EXPLORING CODE PROJECTIONS AS A TOOL FOR CONCERN MANAGEMENT

Ján JUHÁR, Liberios VOKOROKOS
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic,
E-mail: {jan.juhar, liberios.vokorokos}@tuke.sk

ABSTRACT

Comprehending the code is the main activity performed by programmers during programming. Each software system implements many concerns from the problem domain, but their transformation into a source code makes them scattered and hard to find. For this reason, many tools and approaches for working with concerns were created. Evolution of Integrated Development Environments (IDEs) brought a possibility to work with the code on several levels and it became possible to alter the presentation of the code without changing the code itself. Such presentation is referred to as projection*. In this paper we explore possibilities to use projections as a tool for concern management. We present our categorization of different existing approaches for concern management and of projectional tools already available in multiple IDEs. We also review research-originated tools that provide code projections. Based on this review we identify several areas where existing solutions can be improved and propose our concept of configurable projection that can further facilitate the program comprehension process.*

Keywords: *program comprehension, integrated development environments, software concerns, projectional editing*

1. INTRODUCTION

Program comprehension is a cognitive process that involves source code analysis and retrieval of information needed to develop or maintain software systems. Many researchers point out that this process tends to take up to a half of programmers' time during their work with the source code [1, 2, 3]. The main source of hindrance programmers face during this process lies within the wide *semantic gap* that is created between the problem domain and the solution domain. Much information is lost or scattered during the transformation of programmer's mental model of the system into a source code written in a specific programming language.

With the intent to narrow the semantic gap, a whole group of *program comprehension tools* exists with both an industrial and a research origins, either as a stand-alone tools or as parts of an IDE. These are generally considered as helpful in the process of code comprehension (e.g, as reported by Kosar et al. [1] or Storey [3]). However, as Maalej et al. observed, many of such IDE tools are left undiscovered even by professional programmers [2], and, similarly to Damevski et al. [4], they call for *context-aware* tools in development environments.

Although IDEs are the most complete toolsets for working with source code, they do not take into account software concerns related to a task a programmer is working on. Information and views they provide are extracted from the entire software project. Another issue is related to the dominant decomposition of the system. A programmer creating the code is the one who gets to decide on the structure it is going to have. Other programmers working with the code later must use this imposed structure, or, if it does not suit their needs, perform a permanent refactoring (and impose a new structure) [5].

Mainly these issues drive our work towards exploring the idea of *code projections* with connection to a software concerns management. Thanks to the techniques used by modern IDEs that employ parsing of the source code and multi-layered code presentations, it is possible to create various views that "understand" the code structure [6, 7]. If some supporting tool could provide such views of the source code that would present its alternative structure according to chosen concerns without actually restructuring the code stored in files, it may be able to assist a programmer with comprehension tasks.

2. MANAGING SOFTWARE CONCERNS

Modularization of software systems was initiated by attempts to improve their flexibility and comprehensibility and to reduce required development time [8]. Rising abstraction level of programming languages and new programming paradigms were profoundly influential on programmers' ability to express their intentions and to separate some of the concerns into distinct entities provided by a particular language.

Every paradigm is, however, designed with focus on some particular decomposition of concerns (e.g., object-oriented paradigm abstracts data structures) and other concerns do not fit into it cleanly. This results in two main symptoms of insufficient modularity or concern separation [9]: *scattering* of the code related to a single concern and *tangling* of one concern's code with other concerns. And even *concern-focused* programming paradigms specifically designed to capture crosscutting concerns, like *Aspect-oriented programming* or *Language-oriented programming*, do not address the issue of single dominant decomposition. This decomposition is a result of a static source code nature. All of this suggests that languages alone are not sufficient for concern management.

Another take on this problem is present in various approaches and tools supporting concern separation. One of the main goals of these tools is to provide dynamic views of the system's structure built atop the static one created with a language.

The tools take different forms. They are built either as a stand-alone applications, or as extensions of existing pro-

gramming tools (primarily IDEs). Another variable aspect is in the approach itself used to achieve concern separation in a code base. We categorize the existing tools into following three main categories of approaches:

- model-based approaches,
- remodularizing approaches, and
- explicit metadata approaches.

We characterize these types and give examples in the following subsections.

2.1. Model-based Approach

The model-based approaches allow to build an external (with relation to the source code) concern models that connect concerns with related elements of the code base.

As an example of such approach we can present *Concern Graphs* by Robillard et al. [10]. The approach is designed to capture a main code structure that implements a concern, while it abstracts some of the implementation details. A concern graph is created by programmer with the help of *Feature Exploration and Analysis Tool* by iteratively querying a program model and determining which program elements belong to the concern implementation. The main property of the approach is the localization of scattered code contributing to different concerns in a external model.

Robillard et al. conducted a number of case studies that demonstrated usefulness of Concern Graphs for software maintenance tasks. However, due to some abstracted details combined with model granularity at the level of declarations, concerns in highly algorithmic code bases were not expressed sufficiently.

2.2. Remodularizing Approach

A characteristic feature of a remodularizing approach is that it uses manual configuration based on structural patterns and naming conventions to set up a tool that performs code-level remodularization on demand.

An example tool for this category is *Stellation*, a software configuration management (SCM) system for *Eclipse* IDE, presented by Chu-Carroll et al. in [11]. In contrast to most other SCMs that work with the software project at the file level, it uses a finer granularity. Specifically, for Java it uses granularity of class methods and fields. One of the motivations behind such approach is to provide dynamic views on the source code. These views are called *virtual source files* (VSFs) and are constructed from system artifacts with a special query language. With this language a programmer can specify predicates that an artifact must satisfy to be included in a VSF. The queries are also stored in the revision control system. This way a programmer with a detailed knowledge of a concern with scattered code can prepare a VSF that can help other programmers to comprehend it. However, to support external tools, which work with files, the system must provide exporting and importing facility.

2.3. Explicit Metadata-based Approach

Approaches classified in this category use metadata present in the source code to explicitly assign high-level concepts to program elements.

Annotation of program elements with additional metadata that associate these elements with concerns they pertain to is the main idea of the *Concern annotations* approach [12]. It requires a host programming language that supports annotating program elements (e.g, as do Java with annotations or .NET with attributes).

Annotations are typically used as a configuration facility for various tools. With concern-oriented annotations, the focus is on program comprehension, as individual annotations represent mapping between high-level concept (preserved in annotation's name) and annotated program elements. Such concern annotations can be combined with appropriate tool to achieve on-demand concern separation by dynamically creating code views based on these annotations.

2.4. The Role of Dynamic Views

We consider the remodularizing approach as the least practical, because it breaks compatibility with traditional file-based tools, like compilers or version control systems. Another drawback is the requirement to configure concerns through structural and naming conventions, which may present an issue mainly in already existing code that does not follow such conventions.

On the other hand, dynamically created views possible with model-based and metadata-based approaches do not have these issues. The code of a system is not directly modified in order to display concern-oriented view, as only the presentation level is changed with the help of an appropriate tool. Moreover, they are able to portray higher-level concerns present in the system implementation. The generalized idea of such views, called *code projections*, is explored in the next section.

3. UTILIZATION OF PROJECTIONAL TOOLS

The idea behind so-called *projectional editing* was described by Fowler in [6]. He based the notion about this approach on solutions he named *Language Workbenches*, like *Intentional Domain Workbench*[1] or JetBrains *Meta Programming System*[2], and also on mainstream modern IDEs, like JetBrains *IntelliJ IDEA*.

Projections are well known in software engineering from graphical modeling tools (e.g., UML or entity-relational diagrams): users are presented with graphical representation, or *projection*, of an underlying structure of a model they are creating. Projectional editing reuses the idea and creates an alternative to direct source code editing with manipulation of a base system definition through source code projections.

[1] http://www.intentsoft.com/intentional-technology/intentional-platform/
[2] https://www.jetbrains.com/mps/

Fowler recognizes five system representations in a development environment that supports code projections. They are shown in Fig. 1 [6]. *Executable* (e.g., compiled) and *storage* representations are the usual ones. But *abstract* representation is used for manipulation and reasoning about the system definition and can be projected for a programmer as *editable* or non-editable (*visualization*) view.

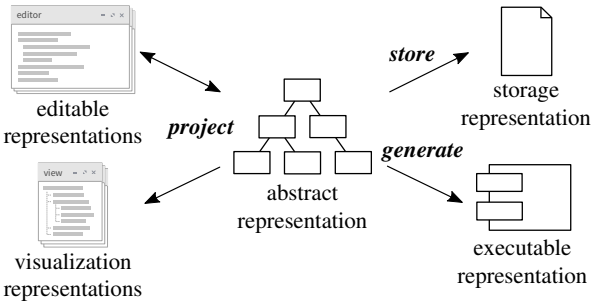

Fig. 1 Multiple system representations in projectional editing

Language Workbenches (LWs) are tools for developing domain-specific languages with tailored syntaxes and supporting IDE tools. Projectional LW do not require created language grammars to be parsable, because code editor works directly with an abstract representation of the code—its abstract syntax tree (AST). The different user experience and tool interoperability issues of such editors hinder their wider adoption, but advances are being made in this direction nonetheless, as reported by Voelter et al. [13].

As our work deals with software concerns and not with DSLs specifically, we looked more closely at the other category of existing projections, that is available in traditional modern IDEs.

3.1. Projections in Parser-based IDEs

The term "IDE" covers various development tools, but projections are available in those that use parsers to create an abstract code representation and to gain structural information about the code. This parser-based approach is present in a significant number of today's IDEs, like *Eclipse*, *IntelliJ IDEA*, *Netbeans* or *Visual Studio*.

For example, *IntelliJ IDEA* uses custom lexers and parsers to convert source code into a special form of AST called *Program Source Interface* (PSI) tree [7]. PSI tree preserves even the white spaces used in original source code so the editor can project the tree in a form that is an exact representation of the originally parsed source code.

Referring back to the Fowler's categorization of system representations, both editable and visualization ones exist also in parser-based IDEs.

Editable projections are those that augment or alter presentation layer of the code in IDE editors. We identified three types of editable projections. First, *source code projections* provide projectional features directly in source code editor. Features like code highlighting, folding or inlining belong here. Second, *modeling projections* project code as editable diagrams that are kept synchronized with the code they represent. Advanced example of such projection is present in *IntelliJ IDEA*, where a programmer can display classes as UML class diagrams and edit them (although with some limitations) in this graphical form. And third type, *domain-specific projections*, take advantage of common application structures, conventional configuration and domain-specific APIs in order to aid in development with supported frameworks.

As for the visualization projections, their content cannot be directly altered. They only reflect changes made to the abstract program representation through some other editable view. These projections usually provide concentrated views on some system property. Again, there is the *source code projection* type in this category. It includes common IDE features, like code completion, or more unique ones, like *CodeLens* in *Visual Studio* that provides a quick overview of a method's usage throughout a project. The new is the *structure projection* type that includes primarily various views showing tree structure of a project. But not only at the file level; they may include also classes, their methods and fields, and even the visibility modifiers.

Apart from these IDE features, there are also tools by which researchers extend IDEs with another projectional capabilities. Dvais and Kiczales created *Registration-based abstraction* editor [14] for *Eclipse* to enable creation of new language construct on the view layer. A functionally similar concept for language enrichment environment that includes projectional editing is presented in the work of Chodarev et al. [15]. It uses pattern recognition to suggest recurring code patterns that could be abstracted into a new language constructs. And a whole framework for extending *Eclipse* with multi-perspective software visualizations is described by Carneiro and Neto in [16]. They use a traditional data visualization process (comprising data collecting, filtering, and transformation steps) adjusted to the needs for displaying various source code dependencies in graphical form, e.g., treemap[3] version of project structure.

3.2. Concern-oriented Projections

The main purpose of projections described in the previous subsection is to facilitate program comprehension of implementation-level concerns. As such, their construction does not require any other information than those obtainable from the source code itself. This ensures that tools using these projections are usable without any additional effort from a programmer. On the other hand, semantic gap between problem and solution domains remains wider.

Concern-oriented projections are focused on higher-level concerns. Such projections require some additional metadata bound to the source code that capture these concerns. A particular projection can be created with a *projection query* specifying relevant concerns or incrementally if creation of final projection requires multiple actions.

Taking into account different format of concerns, Nosáľ [12] recognizes the following three types of concern-oriented projections, listed in the order of increasing abstraction level:

[3]Treemap is a 2D visualization of a tree structure using recursively nested rectangles.

- *Source intrinsic projections* that are based on the analysis of the system's source code. Most IDE projections belong here.
- *Configuration-based projections* that are based either on explicit configuration or on code conventions. These projections can be found in IDEs that provide specific support for some framework or tool.
- *Annotation-based projections* that require metadata explicitly embedded in the source code. Annotations are created solely for the purpose of building a projection.

3.3. Editors for Concern-oriented Projections

In the last five years a number of experimental code editors that can be categorized under concern-oriented projections were presented in the research community. All of them break the traditional file-based editor paradigm in which one editor displays content of a single file.

Sieve Source Code Editor (SSCE) [12] is a tool that complements the approach of annotation-based projections with a possibility to build a kind of a virtual file comprising all the program elements annotated with selected annotations. Experimental evaluation showed that the projection was beneficial for navigating around an unknown code.

There is also a group of canvas-based editors. *Code Bubbles* [17] for *Eclipse* allows to place editable code fragments (*bubbles*) onto a 2D pannable canvas. Similar is the *Code Canvas* [18] extension for *Visual Studio*. In case of both of these tools, bubbles can contain classes or individual methods and related bubbles are interconnected with arrows pointing from a method call to its definition. A simpler approach presents the *Patchworks* code editor [19] that restrict the free-form placement of code fragments into a two-row horizontal strip of positionally fixed editors (*patches*). Evaluations of these editors show that programmers can leverage their spatial memory for quicker navigation (which was most obvious for *Patchworks* where placements of fragments were constrained) and that custom code layouts help comprehend unfamiliar code.

All these alternative editors try to solve the issue of organizing working environment for specific task without the limitations imposed by used storage model. Although canvas-based solutions still work only with source-intrinsic information, we can view the layout of code fragments created by a programmer as "concern-related metadata". Problems however arise when we consider reusing such layout for other tasks as it has no explicit semantic meaning.

3.4. Areas for Possible Improvements

So far we have presented our review of approaches and tools that were designed to facilitate program comprehension through dynamic code views. Considering the overall usability of these tools we see that there are issues regarding the following three areas:

- *Configurability*: it is not possible to configure development environment to provide a focused view on a specific concern or a set of concerns.
- *Composability*: development environments are missing an ability to build concern-oriented projections based on a composition of concern metadata of various sources and types.
- *Reusability*: primarily the projections of canvas-based editors are built for specific tasks only and are hard to reuse for different tasks.

The likely cause behind the first two areas is the fact that the concerned projectional tools were developed in isolation, designed to cope with a particular problem. This is not a problem of these individual tools, but rather of IDEs that do not provide common means to work with custom metadata [5]. On the other hand, the reusability issue is more specific to the design of canvas-based editors.

Thus, for our next work, we want to focus primarily on the composability, as we consider it to have a potential to advance the overall usability of concern-oriented projections.

4. TOWARDS CONCERN-ORIENTED DEVELOPMENT ENVIRONMENTS

In our next work we plan to design an IDE plug-in that will serve two main purposes. First, it will provide a common platform for working with concerns in some way associated with the source code. Here we do not plan to create a method for extracting concerns, but rather reuse existing concern extraction methods. And second, it will provide a projection that will use this platform to create configurable alternative view of the source code. In the following we discuss individual aspects of our preliminary concept for the plug-in.

4.1. Extension of IDE Code Model

In order to process concern-related metadata of various origins and forms, we will need to transform these metadata into a universal structure. As IDE is parsing the code into an AST that is used as a model driving many of its views, we see as one of the viable options to augment this AST with the required metadata. In such a design, each AST node representing an element with associated concerns would receive a reference to metadata nodes of these concerns.

Considering multiple sources of concern-related metadata (e.g., code annotations, structured comments, outputs of feature location tools, and others), the augmentation can be performed through a series of steps. In each step a transformation specific to a particular form of metadata can take place. A projection built upon such augmented code model could cover multiple projection types recognized by Nosáľ. Moreover, by allowing to configure concern source selection, a variable concern abstraction level could be achieved.

4.2. A Question of Concern Granularity Level

With regard to the source code projection we plan to build on the basis of the concern-augmented code model, an important property is the granularity level of program elements at which the concerns can be preserved. All the tools reviewed in section 2 use granularity of declarations (i.e.,

classes, class methods and fields). We also mentioned experiments showing that in some cases it was not sufficient. Except for some automated feature location tools from the survey of Dit et al. [20], no finer granularities were used and we found no studies explicitly evaluating them.

To find out which granularities would be used by programmers to capture concerns in the code if the used tool had no granularity restrictions we conducted a case study[4] with 5 participants who tagged concerns in a known code base with our simple tagging tool. Participants *C1* and *C2* used code written in C language, participants *J1* and *J2* in Java, and participant *P* used Python.

The resulting distribution of concern granularity levels among tagged code fragments is shown in Fig. 2. We can see that each participant used a significant amount of tags covering a statement or a group of consecutive statements inside methods. And 19 out of 85 identified concerns had no tags on a coarser level than statement.

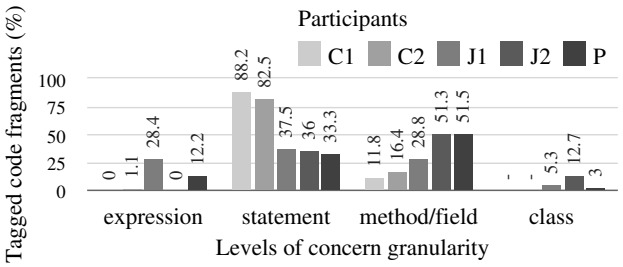

Fig. 2 Distribution of concern granularity levels among tagged fragments

Even though this was only a preliminary study, its results indicate that the usage of sub-method concern granularities should be considered and further evaluated. On the other hand, the expression level seems not to be used significantly enough to outweigh an increased cost of capturing concerns at this fine level.

4.3. Preserving Context of the Projected Code

SSCE provides a *virtual file* projection with the granularity level constrained to elements that can be annotated in Java. When the projection is constructed, the individual annotated methods are "sieved" into a single editor. Additionally, non-editable markings are added to describe the code fragment original location (e.g, file, class). Similarly the canvas-based editors add context headers to each code fragment. This is important to give a programmer required context of a code fragment to prevent ambiguity.

In case of projecting at finer, sub-method granularity levels, the question of the context is even more important. If we consider a virtual file combining fragments of multiple methods, an inadequate surrounding context can easily lead to wrong interpretation of the code by programmers. To solve this issue, we can see a relatively simple solution in projecting a whole method with additional highlighting of related, respectively folding of unrelated code fragments inside method bodies. Design of such projection will need to be thoroughly evaluated for its comprehensibility and usefulness.

[4] More details on the case study can be found in [21].

5. CONCLUDING REMARKS

As we described in the previous section, our next work in the area of projectional code editing will focus on a design and evaluation of an editable projection based on the concept of *virtual files*.

Regarding the evaluation, we are particularly interested in finding answers to two research questions. First, will a projection based on a combination of multiple concern-related metadata make the program comprehension more effective than individual methods that created these metadata? And second, will a code projection with a sub-method concern granularity and additional provided context provide benefits over a projection with coarser granularity?

Answering these questions through experiments with our proposed projection implementation can provide more understanding on effectiveness of parser-based projectional tools for program comprehension tasks.

ACKNOWLEDGEMENT

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10.

REFERENCES

[1] KOSAR, T. – MERNIK, M.: The impact of tools supported in integrated-development environments on program comprehension, 3rd International Conference on Information Technology Interfaces (ITI'11), 2011, pp. 603-608.

[2] MAALEJ, W. – TIARKS, R. – ROEHM, T. – KOSCHKE, R.: On the Comprehensioin of Program Comprehension, ACM Transactions on Software Engineering and Methodology (TOSEM), 23, No. 4, 2014, pp. 31:1-31:37.

[3] STOREY, M. A.: Theories, Methods and Tools in Program Comprehension: Past, Present and Future, IEEE 13th International Workshop on Program Comprehension (IWPC'05), 2005, pp. 181-191.

[4] DAMEVSKI, K. – SHEPHERD, D. – POLLOCK, L.: A field study of how developers locate features in source code, Empirical Software Engineering, 21, No. 2, 2015, pp. 724-747.

[5] NOSÁĽ, M. – PORUBÄN, J. – NOSÁĽ, M.: Concern-oriented source code projections, Proceedings of the 2013 Federated Conference on Computer Science and Information Systems (FedCSIS). 2013, pp. 1541-1544.

[6] FOWLER, M.: Projectional Editing, 2008, `http://www.martinfowler.com/bliki/ProjectionalEditing.html`.

[7] JEMEROV, D.: Implementing refactorings in IntelliJ IDEA, Proceedings of the 2nd Workshop on Refactoring Tools (WRT'08), 2008, pp. 1-2.

[8] PARNAS, D. L.: On the criteria to be used in decomposing systems into modules, Communications of the ACM, 15, No. 2, 1972, pp. 1053-1058.

[9] HANNEMANN, J. – KICZALES, G.: Overcoming the prevalent decomposition in legacy code, Proceedings of the ICSE Workshop on Advanced Separation of Concerns, 2001.

[10] ROBILLARD, M. P. – MURPHY, G. C.: Concern graphs: finding and describing concerns using structural program dependencies, International Conference on Software Engineering (ICSE'02), 2002, pp. 406-416.

[11] CHU-CARROLL, M. C. – WRIGHT, J. – YING, A. T. T.: Visual separation of concerns through multidimensional program storage, Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03), 2003, pp. 188-197.

[12] NOSÁĽ, M.: Leveraging Program Comprehension with Concern-oriented Projections, PhD thesis, Technical University of Košice, 2015.

[13] VOELTER, M. – SIEGMUND, J. – BERGER, T. – KOLB, B.: Towards User-Friendly Projectional Editors, Software Language Engineering, Lecture Notes in Computer Science, Vol. 8706, chap. 3, 2014, pp. 41-61.

[14] DAVIS, S. – KICZALES, G.: Registration-based language abstractions, ACM SIGPLAN Notices, 45, No. 10, 2010, pp. 754-773.

[15] CHODAREV, S. – PIETRIKOVÁ, E. – KOLLÁR, J.: Towards Automated Program Abstraction and Language Enrichment, 2nd Symposium on Languages, Applications and Technologies (SLATE), 2013, pp. 51-64.

[16] CARNEIRO, G. D. F. – NETO, M. G. D. M.: SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment, Enterprise Information Systems, Lecture Notes in Business Information Processing, Vol. 190, 2014, pp. 242-263.

[17] BRAGDON, A. et al.: Code bubbles: a working set-based interface for code understanding and maintenance, Proceedings of the 28th international conference on Human factors in computing systems (CHI'10), 2010, pp. 2503-2512.

[18] DELINE, R. – ROWAN, K.: Code Canvas: Zooming towards Better Development Environments, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10), 2010.

[19] HENLEY, A. Z. – FLEMING, S. D.: The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes, Proceedings of the 32nd annual ACM conference on Human factors in computing systems (CHI'14), 2014, pp. 2511-2520.

[20] DIT, B. – REVELLE, M. – GETHERS, M. – POSHYVANYK, D.: Feature location in source code: a taxonomy and survey, Journal of Software: Evolution and Process, 25, No. 1, 2013, pp. 53-95.

[21] JUHÁR, J. – VOKOROKOS, L.: Separation of Concerns and Concern Granularity in Source Code, Informatics'2015: IEEE 13th International Conference on Informatics, 2015, pp. 139-144.

Received July 1, 2016, accepted September 20, 2016

BIOGRAPHIES

Ján Juhár was born in 1989. In 2014 received his MSc. with distinction at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at the Technical University of Košice. Currently, he is PhD student at the Department of Computers and Informatics at the Techincal university of Košice. His research focuses on program comprehension, programming tools, and projectional editors.

Liberios Vokorokos was born in 1966 in Greece. In 1991 he received his MSc. with honours at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at the Technical University of Košice. He defended his PhD. in the field of programming device and systems in 2000. Since 1995 he has been working for the Department of Computers and Informatics, Technical University of Košice. In 2005 he was appointed as Full Professor in Computer Science and Informatics. Currently he holds the position of Dean at the Faculty of Electrical Engineering and Informatics.