instructions such as jump to the next instruction.)

Object oriented metrics

These metrics are precisely computable from the IL code, but we need some additional data extraction, namely the class hierarchy and the ability to distinguish between internal and external references relative to a class.

To compute the Weighted Methods per Class (WMC) we have to summarize the structural complexity of methods. The class hierarchy is fully extractable by the .NET API, therefore Number of Classes (NOC) and the Depth of Inheritance Tree (DIT) is given.

Internal and external variable and method references are also encoded in method bodies; the IL representation directly disambiguates them. Coupling Between Object Classes (CBO), Response for Class (RFC) and Lack of Cohesion in Methods (LCOM) can be calculated straightforwardly.

5. EXAMPLES AND RESULTS

The presented example here contains two C# and one Visual Basic class. Stack_CS and Stack_VB are two stack implementations and IntStack_CS is the client code.

Method	#i	MC	HB	AV
Stack_VB::Push	51	2	2	62
Stack_VB::Pop	40	3	4	63
Stack_VB::Peek	19	3	4	27
Stack_VB::Contains	40	8	30	143
Stack_VB::Clear	13	1	1	11
Stack_VB::get_Count	4	1	1	4
Stack_VB::main	1	1	1	2
Stack_CS::Push	50	2	2	60
Stack_CS::Pop	34	3	4	52
Stack_CS::Peek	18	3	4	26
Stack_CS::Contains	39	8	30	134
Stack_CS::Clear	13	1	1	11
Stack_CS::get_Count	4	1	1	4
Stack_CS::Main	1	1	1	2
IntStack_CS::Push	5	1	1	6
IntStack_CS::Peek	5	1	1	6
IntStack_CS::Pop	5	1	1	6
IntStack_CS::Contains	5	1	1	6

Table 1.

Stack_CS and Stack_VB classes have the same fields and methods. IntStack_CS shows the collaboration

between classes implemented in two different languages: IntStack_CS is a C# stack of integers inherited from the Stack_VB class.

The quantities above are calculated on per-method basis. (Legend: #i is the number of IL instructions, MC is the McCabe cyclomatic complexity, HB is the Howatt-Baker nested complexity and AV is the AV-graph result.)

There is a significantly high correlation between the Stack implementations. This proves that measuring based on IL results consistent data: language translation has no disturbing effect. The IntStack numbers represent the fact that our measurement tool is also able to work in multilingual environment: where base class and derived class were implemented in different languages.

Class	LCOM	FO	#f	#m	MC	HW	AV
Stack_VB	21	5	2	7	19	43	312
Stack_CS	21	4	2	7	19	43	289
IntStack	6	1	0	4	4	4	24

Table 2.

Object-oriented metrics effected similar result. In the columns we can see the results of the Lack of Cohesion in Methods (LCOM), the Fan Out value (FO), the number of fields (#f) and methods (#m), the summarized McCabe, Howatt and finally the AV-graph results.

Let us notice the difference between the faFOn-out values: as we have seen in our previous example the Visual Basic compiler generates a *System.Array::get_Length()* function call instead of *ldlen* which is one more method reference outside the stack class. (As we presented the fan out metric is the number of references that are pointing out from the class.) The AV-graph complexity also reflected this difference.

6. CONCLUSION AND FUTURE WORK

The Microsoft .NET environment is one of the leading development platforms. Implementing a reliable software measurement tool for .NET can be an important step for designers and developers. As the .NET platform is a multilingual environment by its nature, such tool must measure software written in different languages and handle inter-language connections too.

One way to implement such tool is to target the IL code as the measurement input. This enables to avoid the excessive work dealing with different languages and in the same time it increases consistency between the measurements. Handling inter-language connections are also possible.

In this article we presented a way to implement such tool by evaluating the relationship between the IL encoded information and the required information to calculate some popular metrics. We also demonstrated that these metrics produce reliable results, independently from the source languages.

In the future we continue the empirical evaluation of our .NET metrics tool. We plan to examine the forthcoming C# generics and its relation to IL. It is also promising to develop special metrics for generics and aspects.

7. REFERENCES

[CE00] Czarnecki K., Eisenecker U.W.: Generative Programming Addison-Wesley, (2000).

[Chi94] Chidamber S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Software Engineering, vol.20. pp.476-498, (1994).

[Cop98] Coplien J.O.: Multi-Paradigm Design for C++ Addison-Wesley, (1998).

[DotNet] Microsoft .NET platform: http://www.microsoft.com/net

[Ecl] Eclipse Metrics Plugin: http://www.teaminabox.co.uk/downloads/metrics

[FNP99] Fóthi, Á., Nyéky-Gaizler, J., Porkoláb, Z.: On the Complexity of Class Proc. of the FUSST'99, Tallin, Estonia, pp.221-231 (1999).

[FenNeil] Fenton, N.E., Neil, M. Software Metrics: Roadmap, The Future of Software Engineering. ACM Press, New York, (2000).

[HB89] Howatt,J.W. and Baker,A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting The Journal of Systems and Sofware 10, pp.139-150 (1989).

[HK81] Henry S., Kafura D.: Software Sructure Metrics Based of Information Flow IEEE Trans. Software Engineering, vol.7, pp.510-518 (1981).

[HM81] Harrison,W.A. and Magel,K.I.: A Complexity Measure Based on Nesting Level ACM Sigplan Notices,16(3), pp.63-74 (1981).

[Hen96] Henderson-Sellers, B.: Object-oriented metrics: measures of complexity, Prentice-Hall, pp.142-147, (1996).

[McC76] McCabe, T.J.: A Complexity Measure IEEE Trans. Software Engineering, SE-2(4), pp.308-320 (1976).

[Mul] Multiparadigm Programming Home Page: http://www.multiparadigm.org

[PZ1] Fóthi Á., Nyéky-Gaizler J., Porkoláb Z.: The Structured Complexity of Object-Oriented Programs Mathematical and Computer Modelling 38, pp.815-827 (2003).[Piw82] Piwowarski,P.: A Nesting Level Complexity Measure ACM Sigplan Notices, 17(9), pp.44-50 (1982).