UDC 659.2:004, DOI:10.2298/CSIS090315004D

Metrics for Evaluation of Metaprogram Complexity

Robertas Damaševičius[1] and Vytautas Štuikys[1]

[1]Kaunas University of Technology, Software Engineering Department, Studentų 50, LT-51368, Kaunas, Lithuania
{robertas.damasevicius, vytautas.stuikys}@ktu.lt

Abstract. The concept of complexity is used in many areas of computer science and software engineering. Software complexity metrics can be used to evaluate and compare quality of software development and maintenance processes and their products. Complexity management and measurement is especially important in novel programming technologies and paradigms, such as aspect-oriented programming, generative programming, and metaprogramming, where complex multi-language and multi-aspect program specifications are developed and used. This paper analyzes complexity management and measurement techniques, and proposes five complexity metrics (Relative Kolmogorov Complexity, Metalanguage Richness, Cyclomatic Complexity, Normalized Difficulty, Cognitive Difficulty) for measuring complexity of metaprograms at information, metalanguage, graph, algorithm, and cognitive dimensions.

Keywords: Metaprogramming, complexity evaluation, metaprogram metric.

1. Introduction

Complexity is a difficult concept to define. Though the term "*complexity*" is used in many of 25 roadmaps for software [1] and can, e.g., be found in relation to software development, software metrics, software engineering for safety, reverse engineering, configuration management, empirical studies of software engineering [2], so far there is no exact understanding of what is meant by complexity with various definitions still being proposed. High complexity of a system usually means that we cannot represent it in a short and comprehensive description. L.C. Briand *et al.* [3] state that complexity (of a modular software system) is a system property that depends on the relationships among elements and is not a property of any isolated element. IEEE Std. 610.12:1990 [4] defines software complexity as "*the degree to which a system or component has a design or implementation that is difficult to understand and verify*". Therefore, complexity relates both to comprehension complexity as well as to representation complexity.

The other definition deals with *psychological complexity* (also known *as cognitive complexity*) of programs explaining that "*true meaning of software complexity is the difficulty to maintain, change and understand software*" [5]. There are 3 specific types of psychological complexity that affect programmer ability to comprehend software: *problem complexity*, *system design complexity*, and *procedural complexity* [6]. Problem complexity is a function of the problem domain. It is assumed that complex problem spaces are more difficult for a programmer to comprehend than simple problem spaces.

Knowledge-based perception of software complexity is described in [7] as a process of "*translating*" human-seen complexity into numbers. The process starts with an experiment that involves human beings and provides data with embedded knowledge about human perception of complexity. Data processing and analysis of data models lead to discovery of simple rules which represent human perception of software complexity.

From the organizational viewpoint, complexity of a system is defined with respect to *number, dissimilitude* and *states' variety* of system elements and relationships between them [8]. These complexity variables enable distinction between *structural* (*static*) and *dynamic* complexity. Structural complexity describes the system structure at a defined point in time, and dynamic complexity represents the change of system configuration in time.

How to manage complexity? Many factors influence on better management of complexity. E.g., from the cognitive complexity viewpoint, the major factor is *understandability* [9]. One way to avoid exceeding the cognitive constraints and creating cognitive overload is to reduce the amount of information that needs to be stored in a short-term memory and to decrease the uncertainty of that information [10]. A common method to achieve this would be by creating new and useful *abstractions*. As a program is more than just the informative code during the process of understanding, a programmer's level of expertise in a given domain, i.e., *domain knowledge*, greatly affects program understanding, as well as programmer's knowledge. The commonly recognized principles for managing complexity are *reducing the amount of information, decomposing a system into modules, abstracting or hiding information,* and *providing different levels of abstractions.*

The abstraction level is the level of detail of a software system [11]. In this sense, abstraction is a basic property for understanding the reality and managing complexity of software systems. Abstraction is a gradual increase in the level of representation of a software system, when existing detailed information is replaced with information that emphasizes certain aspects important to the developer while other aspects are hidden. More abstract programming language mechanisms allow to replace complex and repeating low-level operations, and allow to address complex problems with less code and less programming errors.

Software design complexity is also related with design quality. As complexity increases, design quality also tends to decrease. To achieve the levels of quality needed in today's complex software designs, quality must be *designed in*, not tested in. Thus the *design-for-quality* paradigm is becoming extremely important. In this context, M. Keating [12] proposed a simple

software design partitioning rules as a basis for a quantitative measure of complexity: *the number of modules at any level of hierarchy must be 7 +/- 2.*

The complexity growth forces researchers to seek for the adequate means for better management of complexity. A number of techniques has been identified and followed in software design practice that enforces higher program comprehensibility reuse and eases complexity management. These include various lexical conventions, design style conventions and design process conventions. The primary tasks are understanding of the complexity problem and finding of relevant measures for evaluating software complexity. These issues may have a direct influence on testability, performance, efficiency and other characteristics of software systems to be designed.

Software metrics have always been strongly related to the programming paradigm used by the respective researchers. E.g., McCabe's Cyclomatic Complexity [13] was proposed for measuring the testing efforts of structural programs. For object-oriented programs, complexity metrics are based on special object-oriented (OO) features, such as the number of classes, depth of inheritance tree, number of subclasses, etc. [13]. With the arrival of new higher-level programming paradigms such as aspect-oriented programming, generic programming or metaprogramming, new complexity metrics should be defined, because metrics applied to programs implemented in different paradigms than the one they were developed for may report false results [14].

The aim of this paper is to contribute towards research in software complexity measurement and management by defining complexity metrics specifically for metaprograms. The research is relevant because of the importance of ensuring metaprogram testability and reliability and developing effective metaprogram testing procedures, to which metaprogram complexity measures can contribute similarly to the contribution of software metrics to predict critical information about reliability and maintainability of software systems using automatic analysis of source code.

The outline of the paper is as follows. Section 2 discusses related works on complexity metrics. Section 3 analyzes evaluation of metaprogramming and metaprogram complexity. Section 4 describes the proposed metaprogram complexity metrics. Section 5 presents theoretical validation of the proposed metrics. Section 6 gives two examples of metaprogram complexity calculation. Finally, Section 7 presents conclusions and outlines future work.

2. Related Work on Complexity Metrics

Complexity is the intrinsic attribute of systems and processes through which systems are created. Complexity measures allow reasoning about system structure, understanding system behaviour, comparing and evaluating systems or foreseeing their evolution. System design complexity addresses complexity associated with mapping of a problem space into a given representation. An overall rating of system complexity (*System Complexity*) consists of the sum of the individual module complexities associated with the

module's connections to other modules (*Structural Complexity*) and the amount of work the module performs (*Data Complexity*) [15].

Structural complexity addresses the concept of *coupling*, i.e., the interdependence of modules of source code. It is assumed that the higher coupling between modules is, the more difficult it is for a programmer to comprehend a given module. Data complexity addresses the concept of *cohesion*, i.e., the intradependence of modules. In this case, it is assumed that the higher cohesiveness is, the easier it is for a programmer to comprehend a given module. The structural and data complexity measures are based on the module's fan-in, fan-out, and number of input/output variables. These metrics address system complexity at the system and module levels. Procedural complexity is associated with the complexity of the logical structure of a program assuming that the length of a program in Lines of Code (LOC) or the number of logical constructs such as sequences, decisions, or loops determines complexity of the program.

M. Rauterberg [16] addresses a similar problem, i.e., how to measure the cognitive complexity in human-computer interaction. He proposes to derive *cognitive complexity* (CoC) from *behaviour complexity* (BC), *system complexity* (SC) and *task complexity* (TC) as: CoC = SC + TC – BC.

S.D. Sheetz *et al.* [17] address complexity of the OO system at the application, object, method, and variable levels, and at each level propose the measures to account for the cohesion and coupling aspects of the system. Complexity of the OO system at each level is presented as a function of the measurable characteristics such as fan-in, fan-out, number of I/O variables, fan-up, fan-down, and polymorphism. Each measure is defined with adherence to the principles that measures must be intuitive and that they must be applicable to all phases of the OO development lifecycle.

Cyclomatic complexity is one of the more widely-accepted *static* software metrics [13]. It is intended to be independent of the language and language format. The other metrics bring out other facets of complexity, including both structural and computational complexity: *Halstead complexity measures* [18] identify algorithmic complexity, measured by counting operators and operands; *Henry and Kafura metrics* [19] indicate coupling between modules (parameters, global variables, calls); *Bowles metrics* [13] evaluate the module and system complexity, coupling via parameters and global variables; *Troy and Zweben* [13] metrics evaluate modularity or coupling; complexity of structure (maximum depth of a structure chart). Wang's *cognitive complexity measure* [20] indicates the cognitive and psychological complexity of software as a human intelligence artefact. With the arrival of new programming paradigms, new complexity metrics have been proposed for aspect-oriented programming (AspectJ) [21] and generic programming (C++ Standard Template Library) [22].

There were efforts to describe formal properties of complexity metrics that could be used for evaluation and theoretical validation of complexity measures. J. Weyuker [23] introduces a set of syntactic software complexity properties as criteria and examines the strengths and weaknesses of the known complexity measures, which include *statement count*, *cyclomatic*

number, effort measure, and *data flow complexity.* L.C. Briand *et al.* [3] provide a theoretical framework for relating structural complexity, cognitive complexity and external quality attributes.

3. Complexity of metaprogramming and metaprograms

It is a well-known fact that the same algorithm implemented in different programming paradigms or languages can have very different complexity of description (i.e., description complexity is not a property of an algorithm but rather a property of an implementation language). For example, one study [24] shows that complexity of Quick Sort algorithm implementations measured using Halstead Volume, Program Effort and Program Difficulty metrics [18] is highest for C and lowest for Assembly and Visual Basic language programs.

Metaprogramming [25], as a paradigm for developing programs that create other programs, is a level of complexity above traditional programming paradigms. There are two types of metaprogramming: homogeneous metaprogramming and heterogeneous metaprogramming.

In case of homogeneous metaprogramming, we have two subsets of a domain language: one is dedicated for expressing domain functionality, and the other is used for managing variability at meta-level (generic parameters, templates, etc.). The developer has to know only one programming language syntax, the metaprogram is as readable as a domain program written in the same domain programming language, and the development flow uses the same development toolset. Therefore, the complexity of developing metaprograms using homogeneous metaprogramming technique is only slightly higher than complexity of traditional programming.

In case of heterogeneous metaprogramming, we have two different languages: a domain language itself and a metalanguage, which manipulates with source code of domain language programs. As a result, the cognitive complexity of heterogeneous metaprograms expressed in terms of their readability and understandability is significantly higher, because the developer must know, understand and use the syntactical constructs of two different languages in the same metaspecification. The development flow is significantly more complex: not only two development environments have to be used, but also the testing of metaprograms is a significant and time-consuming problem. Therefore, complexity of developing metaprograms using heterogeneous metaprogramming techniques is considerably higher than complexity of traditional programming.

Complexity measures may be helpful for reasoning about metaprogram structure, understanding the relationships between different parts of metaprograms, comparing and evaluating metaprograms. Here we distinguish between: 1) first-order properties, or *characteristics*, which are derived directly from the metaprogram description itself using simple mathematical actions such as counting, e.g., program size (count of symbols in a file); and 2)

second-order properties, or *metrics*, which cannot be derived directly from artefacts, but are calculated from the first-order properties.

Metaprogram complexity can be evaluated at several dimensions:

1) **Information:** Metaprogram as message (sequence of symbols) containing information with unknown syntax and structure.

2) **Metalanguage:** Metaprogram as annotated domain knowledge. Domain knowledge is expressed using a domain language, whereas domain variability is specified using a metalanguage. Such separation of domain and meta levels is a first step towards the creation of a metaprogram.

3) **Graph:** Metaprogram as a graph of execution paths, where a root is a metaprogram, the nodes are the metalanguage constructs, and the leaves are the domain program instances.

4) **Algorithm:** Metaprogram as a high-level program specification (algorithm), which contains a collection of functional (structural) operations. An operation may have one or more operands specified as metaprogram attributes (parameters).

5) **Cognition:** Metaprogram as a number of different information units available for human cognition. A unit may represent either a metalanguage construct (macro, template, function), its argument or a meta-parameter.

4. Metaprogram Complexity Metrics

We use the following metrics for evaluating complexity at different dimensions of a metaprogram: Relative Kolmogorov Complexity (RKC), Metalanguage Richness (MR), Cyclomatic Complexity (CC), Normalized Difficulty (ND), and Cognitive Difficulty (CD).

4.1. Information dimension: Relative Kolmogorov Complexity

There are several methods to evaluate informational software complexity such as Shannon entropy, computational complexity, network complexity and topological complexity. We use the algorithmic complexity metric also known as *Kolmogorov Complexity* [26]. Kolmogorov complexity is a measure of randomness of strings and other objects based on their information content. Kolmogorov Complexity measures complexity of an object by the length of the smallest program that generates it. Suppose, we have an object x and a description system (e.g., a programming language) φ that maps from a description w to this object. Kolmogorov Complexity $K_\varphi(x)$ of an object x is the size of the shortest program in the description system φ capable of producing x on a universal computer:

$$K_\varphi(x) = \min_w \{\|w\| : \varphi_w = x\} \quad (1)$$

Different description systems can provide distinct values of *K(x)*, but one can prove that the differences are only up to a fixed additive constant. Intuitively, Kolmogorov Complexity $K_\varphi(x)$ is the minimal size of information required to generate *x* by an algorithm. Unfortunately, it cannot be computed in the general case and must be approximated. Usually, compression algorithms are used to give an upper bound to Kolmogorov Complexity. Suppose that we have a compression algorithm C_i. Then, a shortest compression of *w* in the description system φ will give the upper bound to information content in *x*:

$$K_\varphi(x) \le C(x) := \min_i \{\|C_i\|, \varphi_{C_i} = x\} \quad (2)$$

Kolmogorov Complexity has been used earlier (under the name of Generative Software Complexity) to measure the effectiveness of applying program generation techniques to software [27]. Program generators were defined as compressed programs, and the shortest generator is assumed to have maximal generative complexity.

Here we evaluate the complexity of a metaprogram *M* using the Relative Kolmogorov Complexity (RKC) metric, which can be calculated using a compression algorithm *C* as follows:

$$RKC = \frac{\|C(M)\|}{\|M\|} \quad (3)$$

where $\|M\|$ is the size of a metaprogram *M*, and $\|C(M)\|$ is the size of a compressed metaprogram *M*.

A high value of RKC means that there is a high variability of text content, i.e., high complexity. A low value of RKC means high redundancy, i.e., the abundance of repeating fragments in metaprogram code.

4.2. Metalanguage dimension: Metalanguage Richness

Metaprogram *M* can be defined as a collection of domain language statements with corresponding annotations (metadata) expressed symbolically: $O = \langle (s, m) \mid s, m \in \Sigma^* \rangle$, where s is a domain language statement, m is the metadata of s, and Σ^* is a string of symbols from alphabet Σ. For the evaluation of metaprogram complexity at the metalanguage dimension, we use the Metalanguage Richness (MR) metric:

$$MR = \frac{\sum_{m \in M} \|m\|}{\|M\|} \quad (4)$$

where $\|M\|$ is the size (length) of a metaprogram *M*, and $\|m\|$ is the size (length) of the metalanguage constructs in a metaprogram *M*.

A higher value of MR means that a metaprogram contains more metadata and its description is more complex.

4.3. Graph dimension: Cyclomatic Complexity

Cyclomatic Complexity (CC) [28] of a program directly measures the number of linearly independent paths through a program's source code from entrance to each exit. For metaprograms, CC is equal to the number of distinct domain program instances that can be generated from a metaprogram.

A metaprogram *M* can be defined as a function $\Phi(M): P \to I$ that maps from a set of its parameters *P* to a set of its domain program instances *I*. Following this definition, CC of a metaprogram is equal to the cardinality of a set of the distinct domain program instances described by a metaprogram.

$$CC = |\mathrm{cod}\ \Phi| = |I| \tag{5}$$

Since Φ is an injective function, which associates distinct metaprogram parameter values with distinct domain program instances, the cyclomatic complexity of a metaprogram *M* can be computed using only the interface description of a metaprogram. For independent parameters, the value of CC can be calculated as a product of the number of allowed parameter values for each parameter of a metaprogram:

$$CC = \prod_{p \in P} |\mathrm{dom}\ p| \tag{6}$$

A higher value of CC indicates higher complexity of the metaprogram's parameter set (meta-interface).

4.4. Algorithmic complexity: Normalized Difficulty

A functional program specification *S* is a sequence of functions $S = (f \mid f \in F)$, where $f : (a, a \in A) \to A$ is a specific function (operator) that may have a sequence of operands as its arguments, and *A* is a set of function operands. For metaprograms we accept that operations are specified as metalanguage functions, and operands are specified as metaprogram parameters. For the evaluation of metaprogram complexity at the algorithm dimension, we use the Halstead complexity metrics [18]. From a metaprogram we derive the number of distinct operators $n_1 = |F|$, the number of distinct operands $n_2 = |A|$, the total number of operators $N_1 = |S|$, the total number of operands $N_2 = \sum_{f \in S} |A|$.

Halstead Difficulty *D* indicates the cognitive difficulty of a program:

$$D = \left(\frac{n_1}{2}\right)\left(\frac{N_2}{n_2}\right) \quad (7)$$

The Halstead Volume *V* measures the size of a program specification:

$$V = N \log_2 n \quad (8)$$

For evaluating metaprogram complexity at the algorithm dimension we propose the Normalized Difficulty (ND) metric, which is a normalized ratio of the cognitive difficulty and size metrics:

$$ND = \frac{n_1 N_2}{(N_1 + N_2)(n_1 + n_2)} \quad (9)$$

The ND metric measures the complexity of a metaprogram as an algorithm. A high value of the *ND* metric means that metaprogram is highly complex in terms of time and effort required to understand it.

4.5. Cognitive complexity: Cognitive Difficulty

Following the works of G. Miller [29] stating that humans can hold 7 (+/- 2) chunks of information in their short-term memory at one time, and M. Keating [12], who claims that the number of modules at any level of software hierarchy must be 7 +/- 2, for evaluating complexity of metaprograms we propose the Cognitive Difficulty (CD) metric. Cognitive Difficulty is calculated as the maximal number of meta-level units (metaparameters *P*, metalanguage constructs N_1, or their respective arguments N_2) in a metaprogram.

$$CD = \max(P, N_1, N_2) \quad (10)$$

The proposed metaprogram complexity metrics are summarized in Table 1.

Table 1. Summary of metaprogram complexity metrics

Metric	Objects of measurement	Meaning for a metaprogram
Relative Kolmogorov Complexity	Object: metaprogram Program: compressed metaprogram	High variability of content
Metalanguage Richness	Data: domain language constructs Metadata: metalanguage constructs	Complexity of description at meta level
Cyclomatic Complexity	Independent paths: number of distinct instances	Complexity of a meta-interface
Normalized Difficulty	Operators: metalanguage functions Operands: metaprogram parameters	Algorithmic complexity of a

		metaprogram
Cognitive Difficulty	Metaprogram parameters, metalanguage functions, metalanguage function arguments	Cognitive understandability of a metaprogram

5. Theoretical validation of complexity metrics

Validation of software metrics is important to ensure that metrics are accepted by the scientific community and used properly. There are two methods of metrics validation: *theoretical* and *empirical* [30]. Theoretical validation ensures that the metric is a proper numerical characterization of software property it claims to measure. Empirical validation relates metrics with some important external attributes of software (such as the number of faults). While both types of validation are necessary, the empirical validation requires much time and many researchers to contribute since many studies need to be performed to gather convincing evidence from many real-world libraries and applications that a metric is valid. The domain of metaprogram complexity research is not mature yet, therefore while there are open metaprogram libraries available (such as Boost [31] in C++) for such research currently there are not sufficient data available publicly on the external characteristics of such metaprograms such as reliability or maintainability.

Therefore, we validate the proposed metaprogram complexity metrics theoretically using Weyuker's properties [23], a set of formal properties that can be used to evaluate any software metrics.

Property 1 (Eq. 11) will be satisfied when we can find two metaprograms of different complexity. All proposed complexity metrics satisfy Property 1.

$$(\exists P)(\exists Q)(|P| \neq |Q|) \quad (11)$$

Property 2 is satisfied when there are finitely many programs of complexity c, where c is a non-negative number. The property is not satisfied for all complexity measures that are size-independent (scaled). Therefore Property 2 is not satisfied for all proposed metaprogram complexity metrics.

Property 3 (Eq. 12) is satisfied if we can find two distinct metaprograms that have equal complexity. The property is satisfied by all proposed metaprogram complexity metrics.

$$(\exists P)(\exists Q)(P \neq Q)(|P| \neq |Q|) \quad (12)$$

Property 4 (Eq. 13) is satisfied if equivalent metaprograms of different complexity can be written. The property is not satisfied by RKC and MR metrics.

$$(\exists P)(\exists Q)(P \equiv Q \,\&\, |P| \neq |Q|) \quad (13)$$

Property 5 (Eq. 14) is satisfied if after concatenating two metaprograms, the complexity of the merged metaprogram increases beyond individual

complexities of original metaprograms. The property is satisfied by all metrics, except MR (because of averaging).

$$(\forall P)(\forall Q)(|P| \le |P+Q| \,\&\, |Q| \le |P+Q|) \quad (14)$$

Property 6 (Eq. 15) is satisfied if concatenation of two equal complexity metaprograms with some other metaprogram gives different complexity metaprograms. The property is satisfied by all metrics (because metaprograms can have common metaparameters, but distinct metabodies).

$$(\exists P)(\exists Q)(\exists R)(|P| = |Q| \,\&\, |P;R| \ne |Q;R|) \quad (15)$$

Property 7 is satisfied if by permuting the order of statements in a metaprogram, the complexity of a metaprogram changes. The property is not satisfied by all metaprogram complexity metrics except RKC metric.

Property 8 is satisfied if renaming of the symbols and variables of a metaprogram does not change the complexity of a program. The property is satisfied for all metaprogram complexity metrics except RKC metric.

Properties 9a (Eq. 16) and **9b** (Eq. 17) are satisfied when a two (or more) metaprograms are concatenated, the sum of complexities of the original metaprograms is less than the complexity of the bigger metaprogram. The property is satisfied by RKC (because the concatenation provides more opportunities for compression), CC (because adding new metaparameters leads to geometrical increase of metaprogram instance number), CD (because two metaprograms can have the same metaparameters, metalanguage constructs or their arguments) metrics. Properties 9a and 9b are not satisfied by MR metric (because combining two metaprograms will not lead to their increased coupling). Only property 9a is satisfied by ND metric.

$$(\exists P)(\exists Q)(|P| + |Q| < |P;Q|) \quad (16)$$

$$(\forall P)(\forall Q)(|P| + |Q| \le |P;Q|) \quad (17)$$

Table 2. Summary of metaprogram complexity validation

Complexity metric	Weyuker's property								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
RKC	+	–	+	–	+	+	+	–	+
MR	+	–	+	–	–	+	–	+	–
CC	+	–	+	+	+	+	–	+	+
ND	+	–	+	+	+	+	–	+	+/–
CD	+	–	+	+	+	+	–	+	+

The results of theoretical validation are summarized in Table 2. Note that Weyuker's properties were developed for procedural languages. Hence, there might be possibility that a proposed metaprogram complexity measure may not satisfy all the properties, but still may be valid for metaprogramming

domain as, e.g., some object-oriented metrics that do not satisfy Weyuker's properties are still considered valid for object-oriented programs [32].

6. Example of metaprogram complexity calculation

6.1. Heterogeneous metaprogramming

We demonstrate the complexity calculation of the heterogeneous metaprogram developed for hardware design domain. In hardware design domain, a great number of similar domain entities exist. For example, the most widely used hardware library components are gates (see Fig. 1; in VHDL), which implement a particular logical function. The hardware designer requires many different gate components implementing different functions and having a different number of inputs. All these components are very similar to each other both syntactically and semantically, and thus they constitute a component family.

```
entity gate is
   port ( X1, X2 : in bit; Y : out bit );
end gate;

architecture behave_gate of gate is
  begin
  Y <= X1 and X2;
end behave_gate;
```

```
entity gate is
   port ( X1, X2, X3 : in bit; Y : out bit );
end gate;

architecture behave_gate of gate is
  begin
  Y <= X1 or X2 or X3;
end behave_gate;
```

Fig. 1. Instances of VHDL gate family: a) 2-input AND gate, and b) 3-input OR gate

Next, we develop a metaprogram, which describes a gate component family. For example, the identified generic parameters and their values for the gate component family are as follows:

Gate_function = { AND, OR, XOR, NAND, NOR, XNOR }
Gate_inputs = { integer numbers from 2 to 8 }

A *gate* metaprogram (see Fig. 2) was developed using Open PROMOL [33] metalanguage. The metaprogram has 2 parameters, 3 metalanguage functions, and its size is 291 B.

```
$
"Enter gate function: "    {and, or, xor, nor, nand, xnor}  f := and;
"Enter number of inputs:"  {2..8}                           num := 2;
$
entity gate is
   port ( @gen[num,{, }] : in bit; Y : out bit );
end gate;

architecture behave_gate of gate is
  begin
  Y <= @gen[num, { @sub[f] }];
end behave_gate;
```

Fig. 2. Generic gate described using Open PROMOL metalanguage

We calculate RKC value using a BWT (*Burrows-Wheeler Transform*) compression algorithm, because currently it allows achieving best compression results for text-based information and thus allows to better approximate information content. The size of the *gate* metaprogram is 271 B. The size of the compressed metaprogram will put the upper limit on its information content. After compression we obtain 245 B, therefore RKC value of a *gate* metaprogram is equal to 245/271 = 0.90.

We calculate MR of the *gate* metaprogram by calculating the size of its metainterface and the length of its metalanguage functions, which is equal to 139 B. Therefore, its MR value is equal to 139/271 = 0.51.

Cyclomatic Complexity of a metaprogram is a number of different program instances that can be generated from it. The metric can be calculated as the number of distinct metaprogram parameter values. Parameters *f* and *num* are independent. Parameter *f* can have 6 different values, and parameter num can have 7 values. The *gate* metaprogram covers a family of $6 \cdot 7 = 42$ different component instances. Therefore, its CC value is 42.

The *gate* metaprogram has 3 metalanguage functions, 2 distinct functions (**@gen**, **@sub**), 4 metalanguage function arguments and 3 distinct arguments (num, {,}, {**@sub**[f]}). Therefore, its ND is equal to: $\frac{2 \cdot 4}{(3+4) \cdot (2+3)} = \frac{8}{35} = 0.23$. From the same values, we calculate that its CD is $\max(2,3,4) = 4$.

The values of the calculated complexity metrics for the *gate* metaprogram are summarized in Table 3.

Based on the metaprogram complexity metric values we can make the following conclusions on complexity of the *gate* metaprogram. The RKC value is high, therefore the metaprogram almost has no repeating fragments, it is coded at a meta-level efficiently and there is hardly room for any additional generalization without introducing new parameters or widening the scope of the metaprogram. The MR value shows that metalanguage constructs cover only about a half of the metaprogram's size, therefore, its understandability and readability is good. Following Frappier *et al.* [34], who introduce the

following boundaries of the CC values based on empirical research and practical implementations of large software systems: simple (1-10), slightly (moderately) complex (11-20), complex (21-50), over-complex and untestable (> 50), we conclude that due to large parameter space of the metaprogram, the exhaustive testability of its instances is complex. The CD value is below lower threshold (< 5) for short-term memorability of chunks of information as formulated by [29], therefore, cognitive complexity of the metaprogram is low.

Table 3. Complexity measures of the *gate* metaprogram

Complexity dimension	**Complexity metric**	**Value**
Information	Relative Kolmogorov Complexity (RKC)	0.90
Metalanguage	Metalanguage Richness (MR)	0.51
Graph	Cyclomatic Complexity (CC)	42
Algorithm	Normalized Difficulty (ND)	0.23
Cognitive	Cognitive Difficulty (CD)	4

Finally, we present complexity values calculated for Open PROMOL metaprograms created from Altera's library for OrCAD VHDL components (Table 4). Altera's library is a large collection of specific components, which are supposed to cover the entire circuit design domain (it contains 282 macrofunctions and 73 primitives, i.e., 355 VHDL components at all). The components were generalized using Open PROMOL metalanguage to create a generic VHDL component library [35].

Table 4. Complexity of Open PROMOL components of generic VHDL library

No.	**PROMOL metaprogram**	**Complexity metric**					**Complexity**
		RKC	**MR**	**CC**	**ND**	**CD**	
1	Serial multiplier	0.219	0.03	4	0.229	27	Simple
2	Trigger	0.271	0.27	80	0.111	52	Over-complex
3	Gate	0.502	0.49	181	0.026	26	Over-complex
4	Adder	0.478	0.25	4	0.169	20	Simple
5	Register	0.457	0.34	512	0.136	94	Over-complex
6	Multiplexer	0.507	0.36	32	0.051	22	Complex
7	Comparator	0.429	0.31	9	0.123	33	Simple
8	Shift Register	0.392	0.31	18	0.091	121	Moderate
9	Subtractor	0.378	0.20	4	0.072	84	Simple
10	Parallel multiplier	0.328	0.38	96	0.092	126	Over-complex
11	Register File	0.358	0.24	36	0.084	255	Complex
12	Counter	0.323	0.28	30	0.044	172	Complex
13	Multiplier	0.331	0.64	8	0.092	28	Simple
14	Divider	0.527	0.38	30	0.096	48	Complex

We evaluate the results presented in Table 4 as follows. Most complex metaprograms are those, which describe components with largest variability in the domain, thus requiring a larger number of parameters for selection of a specific instance and a larger number of metalanguage functions to represent their variability (see values of CC and CD metrics). Such metaprograms are difficult to test and maintain. Their complexity can be decreased by introducing hierarchical decomposition at the metaprogram level.

6.2. Homogeneous metaprogramming

As an example of complexity measurement of homogeneous metaprograms, we analyze Boost C++ Libraries [31]. Boost is a collection of open source libraries that extend the functionality of C++. To ensure efficiency and flexibility, Boost extensively uses C++ template metaprogramming techniques. In C++, the template mechanism provides a rich facility for computation at compile-time. Here we analyze complexity of template functions in a Boost.Math. This library several contributions in the domain of mathematics such as complex number and special mathematical functions. An example of such template function (a fragment) is presented in Fig. 3.

```
template<class T>
inline T fabs(const std::complex<T>& z)
{
   return ::boost::math::hypot(z.real(), z.imag());
}
```

Fig. 3. An example of template function (fabs)

The complexity measurement results using the proposed metaprogram complexity metrics are presented in Table 5.

Template functions in the Boost.Math library are rather simple. They mostly have CC values either 3, 16 or 19 meaning that each template function has a single template parameter, which can accept either 3 floating point, 16 integer or 19 floating point and integer C++ type values. Only `common_factor_ct` has template function `static_lcm`, whose template parameters are numbers of *long* type rather than types. All template functions also have the same ND value, because all template references are to the same template parameter class and have only one template parameter, therefore the number of distinct metaprogram operators and operands is equal to 1, and ND is equal to 0.25. The value of the CD metric is larger for components, which have a larger number of template references. The values of the RKC and MR metrics are larger for smaller components, which have less domain language (C++ non-template) code. When evaluating testability and maintainability of Boost.Math library components, the CD value could be used using the boundaries proposed by Frappier *et al.* [34] (see last column of Table 5.).

Table 5. Complexity of components of Boost.Math library

No.	Template function	Complexity metric					Complexity
		RKC	MR	CC	ND	CD	
1.	acos	0.229	0.037	3	0.25	34	Moderate
2.	acosh	0.488	0.128	3	0.25	3	Simple
3.	asin	0.218	0.037	3	0.25	34	Moderate
4.	asinh	0.488	0.144	3	0.25	2	Simple
5.	atan	0.423	0.116	3	0.25	10	Simple
6.	atanh	0.231	0.035	3	0.25	23	Complex
7.	bessel	0.149	0.157	3	0.25	127	Over-complex
8.	beta	0.134	0.090	3	0.25	199	Over-complex
9.	binomial	0.373	0.201	16	0.25	19	Moderate
10.	cbrt	0.438	0.144	3	0.25	15	Moderate
11.	common_factor_ct	0.171	0.042	1.9E19	0.25	13	Moderate
12.	common_factor_rt	0.160	0.043	3	0.25	32	Complex
13.	cos_pi	0.410	0.216	3	0.25	14	Moderate
14.	digamma	0.224	0.061	3	0.25	45	Complex
15.	ellint_1	0.242	0.133	3	0.25	40	Complex
16.	ellint_2	0.264	0.167	3	0.25	39	Complex
17.	ellint_3	0.213	0.116	3	0.25	48	Complex
18.	ellint_rc	0.391	0.137	3	0.25	17	Moderate
19.	ellint_rd	0.354	0.121	3	0.25	20	Moderate
20.	ellint_rf	0.363	0.127	3	0.25	22	Complex
21.	ellint_rj	0.320	0.105	3	0.25	24	Complex
22.	erf	0.210	0.055	3	0.25	88	Over-complex
23.	expint	0.234	0.038	3	0.25	128	Over-complex
24.	expm1	0.279	0.172	3	0.25	61	Over-complex
25.	fabs	0.666	0.150	3	0.25	1	Simple
26.	factorials	0.244	0.141	16	0.25	47	Complex
27.	fpclassify	0.146	0.086	19	0.25	98	Over-complex
28.	gamma	0.138	0.140	3	0.25	329	Over-complex
29.	hermite	0.421	0.208	3	0.25	14	Moderate
30.	hypot	0.396	0.211	3	0.25	22	Complex
31.	laguerre	0.260	0.164	3	0.25	30	Complex
32.	lanczos	0.225	0.070	3	0.25	638	Over-complex
33.	legendre	0.237	0.136	3	0.25	43	Complex
34.	log1p	0.204	0.112	3	0.25	91	Over-complex
35.	modf	0.293	0.233	3	0.25	10	Simple
36.	next	0.202	0.085	19	0.25	60	Over-complex
37.	octonion	0.030	0.013	19	0.25	599	Over-complex
38.	pow	0.239	0.196	3	0.25	42	Complex
39.	powm1	0.390	0.254	3	0.25	15	Moderate
40.	quaternion	0.059	0.029	19	0.25	348	Over-complex
41.	round	0.274	0.229	3	0.25	20	Moderate
42.	sign	0.516	0.155	19	0.25	5	Simple
43.	sin_pi	0.471	0.209	3	0.25	9	Simple
44.	sinc	0.215	0.098	3	0.25	36	Complex
45.	sinhc	0.224	0.084	3	0.25	30	Complex
46.	spherical_harmonic	0.210	0.150	9	0.25	50	Complex
47.	sqrt1pm1	0.468	0.238	3	0.25	8	Simple
48.	trunc	0.276	0.225	3	0.25	20	Moderate
49.	zeta	0.257	0.041	3	0.25	61	Over-complex

7. Conclusions and Future Work

Complexity of metaprograms and metaprogramming techniques is an important factor in developing and maintaining generic components and software generators. Complexity of metaprograms can be evaluated at several dimensions (information, metalanguage, graph, algorithm, cognition) using a variety of measures adopted from information theory and software engineering domain. Such metrics can be used to rank metaprograms based on their complexity values, to assess testability and maintainability of metaprograms, and can be used by reusable software library developers for evaluating complexity of their work artefacts. Despite the lack of larger-scale empirical validation, we still expect that metaprogram complexity metrics could be used to indicate poorly written or untestable metaprograms, when the metric values exceed predefined maximal or minimal boundaries.

Future work will focus on the empirical validation of proposed metrics using open metaprogram libraries implemented in different metalanguages.

References

1. Bennett, K.H., Rajlich, V.: Software maintenance and evolution: A roadmap. In: A.C. Finkelstein (ed.), Future of Software Engineering. ACM Press, 73-87, 2000.
2. Visscher, B.-F.: Exploring Complexity in Software Systems. Ph.D. thesis. Department of Computer Science and Software Engineering. University of Portsmouth, UK, June 2005.
3. Briand, L.C., Morasca, S., Basili, V.R.: Property-Based Software Engineering Measurement. IEEE Trans. Software Eng. 22(1): 68-86, 1996.
4. IEEE Computer Society: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12 – 1990.
5. Zuse, H.: Software Complexity – Measures and Methods. DeGruyter Publ., 1991. Card, D.N., Agresti, W.W.: Measuring software design complexity. The Journal of Systems and Software, vol. 8, 185-197, 1988.
6. Reformat, M., Musilek, P., Wu, V., Pizzi, N.J.: Human Perception of Software Complexity: Knowledge Discovery from Software Data. In: Proc. of 16th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI 2004), Nov. 15-17, 202-206, 2004.
7. Blecker, T., Abdelkafi, N., Kaluza, B., Kreutler, G.: A Framework for Understanding the Interdependencies between Mass Customization and Complexity. In: Proc. of the 2nd Int. Conf. on Business Economics, Management and Marketing, Athens/Greece, June 24 - 27, 2004.
8. Misra, S., Akman, I.: A Model for Measuring Cognitive Complexity of Software. In: Proc. of 12th Int. Conf. on Knowledge-Based Intelligent Information and Engineering Systems (KES 2008), Zagreb, Croatia, September 3-5, 2008, Part II. LNCS vol. 5178, pp. 879-886. Springer, 2008.
9. Mayrhauser, A.V., Vans, A.M.: From Code Understanding Needs to Reverse Engineering Tools Capabilities. In: Proc. of 6th Int. Conference on Computer Aided Software Engineering (CASE'93), 19-23 July 1993, 230-239.
10. Damaševičius, R.: On the Quantitative Estimation of Abstraction Level Increase in Metaprograms. Computer Science and Information Systems (ComSIS), 3(1): 53-64, 2006.

11. Keating, M.: Measuring Design Quality by Measuring Design Complexity. In: Proc. of the 1st Int. Symp. on Quality of Electronic Design (ISQED'2000), p. 103, 2000.
12. Software Engineering Institute (SEI). Software Technology Roadmap, 2006. Sipos, A., Pataki, N., Porkolab, Z.: On Multiparadigm Software Complexity Metrics. Pure Mathematics and Applications, 17(3-4): 469-482, 2006.
13. Card, D.N., Glass, R.L.: Measuring Software Design Quality. Prentice Hall, 1990.Rauterberg, M.: How to Measure Cognitive Complexity in Human-Computer Interaction. In: Proc. of the 13th European Meeting on Cybernetics and Systems Research, Vienna, Austria, 9–12 April 1996, Vol. 2, 815-820.
14. Sheetz, S.D., Tegarden, D.P., Monarchi, D.E.: Measuring Object-Oriented System Complexity. Proc. of. First Workshop of Information Technologies and Systems, 285-307. MIT Sloan School of Management, Cambridge, MA, 1991.
15. Halstead, M.H.: Elements of Software Science. New York: Elsevier, 1977. Henry, S.M., Kafura, D.G.: Software Structure Metrics Based on Information Flow. IEEE Trans. Software Eng. 7(5): 510-518, 1981.
16. Wang, Y.: On the Cognitive Complexity of Software and its Quantification and Formal Measurement. International Journal of Software Science and Computational Intelligence 1(2): 31-53, 2009.
17. Pataki, N., Sipos, A., Porkolab, Z.: Measuring the Complexity of Aspect-Oriented Programs with a Multiparadigm Metric. In: Proc. of European Conf. on Object-Oriented Programming - Quantitative Approaches in Object-Oriented Software Engineering Workshop (ECOOP-QAOOSE), Nantes, France, 1-10, 2006.
18. Pataki, N., Pocza, K., Porkolab, Z.: Towards a Software Metric for Generic Programming Paradigm. In: 16th IEEE Int. Electrotechnical and Computer Science Conference, 24-26 September 2007, Portorož, Slovenia.
19. Weyuker E.J.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, vol. 14, no. 9, 1357-1365, September, 1988.
20. Olabiyisi, S.O., Ayeni, R.O., Omidiora, E.O.: Comparative Study of Implementation Languages on Software Complexity Measures of Quick Sort Algorithm. Journal of Engineering and Applied Sciences 3 (1): 118-122, 2008.
21. Damaševičius, R., Štuikys, V.: Taxonomy of the Fundamental Concepts of Metaprogramming. Information Technology and Control, 37(2), 124-132, 2008.
22. Li, M., Vitanyi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer Verlag, 1997.
23. Heering, J.: Quantification of structural information: on a question raised by Brooks. ACM SIGSOFT Software Engineering Notes 28(3): 6, 2003.
24. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering, vol. se-2, no. 4, 308-320, 1976.
25. Miller, G.: The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. The Psychological Review, March, 1956.
26. El Emam, K.: A Methodology for Validating Software Product Metrics. National Research Council of Canada, Ottawa, Ontario, Canada NCR/ERC-1076, 2000.
27. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley Professional, 2004.
28. Misra, S., Akman, I.: Applicability of Weyuker's Properties on OO Metrics: Some Misunderstandings. Computer Science and Information Systems (ComSIS), Vol. 5(1), 17-24, June 2008.
29. Štuikys, V., Damaševičius, R., Ziberkas, G.: Open PROMOL: an experimental language for target program modification. In: A. Mignotte, E. Vilar and L. Horobin (eds.), System on chip design languages: extended papers: best of FDL'01 and HDLCon'01. Netherlands: Kluwer Academic Publishers, 235-246, 2002.

30. Frappier, M., Matwin, S., Mili, A.: Software Metrics for Predicting Maintainability. Software Metrics Study: Tech. Memo. 2. Canadian Space Agency, 1994.
31. Damaševičius, R.: Scripting Language Open PROMOL: Extension, Environment and Application. MSc. Thesis. Kaunas University of Technology, Lithuania, 2001.

Robertas Damaševičius received his MSc (2001) and PhD (2005) degrees in informatics from Kaunas University of Technology (KTU), Kaunas, Lithuania. Currently he is an associated professor at Software Engineering Department, KTU. He teaches several computer science, programming and software engineering courses. He is also the member of Design Process Automation Group at Software Engineering Department. His research interests include hardware design, design automation, metaprogramming, software generation and program transformation, as well as domain analysis methods. He is an author of more than 50 scientific papers in the area.

Vytautas Štuikys is a professor at Software Engineering Department of KTU, Lithuania. He received the PhD and *doctor habilitatis* titles from KTU in 1970 and 2002, respectively. He is a lecturer and a researcher as well as a leader of the Design Process Automation Group. His research interests include software and hardware design methodologies, software reuse, component-based programming, metaprogramming and program generation, CAD systems and soft IP design. He has published more than 100 papers in the area. He is an author of several books and a monograph.

Received: March 15, 2009; Accepted: December 04, 2009.