EXPOSING RUNTIME INFORMATION THROUGH SOURCE CODE ANNOTATIONS

Matúš SULÍR, Jaroslav PORUBÄN
Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, E-mail: {matus.sulir, jaroslav.poruban}@tuke.sk

ABSTRACT

Many tools support exploration and analysis of various run-time properties of programs. However, the results produced by these tools are often discarded or stored in custom external files. We discuss an approach where data obtained when executing the program are written as Java annotations over program elements. First, the viability of this approach is demonstrated by a feature annotator, which assigns features to corresponding methods. Second, we describe other types of run-time information that we consider suitable for inclusion in the source code. We also suggest two possible workflows how generated annotations can be used in the development process.

Keywords: *metadata, runtime information, dynamic analysis, feature location, program comprehension*

1. INTRODUCTION

When developers try to understand a program, two most used sources of information are the source code and the executable application. They read the source code in an integrated development environment (IDE) and try to mentally connect the code to the behavior obtained from the running application.

There exists a wide variety of standalone tools, IDE features and extensions to support the inspection of run-time behavior of a program – from simple debugging tools to sophisticated dynamic analysis software. The inspected data include:

- performance data (from profilers),
- run-time callers, called methods and parameter types [1],
- runtime metrics [2],
- mappings from features to code fragments obtained by dynamic analysis [3],
- snapshots of graphical user interfaces (GUIs) [4, 5].

1.1. Motivation

The mentioned tools usually fall into one of two categories:

- The information is displayed while the program is running, and then discarded.
- The collected data are stored in a file separate from the source code.

In the first case, the program must be re-run with the same input each time a developer needs to access the runtime information.

In the second case, each piece of information must link to the corresponding source code element. As the source code changes through time, the references may be no longer valid if the tool managing them is not notified about the modification. For example, if we refer to a source code element using its file name and line number, and a few lines are inserted above the element, the reference becomes invalid. Several approaches exist to cope with the problem of source location tracking [6]. Nevertheless, none of them works flawlessly: Even some of the most precise methods correctly refer to only roughly 90% of identifiers after source code changes [6].

In both cases, there is one more negative aspect. Either the tool is completely separate from the IDE, coercing the developer to switch between two views; or it is dependent on a specific IDE.

If the tool wrote the produced data directly into source code, it could be IDE-independent, while the developers could use their favorite IDE to view the data.

1.2. Goal

This paper discusses the approach of (semi-)automated source code annotation, using data retrieved from a running program.

Java annotations (or attributes in C#) are a form of metaprogramming, marking program elements with additional metadata. Traditionally, they are written manually by programmers. Then, these annotations are programatically read during compilation by annotation processors, or at runtime – using reflection. Consider this example:

```
@Min(3)
@Max(120)
private int age;
```

The programmer is giving "hints" to a system that the age has valid values from 3 to 120. The system can then use these manually supplied information at runtime – to perform data validation.

We hypothesize that an opposite of the traditional process is possible: Annotations are written into the source code automatically (using runtime information), and read later by programmers to aid program comprehension. For instance, during profiling, the methods taking the majority of time during execution ("hotspots") could be automatically marked with an annotation:

```
@HotSpot(timePercent=27.8)
public double processData(double[] data) {
```

```
  ...
}
```

We will try to demonstrate the idea, first on feature location using dynamic analysis, then on various smaller examples.

2. SEMI-AUTOMATED FEATURE ANNOTATION

Software systems consist of a multitude of features. For example, in a diagramming application, they might include shape drawing, erasing, moving; file import and export, etc. When writing programs in traditional object-oriented programming languages, the implementation of a feature is scattered across multiple classes and methods [7]. During the program comprehension and maintenance phrase, it is often necessary to find all or at least some code pertaining to a particular feature [8].

In our approach, each feature will be represented by one annotation type – e.g., `Erasing` for the "erasing feature". The goal is to have annotations (without parameters) above appropriate methods in the source code.

2.1. Differential Code Coverage

We will use a very simple dynamic feature location technique named *differential code coverage* [9] (also called software reconnaissance [10]).

The process is as follows. For each feature, the program is run twice: the first time utilizing the feature of interest, the second time not utilizing it. E.g., for the "erasing" feature of a diagramming application, the first time a shape is drawn and erased, while the second time it is drawn and left on the canvas.

During the runs, sets of all executed methods are recorded. A set of methods pertaining to the feature is calculated as $M_1 - M_2$, where M_1 is the set of methods executed in the first run, M_2 in the second one.

2.2. Semi-automated Annotation

In [11], we used differential code coverage to semi-automatically annotate methods in the source code with its corresponding features.

The input of our preliminary implementation, AutoAnnot, is plain, unannotated source code, and a list of features (annotation types).

For each feature in a supplied list, the developer runs the program twice, and interacts with it – by clicking the appropriate menus, entering data, etc. – according to the rules described in the previous section. This interactions can be optionally replaced by pairs of fully automated tests, if available.

Our implementation, AutoAnnot, collects the lists of executed methods, using BTrace [1]. The set of methods related to each feature is computed using differential code coverage. Above each such method, an annotation is inserted into the original source file.

For example, this would be an excerpt from the resulting annotated source code of a diagramming application:

```
@Drawing
@Erasing
public void changeTool(Tool tool) {
  ...
}

@Erasing
public void clearRegion(Shape region) {
  ...
}
```

2.3. Comparison with Manual Annotation

We demonstrate the viability of this approach on a small (2500 lines of code) desktop Java application for bibliographic note-taking, EasyNotes [2]. We seek to answer the following research question: To what degree do the results of manual and semi-automatic annotation of the same project overlap?

2.3.1. Method

To obtain manually annotated source code to be used as a baseline for comparison, we used the following process.

First, a list of features in the application was drawn from the results of our previous experiment [12]. Among other activities, seven participants were asked to create one annotation type for each concern (i.e., a feature or other intention behind pieces of code) they recognized in the application. The annotation types were then merged by a researcher, grouping the same concerns expressed by different terms into one type. Only annotation types recognized by at least two participants were preserved. Next, we removed annotation types representing non-functional concerns and maintenance notes (like "domain entity" or "unused code"), preserving 9 annotation types representing features visible to end users [11]. For a list, see Table 1.

The author of the EasyNotes application was asked to mark methods in the source code with feature annotations (from Table 1) according to his own discretion, thus producing manually annotated source code.

Using the list of the mentioned 9 features, the AutoAnnot process was executed by a researcher on clean source code, producing semi-automatically annotated code [11].

2.3.2. Results

For each annotation type (feature), we calculated the portion of overlapping annotation occurrences:

$$\frac{M_{manual} \cap M_{auto}}{M_{manual} \cup M_{auto}}$$

where M_{manual} is a set of methods manually annotated with a particular feature, M_{auto} a set of semi-automatically annotated ones. We can see the results in Table 1. The "Total" row is computed analogically, but using pairs $(feature, method)$ as set elements instead of just methods.

[1] `http://kenai.com/projects/btrace`

[2] `http://github.com/MilanNosal/easy-notes`

Table 1 The features and their overlap between manual and semi-automated annotations

Feature (annotation type)	Overlap
Citing	12.50%
Filtering	19.23%
Links	21.74%
NoteAdding	25.00%
NoteDeleting	66.67%
NoteEditing	11.76%
NotesLoading	38.46%
NotesSaving	41.67%
Tagging	6.25%
Total	22.22%

2.3.3. Discussion

The overall overlap was 22.22%. There are multiple reasons why this number is relatively low.

Often it is difficult to avoid execution of feature-unrelated methods in the second run. For instance, the method `Note.getTags` is called as soon as a list of of notes is shown, even if the notes do not contain any tags [11].

Similarly, a feature-related action may sometimes be missed in the first run. The `Note.isUsed` method is an example. The EasyNotes program contains a filtering drop-down list with an item called "not used". This filter calls the method `Note.isUsed`, which, in turn, determines whether a note does not have a "new" tag [11]. Therefore, it is only indirectly related to the "tagging" feature.

Differential code coverage is a simple approach, which has its flaws. Due to its nature, it recognizes only task-specific code, i.e., not general utility methods used in many features. More sophisticated feature location approaches were devised over time [3]. Differential code coverage was selected only for demonstration purposes – it could be replaced by any other method using dynamic (or static) analysis. Our goal was not to compete with state-of-art feature location techniques, but rather to show how the results produced by a feature location tool can be in principle written to the source code in a form of annotations.

3. OTHER RUNTIME INFORMATION

While initially intended only for feature tagging, our approach is not limited to it. We will now explore the possibilities (although currently not implemented) of automated annotation of source code using runtime-originated information.

3.1. Profiling

Output of a profiler is often used by a single programmer, not shared among developers. Hotspots, i.e., the methods which are executed for the longest time according to the profiler, could be automatically marked with the `HotSpot` annotation. The parameters might include the time percentage and an optional programmer-supplied use case name:

```
@HotSpot(selfTimePercent=32, useCase="Order")
public BigDecimal getPrice() {
  ...
}
```

The method can be then optimized by another member of the team. Re-running a tool would automatically remove the annotation if it was no longer a hotspot. Alternatively, it could be removed manually when fixing a performance-related bug.

3.2. Run-time Call Graph

A call graph consists of methods as nodes, and all possible calls between them as directed edges. While a static call graph can be generated from the source code itself, it does not distinguish whether a particular call was really executed. A dynamic call graph contains just actual method calls for a particular execution. We could annotate methods with their direct callers:

```
@Caller(clazz=Order.class, method="getTotal")
@Caller(clazz=Cart.class, method="add")
public BigDecimal getPrice() {
  ...
}
```

Call graph exploration tools were shown to improve maintenance efficiency [13]. In our case, the exploration itself would require no additional tool except a standard IDE. However, sharing between developers would have a limited use.

3.3. Dynamic Type Information

Due to polymorphism, a method with a parameter (or a return value) of a given abstract type can accept (or return) an object of any inherited concrete type. However, while reading the source code in an IDE, only the abstract type is apparent to the developer [1]. Suppose we have a system with a complicated shape hierarchy, containing many abstract and concrete classes. Consider the following example:

```
public void clearRegion(Shape region) {
  ...
}
```

By looking at the method, the developer has no idea what types of regions were cleared during the testing of an application. Using a debugger, it is possible to find the concrete types. However, such process is laborious and the data are discarded as soon as the debugging session ends. The list of concrete classes could be determined by a tool, which would annotate the method (using Java 8 parameter annotations):

```
public void clearRegion(
 @Types({Square.class, Dot.class})
 Shape region
```

```
) {
  ...
}
```

3.4. Faults and Flaws

There exist many tools for automatic or semi-automatic detection of potential faults, resource leaks or security flaws. However, the results of their analysis, which could include human work, are often thrown away after the programming session ends.

We propose that such tools should annotate the source code with the found flaws. For example, if a monitoring tool determines that in the moment of program termination, a particular file (represented by a member variable) was not yet closed, it could annotate it:

```
@ResourceLeak(Leak.FileNotClosed)
FileReader inputFile;
```

Thanks to this, the flaw remains permanently noted until it is fixed – when it is removed either by a programmer or by the automated tool. Furthermore, all team members working with this piece of code become aware of it.

4. WORKFLOW

The annotated source code can be checked into a version control system, and thus shared with the whole development team. This improves awareness about non-functional software properties like performance and security, which are otherwise not obvious by looking at the clean code.

However, not all dynamic data are suitable to be shared across the whole team. They could cause unnecessary pollution and problems when merging the code. For this reason, we devised two types of scenarios: a local-only and shared workflow.

4.1. Local-Only Workflow

Before performing an enhancement or a bug-fix requiring extensive comprehension of the source code, the developer executes a tool, which annotates the source code according to collected runtime information.

Once the annotations are written, all standard tools which can be used on annotations are applicable. For example, the Find Usages capability of an IDE may be used to find all hotspots or potential flaws in the code. It is possible to use standard mouse and keyboard navigation (e.g., Ctrl+click) on the class names in the caller lists or concrete type annotations. Each flaw can have its documentation, displayed in a tooltip above the flaw type.

After the developer finishes the work, the tool will delete all annotations previously inserted into the source code. This way, unnecessary code pollution and merging problems are prevented.

An advantage of this approach is that the tool is completely independent of a specific IDE. At the same time, the developer can utilize an IDE of his choice for comprehension, without switching between it and a separate comprehension tool.

4.2. Shared Workflow

When using a shared workflow, the tool is executed and the IDE is then used in the same way as in the local-only workflow. However, the annotations are not deleted after the issue is resolved. The annotated source code is checked into a version control system, where also other team members can see it.

Although the annotations can be helpful to the team, they can cause merging problems and make a version control log hard to read. Furthermore, once the data are written, they can become obsolete as the code changes.

4.3. Workflow Selection

For a specific kind of annotations, it is necessary to decide which workflow to use.

In our EasyNotes study, semi-automated feature annotation marked each method with only 0.43 annotations on average. Although in more realistic use cases, the number is expected to grow, it should not affect readability significantly. In controlled experiments [14], we showed that feature (and other concern) annotations are useful when shared with other developers – they improve program comprehension efficiency. Nevertheless, keeping annotations synchronized with source code changes could be challenging.

Profiling data can be easily trimmed – for example, we can limit the annotation process to 3 most resource-hungry methods. In this case, they are suitable for sharing. After performance is improved, the annotations can be removed by the programmer committing the fix.

A number of methods callers tends to be large: even in a small-scale application like EasyNotes, there are more than 160 method calls within the project itself (determined by Eclipse's references search using static analysis). However, it is possible to limit the amount of annotations using dynamic analysis and restricting to a specific use case the programmer is trying to debug. Therefore, we suggest to use call graph annotations only in the local workflow. A similar recommendation holds for dynamic type information annotations.

Faults and flaws should be shared in a version control system to inform all developers about them. If a specific flaw is fixed, the annotation should be removed in the same commit. We can consider this a lightweight issue tracking approach for small issues tightly bound to a specific piece of source code.

5. RELATED WORK

Now we will present related works.

5.1. Annotations

Concern annotations [12, 14] are annotations acting as hints for developers, informing them about intentions behind a code element. Originally, they were intended to be inserted manually into the source code, preferably by programmers writing the code. However, as any form of "documentation" without any direct effect on the application behavior, we can expect developers to ignore writing them, as

it is time-consuming. For this reason, we see potential in automated or semi-automated concern annotation.

Joy et al. [15] presented an approach of automatic C source code back-annotation using information obtained at runtime. However, they collected only timing and power information, which is later used in embedded software simulation. Therefore, their annotations are not intended for human consumption and program comprehension, as in our approach.

The built-in Java annotation model has two drawbacks. First, only a selected subset of elements (most notably, classes, fields and methods) can be annotated. @Java [16] is a Java language extension supporting annotations over statements and expressions. Second, expressing the constraints above annotations is very limited – e.g., we cannot restrict an annotation type to be applicable only to private final methods. Ann [17] is a domain-specific language enabling such advanced constraints.

5.2. Exposing Runtime Information

Senseo [1] displays various information like a number of invoked methods or created objects, callers, callees and concrete argument types, in an IDE window. However, since Senseo is an Eclipse plugin, it is IDE-dependent.

An in-situ profiler [18] displays small visuals next to method declarations and calls, presenting performance data obtained by profiling. The results are not persisted in any way, though.

5.3. Workspace Sharing

Code Bubbles [19] is an unconventional development environment. Among other features, it supports sharing parts of source code annotated with values of variables obtained by debugging, notes, image flags, etc. While the presentation is visually attractive, a specialized tool needs to be used by all developers to utilize such features. In the case of Java annotations, any standard IDE can be used.

TagSEA [20] is an IDE extension enabling "social tagging". The developers can insert specially-formatted comments into source code. Such marks are then shared in the development team. There are two shortcomings: First, the tags in comments are not a part of Java syntax, thus requiring a special tool support to be processed. Second, they are inserted into source code manually by programmers.

5.4. Feature Mapping

A commonly used technique to assign features to source code fragments are `#ifdef` preprocessor macros in C and C++. They enable variability [21]: when a particular feature is configured to be included in the produced software, the implementing code is compiled, otherwise not. Although the C preprocessor is language-independent [21] and can be in theory used also for Java and C#, it rarely is in practice. Besides, our goal was not to support feature variability via feature-oriented software product lines [22], but rather to ease program comprehension.

The CIDE tool [23] enables annotation of source code elements with features, visualizing them with colors, hiding irrelevant features, and exporting parts into modules. In contrast to our approach, it maps only features, not various run-time hints, to parts of source code. Furthermore, they save the mappings to external files, while we overwrite the source code itself.

Ji et al. [24] assessed the cost of manually annotating features in source code via specially formatted comments. According to the study they performed, the cost of adding and maintaining such annotations is small compared to the cost of the whole development process.

6. CONCLUSION AND FUTURE WORK

We presented an approach of (semi-)automated source code annotation with data obtained during execution of a program.

First, we focused on semi-automated annotation of program elements with their corresponding features, using a technique of differential code coverage. We demonstrated the viability of our approach and compared the manually annotated code with the results of the semi-automated process.

Next, we discussed the types of dynamic information which are potentially suitable for exposure in source code.

Regarding future work, an experimental implementation of some of the mentioned annotators (except the feature annotator, which is already implemented) is necessary. Then, we could proceed to evaluation of usefulness of this approach, via use cases or controlled experiments.

An interesting future extension is creation of a framework for IDE-independent program comprehension tools. Various plugins would feed the framework with program comprehension hints. The framework itself would annotate the source code using the supplied information. A programmer could then used any IDE and its already supported features to utilize the annotations. Finally, the framework would clean up the annotations.

Although in this paper we are focused on runtime information, the approach can be extended to map practically any machine-produced information to a piece of code – output from static analysis [25], build log messages [26], etc.

Since annotations can be programatically queried [27], by annotating the source code of a program, we could also ease querying of run-time information associated with source code elements.

ACKNOWLEDGEMENT

Thanks to the EasyNotes expert, Milan Nosáľ, for producing a manually annotated version of the source code.

This work was supported by project KEGA No. 047TUKE-4/2016 Integrating software processes into the teaching of programming. This work was also supported by the FEI TUKE Grant no. FEI-2015-23 Pattern based domain-specific language development.

REFERENCES

[1] RÖTHLISBERGER, D. et al.: Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks, *IEEE Transactions*

on Software Engineering, Vol. 38, No. 3, 2012, pp. 579–591.

[2] BECK, F. – HOLLERICH, F. – DIEHL, S. – WEISKOPF, D.: Visual monitoring of numeric variables embedded in source code, *First IEEE Working Conference on Software Visualization, VISSOFT 2013*, 2013, pp. 1–4.

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

[4] BAČÍKOVÁ, M. – PORUBÄN, J. – LAKATOŠ, D.: Defining domain language of graphical user interfaces, *2nd Symposium on Languages, Applications and Technologies, SLATE'13*, 2013, pp. 187–202.

[5] BAČÍKOVÁ, M.: Domain analysis of graphical user interfaces of software systems, *Information Sciences and Technologies, Bulletin of the ACM Slovakia Chapter*, Vol. 6, No. 4, 2014, pp. 17–23.

[6] REISS, S. P.: Tracking source locations, *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, 2008, pp. 11–20.

[7] SULÍR, M.: Program comprehension: A short literature review, *15th Scientific Conference of Young Researchers, SCYR 2015*, 2015, pp. 283–286.

[8] SULÍR, M. – PORUBÄN, J.: Locating user interface concepts in source code, *5th Symposium on Languages, Applications and Technologies, SLATE'16*, 2016, pp. 6:1–6:9.

[9] SHERWOOD, K. D. – MURPHY, G. C.: Reducing code navigation effort with differential code coverage, *Technical report, Department of Computer Science, University of British Columbia*, 2008.

[10] WILDE, N. – CASEY, C.: Early field experience with the software reconnaissance technique for program comprehension, *International Conference on Software Maintenance, ICSM 1996*, 1996, pp. 312–318.

[11] SULÍR, M. – PORUBÄN, J.: Semi-automatic concern annotation using differential code coverage, *2015 IEEE 13th International Scientific Conference on Informatics*, 2015, pp. 258–262.

[12] SULÍR, M. – NOSÁĽ, M.: Sharing developers' mental models through source code annotations, *Federated Conference on Computer Science and Information Systems, FedCSIS 2015*, 2015, pp. 997–1006.

[13] KARRER, T. – KRÄMER, J.-P. – DIEHL, J. – HARTMANN, B. – BORCHERS, J.: Stacksplorer: Call graph navigation helps increasing code maintenance efficiency, *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, UIST '11*, 2011, pp. 217–224.

[14] SULÍR, M. – NOSÁĽ, M. – PORUBÄN, J.: Recording concerns in source code using annotations, *Computer Languages, Systems & Structures*, Vol. 46, 2016, pp. 44–65.

[15] JOY, M. – BECKER, M. – MUELLER, W. – MATHEWS, E.: Automated source code annotation for timing analysis of embedded software, *18th Annual International Conference on Advanced Computing and Communications, ADCOM 2012*, 2012, pp. 12–18.

[16] CAZZOLA, W. – VACCHI, E.: @Java: Bringing a richer annotation model to Java, *Computer Languages, Systems & Structures*, Vol. 40, No. 1, 2014, pp. 2–18.

[17] CÓRDOBA-SÁNCHEZ, I. – DE LARA, J.: Ann: A domain-specific language for the effective design and validation of Java annotations, *Computer Languages, Systems & Structures*, Vol. 45, 2016, pp. 164–190.

[18] BECK, F. – MOSELER, O. – DIEHL, S. – REY, G.: In situ understanding of performance bottlenecks through visually augmented code, *21st IEEE International Conference on Program Comprehension, ICPC 2013*, 2013, pp. 63–72.

[19] BRAGDON, A. et al.: Code Bubbles: Rethinking the user interface paradigm of integrated development environments, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, 2010, pp. 455–464.

[20] STOREY, M.-A. – CHENG, L.-T. – BULL, I. – RIGBY, P.: Shared waypoints and social tagging to support collaboration in software development, *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work, CSCW '06*, 2006, pp. 195–198.

[21] LIEBIG, J. – KÄSTNER, C. – APEL, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code, *Proceedings of the Tenth International Conference on Aspect-oriented Software Development, AOSD '11*, 2011, pp. 191–202.

[22] TÁBORSKÝ, R. – VRANIĆ, V.: Feature model driven generation of software artifacts, *Federated Conference on Computer Science and Information Systems, FedCSIS 2015*, 2015, pp. 1007–1018.

[23] KÄSTNER, C. – APEL, S. – KUHLEMANN, M.: Granularity in Software Product Lines, *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, 2008, pp. 311–320.

[24] JI, W. – BERGER, T. – ANTKIEWICZ, M. – CZARNECKI, K.: Maintaining feature traceability with embedded annotations, *Proceedings of the 19th International Conference on Software Product Lines, SPLC '15*, 2015, pp. 61–70.

[25] KOLLÁR, J. – CHODAREV, S. – PIETRIKOVÁ, E. – WASSERMAN, L': Identification of patterns through Haskell programs analysis, *Federated Conference on Computer Science and Information Systems, FedCSIS 2011*, 2011, pp. 891–894.

[26] SULÍR, M. – PORUBÄN, J.: A quantitative study of Java software buildability, *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2016*, 2016, pp. 17–25.

[27] NOSÁĽ, M. – SULÍR, M. – JUHÁR, J.: Language composition using source code annotations, *Computer Science and Information Systems*, Vol. 13, No. 3, 2016, pp. 707–729.

Received October 13, 2016, accepted February 14, 2017

BIOGRAPHIES

Matúš Sulír is a PhD student at the Department of Computers and Informatics, Technical University of Košice. He graduated with a master's degree in Computer Science in 2014. His current research is focused on program comprehension, source code annotations, and empirical methods in software engineering.

Jaroslav Porubän is an Associate Professor and Head of the Department of Computers and Informatics, Technical University of Košice, Slovak Republic. He received his MSc. in Computer Science in 2000 and his Ph.D. in Computer Science in 2004. Currently the main areas of his research are computer language engineering, domain-specific languages and program comprehension.