Measuring Software Complexity

Doctoral Qualifying Examination

Nenad Ukić

Ericsson Nikola Tesla d.d.
Research & Innovations
nenad.ukic@ericsson.com

Postgraduate study in
Electrical Engineering and Information Technology

Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split
Croatia

Contents

1 Introduction

Software systems are among the most complex human-constructed systems [10]. Unlike physical systems, such as computers, cars, or buildings, where certain elements are often repeated frequently, in good software systems usually there are no similar parts – if there are, they are abstracted into modules and referenced wherever needed. Similarity between building elements of software is probably the lowest among all human products. This implies that scaling up a software entity is not merely a repetition of same elements in larger sizes, usually we have to come up with new, not-yet-existing elements. In addition, new elements often interact with many existing elements so, as a result, complexity of software system as a whole grows much faster then linearly with size [10].

We differentiate between two fundamental groups of software: *essential* and *accidental* software complexity[10]. Essential software complexity is inherent to the problem that is being solved, and cannot be minimized or removed. Accidental software complexities include limitations and imperfections of the software methodology, tools or languages being used. By improving those aspects we can reduce accidental complexity and improve the productivity of the software development process. On the other hand, essential software complexity will always be present and it cannot be reduced. For example, implementing chess-playing software will always be a complex task, no matter which language, tool or methodology we use to develop it.

A common approach for reducing accidental complexity is raising the level of abstraction. Practically, this means focusing on important aspects of software development process. Instead of dealing with bits, registers and memory allocation, software languages with increased level of abstraction allow us to focus on business objects and the logic of the problem domain. In addition, the translation to a lower level of abstraction needed for actual execution is often automated and abstracted away from the software developer. This simplifies software development and enables higher quality of end products.

One of the core problems with software is its *invisibility* [10]. Ability to visualize software structures has been recognized as an important aspect of software development long time ago, but up until early 1990's there was no standard way of doing it. With the rise of object-oriented (OO) languages and Unified Modelling Language (UML) [55], the visualization of software and the word "model" entered the mainstream of software development.

UML models are typically used as design time constructs mainly because of the traditional work-flow they have been used in. This work-flow assumed the creation of a (mainly) class model of the software and the generation

of a source-code skeleton which was then used as a staring point for the implementation phase. All later phases, even the structural modifications, were usually done in the source code rather than in the model. The main reason for this is the additional effort required to maintain the model. After some time, the initial model was usually deprecated and used only for documentation purposes.

However, Model Driven Architecture (MDA)[59] and executable models changed this paradigm by making the models the main software artefact. The term "executable models" means that models are *turing complete*. Since executable, models can be used for specification of complete application functionality which can be verified by executing the model-based tests in the model interpreter. Such workflow considers models as the main intellectual property of the software development process. Similarly as the machine code in the traditional workflow, the source code is considered only as a temporary form towards the binary executable used in a production. Note also that semantic completeness of executable models enables automated implementation phase in which the **complete** source code is generated from a model. Model executablity and translatability are key features of MDA.

Although there exist software development methodologies that share the MDA vision [49] [65], they are not widely accepted within the industry. The main reasons for this are that they target real-time and embedded systems, use non-standard "flavor" of UML or are supported only by proprietary tools. However, development of executable UML standards for the general software development is very slow and still in the development [57].

In the traditional workflow, different software qualities are measured and estimated using source-code complexity metrics. By changing the core artefact of software development process, complexity metrics should be updated and adapted as well. Measuring complexity of executable UML models is a relatively new area and has not yet been investigated thoroughly. When defining software complexity metric on executable UML models, we need to take into account the twofold purpose of models: not only are they used as specification of complete software functionality, but they are also used as a visualization, documentation and a communication tool.

Executable UML models usually use the same or simplified syntax as the traditional UML models. An important difference added by the executable models are clear execution and timing semantics. However, the most of the complexity metrics applied to the traditional UML models can be applied to the executable models as well. Typically, executable models include class models, state-machine models and component models. However, on the action level, for specifying detailed processing, a textual *models* are typically used. From the complexity point of view, this processing model (language)

follows similar control- and data-flow rules as many other *traditional* languages. This implies that the most traditional, source-code level complexity metrics apply on processing model as well.

In this report we will give an overview of the state-of-the-art in software and UML model complexity metrics (section 2 and 3 respectively). In the future we expect to create a selection of complexity metrics that will be adapted and applied to xtUML models. Therefore, the section 4, gives an overview of executable UML methodologies and standards with special focus on the xtUML methodology. In the end, a conclusion and the future work is briefly described.

2 Software Complexity

On the action level, for specifying detailed processing, executable UML models use the textual notation, similarly as all the traditional programming languages. The important difference is in the fact that this textual action language allows specification of only processing part, no structural elements can be defined using it. From the complexity point of view, this textual processing model follows similar control- and data-flow rules as many other *traditional* languages. This implies that the most traditional, source-code level complexity metrics apply on the processing model as well. This chapter will investigate those metrics.

With growing software complexity, it's measuring and management become of paramount importance. Still, there are many unanswered questions about nature of software complexity and its properties. Basili [6] defines software complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task. If the interacting system is a computer and the task is program execution, complexity may be defined as memory and execution time required to perform the calculation. If the interacting system is an engineer, and the task is understanding, maintaining or debugging the software, then complexity can be defined as the difficulty of performing those tasks in a given time-period.

Software complexity is usually estimated or measured in order to be able to do better work effort estimations, to judge about design alternatives or to be able to predict erroneous software modules. Early estimations therefore often rely on some kind of requirements analysis. During design phase, more precise estimations are possible in order to choose among different design alternatives. Once a software project is delivered and the maintenance phase has started, a complexity measure may be used to predict modules which might contain disproportional amount of errors and to predict the effort of maintenance tasks.

In literature, software complexity is studied from several different perspectives. Each of those perspectives focuses on one single aspect of software complexity but those perspectives are not necessarily orthogonal i.e. in certain cases high correlation can be observed. Some of the most frequently mentioned perspectives to software complexity are:

- **Software size** is the most frequent complexity metric focusing on counting basic building blocks, such as lines of code
- **Psychological and cognitive complexity** focuses on human understanding aspect of complexity

- **Cyclomatic complexity** focuses on control flow complexity of software
- **Computational complexity** is primarily concerned with algorithmic complexity of software
- **Data/information flow complexity** analyses complexity of inputs, outputs and data interdependencies within software or some of it's parts
- **Functional size** as complexity metric measures value that software brings to the end users

2.1 McCabe's Cyclomatic Complexity and its variants

McCabe[48] presented graph-theoretic view on software complexity by abstracting software as a control flow graph. In that graph, nodes are groups (blocks) of commands of a program that can be executed only in (one possible) sequence. A directed edge connects two nodes if the first command in the target node *might* be executed immediately after the last command in the source node. Typically, edges in a control flow graph are result of conditional execution of a block of commands, such as *if* branches or conditional loops.

Presenting software as a control flow graph is interesting because it visualizes possible execution paths in the source code. Since the total number of possible paths may be hard, impractical or even impossible to calculate, McCabe introduced *basic paths*, a set of paths that can be used to construct all other paths.

For any single connected graph, the cyclomatic number or circuit rank represents the number of linearly independent cycles and is given with:

$$V(g) = e - n + 1$$

where e is the number of edges and n is the number of nodes. For a more general case, when a graph contains several *connected components* (i.e. islands of connected nodes) the formula will have the following form:

$$V(g) = e - n + p$$

where p is the number of connected components.

Assuming control flow graph has single entry and single exit node, control flow graph becomes strongly connected (each node is reachable from every other node) if we **add single artificial directed edge from last to first node**. McCabe defines cyclomatic complexity of a program (module) as a

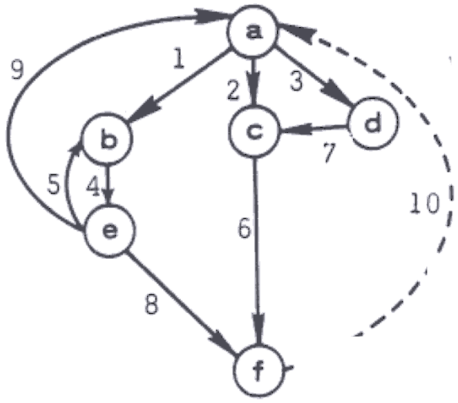
cyclomatic number of a corresponding strongly connected graph calculated with:

$$V(g) = e - n + 2$$

or in general case with p connected components:

$$V(g) = e - n + 2p$$

Figure 1: Example of a graph with 10 edges, 6 nodes, 6 regions and cyclomatic complexity equal to 6 (source [48])



Connected components in a control flow graph represent individual functions or subroutines of a program. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program. It is frequently used as basis for testing methodology because cyclomatic complexity defines minimal number of test cases needed to achieve complete branch coverage (each possible branch of execution is exercised).

Properties of cyclomatic complexity:

- Always larger or equal to 1: $V(G) \leq 1$
- It is maximum number of linearly independent paths in graph: it is the size of basic set
- Inserting or deleting functional statement does not affect V(G)
- Graph has only one path if and only if v(G)=1
- Inserting a new edge increases v(G) by one
- v(G) depends only on the decision structure of G

The number of edges and nodes is not trivial to calculate for larger source codes, a simplification formula may be applied: $v(G) = \pi + 1$ where π is the number of predicates (not conditions!) in the source code. Because of its simplicity, this way of calculating cyclomatic complexity is probably the most widely used. For visually presented strongly connected control flow graphs, there is the additional simplification for calculating cyclomatic complexity by counting the number of regions of that graph.

2.1.1 Cyclomatic complexity and modularization

Modularization of code assumes splitting a bigger block of code into smaller units, typically through the use of subroutines. There are two main reasons for modularization: *abstraction* and *code* reuse. When modularization is done for the sake of abstraction, a logical, self-contained, part of processing code is identified within a bigger block of code and factored out, hopefully by using a name which clearly describes its functionality and follows all agreed naming conventions. Modularization for the sake of code reuse is commonly used to remove code duplication. Such modularization assumes one definition and several usages of the created subroutine. Those two types of modularization are fundamentally different from testing difficulty perspective. Modules that are reused several times, cannot be considered as a separate component each time they are called and need to be tested only once. This means that reused modules should only add to the cyclomatic complexity once, not as many times as they are used (called). However, this may not be the case if we consider cyclomatic complexity in the context of a general software complexity measure (or cognitive software complexity measure), since multiple calls to the same subroutine may actually increase general software complexity.

Even if we observe cyclomatic complexity only as a testing difficulty measure (not as general or cognitive complexity measure) it may be calculated in several different ways. McCabe's paper [48] considers each component (subroutine) of the program separately. As a consequence, each of those components needs to be strongly connected (for each subroutine an artificial edge from end to start node has to be added). On contrary to McCabe's cyclomatic complexity which is focused on module (or unit) testing, Henderson-Sellers metric [29], [66] is focused on the integration test paths. This means it is interested in counting basic test paths of one big control flow graph and not many smaller ones. The merging of modules into one big control flow graph is achieved by the Henderson-Sellers splitting technique.

The graph on figure 2 is equal to the single graph from figure 3 when this technique is applied: each node calling a subroutine is replaced by 2 nodes (one for the entry in a subroutine and one for the exit) and 2 new edges

Figure 2: Control flow graph as set of separate connected components (source [29])

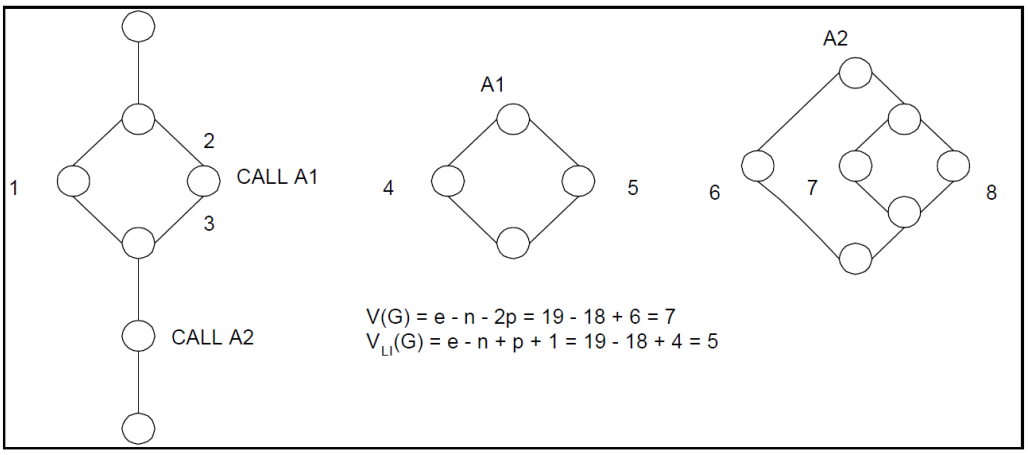
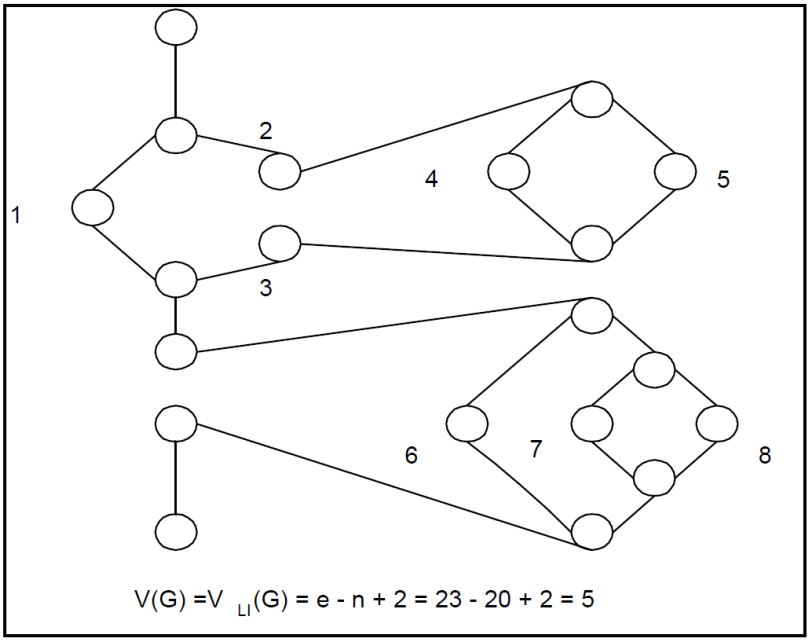


Figure 3: Control flow graph as single integrated connected graph obtained by splitting technique of Henderson-Sellers (source [29])



are added (one for the entry and one for the exit from a subroutine). Now we can apply the well-known McCabe's formula for calculating cyclomatic complexity of a single-component control flow graph : $V(g) = e - n + 2$. The merging increases cyclomatic complexity by one (1=2-1) **for each connected component except for the one which we are merging into**.

$$V_{LI}(G) = e' - n' + 2 = [e + 2(p-1)] - [n + (p-1)] + 2 = e - n + p + 1$$

where e' and n' respectively represent the number of edges and nodes in a single, merged, control flow graph. This altered cyclomatic complexity metric has different properties with regard to modularization and is much more suitable for integration testing. The value of $V_{LI}(G)$ for the full program is equal to the total number of decisions, D, plus one:

$$\sum_{i=1}^{p} d_i + 1 = D + 1$$

The value of $V_{LI}(G)$ is unchanged when subroutines are merged back into the program either by nesting or sequence (see figures 2 and 3). This confirms the argument that the integration testing procedures are unchanged by modularization.

Relationship between the whole and the sum of the parts is also different. McCabe (1976) shows that $V(G) = \sum V(G_i)$ while Henderson-Sellers deduce that:

$$V_{LI}(G) = \sum V_{LI}(G_i) - (p-1) = \sum_{i=1}^{p}(e_i - n_i + 2) + 1 - p = e - n + p + 1$$

where e and n are the total number of edges and nodes, respectively. To summarize, McCabe (V(G)) treats modules in programs essentially independently, while Henderson-Sellers ($V_{LI}(G)$) retain an interpretation with respect to testing paths both at the unit level and at the complete program level.

Note that the last equation disregards modularization due code reuse by counting only the number of components and not the number of times they are called.

Multiple calls of a single subroutine do not add to the value of $V_{LI}(G)$ since it does not introduce additional control flow paths. This is aligned with the premise that testing difficulty should not be affected by multiple calls to same subroutine. It should be noted that this cannot be extrapolated to cognitive complexity or understandability: multiple calls for same subroutine may make programs harder to understand and therefore more cognitively complex.

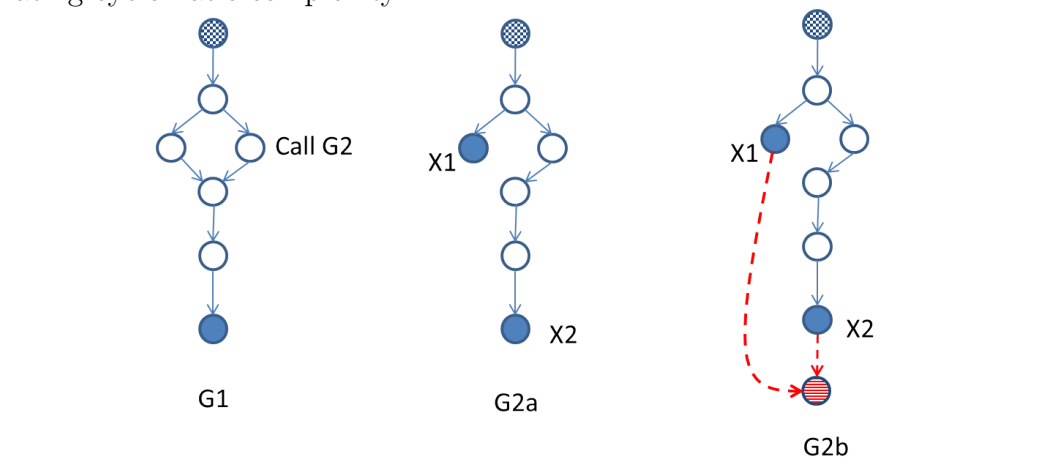
2.1.2 Cyclomatic complexity for modules with multiple entry and/or exit nodes

One of the main assumptions in calculation of cyclomatic complexity is that each component has a single entry and a single exit node. Here we will

consider cases where this assumption is not fulfilled and observe its effect on cyclomatic complexity.

Multiple entries in a module represent module reuse mechanism. Thus, equation $V_{LI}(G) = \sum V_{LI}(G_i) + 1 - p$ applies to *multiple entry, single exit* (MESE) modules as well.

Figure 4: Handling *single entry, multiple exit* (SEME) modules when calculating cyclomatic complexity.



Generally, *single entry multiple exit* (SEME) nodes typically occur in branches of *if* structures where an immediate return ("early exit") to the calling module occurs. One way to handle such cases is to add an additional (virtual) node for each early end and connect it with virtual edges to early and normal exit nodes. This means we need to add one for each exit point (+1 = 2 new edges - 1 new node). So the formula is given with:

$$V_{LIseme}(G) = e - n + p + 1 + \sum_{j=1}^{p-1}(r_j - 1)$$

where r_j is the number of exit points in the graph representation of *j-th* module and there are *p-1* such modules (subroutines). This means that modules with multiple exit points increase complexity.

In addition, *multiple entry, multiple exit* (MEME) modules can be treated as modules with multiple exit points, and the previous equation applies.

2.1.3 A critique of cyclomatic complexity as a software metric

McCabe's cyclomatic complexity is based on solid theoretical foundations and is useful as a measure of software testing difficulty, but it can also be considered as a general purpose software complexity measure or metric of cognitive software complexity. In that case, cyclomatic complexity may be criticised on several grounds [70].

First, there is an issue of compound predicates. We can choose either to count predicates or individual conditions, but both alternatives may be too simplistic. Myers [54] proposes a complexity interval which will have *number of predicates + 1* as the lower bound and the *number of individual conditions + 1* as the upper bound.

The second issue of cyclomatic complexity is the failure to distinguish between *if* and *if/else* constructs: they have the same cyclomatic complexity because they have the same number of execution paths, the difference is only that the alternative ("else") path is explicitly stated. Third issue is handling switch-case construct. Although it has the same cyclomatic complexity as equivalent *if* construct, it is significantly simpler and should contribute to complexity with logarithmic scale $log_2(n)$.

There is also an additional issue that cyclomatic complexity remains one for any linear sequence of statements.

A more fundamental problem of cyclomatic complexity measure is the fact that **generally accepted techniques for modularization (splitting in subroutines) effectively increases complexity because the number of connected components (p in the equation) increases**. The problem is even more complicated with observation that graph complexity may be reduced when modularization eliminates code duplication (when the same subroutine is called more than once). Cyclomatic complexity is then given with:

$$v(P') = v(P) + i - \sum_{j=1}^{i}((v_j - 1) * (u_j - 1))$$

where P is equivalent to P' but with single connected component, i is number of modules, v_j is complexity of *j-th* module, and u_j is number *j-th* module is called.

This means that the complexity of program increases with modularization, but decreases with factoring out duplicate code. This leads to conclusion

that programs should only me modularized for the parts that are reusable, but this is not acceptable guideline to reduce complexity.

Suleman [33] indicated an additional problem with the way how cyclomatic complexity handles nested control structures. Although they have the same cyclomatic complexity as non nested (sequential) control structures, nested counterparts have greater *computational* complexity. As far as testing is concerned, this works fine, but when it comes to benchmarking the code, this is not acceptable. There are existing solutions for nested *if* problems but those cannot be applied to nested loops. Suleman proposes a solution to this problem by adding total number of iterations of nested loop to cyclomatic complexity:

$$V(G)^* = V(G) + \prod_{i=1}^{n} P_i$$

where n is nesting depth (the number of nested loops) and P represents the number of iterations in each loop. Problem with this approach is that number of iterations does not need to be known in design time, only at runtime. This makes the calculation of computational complexity hard. Also, note that this formula works only if we have a single top-level loop.

In addition to theoretical objections, there are also some empirical objections related to cyclomatic complexity ass general software complexity metric. The number of empirical studies have been carried out with different interpretations: The problem is that there is no explicit hypothesis being evaluated. Possible *a posteriori* hypothesis which can be used to examine empirical work are:

1. Total program cyclomatic complexity can be used to predict various software characteristics (such as development time, incidence of error and program comprehension)

2. Programs with cyclomatic complexity lower than 10 are easier to test and maintain than those where this is not the case (original McCabe's hypothesis)

Empirical data does not a give great deal of support for either of the hypothesis. The clearest empirical result indicates strong relationship between the number of lines of code (LOC) and cyclomatic complexity (CC). This is not very good because one of the motivations for cyclomatic complexity is inadequacy of LOC as complexity metric. However, there are many studies that show that LOC actually outperforms CC.

A lot of empirical validations are done by measuring correlation with Pearson's product moment as a coefficient. That correlation gives a value

between -1 and 1 where 0 indicates no correlation while -1 and 1 indicate strong, negative or positive correlation. However, that coefficient requires roughly normal distribution, which is particularly problematic when correlating cyclomatic complexity and module error rates because it is impossible to get negative error count and corresponding distribution is skewed.

Empirical validation is very often done in two scenarios: large uncontrolled and small controlled ones. Both have issues. Large scale empirical validations have problem of not taking into account the individual ability of single programmer. Small scale empirical validations are usually done on small (trivial) program samples (up to 300 LOC) and do not take into account programmer familiarity with the problem.

One might argue that a search for a general complexity metric based upon program properties can be considered as a futile task, because of range of programmers, programming environments, languages and tasks. A more fruitful approach may be to derive a complexity metric from more abstract notations and concepts of software design. This would have the additional benefit that design metrics would be available earlier in the software development process.

2.2 Halstead metric suite

In addition to McCabe's cyclomatic complexity, another traditional set of metrics, the Halstead metrics [26], are often used. Halstead metrics are an attempt to establishing an empirical science of software development. He realized that software metrics should reflect the implementation of algorithms in different languages, but that they are independent of their execution. Halstead metrics are computed statically from the code [78]. Halstead defines following basic elements:

- **n1**: number of unique or distinct operators appearing in a program
- **n2**: number of unique or distinct operands
- **n=n1+n2** represents vocabulary
- **N1**: total number of operators (implementation)
- **N2**: total number of operands (implementation)
- **N=N1+N2**: represents counted (implementation) length

From this basic element, several metrics is calculated. **Program volume** (V) represents the information content of the program and is measured in

bits:

$$V = N * log_2(n)$$

The program volumene is the actual size of a program in a computer if a uniform binary encoding for the vocabulary is used [69], but Halstead interpreted it as number of *mental comparisons* needed to write a program of length N. Volume metric is based on the assumption that humans use the binary search algorithm in selecting the next token from vocabulary of n symbols.

Program difficulty (D) or error-proneness is based on simple cognitive complexity theory that adding new operators and reusing old operands increases difficulty of algorithm (program) understanding [52]. It is calculated by using the equation:

$$D = (n1/2) * (N2/n2)$$

Program effort is the mental effort required to implement or comprehend the algorithm, measured in *elementary mental discriminations*. For each mental comparison (in volume, V), depending on difficulty (D), the human mind needs to perform several elementary mental discriminations. Therefore, effort is proportional to volume (V) and difficulty (D):

$$E = D * V = \frac{n_1 * N_2 * N}{2n_2} * log_2(n)$$

One of the hypothesis of Halstead is that the length of well structured algorithm (program) is dependant only on the number of unique operators (n_1) and operands (n_2). He gave a formula for **estimating program length** ($\hat{N}$):

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

One of the more controversial Halstead's metrics is **time** (T) required to implement an algorithm (program). It is simply calculated by dividing effort (E) with 18 and result represents the time in seconds. Number 18 in this equation is considered as a Stroud number. This is based on weak and, from current perspective, irrelevant psychological theory made by John Stroud, that human beings are able to detect between 5 and 20 discrete events per second [72].

Since these metrics are applicable to any program or language, it is important to clearly understand the rules for selection between operators and operands. Initially Halstead metrics ignored variable declaration section from calculation and were only applied to algorithms (not programs). Algorithms were usually written in Algol and Fortran [69], so the distinction was rather simple, but with more complex languages this may not be as simple as it seems.

2.2.1 Criticism and empirical validations of Halstead metrics

Although the theory behind the Halstead's metrics is weak, the metrics have been validated through a number of empirical studies. The first one, conducted by Halstead himself [26], confirmed the validity of his metrics. However, those validations have been criticized on the following grounds:

- The sample size was too small and parametric statistic (Pearson correlation coefficient) was used. Parametric statistics requires normal distributions and it is very hard to show that data set below 30 samples follows normal distribution. Halstead frequently used sample size less the 10.
- Most of the programs were very small, less then 50 lines of code
- Many experiments, especially ones involving time, contained only single subjects. Unless the subject programmer was perfectly typical, we cannot generalize the conclusion
- Subjects involved in experiments were mainly students and metrics may not apply to professional programmers.

Shen [69] has shown that the length equation cannot be justified theoretically in the manner proposed by Halstead. However, there are empirical evidences to suggest its validity, although it appears to work best in the range of N between 2000 and 4000. In addition, published data do seem to sustain the usefulness of the difficulty metric (D) as a measure of error-proneness. Results also suggest that the software science effort (E) is at least as good an effort measure as most others being used.

Basili [5] reports significant correlations between real and calculated effort measured on large, real world software project written in FORTRAN. They have analysed ground support software for satellites ranging from 50000 to 110000 LOC and having 200-500 modules.

Smith [71] summarizes findings of analysis of several large IBM products. Analysis focuses on Halstead's length (N) and volume (V) metrics in comparison to the traditional lines-of-code measure. It is shown that the size of a program can be estimated with reasonable accuracy once the vocabulary is known.

2.3 Data and information flow complexity metrics

Henry and Kafura [31] defined a set of software metrics based on the information flow between system components. This set includes metrics for

procedure complexity, module complexity and module coupling. The following information flow types are identified:

1. *Global information flow* between module A and B happens when there is structure D in which module A writes and module B reads

2. *Direct local flow* from module A to B happens if A calls B

3. *Indirect local flow* from A to B happens if B calls A and A returns a value which B later uses or in case when the third module C calls A and B, passing output from A as a parameter into B

Metrics based on information flow are measuring *simplicity of relations between modules*. For single procedure, Henry and Kafura defined **fan-in** and **fan-out** as:

- **Fan-in** of a procedure is the number of local flows into the procedure plus the number of data structures from which A retrieves information

- **Fan-out** of a procedure is the number of local flows from the procedure plus number of data structures which procedure updates

Procedure complexity depends on two aspects: complexity of its internal code and, complexity of procedure's connections with its environment. With the number of lines of code as measure of complexity of its internal code, procedure complexity can be formalized as [31]:

$$length * (fan_{in} * fan_{out})^2$$

Order of complexity	Number of procedures	Number of procedures with changes	Percentage
0	17	2	12
1	38	12	32
2	41	19	46
3	27	19	70
4	26	15	58
5	12	11	92
6	3	2	67
7	1	0	0

Table 1: Relationship between procedure complexity to changes (taken from [31])

Fan-in and fan-out are lifted on the power of 2 because of the **belief** that complexity is more than linearly dependant when it comes to connections to its environment. If the effect of the number of lines of code ($length$) is estimated or ignored, the metric can be calculated even before the implementation phase. This is important in order to achieve the *design-measure-redesign* cycle instead of the usual *design-implement-test-redesign* cycle.

In the previous formula, the product of fan_{in} and fan_{out} represents total number of input-output flow combinations, which is a result of a simplistic assumption that each input of procedure affects each output. Detailed data-flow analysis, similarly to the one typically done in compiler optimizations, may be performed to improve this [13], [28], [61].

Analysis of the formula for procedure and module complexity may lead us to some useful conclusions. Procedures with high fan-in and fan-out numbers have many connections which might indicate that they perform more than one task. In addition, such procedures are considered as stress points where changes have many effects to the environment. Procedure complexities in a module are summed to obtain module complexity. High module complexities typically indicate improper modularizations. High global flows and low or average module complexity indicate poor internal construction of modules. In that case procedures directly access data structures, but there is little communication between procedures. In the case of low global flows and complicated module complexity, it is probable that functional decomposition within module is poor or there is a complicated interface between modules.

An important aspect of any metric is that its calculations is completely automated so it can easily be applied to large scale systems. Henry and Kafura validated their metric set on the source code of the Unix operating system (version 6.0) and found strong correlation (0.94) with occurrence of changes (errors). Additionally, they noticed that most of the modules complexity (85%) comes from three most complex procedures [31].

Unlike Henry and Kafura [31], Oviedo [61] did not assume that every input of procedure affects every output variable. Inspired by compiler optimization techniques, he defined precise *data flow complexity* metric which counted the number of all prior (re)definitions of all locally used variables. Following text explains the metric in little bit more details using concepts:

- *Variable definition* appears as left side expression in assignment statements or as input parameters of a subroutine.

- *Variable reference* typically appear in right side expression of assignment, as part of predicates or is used in subroutine output statement (i.e. "return" statement).

- *Locally available variable (re)definition* is (re)definition of the variable within the block.
- *Locally exposed* variable reference is a variable that is used (referenced) within a block, but is not (re)defined within it.
- Variable definition defined in block k is said to *reach* block i if it has not been (re)defined along the path from block k to block i.
- Variable (re)definition in the block *kills* all previous definitions of that variable that might *reach* the block.

Let V_i be set of locally exposed variables (set of variables used) in block i and R_i be set of variable definitions reaching block i. Note that, for each variable used (referenced) in block i there might be several definitions, depending on the number of possible control paths leading to the block i.

Data flow complexity of block i is defined as the number of (re)definitions reaching block i of all the variables used (i.e. referenced, *locally exposed*) in that block, or formally:

$$DF_i = \sum_{j=1}^{|V_i|} DF(v_j)$$

where $|V_i|$ is the number of variables used in block i and $DF(v_j)$ is number of definitions of variable v_j reaching block i. *Data flow complexity* of program body is then defined as sum of data flow complexities of all the blocks in the program. Note that only inter-block data flows contribute to the data flow complexity. Such definition is closely related to the "all-uses" test data selection criteria which requires that all *definition-reference* pairs are exercised [63].

Oviedo's data flow complexity metric is one of the first complexity metrics that has focused on the complexity of data manipulation within a block of processing code. That makes this metric context dependant, which is a property not typically present among other complexity metrics. In its nature, data flow complexity metric is orthogonal (at least theoretically) to control-flow complexity described by McCabe's cyclomatic complexity. The combination of those two types of metrics is one of the ideas behind cognitive complexity metrics which we will analyse in the next chapter.

2.4 Cognitive and psychological complexity metrics

Since software is the result of human creative activity, cognitive informatics plays an important role in understanding its fundamental characteristics.

[68]. Many traditional complexity metrics, such as McCabe's cyclomatic complexity[48] and especially Halstead software science [26] actually targeted measuring (among others) psychological complexity. Curtis *et.al* [15] gave evidence that Halstead and McCabe software complexity metrics are related to the difficulty programmers experienced in understanding and modifying software. However, correlations are not as high as in Halstead's study [26] in which he verified his theory. Curtis also noticed that complexity of metrics were more highly related to performance of less experienced programmers and that they may not be suited for predicting performance of experienced programmers. Recently, a new approach in measuring psychological (cognitive) software complexity emerged.

Wang [75] stated that cognitive weight of software is difficulty expressed as relative time (effort) for comprehending the given program. Most cognitive metrics are based on *basic control structures* (BCS) which represent a set of essential control flow mechanisms used to build logical software architecture.

- Sequence
- Branching: if-then-else
- Branching: switch-case
- Iteration (loops)
- Embedded component: Function call
- Embedded component: Recursion
- Concurrency: parallel execution
- Concurrency: interrupted execution

Basic control structures are assigned with *cognitive weight* based on the axiom that time required to understand functionality and semantics of given BCS is proportional to the cognitive complexity of BCS. This approach is justified with the fact that relative effort (time or weight) of BCSs is statistically stable, although it can vary from person to person [75].

There is no agreement on cognitive weights of different BCS and they vary from author to author. After performing experimental results on 126 undergraduate and graduate students of software engineering, Wang [75] came with his relative weights. Gruhn [24] summarizes cognitive weights found in literature indicating that current weights for BCSs are not yet satisfying.

Whatever cognitive weights of BCS are, there is an additional problem of their combination when calculating cognitive weight of complete software.

	BCS	Wang's cognitive weight [75]	Shao's cognitive weight [68]
1	Sequence	1	1
2	Branching: if-then-else	3	2
3	Branching: switch-case	4	3
4	Iteration: "for" loop	7	3
5	Iteration: "repeat-until" loop	7	3
6	Iteration: "while" loop	8	3
7	Embedded component: Function call	7	2
8	Embedded component: Recursion	11	3
9	Concurrency: parallel execution	15	4
10	Concurrency: interrupted execution	22	4

Table 2: Comparison of different BCS cognitive weights (taken from [24])

The simplest way is just to sum the cognitive weights of all other control structures in all the software components (methods)[74]. However, basic control structures can be combined in sequence or nested in a tree which may the affect overall cognitive complexity [68]. Taking into account this idea, the following rule can be implied: for BCSs in sequence, simply sum their weights, for nested BCSs, multiply their weights. In the general case, the total cognitive weight of a software component having q linear blocks of code, each of them having m nested layers and each layer having n individual BCS in sequence is defined with [68]:

$$W_c = \sum_{j=1}^{q}[\prod_{k=1}^{m}\sum_{i=1}^{n} W_c(j,k,i)]$$

Although Misra [51] considered that the formula satisfies the calculation of complete cognitive complexity of software, other authors [68] [74] considered it only as the *operational* part of the overall complexity. They indicated a need to consider *architectural* (structural, data) complexity as well.

Initially Shao and Wang[68] defined **cognitive functional size** (CFS) metric of a single method (component) as the product of total weighted cognitive complexity(W_c) and the number of it's inputs (N_i) and outputs

(N_o):

$$S_f = f(W_c, N_i, N_o) = (N_i + N_0) * W_c$$

For a complex component consisting of n methods, cognitive function size is then calculated with:

$$S_{ftotal} = \sum_{j=1}^{n} S_f(j)$$

In a similar manner, the complexity of a system of components is calculated as the sum of complexities of all components. *Unit of cognitive weight* (CWU) used for *cognitive function size* (CFS) is the complexity of a single method having one input/output and consisting of a single sequential operation.

Later, Wang [74] has made modifications to his approach, by simplifying the calculation of operational complexity and extending the architectural part by adding complexity introduced by internal data handling of the methods. Operational complexity is simplified in a way that it does not take into account the nesting of the basic control structures, but instead, more complex control structures now have significantly higher weights (see Table 2). This cognitive complexity metric is simply called ***cognitive complexity*** of system S and is calculated using the following formula:

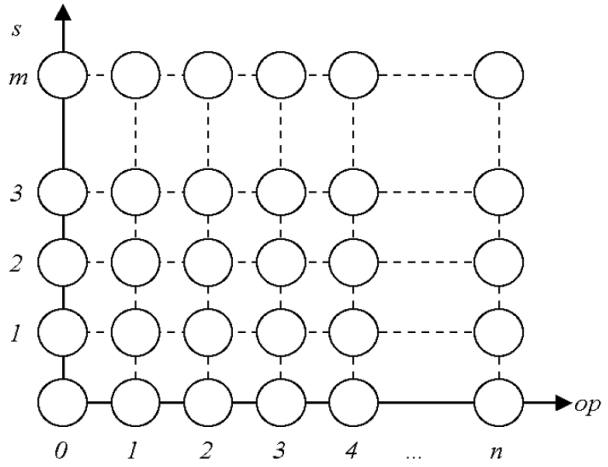
$$C_c(S) = C_{op}(S) * C_a(S) = \sum_{k=1}^{n} \sum_{i=1}^{m} w(k,i) * [\sum_{j=1}^{g} O_{glob} + \sum_{l=1}^{v} O_{loc}]$$

where n is the number of components, m the number of methods, g is the number of global variables and v is the number of local variables used in the system. Wang considered operational and architectural (data) complexities as orthogonal dimensions of the overall software complexity and defined the *semantic space* of programs as a Cartesian product of program variables and all execution steps of a program (see Figure 5). The unit of operational complexity is defined with the simplest *function* of the system, one sequential operation, indicated with *F*. The unit of architectural complexity is considered one *object* onto which function is applied. Consequently, the unit of cognitive complexity is one *function-object* (FO) which is complexity of simplest system that performs single function onto single object.

2.5 Entropy-based complexity metrics

Several authors [1], [27] [40] considered software as an information source and computed software complexity as some variation of classical Shannon ***entropy*** [67] of the source code:

Figure 5: Semantic space of a program (source [74])



$$H_n(p) = -\sum_{k=1}^{n} p_k log_2 p_k = \sum_{k=1}^{n} p_k log_2(1/p_k)$$

where $p_k \geq 0 (k = 0, 1, 2, 3...n)$ and $\sum_{k=1}^{n} p_k = 1$, n is the number of symbols (events), and p_k is the probability of *k-th* symbol. The main issue with this entropy complexity metrics is the selection of symbols. Once defined, their probabilities are calculated by counting their appearance in the source code. Abd-El-Hafiz [1] counted only function/method calls, while Harrison [27] additionally counted reserved words and special symbols.

Regarding the validation of entropy-based metrics, some authors reported [1] meaningful and intuitive results which correspond to subjective measure of source code complexity. Others [27] took a more rigorous approach and performed empirical validation on real industrial software which indicated significant correlation (0.92 and 0.73 in repeated experiment) between entropy complexity metric and average error-span (average number of tokens per error).

Kim et. al. [40] took a somewhat different approach. They constructed intra- and inter- class dependency graphs built from object oriented software and used entropy-based approach for calculating complexity. For each class, they defined a *Data and Function Relationship* (DFR) graph that represents dependencies between data and function members of a class and between function members as well. They represent each data and function member as a node, using weights on arcs to denote the number of times a data member is read or written, or, in the case of a function, how many times it has been

called or how many times it calls other functions. The symbols used in entropy calculations are nodes in the graph. The probability of each symbol (node) required for entropy function is calculated as the division of the sum of node weights (on all incoming and outgoing arcs) and duplicated sum of all weights in the graph (because each weight is counted twice, for originating and terminating node)

Figure 6: DFR graph of example source code(source [40])

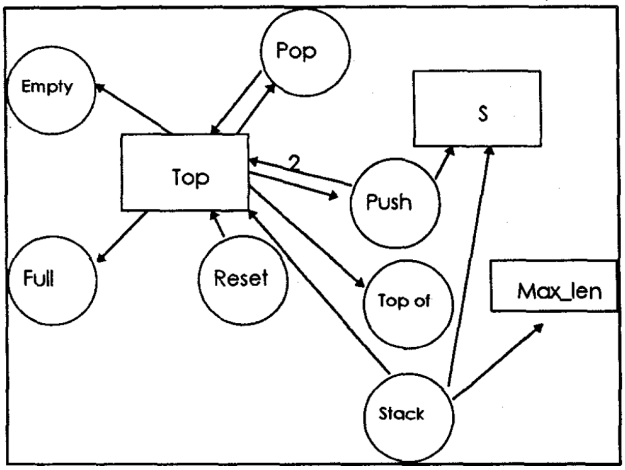


Figure 7: Example source code(source [40])

```
class array {
    char s[max_len]; int i;
    public:
    void reset() {i=0;}
    void insert(char c){s[i]=c; i++;}
    char last_delete(){return (s[i--]);}
    char view{return(s[i]);}
    boolean empty(){return(i<0);}
    boolean full(){return(i==max_len-1);}
};
```

In similar way, the inter-object graph, called *Object Relationship* (OR), is constructed, with nodes being classes, arcs being messages between them, and weights being the number of times a given message is called from the originating class. Weights in that graph are then used to calculate intra-object entropy representing intra-class complexity of software. Total complexity of software is then calculated as the sum of intra- and inter-object entropy of all the classes in the software.

Kim verified the metrics theoretically against Weyuker's properties [77] resulting in only 2 of them (5 and 7) not being satisfied. Regarding empirical validation, authors report significant correlation (0.9047 and 0.8750 respectively) between (intra-)class complexity and *Weighted Methods per Class* (WMC) and *Lack of Cohesion In Methods* (LCOM) [12]. For inter-class complexity, they measured correlation with *Response For a Class* (RFC) and *Coupling Between Objects* (CBO)[12] also resulting in significant correlations (0.8735 and 0.8950, respectively).

2.6 Theoretical evaluation of software complexity metrics

Over the last few decades, plenty of software complexity metrics have been proposed. As s consequence, the question of their validation emerged.

Kearney et. al. [37] indicated several problems with software complexity metrics and proposed some guidelines for their creation and validation. First, they indicated that a metric should always be made with clear and specific goal and anticipated usage in mind. Before a metric is developed, a specification of what will be measured, and why, should be stated. A relation between practical usage and desired metric goal of a metric should be confirmed by underlying theory. For example, if index of program comprehension is wanted, more details about the human understanding of programs should be studied. If index of errors is wanted, causes of errors must be determined. In addition, authors state that a metric should not only index the level of complexity but also suggest a method for its reduction. A norm that identifies acceptable complexity level should also be specified.

An empirical validation presents a natural way to validate the new metric, however, designing and performing quality experiment is not a trivial task. Problems with metric experiment validation is primarily the large number of factors that can influence the outcome of an experiment. These typically include subject, language and task selection problems. The danger of experiments is that the results are easily misinterpreted. In case the large number of experimental conditions is examined, the likelihood of finding accidental relationship is high. As a consequence, type 1 errors happen: inferring the existence of non-existent relationship. Even if high correlation is found, without understanding underlying process that lead to the relations, it is difficult to know how to use the result to an advantage.

In order to perform the initial metric assessment, researchers usually use theoretical evaluations, which are much more practical and easier to perform. One of the most popular and widely used theoretical evaluations of software

complexity metrics is the one proposed by Weyuker [77].

She defined 9 evaluation criteria for software complexity and evaluated 4 most important complexity metric (at the time) using those criteria: lines of code (LOC), Halstead effort [26] , McCabe cyclomatic complexity [48] and Oviedo's data flow complexity [61]. Those 9 desirable properties of software complexity metric are:

- **Property 1 (Nonuniformity)**: Metric should not assign same complexity to all programs because then it is not a metric at all

$$(\exists P)(\exists Q)(|P| \neq |Q|$$

- **Property 2 (Fine granularity)**: Complexity metric is not "sensitive" enough if it categorizes programs in only few complexity values. There should be only finitely programs with complexity c, where c is non-negative number.

- **Property 3 (Coarse granularity)**: Exactly the opposite of second property, a metric should not be of too fine granularity. There should exist two distinct programs with same complexity

$$(\exists P)(\exists Q)(|P| = |Q|)$$

- **Property 4 (Implementation dependence)**: Complexity of a software metric should measure complexity of *implementation*, not the functionality performed by the software. This means that the complexity metric should depend on syntactic features of program not its semantic. In another words, equivalent programs (performing same functionality) do not necessarily have same complexity.

$$(\exists P)(\exists Q)(P \equiv Q and |P| \neq |Q|)$$

- **Property 5 (Monotonicity)**: Components (parts) of a program cannot be more complex then the whole program. This is one of the central properties of any syntactic software complexity metric.

$$(\forall P)(\forall Q)(|P| \leq |P;Q| and |Q| \leq |P;Q|)$$

- **Property 6 (Interaction sensitivity, context dependence)**: Two components of a program result in different complexities depending on how they interact with each other. This metric is frequently not satisfied by complexity metrics.

$$(\exists P)(\exists Q)(\exists R)(|P| = |Q| and |P;R| \neq |Q;R|)$$

- **Property 7 (Permutation sensitivity)**: Two programs which have same statements but in different order should not necessarily have same complexity. Order of statements should affect complexity.

- **Property 8 (Renaming)**: If program P is "renaming" of Q then they have same complexity. Naming conventions used in program should not affect its complexity. Weyuker explicitly stated that this rule does not apply for metrics evaluating cognitive (psychological) complexity.

- **Property 9 (Increasing growth)**: There should be cases when concatenation of two components is more complex then sum of complexities of components individually. This property is stronger alteration of property 5. The metric should allow that potential interactions between programs might add to overall complexity of their composition:

$$(\exists P)(\exists Q)(|P| + |Q| < |P; Q|)$$

- **Property 9b (stronger version)**: Concatenation of two components should always (not only in some cases) be more complex then sum of complexities of components individually.

$$(\forall P)(\forall Q)(|P| + |Q| \leq |P; Q|)$$

Property number	LOC	McCabe [48]	Halstead Effort [26]	Oviedo Data Flow [61]
1	YES	YES	YES	YES
2	YES	NO	YES	YES
3	YES	YES	YES	YES
4	YES	YES	YES	YES
5	YES	YES	NO	NO
6	NO	NO	YES	YES
7	NO	NO	NO	YES
8	YES	YES	YES	YES
9	NO	NO	YES	YES

Table 3: Comparison of properties for 4 well known complexity metrics([77])

Although widely used for theoretical metric validations, Weyuker's evaluation properties are not without criticism. Cherniavsky and Smith [11] constructed a metric which satisfies all 9 Weyukers complexity properties but should should be viewed sceptically as metric any type of complexity.

They stress that Weyuker's properties present necessary but not sufficient condition for software metrics and should be taken carefully and intelligently. They also indicate that Weyuker properties apply only on traditional programming languages and that new (or adapted) set of rules should be found for object oriented languages.

Maneely et.al. [50] gave summary of metric validation criteria found in literature and provide practical guide for researchers working on new software metrics. They found total 47 unique validation criteria and performed analysis to explore relationships among them. They also identified 11 possible advantages of the validation metrics and associated validation criteria to their advantages. Starting with clear use of the metric, a researcher can decide which advantages are relevanr, and then choose validation criteria for the new metric.

2.7 Empirical evaluations of software complexity metrics

2.7.1 Dimensionality of software metrics

Many authors noticed relatively high correlations between software complexity metrics. This rose the question of their interdependence and dimensionality of software complexity. One of the first studies on this topic is performed by Henry et.al [32]. They studied relationship (correlation) between three metrics (McCabe cyclomatic complexity [48], Halstead's effort [26] and Henry and Kafura's information flow [31]) on 165 methods in UNIX operating system. They also studied correlation of those metrics to errors.

The study showed that Halstead and McCabe metrics are highly correlated (0.84) to each other but relatively weakly to the information flow (0.38 and 0.35 respectively) which is therefore considered as an orthogonal metric. Regarding correlations to the error count, all metrics showed high correlations, information flow having 0.95, McCabe 0.96 and Halstead effort 0.89.

Considering obvious orthogonality of the metrics, authors also proposed usage of McCabe and Halstead metrics on intra-module level, while using information flow on inter-module level

Another study on dimensionality among metrics is performed by Munson and Khoshgoftaar [53] [39] also noticed high correlations between software complexity metrics which suggests existence of hidden common dimensions to those metrics. In order to find those dimensions, they performed *factor analysis* of five empirical studies dealing with software complexity metrics [42], [17], [34], [64], [30].

Figure 8: Factors imputed from five different studies ([53])

Associated Metric	Imputed Factors				
	Control	**Volume**	**Action**	**Modularity**	**Effort**
Data	***				
Diff	***				
Ueff	***				
Blocks	***				
Cond	***				
Band	***				
CL	***				
Scope	***				
Nodes	***				
Edges	***				
Knot	***				
V(G)	***				
η	***	***			
η_2	***	***			
Size	***	***			
Stmnts	***	***	***		
I		***			
N_2		***			
N_1		***	***		
N		***	***		
Lines		***	***		
$\hat{N}$		***			
N_F		***			
V		***			
Inlines		***			
Iden			***		
Nest				***	
Mnest				***	
Depth				***	
Inflow				***	
Call				***	
St10				***	
η_1			***	***	
$\hat{E}$	***				***
$\hat{\hat{E}}$					***

Five studies analysed are completely independent of each other and no direct statistical inference can be made. However, although true underlying dimensions cannot be determined directly, results indicate that software complexity domain is not unrestricted and that there are underlying dimensions that can generally be used to describe complexity of any software. Those imputed *orthogonal* factors are **Control, Volume, Action, Effort and Modularity**. The **control** dimension is related to metrics measuring control flow complexity or difficulty. Unifying theme for metrics in the **action** group is unique and total number of operands (Halstead n_2 and N_2). **Volume** dimension is typically associated with metrics depending on size or item count perspective of complexity. Mental **effort** dimension grouped metrics that depend on multiplicative effect of interaction of program operators and operands. The **modularity** dimension is mostly affected by number of *call* statements indicating level software is modularized. Identification of those factors actually reduced dimensionality of software complexity where each of complexity metric maps onto at least one factor dimension. Existence of complexity dimensions, clearly indicates that it *is* possible to construct single measure of absolute complexity of software systems that can be used for comparison of any two programs.

Figure 9: Correlation between 18 metrics in FORTRAN programs

CORRELATION COEFFICIENTS AMONG 18 SELECTED METRICS

	STMTS	LN-CM	NODES	EDGES	McCBE	SCOPE	n2	N1	N2	n	N	N^	V	IC	E^	E^^	CL
STMTS																	
LN-CM	.983																
NODES	.924	.906															
EDGES	.914	.875	.982														
McCBE	.908	.891	.964	.971													
SCOPE	.848	.797	.910	.947	.892												
n2	.898	.877	.896	.889	.872	.826											
N1	.977	.971	.916	.898	.905	.833	.925										
N2	.942	.933	.917	.903	.915	.828	.953	.976									
n	.907	.893	.920	.899	.886	.832	.987	.933	.950								
N	.968	.960	.921	.906	.915	.836	.943	.996	.992	.946							
N^	.896	.878	.913	.898	.881	.837	.989	.925	.947	.998	.940						
V	.960	.949	.927	.914	.918	.852	.956	.990	.992	.959	.997	.958					
IC	.865	.834	.810	.824	.796	.780	.956	.882	.891	.907	.891	.912	.900				
E^	.914	.913	.905	.873	.897	.805	.845	.940	.937	.884	.944	.880	.947	.728			
E^^	.886	.882	.917	.881	.892	.813	.887	.914	.925	.931	.924	.931	.938	.748	.976		
CL	.878	.830	.930	.978	.969	.932	.851	.862	.872	.848	.872	.850	.880	.803	.830	.828	
KNOT2	.871	.830	.919	.948	.923	.877	.855	.861	.872	.848	.871	.845	.873	.815	.803	.799	.943

Li [42] performed empirical study on 31 metrics which are categorized in 3 different types: metrics that measure size (volume) of the software, metrics that measure usage, visibility and interaction of data within a program

and metric that measure control organization within that program. Experiment was applied to 255 student assignment FORTRAN programs. Results confirm high correlation between all metrics. In general, measures based on program size have been the most successful in predicting maintenance costs. Many volume metrics have similar performance, while some control metrics surprisingly correlate well with typical volume metrics.Li indicated incomletness of most of these metrics and proposed a flexible class of metrics combining volume and control.

In addition to volume, control and data complexity, Banker [80] indicated **modularity** as independent dimension of software complexity. He also stressed the fact that most of the metrics are affected by program size which calls for metrics that are normalized for size. Banker assumed organization of software system as collection of programs which are then divided into modules (sub-programs). He proposed five metrics for such software system: **Program size** as average number of LOC; **modularity** measured at module level as an average number of statements per sub-program, **decision structure complexity** measured at statement level as proportion of *IF* statements in program, and program **decomposability** measure at statement level by the portion of GOTO statements within a program. Additional fifth metric is also introduced to measure total size of application system.

2.7.2 Effect of software complexity on software maintenance process

Several research studies focused on the relation between software complexity and software maintenance process. There are several underlying theories used to justify this relation. Each of those theories is addressing complexity effect on maintenance effort in different phase of software development process. The first hypothesis assumes effect of complexity on correctness of the initial implementation. The second one assumes effect of complexity on testing difficulty. The third hypothesis accounts the effect of complexity on cognitive effort required for understanding the software during maintenance process.

Gill [38] indicated a need to improve software maintenance process because 40-75% of all software efforts are spent on maintenance[73]. He used modified McCabe cyclomatic complexity metric which was normalized by size (measured in number of lines of code) obtaining *cyclomatic complexity density*. The main research question was whether or not cyclomatic complexity effect maintainable productivity and empirical test was performed. Although data set was small (only 7) results indicated that cyclomatic complexity density can be used as predictor of software maintainability.

Curtis [15] compared effect of three complexity metrics (Halstead effort [26] , McCabe [48] and length) to programmer performance on two software maintenance tasks. *Understanding task* required students to recall the statements of the algorithm. The number of correctly recalled statements is counted as measure. The second maintenance task was to accurately *implement modification to the algorithm*. Both, the accuracy and the time required to accomplish the task is measured. In the first experiment, a correlation between the percent of statement correctly recalled and complexity metric is found. All correlations are found to be negative. Length and *McCabe* are moderately correlated (-0.61 and -0.55 respectively) with performance while Halstead effort has little relationship with performance (-0.36). In the second experiment, a correlation between the effort, cyclomatic complexity and length metric and accuracy of modification was relatively low (-0.34, -0.35 and -0.37 respectively). Correlation between the metrics and the time required to implement the modification was somewhat higher but still not significant (0.47, 0.55 and 0.55 respectively). This study provided the evidence that software complexity metrics are related to the difficulty programmers experience in understanding and modifying software, but correlations are not as high as in the initial Halstead's study [26]. Curtis also stressed that complexity metrics were more highly related to performance of less experienced programmers, and that they cannot be applied for estimating performance of professional programmers.

A more recent study [79] used complexity based method for predicting defect-prone components. Three code-level complexity measures were used as input (McCabe cyclomatic complexity [48], Halstead effort [26] and LOC) of 12 different classification models in order to classify component as defective or not. Results of the study indicate that static code complexity measures can be useful indicators of component quality. Another benefit of the study is that the number of metrics required is much less than number required by other defect prediction models. Prediction model is tested on NASA dataset with satisfactory accuracy. Authors also indicate that same methodology cannot be applied on functions because only 4.4% of total functions is found to be defective. This imbalance in class distribution makes classification learning hard.

2.7.3 Practical applicability of metrics

De Silva et. al. [**silva2012**] indicated lack of studies comparing practical applicability of metrics. They compared McCabe cyclomatic complexity [48], Halstead volume metric [26] and Shao and Wang's cognitive functional size [68]. The study is done on 10 freely available Java programs. First, Java

programs are ranked according to their complexity by 30 programmers and then an average rank is calculated for each of the programs. Then, all three metrics are calculated for each of the programs and a correlation is calculated between expert's ranking and ranking according to the metrics. Results show that McCabe complexity has lowest correlation of 0.752, Halstead metric has correlation of 0.851 and Sho and Wang's CFS metric has highest correlation of 0.863. They conclude that Cognitive functional size (CFS) is the most suitable complexity metric to be used in real world.

In his later study, De Silva concentrated on cognitive complexity metrics[16]. In that study 3 cognitive complexity metrics are compared: Cognitive Functional Size (CFS) [68], Cognitive Information Complexity Measure (CICM) [41] and Cognitive Weight Complexity Measure (CWCM) [51]. Similarly as in previous study De Silva conducted study of 10 freely available Java programs and used 30 experienced programmers to rank them by complexity. Their ranking is then compared with rankings calculated using all 3 cognitive complexities. Results show that CFS has greatest correlation of 0.903 indicating it as the best suitable cognitive complexity measure. The two other cognitive metrics used showed significantly lower correlation to expert's ranking (CWCM 0.503, CICM 0.588).
Correlation of 3 cognitive complexity metrics with experts ranking:

			Expert Rankings	CICM	CFS	CWCM
Spearman's rho	Expert Rankings	Correlation Coefficient	1.000	.588	.903**	.503
		Sig. (2-tailed)	.	.074	.000	.138
		N	10	10	10	10

3 Complexity of UML models

Except processing code used for specifying actions, executable UML models typically use one or more abstraction levels that are expressed visually, using the same or simplified syntax as the traditional UML models. This typically includes class models, state-machine models and component models. The most of the complexity metrics applied to the traditional UML models can be applied to the executable models as well. This chapter will investigate existing class, state and component model metrics.

3.1 Class model complexity metrics

Class models are the most popular and the most widely used UML models. They are used for conceptual modelling as well as for detailed specification of software implementation objects. Class models became popular with object-oriented (OO) programming languages as an appropriate way for visualizing static structure of the OO programs. Object oriented software and class models are closely related in the sense that most object oriented metrics are actually class model metrics and can be applied directly.

3.1.1 Chidamber and Kemerer's metrics

Chidamber and Kemerer [12] have developed a suite of object-oriented metrics. Their key contribution is the theoretical development and empirical validation of a set of metrics that can be applied to OO software. Since the focus of object-oriented software is class design (model), most of these metrics are applied on classes. This implies limitations that these metrics do not capture, possible dynamic behaviour of a system.

The theoretical validation of the proposed metrics is done by using the 9 Weyuker's metric properties [77]. Although not without criticism [11], these properties provide necessary (but not sufficient) conditions for good complexity metrics. Empirical validation and analysis of metrics is done at two sites on source code of large real-world applications written in Smalltalk and C++.

The first metric proposed by Chidamber and Kemerer is ***Weighted Methods Per Class*** (WMC). It is defined as the sum of complexities of all methods in a class. If we consider that all methods have complexity equal to 1 (which is typically done), then WMC is equal to the number of methods. This metric can be used as a predictor of time and effort required to develop and maintain a class. Additionally, a large number of methods in a class indicates that sub-classing of a class will have a large impact on child classes

since they inherit all the methods from the parents. This may also indicate that a class is more application specific which limits its possibility for reuse.

The second metric, ***Depth of Inheritance***(DIR), presents the length (in the number of hops) from the class to the root of the inheritance tree. If there are multiple inheritance involved, the biggest number is taken. Authors find several interesting viewpoints for this metric. First they indicate that deeper a class is in a hierarchy, the greater the number of methods it is likely to inherit, thus making it more complex to predict class behaviour. Also they notice that deeper trees indicate greater design complexity since more methods and classes are involved. Finally, they state that deeper the class is in hierarchy, there is greater potential for reuse of inherited methods.

The third metric, ***Number of Children*** (NOC), presents the number of immediate subclasses of a given class. Since inheritance is form of a reuse, greater the number of children, greater the reuse. Also, if a class has a large number of children, it may be a case of misuse of sub-classing.

The fourth metric, ***Coupling Between Objects*** (CBO) represents the number of other classes the given class is coupled to. A class is coupled to another class, if it uses its (instance) methods or attributes. Excessive coupling of classes prevents reuse and breaks modular design principles. Higher CBO metric of a class indicates sensitivity to other parts of the design which makes maintenance of a class more difficult. Coupling metric may also be used to predict effort required to tests certain parts of the design. The higher the coupling, the more rigorous testing needs to be performed.

The fifth metric, ***Response For a Class*** (RFC) is defined as the number of methods (defined outside of the studied class) that are called from the given class. This metric is a measure of communication between the given class and all other classes. If a large number of methods can be invoked as a result of invocation of a method of a given class, the testing and debugging of a class becomes more complicated because it requires greater level of understanding from the tester. For this reason this metric may be involved in testing effort estimation. Larger RFC values of a class, in general, indicates larger complexity of a class.

The sixth metric, ***Lack of Cohesion in Methods*** (LCOM) is defined as the difference between the count of method pairs whose similarity is zero and the count of method pairs whose similarity is not zero. The similarity of two methods of a class is determined by the number of instance attributes (of that class) used by both methods. The larger the number of similar methods, the more cohesive the class is. Methods in a highly cohesive class operate on the same set of instance attributes. In case there are more similar method pairs than those that are not, LCOM is declared zero. Cohesiveness of methods within a class is desirable. A lack of cohesion in a class indicates that a class

should be split into two or more subclasses because there is disparateness in functionality provided by the class. Also, low cohesion indicates increased complexity, which results in increased likelihood of errors.

Regarding the theoretical evaluation of metrics using Weyuker's properties, most of the properties are satisfied. However, Weyukers's ninth property stating that interaction should increase complexity is not satisfied for any of the metrics. When applied to classes, Weyuker's ninth property states that when two classes are merged into single class, complexity should increase because of possible interactions between two classes. Failing to satisfy this property, metrics actually indicate that merging two classes into a single one could *reduce* complexity. Authors do not find this as an essential problem of OO software design.

3.1.2 Li and Henry's metrics

Li and Henry [43] proposed a set of object-oriented metrics that can be used for estimating maintainability effort. They created a regression model and validated their metrics on two commercial systems showing strong relationship to maintenance effort. They have introduced the following metrics: i) Message Passing Coupling (MPC), ii) Data Abstraction Coupling (DAC) and iii) Number of Local Methods (NOM).

Message Passing Coupling (MPC) metric is defined as the number of send statements defined in the class, where messages may be sent synchronously or asynchronously. Another type of coupling measure they identified is ***Data Abstraction Coupling*** (DAC) which counts the number of attributes within a class that have another class as their attribute. They also introduced the ***Number of Local Methods*** (NOM) as a metric of understandability. Finally, they introduced a new size metric, which counts the number of properties of a class, including attributes and local methods.

Li and Henry analysed effect of size metrics on maintenance effort by creating and comparing the two regression models. The first one included only size metrics, while the other one included all metrics. The conclusion is that the size metrics account for a large portion of variance in the maintenance effort, but prediction can be significantly improved by using all the metrics.

3.1.3 Briand's metrics

Another study on class design metrics was performed by Briand et. al [9]. They measured the coupling between classes in C++ but their metrics can be easily tailored to other OO languages. The metrics distinguish among the class relationships, different types of interactions, and the locus of the impact

of the interaction. The acronyms for the measures indicate what interactions are counted. The first two letters indicate the *type of coupling* which may be coupling to ancestor classes (A), descendants (D), friend classes (F), invert friend coupling (IF) or other (O). The next two letters indicate the *type of interaction* between classes which may be class-attribute (CA) interaction, class-method (CM) interaction or method-method (MM) interaction. The last two letters indicate whether a class in an interaction is using other class (import coupling, IC) or is being used by other classes (export coupling, EC). Hypotheses stated by Briand et.al. in this study are:

- **H1:** The higher the export coupling of class C, the greater the impact of a c change to C to another classes. Many classes depend on the design of the class C and thus there is greater likelihood that a failure is tracked back to the class C.

- **H2:** The higher the import coupling of the class C, the greater the impact of a change on another classes on the class C itself. Thus, class C depends on many other classes and understandability of class C may be more difficult. This makes class C more fault-prone.

- **H3:** Coupling based on friendship between classes [1] is in general likely to increase the likeliehood of fault even more then other types of coupling since friendship violates modularity in OO design.

Authors performed empirical validation showing that some of those metrics can be used as significant predictors of fault detection and that those metrics are complementary to Chidamber and Kemerer's metrics [12]. The two friend-based coupling metrics [2] showed the greatest relation to probability of fault detection which supports the hypothesis 3. Other then friend-based coupling metrics, 3 metrics dealing with import coupling from other classes[3] showed significant relation to probability of fault detection which supports hypothesis 2. Finally, 2 metrics dealing with export coupling to other classes [4] showed to be significantly related to fault-proneness as well, supporting the first hypothesis.

[1] In C++, friend classes have access to private and protected members of the given class

[2] IFMMIC (Inverse Friend, Method-Method, Import Coupling) and FMMEC (Friend, Method-Method, Export Coupling)

[3] OCAIC(Other type, Class-Attribute, Import coupling), OCMIC (Other type, Class-Method, Import Coupling) and OMMIC (Other type, Method-Method, Import Coupling)

[4] OMMEC(Other type, Method-Method, Export Coupling), OCMEC(Other type, Class-Method, Export Coupling)

3.1.4 Genero's metrics

Genero et.al. [23] [20] [21] studied early indicators of UML class model quality in the sense of understandability and modifiability. They performed an empirical study in order to determine which UML class model metrics have the greatest effect and which metrics do not affect the quality of a class model. By their findings, the metric that have the most impact on the maintainability of the class model are (i) the number of methods (NM), (ii) the number of attributes (NA), (iii) the number of generalizations ($NGen$), (iv) the number of dependencies ($NDep$), (v) the maximum depth of inheritance tree ($MaxDIT$) and (vi) the maximum height of the aggregation hierarchies ($MaxHAgg$). The metrics that do not have an impact on the maintainability are (i) the number of classes (NC), (ii) the number of associations ($NAssoc$), (iii) the number of aggregation hierarchies ($NAggH$) and, (iv) the number of generalization hierarchies ($NGenH$).

3.1.5 Empirical validations of class model metrics

Except empirical validations usually performed by the authors themselves, several independent empirical studies have been performed to validate different metric sets. Genero et. al. [22] gave a detailed summary about UML class model metrics, their goals and evaluations. The most attention is given to the metrics proposed by Chidamber and Kemerer (CK) [12] and by Briand [9].

Basili et.al. [4] performed an experiment on 8 medium sized systems implemented in C++ in attempt to use CK metrics as predictors of fault-prone classes. They concluded that metric suite is well suited for the task. The only metric that can be considered as insignificant in this context is Chidamber and Kemerer's Lack of Cohesion Methods (LCOM).

Khaled [18] focused at CK and Briand's metric suite and gave cognitive theory that relates object-oriented metrics and fault-proneness. They also claim that the most important metrics to be measured are different types of export and import coupling.

Prechelt et.al. [62] performed two controlled experiments in which they compared the performance on code maintenance tasks for three equivalent programs with 0, 3 and 5 levels of inheritances. The conclusion is that higher inheritance (DIT metric) requires higher understanding effort.

3.2 State machine model complexity metrics

State machine models are typically used to describe discrete event-driven behaviours of the software [55]. Since they focus on behaviour instead on object structure, state machine models are significantly less used in practice and, consequently, much less interesting to researchers.

Syntactically, we differentiate between two main types of state machines, *flat* and *hierarchical*. The main difference is that hierarchical state machines allow existence of *composite states* (sometimes called *superstates*) which are logical collections of sub-states, while flat state machines allow only **simple states** without any sub-states. A state defines a situation in which some invariant condition holds. States are often abstracted with some meaningful names, representing stages in the life-cycle of an entity which they are associated to. A transition represents a path from one state to another. Transitions are usually triggered by some events which represent any observable occurrence. States and transitions may have associated actions specifying processing to be done. There are state *entry-actions* executed when the state in entered, **state action** which is executed while in state and **exit-actions** executed when leaving the state. Transitions may have **guards** associated, which are boolean expressions enabling fine-grained control whether or not a transition will will be actually taken. Such transitions are called **complex transitions**. This chapter presents the state-of-the-art in state machine complexity metrics.

3.2.1 State machine structural complexity metrics

A set of basic state machine metrics identified by Genero[46] and Jose [14] [36] are (i) the number of entry actions (NEntryA), (ii) the number of exit actions (NExitA), (iii) the number of activities (NA), (iv) the number of simple states (NSS), (v) the number of composite states (NCS), (vi) the number of events (NE), (vii) the number of guards (NG), (viii) McCabe's cyclomatic number (CC), (ix) the number of transitions (NT), (x) the number of complex transitions (NCT), (xi) the number of composite states (NCS) and (xii) the number of actions associated to transitions (NIA).

Genero [46] performed theoretical and empirical validation of UML state machine metrics with respect to their capability of being used as understandability indicators. Understandability is identified as a key external quality attribute since it affects several other quality characteristics, among others, maintainability [19]. Theoretical validation is done by using a property based framework proposed by Briand et. al. [8]. Regarding empirical validation, an experiment is performed in which a correlation with understandability

time is calculated for each of the metrics. Results show that NA, NSS, NG and NT are highly correlated with understandability time.

Similar experiment is done by Cruz et.al. [14] who studied the impact of structural complexity on the understandability of state machine diagrams and performed repeated experiments to validate the metrics. As it was clear that there will be a large degree of correlations between structural metrics, Cruz performed principal component analysis (PCA) [35] to find underlying orthogonal dimensions of structural complexity and to reduce the number of metrics. According to their effect, metrics are then grouped into one of the 3 principal components:

- **Simple states features** (SSF) comprising of NSS, NE, NG, NT and CC metrics
- **Activities Within States** (AWS) comprising NEntryA and NExitA metrics
- **Number of Activities** (NA) is dimension represented by single metric of the same name

They also build a regression model that corroborated the hypothesis that structural complexity principal components influence understandability time and efficiency (defined as correctness/understandability time ratio).

3.2.2 Measuring complexity of hierarchical state machines

The study performed by Hall [25] was focused on hierarchical state machines and calculation of their cyclomatic complexity. The first approach presented in this study was to calculate cyclomatic complexity by ignoring superstates [46]. This complexity is called *structural cyclomatic complexity* (SCC). A completely opposite approach is taken when calculating *top-level cyclomatic complexity* (TLCC) of a state machine where only top level composite states are taken into account. Finally, the combination of these approaches (HCC) is possible by adding the cyclomatic complexity of each *layer* in a hierarchy with its own weight, where each layer weight is inversely proportional to the depth of the layer in the hierarchy ($weight = 1/n$, where n represents the depth of state machine layer)

In a study performed by Jose [36], effect of composite (nesting) states on understandability of state machines is analysed. After performing several experiments, he concluded that the level of nesting has a negative effect on understandability and that flat state machines should be preferred over hierarchical ones.

3.2.3 State machine cohesion and coupling metrics

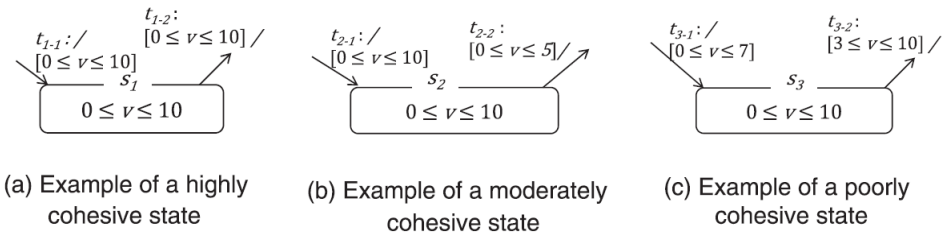
Jung [3] had somewhat different approach to the measurement of state machine understandability. He considered metrics proposed by Genero and Jose [46] [14] [36] merely as structural complexity metrics which have only limited effect on the overall state machine understandability. More precisely, Jung argues that there are situations in which structurally simpler state machines are harder to understand than state machines which are structurally more complex. His research focused on cohesiveness and coupling metrics of state machines by observing the level of "overlapping" between state conditions and pre- and post- conditions of its incoming and outgoing transitions. Jung assumes that the system is modelled with *aggregated* states where each state is defined with a formal condition expressed over system variables. Except state conditions, each outgoing transition is associated with its pre-condition and each incoming transition with its post-condition.

The notion *aggregate granularity states* is first introduced by Binder [7]. He differentiates between primitive and aggregate **granularity levels** of states where **primitive granularity** defines a state as one combination of system variables. This implies that there are as many states as there are combinations of values of variables. Instead using a combination of system variable values, an **aggregate granularity** system state is defined using ranges of variable values that the system has when residing in the given state. The number of states in a system is then determined by the abstraction level used to detect states. The higher the abstraction level, the lower the number of states. **To maintain proper level of abstraction, each state should represent a single semantic in which a user is interested.** That is, each state should have a single semantic and each single semantic should be represented by a single state.

Following this idea, Jung defines the number of semantics within a state as a number of possible partitions we can make in a set of conditions comprising of preconditions, postconditions and state condition. If there is only one possible partition, it means all that pre- and post- conditions are the same as state condition which implies state has a single meaning. **Cohesiveness of a state** (COS) is then defined as reciprocal to the number of semantics of a state, meaning the more semantic a state has, the less cohesive it is.

In addition, Jung defined **similar states** as those having (at least) one *same* incoming and outgoing transition. In this context, *same transition* means, for incoming transition, that they have the same start state and the same associated event. For outgoing transitions, it means that they have the same destination state and the same associated event. If a state has exactly one semantic, the two states split from single state have same incoming and

Figure 10: Example of states with different degree of cohesiveness. Note the difference in pre- and post-conditions (source [3])



(a) Example of a highly cohesive state

(b) Example of a moderately cohesive state

(c) Example of a poorly cohesive state

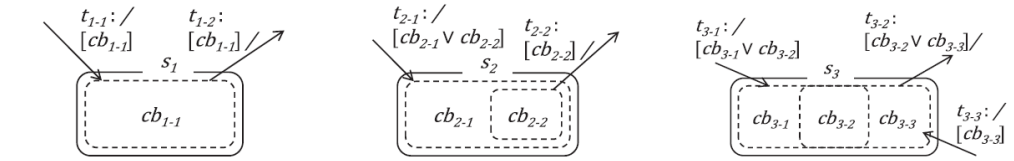
outgoing transition. Similar states represent the same semantics as a result of unnecessary partitioning.

Jung shows that understandability of state diagrams can worsen if states in state diagrams have many similar states. He defined **average cohesiveness of states** (ACOS) of state diagram SD as average cohesiveness value of all states:

$$ACOS(SD) = (\sum_{s} COS(s))/ \mid S \mid$$

Since $COS(s) \leq 1$, ACOS is always between 0 and 1. If it is 1, all states have exactly one semantics.

Figure 11: Example of state partitions (source [3])

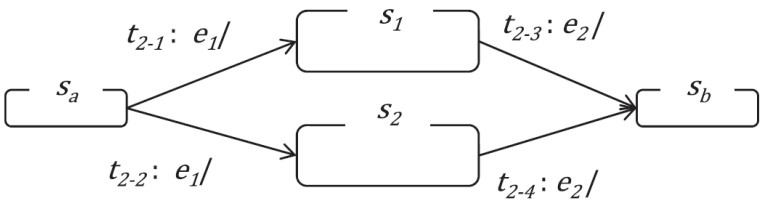


(a) State with one condition block

(b) State with two condition blocks

(c) State with three condition blocks

Figure 12: Example of similar states (source [3])



Another metric proposed by Jung is the **average number of similar states of states** (ASSOS). It is defined as the average number of similar states for all states within state a diagram. ASSOS is equal or greater then zero. If it is zero, then there is no pair of similar states, different semantic is represented with different state. High ASSOS implies there are many similar states that represent same semantics.

$$ASSOS(SD) = (\sum_{s} \mid SS(s) \mid)/ \mid S \mid$$

In his later work [2], Jung introduced and empirically evaluated another metric, **state machine understandability metric**(SUM) which is based on ACOS and ASSOS metrics. This empirical study confirmed that simplicity does not positively affect understandability of state machines.

Problem with Jung's metrics is that it assumes explicit existence of state and transition conditions, which are usually not expressed in formal way. Although it is possible to express conditions and constraints in formal way using **Object Constraint Language** (OCL) [76], this is rarely the case, even in software development methods that use state machines in formal way (i.e. generate code from them) [49] [65]. Usually states are identified as abstractions of life-cycle of entity they describe which implicitly define constraints on variable values.

3.3 Component model complexity metrics

UML standard version [55] defines component rather vaguely. It defines a component "as a modular unit with well-defined interfaces that is replaceable within its environment". A reuse aspect of components is an important one since components should always be considered as an autonomous unit with a system. They have one or more provided and/or required interfaces and its internals are hidden and inaccessible to external entities. Component dependencies are defined only by its required interfaces, resulting in components being rather independent. As a result, components can be flexibly reused and replaced by connecting ("wiring") them together. Autonomy and reuse of components applies at deployment time as well as to design time. This means that implementation of a component should also be possible to deploy and re-deploy independently, for example to update a running system.

There is very little literature about UML component complexity metrics. Because of lack of precise semantics, associated standards and tools, UML component diagrams are rarely used in practice. When they are used, it is usually in informal fashion. However, thigs are about to change in this

field. Formal software development methodologies based on UML such Executable UML [49] and ROOM [65] have already introduced concepts of components and composite structures with rather precise semantics. However, these methodologies are typically used in very specific (mostly real-time) domains and they are far from "mainstream" UML users. This situation will hopefully soon change since beta version of *Precise semantics of UML composite structures* standard [56] is published by Object Managament Group (OMG).

Most of the existing techniques for calculating component complexity assumes availability of the source code, and are dependant on language of implementation. One of the rare studies about component complexity measured on UML models is done by Mahmood [44] [45]. By measuring component complexity from UML specification we can get complexity estimates very early in the software development process. He identified that component complexity depends on **complexity of interfaces**, **constraints** on those interfaces and **interactions** on those interfaces

Interface complexity is based on number of operations (NO) and number of parameters (NP). First we check the NO and NP values for the component and determine component. complexity from the table shown on figure 13.

Figure 13: NO/NP complexity metrics

NO\NP	1 – 19	20 – 50	51 +
1	Low	Low	Average
2 – 5	Low	Average	High
6 +	Average	High	High

After that, manual (or automated [58]) function point analysis is done to categorize interfaces as ILF (Internal Logical File) or ELF (external logical file). Interfaces that have operations that change attributes of other interfaces are considered as ILF, all other are categorized as ELF.

Total interface complexity IC_i of component i is then calculated as sum of ILF_i and ELF_i which are weighted values for component interfaces classified based on their complexity.

Interface constraints are exposed as OCL [76] pre- and post-conditions for each operations. Pre-conditions are assertions that component assumes before operation is invoked and are usually expressed as predicates over operations input parameters. Post-conditions are component guarantees that will hold after operation has been invoked. They are expressed as predicates

Figure 14: Weights for calculation of interface complexity are based on two classification: NO/NP complexity classification and classification based on function point analysis

Data Type	Low	Average	High	Total
ILF*i*	--- x 7 =	--- x 10 =	--- x 15 =	
EIF*i*	--- x 5 =	--- x 7 =	--- x 10 =	

over both, input and output parameters. In addition, operation pre- and postconditions depend on the state maintained by the component. This set of invariants associated with interface is also expressed as *predicates over interface state model* that will always hold. Finally, each component specification has inter-interface conditions that are predicates over the state model of all component interfaces. Mahmood proposed using McCabe's cyclomatic complexity to measure component constraint complexity.

For calculating **complexity of interactions over interfaces**, UML collaboration diagrams are used. There are two factors affecting interaction complexity: **frequency** and **interaction content**.

Each collaboration diagram shows one or more interactions where each interaction shows one possible execution flow. **Interaction frequency** IF_x for operation O_x is defined as ratio of the number of interactions exchanged in that operation, over the total number of interactions exchanged during scenario S_x: $IF_x = Mo/Mi$ where Mo is number of interactions involving operation, and Mi is total number of interactions in given scenario.

Information content is characterized by information received and sent during interaction. Measure of complexity of information content is done by presenting data types used on interfaces as hierarchical directed graph called component *data type graph*. Number of hierarchical levels and number of different data types are two actors affecting information content complexity:

$$CM(W, p) = p + \sum_{i=1}^{n} CM(Y_i, p + 1)$$

where CM is complexity of data structure graph W, p is number of the level where this data type occurs, Y_i is the data type Y of i-th data field in structured data type including n fields.

Overall interaction complexity (CC) is then defined as:

$$CC = \sum_{i=1}^{max} (IF_i * \sum_{j=1}^{n} CM_j)$$

where i is the interface operation for component and j is the number of data types involved in information content exchange for interface operation.

This technique enables designers and developers to measure component complexity by quantitative and objective metrics early in the specification phase. However, problem with this approach is that it assumes existence of OCL-based constraints over interfaces and components as well as interaction diagrams. However, this assumption is frequantly not fulfilled.

4 Introduction to Executable UML modelling

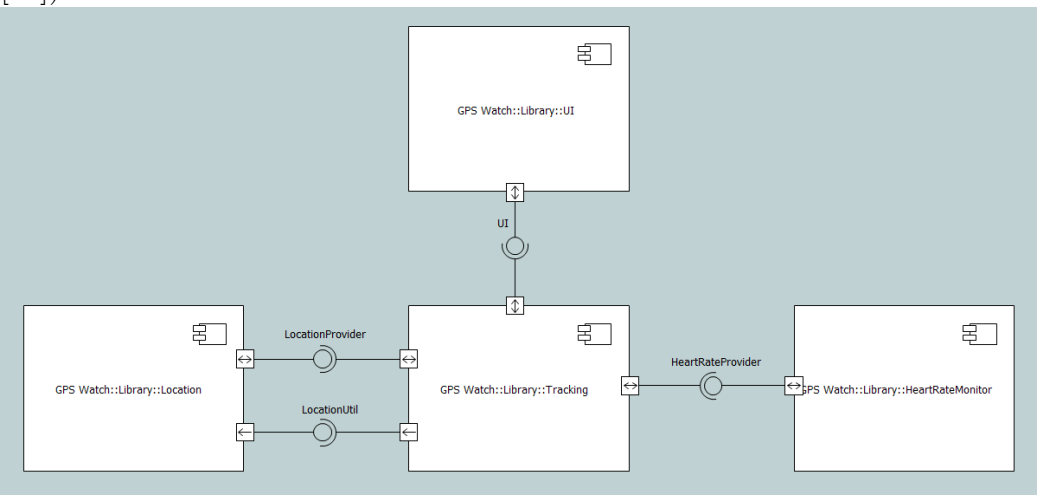
As part of doctoral thesis work, a selection of previously analysed source code and UML model metrics will be adapted and applied to xtUML models. This chapter will give an introduction to the xtUML methodology, as well as brief description of other similar UML methodologies and standards in the field.

4.1 Executable and translatable UML (xtUML)

XtUML originates from Shlaer-Mellor methodology [49]. XtUML stands for *executable* and *translatable* Unified Modelling Language. It uses graphical notation of the standard UML but also defines precise execution and timing rules. That makes it formal software development method but also differs it from standard UML which is merely a graphical notation. Models created by xtUML method are semantically (Turing) complete. This mean that information about software structure, behaviour and processing is semantically integrated in such a way that models can be executed.

XtUML model is the *whole* made from 4 different interconnected models: **component model** defining overall system architecture, **class model** defining concepts and relations within a component, **state machine model** defining class instance life-cycle and **processing model** specifying execution details. First three models are graphical while processing is textual.

Figure 15: Component diagram example taken from BridgePoint xtUML tool [60])

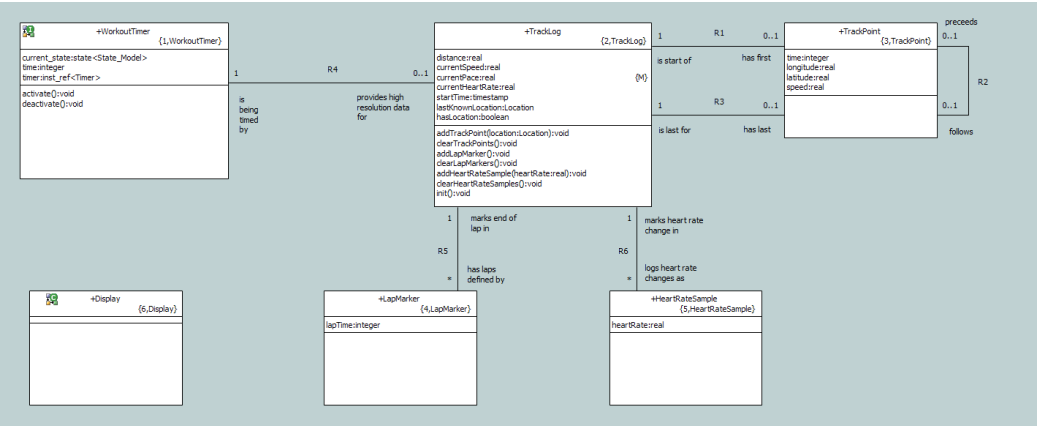


Each component is a considered as black-box from outside and only uses interfaces to communicate with other components. In order to minimize

complex data types on interfaces, each component is usually mapped to single subject matter domain. Good domain separation presents a challenge in xtUML.

XtUML models support synchronous and asynchronous execution and communication between components. Synchronous inter-component communication is enabled with interface operations while asynchronous use interface signals. Synchronous execution within component is realized, among others, using class and instance operations. Classes use state machines and event dispatching mechanism for asynchronous execution.

Figure 16: Class diagram example taken from BridgePoint xtUML tool [60])



Processing (code) can be specified in many places in xtUML model. Component interfaces, from inside perspective have *ports*. Port processing code will be executed when receiver component receives signal or has interface operation invoked. Classes may have operations (instance and class based) which can also contain processing code. In state machines, processing code can be specified in states and transitions so Meally, Moore and hybrid state machines are possible.

Executablity of models is key feature that makes a difference. Executable models are testable, making it possible to debug them, even before any code is generated. Special "virtual machines" exist that are capable to interpret executable model. Executable models also imply existence of *application* and *test* models.

Executable models also enable completeness of generated code. It contains all required information and therefore can be compiled into binary without any "handcoding". Even more, method suggests refraining from any "manual" interventions in generated code since it is considered as only a temporary form of application towards binary form used in exploitation.

Figure 17: State machine diagram example taken from BridgePoint xtUML tool [60])

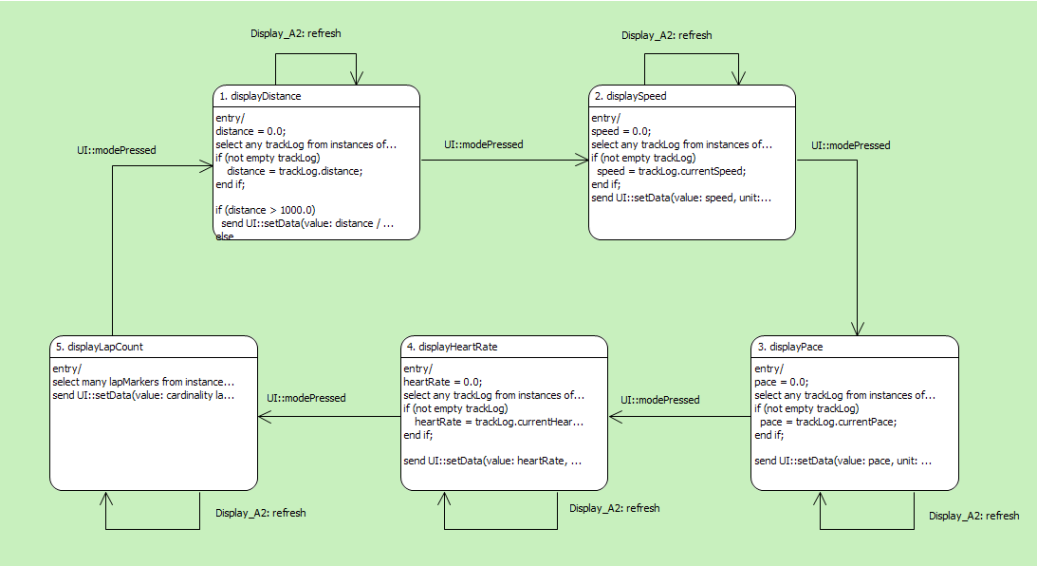


Figure 18: Processing model example taken from BridgePoint xtUML tool [60])

```
distance = 0.0;
select any trackLog from instances of TrackLog;
if (not empty trackLog)
    distance = trackLog.distance;
end if;

if (distance > 1000.0)
  send UI::setData(value: distance / 1000.0, unit: Unit::km);
else
  send UI::setData(value: distance, unit: Unit::meters);
end if;
```

Similar approach most of today's programming languages have towards assembly code. If your C++ code is not working as expected, you will not correct it in assembly, you will do in C++. In rare occasions, if there is a bug in compiler, an interventions will be needed there as well. It is similar in xtUML: bugs will be corrected in xtUML model or, rarely, in model compiler used to generated code from it. This is a reason why xtUML is often considered as higher abstraction software development language.

Traditionally used only in real-time domain and supported by only one proprietary tool [60], this methodology is relatively unknown in the industry.

In attempt to change this, this relatively mature tool became completely open source in November 2014, including its previously protected model interpreter and industry-grade code generators.

4.2 Real-time Object-Oriented Modeling (ROOM) methodology

UML-RT is a UML profile defined by IBM for the design of distributed event-based applications. The profile is based on the Real-Time Object-Oriented Modeling (ROOM) methodology [65] and was implemented in the RSA-RTE [47] product, an extension of IBM's Rational Software Architect (RSA) product.

The basis of UML-RT is the notion of *capsules* that at the same time might have both internal structure (via capsule parts) and behavior (via a state machine). Capsules can be nested, and can communicate synchronously and asynchronously via messages that are sent and received through ports. The types of messages that a port can transmit are defined by protocols. Unlike xtUML components, capsules can be created both statically at design time and dynamically at run-time.

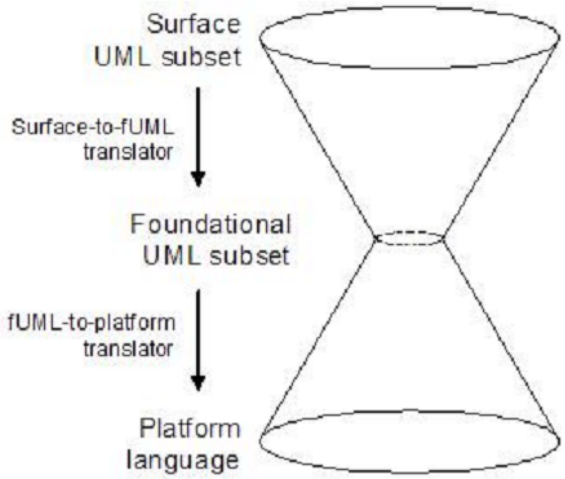
RSA-RTE tool allows several languages to be used to specify actions. This includes C, C++, Java and some other. Model execution is only possible by translation to source code, but no model-level interpreter is possible, at least not in the sense provided by BridgePoint [60] tool.

4.3 Foundational Subset for Executable UML Models (FUML)

Realizing importance of simplicity and clear semantics that are missing from UML standards, OMG defined semantics of a *foundational subset for executable UML models*(fUML)[57]. Main goal of this standard is to act as intermediary language between "surface subsets" of UML used for modeling and computational platform languages used as the target for model execution. FUML is designed to be **compact** in order to facilitate definition of a clear semantics and implementation of execution tools. In addition, it is supposed to be **easily translated** from common surface subsets of UML to fUML and from fUML to common computational platform languages. However, if the feature to be translated from surface UML is excluded from fUML, it is required for the *surface-to-fUML* translator to generate a *coordinated set of fUML elements* that has the same effect as that feature. Then the *fUML-to-platform* translator would need to recognize the pattern generated by the

surface-to-fUML generator, in order to map this back into the desired feature of the target language. Compactness can therefore conflict with ease of translation.

Figure 19: Translation to and from fUML models (source [57])



It is clear that future of UML is in executable modelling which assumes simplicity and clear execution semantics. However, the problem with fUML is that tool support is practically non-existent and it still remains to be seen how it will be accepted by the community.

5 Conclusion and future work

In this paper, we have presented the state-of-the-art in software and UML model complexity metrics, as well as a framework for theoretical evaluation of those metrics.

Software complexity metrics typically measure certain internal software quality attributes, but are usually used to predict some external software quality such as maintainability or understandability. We have also presented results of a series of empirical studies that have explored the correlation between metrics and those external software attributes.

The most frequently used metrics include the size expressed as the number of lines of code, McCabe's cyclomatic complexity, and the Halstead's effort. However, several empirical studies have shown that there are better metrics to be used, for example cognitive metrics which assign cognitive complexity weights to basic control structures. Another interesting group of metrics is the one based on data and information flow, e.g. Oviedo's data flow complexity is particularly interesting since it shows context dependence.

Regarding model complexity metrics, class model metrics are by far the most investigated. Except for typical size and structural complexity metrics, most of these metrics measure some form of coupling between objects.

In the end, we gave an overview of executable UML technologies and standards with a special focus on xtUML methodology. In future work, we will make a selection of metrics analysed in this report, adapt them, and apply to xtUML models. We will implement algorithms for their automated calculation and perform an empirical study in order to evaluate how the distribution of these complexity metrics affects the understandability of xtUML models.

References

[1] S.K Abd-El-Hafiz. "Entropies as measures of software information". In: *Software Maintenance, 2001. Proceedings. IEEE International Conference on*.

[2] Jung Ho Bae, Heung Seok Chae, and Carl K Chang. "A metric towards evaluating understandability of state machines: An empirical study". In: *Information and Software Technology* (2013).

[3] Jung Ho Bae et al. "Semantics Based Cohesion and Coupling Metrics for Evaluating Understandability of State Diagrams". In: *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. 2011.

[4] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *Software Engineering, IEEE Transactions on* (1996).

[5] Victor R Basili and Tsai-Yun Phillips. "Evaluating and comparing software metrics in the software engineering laboratory". In: *ACM SIGMETRICS Performance Evaluation Review* (1981).

[6] VR Basili. "Qualitative software complexity models: A summary". In: *Tutorial on Models and Methods for Software Management and Engineering* (1980).

[7] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. 2000.

[8] Lionel C. Briand, Christian Bunse, and John W. Daly. "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs". In: *Software Engineering, IEEE Transactions on* (2001).

[9] Lionel Briand, Prem Devanbu, and Walcelio Melo. "An investigation into coupling measures for C++". In: *Proceedings of the 19th international conference on Software engineering*. 1997.

[10] Frederick P Brooks Jr. "No silver bullet - Essence and accidents of software engineering". In: (1956).

[11] John C. Cherniavsky and Carl H. Smith. "On Weyuker's axioms for software complexity measures". In: *Software Engineering, IEEE Transactions on* (1991).

[12] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design". In: *Software Engineering, IEEE Transactions on* (1994).

[13] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. "Iterative dataflow analysis, revisited". In: *Proceedings of the PLDI'02* (2002).

[14] José A Cruz-Lemus et al. "The impact of structural complexity on the understandability of UML statechart diagrams". In: *Information Sciences* (2010).

[15] Bill Curtis et al. "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics". In: *Software Engineering, IEEE Transactions on* (1979).

[16] DI De Silva et al. "Applicability of three cognitive complexity metrics". In: *Computer Science & Education (ICCSE), 2013 8th International Conference on*. 2013.

[17] James L Elshoff. "Characteristic program complexity measures". In: *Proceedings of the 7th international conference on Software engineering*. 1984.

[18] Khaled El-Emam. "Object-oriented metrics: A review of theory and practice". In: (2001).

[19] Norman E Fenton and Shari Lawrence Pfleeger. *Software metrics: a rigorous and practical approach*. 1998.

[20] Marcela Genero. "Defining and validating metrics for conceptual models". In: *Computer Science Department* (2002).

[21] Marcela Genero, Mario Piatini, and Esperanza Manso. "Finding" early" indicators of UML class diagrams understandability and modifiability". In: *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. 2004.

[22] Marcela Genero, Mario Piattini, and Coral Calero. "A survey of metrics for UML class diagrams". In: *Journal of object technology* 4.9 (2005), pp. 59–92.

[23] Marcela Genero, Mario Piattini, and Coral Calero. "Early measures for UML class diagrams". In: *L'Objet* (2000).

[24] Volker Gruhn and Ralf Laue. "On experiments for measuring cognitive weights for software control structures". In: *Cognitive Informatics, 6th IEEE International Conference on*. 2007.

[25] Mathew Hall. "Complexity metrics for hierarchical state machines". In: *Search Based Software Engineering*. 2011.

[26] Maurice H Halstead. "Elements of software science". In: (1977).

[27] Warren Harrison. “An entropy-based measure of software complexity”. In: *IEEE Transactions on Software Engineering* (1992).

[28] Matthew S Hecht. *Flow analysis of computer programs*. 1977.

[29] B. Henderson-Sellers and D. Tegarden. “The theoretical extension of two versions of cyclomatic complexity to multiple entrylexit modules”. In: *Software Quality Journal* (1994).

[30] Sallie M Henry and Calvin Selig. “A Metric Tool for Predicting Source Code Quality from a PDL Design”. In: (1987).

[31] Sallie Henry and Dennis Kafura. “Software structure metrics based on information flow”. In: *Software Engineering, IEEE Transactions on* (1981).

[32] Sallie Henry, Dennis Kafura, and Kathy Harris. “On the relationships among three software metrics”. In: *ACM SIGMETRICS Performance Evaluation Review*. 1981.

[33] Suleman Sarwar M.M.; Shahzad S.; Ahmad I. “Cyclomatic complexity: The nesting problem”. In: *Preceedings of 2013 Eighth International Conference on Digital Information Management (ICDIM)*. 2013.

[34] Howard A. Jensen and K Vairavan. “An experimental study of software metrics for real-time software”. In: *Software Engineering, IEEE Transactions on* (1985).

[35] Ian Jolliffe. *Principal component analysis*. 2005.

[36] Mario Piattini José A. Cruz-Lemus Marcela Genero. “Using Controlled Experiments for Validating UML Statechart Diagrams Measures”. In: *Software Process and Product Measurement*. 2008.

[37] Joseph P Kearney et al. “Software complexity measurement”. In: *Communications of the ACM* (1986).

[38] Geoffrey K. Gill; Chris F. Kemerer. “Cyclomatic Complexity Density and Software Maintenance Productivity”. In: *Journal IEEE Transactions on Software Engineering* (1991).

[39] Taghi M. Khoshgoftaar and John C. Munson. “Predicting software development errors using software complexity metrics”. In: *Selected Areas in Communications, IEEE Journal on* (1990).

[40] Kapsu Kim, Yeongil Shin, and Chisu Wu. “Complexity measures for object-oriented program based on the entropy”. In: *Software Engineering Conference, 1995. Proceedings., 1995 Asia Pacific*. 1995.

[41] Dharmender Singh Kushwaha and A. K. Misra. "A Modified Cognitive Information Complexity Measure of Software". In: *SIGSOFT Softw. Eng. Notes* (2006).

[42] Hon Fung Li and William Kwok Cheung. "An empirical study of software metrics". In: *Software Engineering, IEEE Transactions on* (1987).

[43] Wei Li and Sallie Henry. "Object-oriented metrics that predict maintainability". In: *Journal of systems and software* (1993).

[44] R. Mahmood S.; Lai. "Measuring the complexity of a UML component specification". In: *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*. 2005.

[45] Sajjad Mahmood and Richard Lai. "A complexity measure for UML component-based system specification". In: *Software: Practice and Experience* (2008).

[46] Mario Piattini Marcela Genero David Miranda. "Defining Metrics for UML Statechart Diagrams in a Methodological Way". In: *Conceptual Modeling for Novel Application Domains*. 2003.

[47] IBM Mattias Mohlin. *Modeling Real-Time Applications in RSARTE*. 2013. URL: https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/RSARTE%20Concepts.

[48] Thomas J McCabe. "A complexity measure". In: *Software Engineering, IEEE Transactions on* (1976).

[49] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. 2002.

[50] Andrew Meneely, Ben Smith, and Laurie Williams. "Validating software metrics: A spectrum of philosophies". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2012).

[51] Sanjay Misra. "A complexity measure based on cognitive weights". In: *International Journal of Theoretical and Applied Computer Sciences* (2006).

[52] Subhas Chandra Misra and Virendrakumar C Bhavsar. "Measures of software system difficulty". In: (2003).

[53] John C Munson and Taghi M Khoshgoftaar. "The dimensionality of program complexity". In: *Proceedings of the 11th international conference on Software engineering*. 1989.

[54] Glenford J Myers. "An extension to the cyclomatic measure of program complexity". In: *ACM Sigplan Notices* (1977).

[55] OMG Object Management Group. *UML standard version 2.5, In-Process version*. 2012. URL: http://www.omg.org/spec/UML/2.5/Beta1/.

[56] OMG. *Precise Semantics Of UML Composite Structures (PSCS)*. 2014. URL: http://www.omg.org/spec/PSCS/.

[57] OMG. *Semantics Of A Foundational Subset For Executable UML Models (FUML)*. 2013. URL: http://www.omg.org/spec/FUML/.

[58] Object Management Group (OMG). *Automated function Points*. 2014. URL: http://www.omg.org/spec/AFP/1.0/.

[59] Object Management Group (OMG). *Model Driven Architecture (MDA)*. 2014. URL: http://www.omg.org/mda/.

[60] OneFact. *BridgePoint xtUML tool*. 2014. URL: https://www.xtuml.org/download/.

[61] Enrique I Oviedo. "Control flow, data flow, and program complexity". In: *Proceedings of IEEE COMPSAC*. 1980.

[62] Lutz Prechelt et al. *A Controlled Experiment on Inheritance Depth as a Cost Factor for Maintenance*. 2001.

[63] Sandra Rapps and Elaine J. Weyuker. "Data flow analysis techniques for test data selection". In: *Proceeding ICSE '82 Proceedings of the 6th international conference on Software engineering* (1985).

[64] Anne Schroeder. "Integrated program measurement and documentation tools". In: *Proceedings of the 7th international conference on Software engineering*. 1984.

[65] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time object-oriented modeling*. 1994.

[66] B Henderson Sellers. *Modularization and McCabe Cyclomatic Complexity*. 1992.

[67] Claude Elwood Shannon. "A mathematical theory of communication". In: *ACM SIGMOBILE Mobile Computing and Communications Review* (2001).

[68] Jingqiu Shao and Yingxu Wang. "A new measure of software complexity based on cognitive weights". In: *Electrical and Computer Engineering, Canadian Journal of* (2003).

[69] Vincent Yun Shen, Samuel D. Conte, and Hubert E. Dunsmore. "Software science revisited: A critical analysis of the theory and its empirical support". In: *Software Engineering, IEEE Transactions on* (1983).

[70] Martin Shepperd. “A critique of cyclomatic complexity as a software metric”. In: *Software Engineering Journal* (1988).

[71] Charles P Smith. “A software science analysis of programming size”. In: *Proceedings of the ACM 1980 annual conference*. 1980.

[72] John M Stroud. “The fine structure of psychological time.” In: (1956).

[73] Iris Vessey and Ron Weber. “Some Factors Affecting Program Repair Maintenance: An Empirical Study”. In: *Commun. ACM* (1983).

[74] Yingxu Wang. “Cognitive complexity of software and its measurement”. In: *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on*. 2006.

[75] Yingxu Wang. “Psychological experiments on the cognitive complexities of fundamental control structures of software systems”. In: *Cognitive Informatics, 2005.(ICCI 2005). Fourth IEEE Conference on*. 2005.

[76] Jos B Warmer and Anneke G Kleppe. “The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)”. In: (1998).

[77] Elaine J. Weyuker. “Evaluating software complexity measures”. In: *Software Engineering, IEEE Transactions on* (1988).

[78] Wikipedia. *Halstead complexity measures*. URL: http://en.wikipedia.org/wiki/Halstead_complexity_measures.

[79] Hongyu Zhang, Xiuzhen Zhang, and Ming Gu. “Predicting defective software components from code complexity measures”. In: *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*. 2007.

[80] R. D. Banker; S. M. Datar; D. Zweig. “Software complexity and maintainability”. In: *Proceeding ICIS ’89 Proceedings of the tenth international conference on Information Systems*. 1989.