UDC 004.4’41

On the Quantitative Estimation of Abstraction Level Increase in Metaprograms

Robertas Damaševičius

Software Engineering Department,
Kaunas University of Technology
Studentų 50-415, 51368 Kaunas, Lithuania
robertas.damasevicius@ktu.lt

Abstract. Higher-level programming such as metaprogramming introduces a layer of abstraction above the domain language programs. Metaprogramming allows describing generic components and managing variability in a domain. It is especially useful for developing program generators for domains, where a great deal of commonalties exists. It allows increasing the level of abstraction and hiding details that are unnecessary to the designer. Information abstraction and hiding reduces the amount of “user-visible” information. In this paper, we estimate the increase of abstraction by evaluating the information content at the lower (domain) and higher (meta) layers of abstraction. The estimation method is based on the Kolmogorov complexity and uses a common compression algorithm. The method is evaluated experimentally on families of DSP components.

1. Introduction

The abstraction level is the level of detail of a software system (model, component, program, etc.). In this sense, abstraction is a primary concept in software engineering and is, in fact, a basic property for understanding the reality and managing the complexity of software systems [1].

The simplest interpretation of abstraction is hiding of irrelevant details, though there are many different views what “irrelevant” is [2]. 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.

Abstraction is primarily responsible for the evolution of programming languages by stimulating adoption of higher-level mechanisms and constructs for programming. More abstract programming language mechanisms allow to replace complex and repeating low-level operations. Better abstraction allows to address complex problems with less code and less programming errors.

Though different layers of abstraction represent a *qualitative* leap in the level of abstraction that allows achieving higher productivity and faster development times, an interesting problem would be to evaluate the level of abstraction in a software system *quantitatively*. The problem is not a trivial one, because the level of abstraction is related the concepts of *software complexity* [3] and *information content* [4].

Indeed, a representation of a software system at a higher layer of abstraction contains less detail and usually has less source code lines than a corresponding representation at a lower layer of abstraction. Though, Lines of Code (LOC) is often used as a measure of software size, it is often criticized as ambiguous and even meaningless. An effort to measure the abstraction level using the LOC metric fails, because shorter code does not mean less software complexity or less information content. Furthermore, a program may contain redundant code, which is not taken by the LOC metric into account.

In this paper, we use Kolmogorov complexity [5] based metric to estimate the increase in the level of abstraction in metaprograms *quantitatively*. Metaprograms are generic programs (or program generators) that encapsulate families of similar software components. We evaluate the level of abstraction in *metaprograms as compared to families of domain programs* by estimating and comparing the information content at the metalevel and domain level of abstraction using a common compression algorithm.

The rest of the paper is organized as follows. Section 2 overviews the related works. Section 3 describes the layers of abstraction in software systems and describes the principles of metaprogramming. Section 4 describes the problem of estimating information complexity, quantity and content in software. Section 5 presents the experimental results of estimating information content and level of abstraction in component families and metaprograms. Section 6 presents conclusions and future work.

2. Related works

Other authors also use this Kolmogorov complexity in related research. For example, Li and Vitanyi [6] propose a metric for measuring the amount of shared information between two computer programs, thus allowing plagiarism detection. This metric is approximated by a heuristic compression algorithm.

Evans *et al.* [7] apply the Kolmogorov complexity-based metric to the problem of Information Assurance. The metric allows to detect abnormal system behavior and perform analysis of data and process vulnerability.

Bush and Hughes [8] use Kolmogorov complexity to identify different data types (semantic types) by estimates of their complexity. This allows to discover, e.g., executable data embedded within passive data types in network data flows.

Gács *et al* [9] use Kolmogorov complexity as a measure for the relation between an individual data sample and an individual model summarizing the information in the data and develop the algorithmic theory of statistic.

Taha *et al.* [4] propose a view motivated by Kolmogorov's notion of string complexity. A program generator captures the essential complexity of the programs it produces, since it contains all the information needed to reconstruct the full code base.

Keogh *et al.* [10] use the compression-based dissimilarity metric for detecting and clustering datasets. The idea is that similar datasets have similar content and complexity, which can be detected by compressing datasets and comparing their compressed size.

Campani *et al.* [11] apply Kolmogorov complexity to the characterization of systems and processes by calculating the amount of information, and the evaluation of computational models based on the idea of data compression (understood as a measurement of the amount of information).

Veldhuizen [12] uses Kolmogorov complexity to develop a theoretical model of reuse libraries and estimate the amount of reuse in software systems, the size of reuse library components, and the bounds of reuse.

The related works can be summarized as follows: Kolmogorov complexity already has been used to derive several software metrics in order to estimate software complexity, similarity, anomaly, vulnerability or reuse. The novelty of this paper is the application of the Kolmogorov information complexity theory to the problem of quantitative measuring of software abstraction.

3. Layers of abstraction in software

3.1. Classification of programming languages

Modern software systems often have many layers of abstraction such as assembly-level, object-level, scripting-level, web-level, meta-level, etc. Here, we can distinguish we following layers of software abstraction (see Fig. 1):

- *Register layer* – assembly languages that describe operations on a processor (register) level. The level of abstraction is very low and quantity of information available for a developer is very high and detailed.
- *Algorithm layer* – common procedural or (semi-)formal languages such as C, Pascal or LISP that hide the low-level details of handling data and rather focus on the description of software algorithms that manipulate with data. The developer is no longer dealing with data in individual registers. Much of the information content is abstracted away in the libraries.
- *Entity layer* – high-level programming languages such as C++, Java, or VHDL that focus on the description of domain entities. Domain entities are represented using components, aspects [13] or objects, which encapsulate code fragments (methods, concerns) on the same abstraction level. Modeling languages such as UML are also used on this layer, though these represent the domain entities graphically.
- *Composition layer* – compositional, glue and scripting [14] languages such as Python or TCL that assume a collection of existing components and focus on wiring of existing software artifacts into larger systems. The

primary concern is not data or algorithms, but software components. Thus, the level of abstraction is raised even higher, complexity of programming language is reduced, and repeating code is abstracted away leading to the decrease of information quantity available to the developer.

– *Meta layer* – metalogic, metaprogramming and metamodeling languages that are used for describing generic components, developing code generators, and capturing domain commonalties in software product lines [19]. Meta layer languages are based on higher-level programming methodologies, such as metaprogramming [15] or Generative Programming [16]. The examples are macro languages, preprocessing languages, MetaAspectJ, Open PROMOL [17], etc.

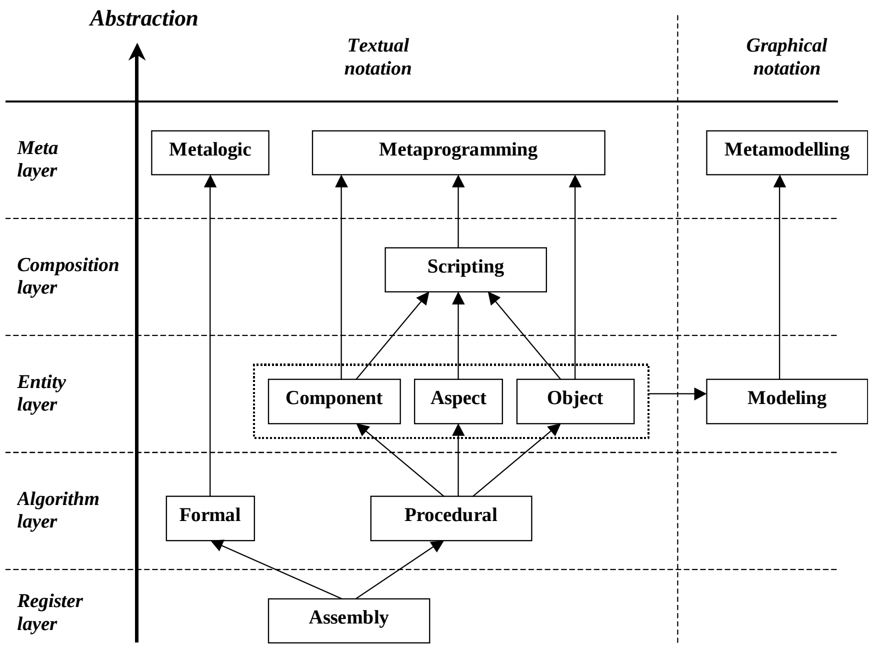

Fig. 1. Layers of abstraction in software programming languages

The presented classification of programming languages is by no means comprehensive. It omits many other classes of languages, such as domain-specific languages, architecture description languages, markup languages, etc. Rather it demonstrates the typical software abstraction layers the programming languages pertain to.

The abstraction level is generally increasing from the lower layers of abstraction to the higher layers, though it may differ at the same layer depending upon syntax and semantics of a particular programming language.

Thus, the level of abstraction in software engineering grows from data to algorithms, to objects, to software components and beyond.

Recently, the MDA (*Model-Driven Architecture*) approach was introduced, which allows to raise the level of abstraction above programming languages by defining implementation-independent models and metamodels that describe fundamental relationships between design concerns [18]. However, MDA requires metaprogramming as well for implementing code generation between high-level models and low-level system implementation.

3.2. Overview of metaprogramming

Now we look more closely at the principles of metaprogramming. Often, there is a great deal of similar components in a domain, e.g., there are 71 similar Java Buffer classes in JDK 1.5.01 source library. Such large number of components is difficult to maintain and reuse, changes are difficult to implement, etc. Higher-level programming, such as metaprogramming, raises the level of abstraction by introducing an additional level of generic parameters for managing variability of domain source code. This higher-level hides a common part of the component family at a lower (domain) layer of abstraction. The result of metaprogramming is a *metaprogram*. According to Batory [19], a meta-program is "*a program that generates the source of the application ... by composing pre-written code fragments*". In other words, a metaprogram is a set of instructions, descriptions, and means of control (possibly *generation*) of sets of domain programs. Metaprograms may be written using the same programming principles and constructs (*if, case, for loop*) as domain programs, however, they manipulate on *program* representations, not on *data*.

Metaprogram describes generic parameters and domain code modifications required to generate a particular customized component instance. The layers of abstraction are clearly separated. At a lower layer of abstraction, there is domain language code that describes common parts of component family. At a higher layer of abstraction, there is metalanguage code that describes variable parts of component family.

To achieve the prescribed aims, metaprogramming uses *separation of concerns, parameterization,* and *parameter dependency* knowledge. The principle of separation of concerns separates each domain problem into a distinct generic component or sets of components used to generate target program. Parameterization increases reusability by providing parameterized components, which can be instantiated for different choices of parameters. Parameter dependency knowledge allows capturing specific information about the parameter dependencies, default settings and illegal combinations.

A high-level language is the main abstraction for expressing the domain content and coding software components and generators. Some languages (e.g., Ada, VHDL, C++) do provide a meta-programming mechanism, called *generics* (or *templates,* in case of C++), for writing parameterized components. However, here we deal only with metaprogramming using a separate metalanguage (*heterogeneous metaprogramming*), which allows us achieving higher parameterization flexibility, better separation of concerns and layers of abstraction and generation of specific component instances.

In next section, we consider the problem of evaluating quantity (content) of information at a higher level of abstraction, in general.

4. Evaluation of information quantity and abstraction

4.1. Kolmogorov complexity

The problem considered in this paper is how to evaluate the raise of abstraction introduced by a higher-level language quantitatively. We argue that it can be evaluated relatively by comparing information content at both layers of abstraction. Since some of information is abstracted away at a higher layer of abstraction, we expect that information quantity directly represented at a higher level of abstraction generally should decrease, because much of it is hidden in the underlying tools (preprocessors, compilers, translators, etc.) and software libraries used. However, the entire content of information required to solve a certain problem should remain the same, as stipulated by the *Law of Conservation of Information*, which states that information in a closed system of natural causes remains constant or decreases [20]. Therefore, a relationship between the content of information at a higher and lower levels of abstraction is a metric of abstraction.

Therefore, we can estimate the increase/decrease of abstraction in software by measuring the content of information as different layers of abstraction. There are several methods to evaluate information content/ complexity such as computational complexity, Shannon entropy and topological complexity [21]. We use the Algorithmic information content metric also known as *Kolmogorov Complexity* [5]. Kolmogorov complexity is a measure of randomness of strings based on their information content. It was proposed by A.N. Kolmogorov in 1965 to quantify the randomness of strings and other objects in an objective and absolute manner.

The main idea of Kolmogorov complexity is to measure the 'complexity' of an object by the length of the smallest program that generates it. In general case, we have a domain object *x* and a description system (e.g., programming language) *φ* that maps from a description *w* (i.e., a program) to this object. Kolmogorov complexity *K(x)* of an object *x* in the description system *φ*, is the length of the shortest program in the description system *φ* capable of producing *x* on a universal computer such as a Turing machine:

$$K_{\varphi}(x) = \min_{w}\{\|w\| : \varphi_w = x\} \qquad (1)$$

Different programming languages will give rise to distinct values of *K(x)*, but one can prove that the differences are only up to a fixed additive constant.

Intuitively, *K(x)* is the minimal quantity of information required to generate *x* by an algorithm.

Kolmogorov complexity is the ultimate lower bound among all measures of information content. Unfortunately, it cannot be computed in the general case [4, 5]. Consequently, one must approximate it.

Some authors criticize the usage of Kolmogorov complexity and compression algorithms for evaluating information content (e.g., [22]). The objections are mostly focused on the concept of randomness. For example, a random string generated by a computer program would have much higher information content than the program itself.

In our view, the critics miss three important points as follows:

(1) Kolmogorov complexity measures the content of information only at the same level of abstraction.

(2) Random strings may not be meaningless and also can carry information, if they are considered as labels [23]. We can not consider a program as a closed system with bounded amount of information, because programs do not exist on their own, but in the context of other programs and data.

(3) Information complexity is not the same as content. Higher complexity, in fact, may mean lower content and *vice versa*.

4.2. Compression-based metric of abstraction increase

In this paper, Kolmogorov complexity is used to estimate the abstraction increase. Usually, the universal 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*:

$$K_{\varphi}(x) \leq \min_{i}\{C_i(\varphi_w)\} \tag{2}$$

As abstraction hides the complexity, abstraction of an object *x* in a description system *φ* can be defined as an inverse of complexity of *x* estimated in terms of Kolmogorov complexity:

$$A_{\varphi}(x) = \frac{1}{K_{\varphi}(x)} \tag{3}$$

The increase of abstraction level between a program φ_w that is a representation of *x* in a description system *φ,* and a program ψ_w, that is a representation of *x* in a description system *ψ* at a higher level of abstraction can be defined as follows:

$$A(\psi \mid \varphi) = \frac{K_{\varphi}(x)}{K_{\psi}(x)} \tag{4}$$

Having in mind Eq. 2 and that a metaprogram *MP* is a concise representation of a component family λ, which is a union of all its members P_j, we estimate the increase of abstraction level *A* in a metaprogram as compared to a domain program as follows:

$$A\left(MP^{\lambda} \mid P^{\lambda}\right) = \frac{\min\limits_{i} C_i\left(\bigcup\limits_{j} P_j^{\lambda}\right)}{\min\limits_{i} C_i\left(MP^{\lambda}\right)} \tag{5}$$

where C_i is a compression algorithm.

In the next Section, we demonstrate how compression algorithm based on Kolmogorov complexity metric can be used to evaluate information content and, consequently, the raise of abstraction introduced by the application of metaprogramming techniques in hardware and embedded systems design domain.

5. Empirical evaluation of abstraction increase in metaprogramming

In hardware and embedded systems design domain, a great number of similar domain entities exist. For example, the most-widely used hardware library components are gates (see Fig. 2), 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, thus their 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. 2. Instances of gate component family (in VHDL): a) 2-input AND gate, and b) 3-input OR gate

The content of information in component families can be estimated using the compression-based information content metric. We have selected the BWT (*Burrows-Wheeler Transform*) compression algorithm, because currently it allows to achieve best compression results for text-based information [24] and thus better approximate information content. The lowest size of the compressed components will put the upper limit on the estimated information quantity in the analyzed component family.

Next, we develop a metaprogram, which describes a component family at a higher level of abstraction. 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 16 }

A metaprogram (see Fig. 3) was developed using Open PROMOL [17] metalanguage. This metaprogram describes a generic gate and covers a family of 90 different component instances, which can be generated from it.

```
$
"Enter gate function: "      {and, or, xor, nor, nand, xnor}  f := and;
"Enter number of inputs:" {2..16}                             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. 3. Generic gate described using Open PROMOL metalanguage

Then, we evaluate the content of information at a higher level (metalevel) of abstraction. We again compress the developed metaprogram using a selected compression algorithm, which in our case is BWT. The lowest size of the compressed metaprogram will put the upper limit on the estimated information content at the metalevel.

The increase of abstraction between metalevel and domain level shall be the ratio of estimated information content at the metalevel and domain level, as stipulated in Eq. 5. For example, the size of the metaprogram given in Fig. 3 is 291B. The size of the metaprogram compressed using the BWT algorithm is 245B, which is the estimated quantity of information at metalevel. Next, we generate all instances of this metaprogram for all possible values of the generic parameters *f* and *num*. We obtain 90 different component instances (2 of them are given in Fig. 2, a & b). The total size of these instances is 21,426B when uncompressed and 726B after compression. Next, we apply Eq. 4 to obtain the estimated abstraction increase for the *gate* component family:

$$A = \frac{K_{domain}(gate)}{K_{meta}(gate)} = \frac{726}{245} = 2.96 \quad (6)$$

Thus, we estimate that the introduction of metaprogramming for describing generic gate components using VHDL as a domain language and Open PROMOL as a metalanguage allowed to increase abstraction by ~ 3 times.

We have performed the experiments with the following VHDL component families and metaprograms: gate, RSA coding processor, serial multiplier, register, shift register, multiplexer, and majority function for voting in fault-tolerant systems. We also have performed the experiments with the DSP algorithms implemented as embedded software in C as follows: DCT, FFT, Romberg integration, Chebyshev approximation and Taylor series expansion of popular mathematical functions. The results are summarized in Table 1.

The statistical evaluation of the obtained results for abstraction increase (mean = 2.9; std. deviation = 0.992; std. error = 0.286) was performed using one-sample Student's t-test. The mean is within 95% confidence interval.

Table 1. Summary of experiments

Component family	No. of instances	No. of generic parameters	Total source code size, B	Est. information quantity at domain level, B	Metaprogram size, B	Est. information quantity at metalevel, B	Abstraction increase
Gate	90	2	21,426	726	291	245	**2.96**
RSA	32	2	1,517,027	72,701	254,049	27,295	**2.66**
Serial multiplier	10	1	96,849	3,497	7,198	1,827	**1.91**
Register	1024	6	2,380,336	1,803	1,384	827	**2.18**
Shift register	72	5	38,800	1,178	2,896	786	**1.50**
Mux	54	4	48,636	3,107	1,848	1,625	**1.91**
Majority	192	2	240,345	1,940	1,017	529	**3.67**
DCT	14	2	5242	655	941	469	**1.40**
FFT	6	1	32,956	2,112	1989	394	**5.36**
Romberg integration	5	1	34,905	1,713	457	313	**5.47**
Taylor series	240	3	43,552	2,017	869	679	**2.97**
Chebyshev approx.	8	1	7,905	912	464	331	**2.76**

The content (or quantity) of information has decreased by 2.9 times on average in metaprograms as compared with domain program families. This number varies depending upon the type and size of components, the number of component instances in a component family, the number of generic parameters in a metaprogram, similarity of components within a component family, and syntactic characteristics of domain language and metalanguage.

In general, we estimate that the level of abstraction in metaprograms is about 3 times higher than the level of abstraction in domain programs.

6. Conclusions and Future work

In this paper, we have analyzed information content in higher-level programs (metaprograms) and compared it with information content in lower-level (domain) program families. We have proposed to estimate the abstraction level of a program as an inverse of its complexity as defined by Kolmogorov Complexity metric measured using a standard text compression algorithm. Based on the performed experiments, we estimate that metaprogramming decreases information content and thus increases the level of abstraction in analyzed domains by approx. 3 times.

Future work will focus on the estimation of component similarity using information content metrics. The more similar are software components, the more easily they can be generalized when developing generic components and thus the level of abstraction and reuse can be raised.

7. References

1. Albin, S.T.: The Art of Software Architecture: Design Methods and Techniques. Willey. (2003)
2. Liu, X., Yang, H., Zedan, H., Cau, A.: Speed and scale up software reengineering with abstraction patterns and rules. In Proceedings of International Symposium on Principles of Software Evolution, 90-99. (2000)
3. Glass, R.L.: Sorting out software complexity. In Source Communications of the ACM archive, Vol. 45, No. 11, 19 - 21. (2002)
4. Taha, W., Crosby, S., Swadi, K., A New Approach to Data Mining for Software Design. In Proceedings of the Int. Conf. on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'04), Cairo, Egypt. (2004)
5. Li, M., Vitanyi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer Verlag. (1997)
6. Evans, S., Bush, S. F., Hershey, J.: Information Assurance through Kolmogorov Complexity. DARPA Information Survivability Conference and Exposition II (DISCEX-II 2001), June 12-14, 2001, Anaheim, California. (2001)
7. Bush, S. F., Hughes, T.: On The Effectiveness of Kolmogorov Complexity Estimation to Discriminate Semantic Types. SFI Workshop: Resilient and Adaptive Defense of Computing Networks 2003, Santa Fe, NM, USA. (2003).
8. Gács, P., Tromp, J. T., Vitányi, P.: Algorithmic Statistics. IEEE Transactions On Information Theory, Vol. 47, No. 6, September 2001, 2443-2463. (2001)
9. Chen, X., Francia, B., Li, M., Mckinnon, B., Seker, A.: Shared Information and Program Plagiarism Detection. IEEE Trans. Information Theory, July 2004, pp. 1545-1550. (2004)
10. Keogh, E., Lonardi, S., Ratanamahatana, C.A.: Towards Parameter-Free Data Mining. In Proceedings of the 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, Seattle, Washington, USA, 206-215. (2004)
11. Campani, C.A.P., Menezes, P.B.: On the Application of Kolmogorov Complexity to the Characterization and Evaluation of Computational Models and Complex Systems. In Proceedings of the Int. Conf. on Imaging Science, Systems and Technology (CISST'04), Las Vegas, Nevada, USA, 63-68. (2004)

12. Veldhuizen, T.: Software Libraries and Their Reuse: Entropy, Kolmogorov Complexity, and Zipf's Law. To be published.
13. Kiczales, G.: Aspect-oriented programming. In Proceedings of 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA. pp. 730. (2005)
14. Ousterhout, J.K.: Scripting: Higher Level Programming for the 21st Century. IEEE Computer 31(3), 23-30. (1998)
15. Štuikys, V., Damaševičius, R.: Metaprogramming Techniques for Designing Embedded Components for Ambient Intelligence. In Basten, T., Geilen, M., de Groot, H. (eds.): Ambient Intelligence: Impact on Embedded System Design. Kluwer Academic Publishers, Boston, 229-250. (2003)
16. Czarnecki, K., Eisenecker U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley. (2000)
17. Štuikys, V., Damaševičius, R., Ziberkas, G.: Open PROMOL: An Experimental Language for Target Program Modification. In Mignotte, A., Villar, E., Spruiell, L.S. (eds.): System-on-Chip Design Languages. Kluwer Academic Publishers. (2002)
18. Mellor, S. J., Clark, A. N., Futagami, T.: Guest Editors' Introduction: Model-Driven Development. IEEE Software 20(5), 14-18. (2003)
19. Batory, D.: Product-Line Architectures. Invited Presentation, Smalltalk and Java in Industry and Practical training, Erfurt, Germany, October, 1998, 1-12. (1998)
20. Dembski, W.A.: Intelligent Design as a Theory of Information. Intervarsity Press. (1999)
21. Edmonds, B.: Syntactic Measures of Complexity. Doctoral Thesis, University of Manchester, Manchester, UK. (1999)
22. Griffiths, T.L., Tenenbaum, J.B.: From algorithmic to subjective randomness. In Advances in Neural Information Processing Systems 16. (2004)
23. Feynman, R.: Feyman Lectures on Computation, Perseus Pub. (1996).
24. Manzini, G.: The Burrows-Wheeler Transform: Theory and Practice. Lecture Notes in Computer Science, Vol. 1672, 34-47. Springer Verlag. (1999).

Robertas Damaševičius graduated at the Faculty of Informatics, Kaunas University of Technology in Kaunas in 1999, where he received a B.Sc. degree. He finished his M.Sc. studies in 2001, and he completed his Ph.D. thesis at the same University in 2005. Currently, he lectures in several computer science and programming courses in Kaunas University of Technology. His research interests include hardware design, generative programming, program generators and metaprogramming. He is the author or co-author of over 25 papers in the area.