MetPurity: A Learning-Based Tool of Pure Method Identification for Automatic Test Generation

Runze Yu
runzeyu@whu.edu.cn
School of Computer Science
Wuhan University
Wuhan, China

Youzhe Zhang
youzhezhang@whu.edu.cn
School of Computer Science
Wuhan University
Wuhan, China

Jifeng Xuan*
jxuan@whu.edu.cn
School of Computer Science
Wuhan University
Wuhan, China

ABSTRACT

In object-oriented programming, a method is pure if calling the method does not change object states that exist in the pre-states of the method call. Pure methods are widely-used in automatic techniques, including test generation, compiler optimization, and program repair. Due to the source code dependency, it is infeasible to completely and accurately identify all pure methods. Instead, existing techniques such as ReImInfer are designed to identify a subset of accurate results of pure method and mark the other methods as unknown ones. In this paper, we designed and implemented MetPurity, a learning-based tool of pure method identification. Given all methods in a project, MetPurity labels a training set via automatic program analysis and builds a binary classifier (implemented with the random forest classifier) based on the training set. This classifier is used to predict the purity of all the other methods (i.e., unknown ones) in the same project. Preliminary evaluation on four open-source Java projects shows that MetPurity can provide a list of identified pure methods with a low error rate. Applying MetPurity to EvoSuite can increase the number of generated assertions for regression testing in test generation by EvoSuite.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Method purity, static analysis, debugging, machine learning, test generation, regression testing

ACM Reference Format:
Runze Yu, Youzhe Zhang, and Jifeng Xuan. 2020. MetPurity: A Learning-Based Tool of Pure Method Identification for Automatic Test Generation. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415292

*Corresponding author

1 INTRODUCTION

In object-oriented programming, a method is *pure* if calling the method does not change object states that exist in the pre-states of the method calling [15]. A pure method is also called a side-effect-free method. Similarly, a not pure method is called an *impure* one. Taking Class `ArrayList` in Java as an example, a method `size()` is pure since this method returns the number of elements in the list without mutating the object states; another method `add(Object obj)` is impure since this method modifies the state of the list via adding an element `obj`.

The property that whether a method is pure or not is referred to as *method purity* [15]. Method purity is widely-used in automatic techniques of program analysis and debugging, including Java bytecode optimization [3], model checking [16], type inference [4], test generation [5, 12], and program repair [17]. In automatic test generation, a test case that calls a pure method cannot change the existing object states. Thus, test generation techniques can arbitrarily insert assertions of pure methods and monitor the current object states. These assertions are mainly used for regression testing (i.e., detecting whether a correct state is violated in regression) and killing mutants (i.e., examining the adequacy of test cases) [5]. In Randoop, a tool of feedback-directed random testing, a pure method (called an observer method) is used to generate assertions to detect object changes [12]. In EvoSuite, a tool of evolutionary testing, a pure method with no parameters and returning primitive values is used to generate an inspector assertion [6]. For instance, if an object `list` of Class `ArrayList` whose list size is 10 during test generation, an inspector assertion like `assertEquals(10, list.size())` will be generated for regression testing. This assertion is used to detect whether a new fault is added via code changes in the future. The test generation tools can add such assertions to test cases without involving any errors since the method `size()` is pure and calling pure methods in a test case does not mutate the state of existing objects.

However, automatic identification of method purity is infeasible due to the complicated dependency in source code. Existing techniques, such as JPPA [15], JPure [14], and Purano [18], are designed to identify method purity for a small subset of methods and skip the others. To the best of our knowledge, ReImInfer by Huang et al. [9, 10] is the state-of-the-art approach that can be applied to large-scale programs. ReImInfer conducts context-sensitive reference inference via static program analysis to detect pure methods. The design of ReImInfer follows the existing work and returns an accurate (called *sound* in program analysis) but incomplete list of pure methods: a subset of pure methods are detected and all the other methods are marked as *unknown* [10]. To make the identified

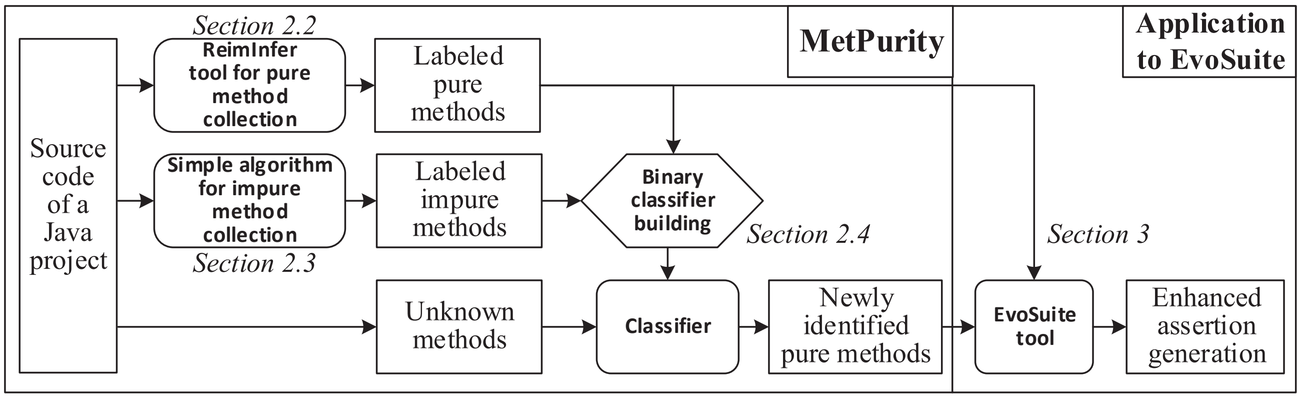
MetPurity
Application to EvoSuite
Section 2.2
ReimInfer tool for pure method collection
Labeled pure methods
Source code of a Java project
Simple algorithm for impure method collection
Labeled impure methods
Binary classifier building
Section 2.4
Section 3
Section 2.3
Unknown methods
Classifier
Newly identified pure methods
EvoSuite tool
Enhanced assertion generation

Figure 1: Overview of MetPurity and its application to EvoSuite.

method purity accurate, many pure methods are directly marked as unknown without identification.

In this paper, we design and implement MetPurity, a learning-based tool of pure method identification. MetPurity converts pure method identification into binary classification and returns a list of pure methods for developers. First, given the source code of a project, the training data of pure and impure methods are automatically collected via two static analysis techniques, including ReImInfer; methods that are not in the training data are treated as the unknown methods. Second, a binary classifier (implemented with the random forest classifier) is built on the training data. Third, the purity of the above unknown methods is identified with the classifier.

We preliminarily evaluate MetPurity on four open-source Java projects, including Joda Time and Apache Commons Math. Application to the test generation in EvoSuite can increase by 2.38% to 30.89% newly generated assertions for regression testing. The new assertions can improve the testability of automatic tools of test generation.

Motivation. Existing techniques on method purity can identify a subset of pure methods and leave the other methods as unknown ones. In this work, MetPurity increases the number of identified pure methods with a low error rate. Developers can check and confirm a small number of pure methods with the support of MetPurity, instead of checking all unknown methods.

This paper makes the following major contributions,

- A learning-based tool MetPurity of pure method identification. To the best of our knowledge, this is the first predictive tool that identifies pure methods and enhances the number of pure methods in existing tools. The error rate of pure methods identification is 0.00% to 12.86%. Developers can manually check the identified pure methods by MetPurity for further applications.
- Preliminary evaluation on enhancing assertion generation of EvoSuite, an off-the-shelf tool of test generation. Up to 30.89% of assertions can be newly generated for regression testing via applying MetPurity.

2 FRAMEWORK OF METPURITY

The goal of the proposed tool MetPurity is to provide a list of identified pure methods for developers. MetPurity takes the source code of a Java project as input and returns a list of pure methods as output.

2.1 Overview of MetPurity

Automatically identifying all pure methods is infeasible due to undetected dependencies in source code [10]. The proposed tool MetPurity is a learning-based tool of identifying pure methods, i.e., predicting whether a method is pure or not.

Figure 1 shows the overview of our proposed tool MetPurity and its application to EvoSuite. This tool consists of two major phases: the building phase and the identification phase. In the building phase, pure methods and impure methods are accurately labeled via two techniques of static program analysis. That is, labeling data in the building phase does not require any manual work. Then, a binary classier (implemented with a random forest classifier [1]) is built based on the labeled data. In the identification phase, the classifier is used to predict whether an unknown method is pure. A list of identified pure methods is returned to developers as the final output. Based on this output, developers can manually check the purity of the methods in the list since the identification in MetPurity is inaccurate (with a low false-positive rate).

The major benefit of using MetPurity is to increase the number of identified pure methods for further application. Developers do not need to check the purity of all methods; instead, developers can only check methods in the list that is provided by MetPurity. We apply MetPurity to a widely-studied test generation tool, EvoSuite. The pure methods by MetPurity can be used to enhance the test assertion generation in EvoSuite. Preliminary evaluation can be found in Section 3.

Given the source code of a Java project, two techniques in MetPurity can accurately detect a set of pure methods (Section 2.2) and a set of impure methods (Section 2.3). These two sets of methods serve as the accurately labeled data and are combined as the training set in the building phase. All the other methods whose labels of purity are unknown, are formed as the test set in the identification phase.

2.2 Accurate but Incomplete Collection of *Pure* Methods

To build a classifier of method purity identification, MetPurity collects two sets of pure methods and impure methods in the building phase.

To the best of our knowledge, ReImInfer is a state-of-the-art tool that can accurately collect a subset of pure methods [9, 10]. ReImInfer is a sound but incomplete approach, which uses static program analysis to collect the known dependency among source code. That is, all the identified methods are actually pure and the other methods are treated as *unknown* methods, including pure or impure ones. The major reason for unknown methods is due to the undetected dependency among source code, including native methods, polymorphism, and library without known source code.

In MetPurity, ReImInfer plays the role of labeling pure methods in the training set. Note that identifying pure methods of a project via ReImInfer requires the local configuration of the project. Therefore, before running MetPurity, we locally configured the project under analysis and keep the source code compilable.

2.3 Accurate but Incomplete Collection of *Impure* Methods

To build a classifier, both pure methods and impure methods should be provided as a training set. However, ReImInfer as well as other existing tools cannot accurately identify impure methods. It is necessary to provide a set of accurately impure methods for the classification. Since it is time-consuming to manually identify impure methods, we followed the idea of ReImInfer and designed a simple algorithm to collect impure methods. This algorithm is also accurate but incomplete. That is, many impure methods cannot be identified and all identified ones are actually impure.

The algorithm of identifying of impure methods takes the set of all methods in a project as input and returns a subset of impure methods. The general idea of identifying impure methods consists of three major rules:

- If a method contains at least one assignment whose left-value is a field or the object reference of `this`, this method is added to a set of impure methods X. This rule is designed to indicate that an object is to be modified via assignments.
- If a method contains at least one assignment whose left-value is an object parameter or its field, this method is added to a set of impure methods Y. This rule is designed to show that an object in a parameter is to be modified.
- If an impure method x in X is invoked by a field or the object reference of `this`, the method that invoking x via calling the field or the `this` is added to the set X. This rule is repeatedly applied until there is no newly added method.

All methods that satisfy these rules, i.e., the union set of X and Y, are collected. These methods are indeed impure since existing objects are modified. MetPurity uses these methods as the training data of impure methods.

2.4 Learning from Collected Pure and Impure Methods

The general idea of MetPurity is to build a binary classifier from collected pure and impure methods. This classifier can be used to predict whether an unknown method is pure or not. Generally, any binary classifier can be used in MetPurity. In our implementation, we used the random forest classifier because of its performance and robustness [1]. A data processing technique, Synthetic Minority Oversampling TEchnique processing (SMOTE) [2], is used to address potential data imbalance of pure and impure methods in the training set.

Feature extraction. To characterize one method into a numeric vector, MetPurity extracts 103 features via static program analysis. The static analysis in MetPurity is conducted via parsing the parameter list and source code of each given method. The 103 features can be roughly divided into seven categories: statements (17 features, e.g., the number of assignments and lines of executable code), variables (27 features, e.g., numbers of defined or used local variables or fields), control nodes (29 features, e.g., numbers of conditions, loops, and nested loops), invocations (8 features, e.g., numbers of static or non-static methods), complexity (17 features, e.g., items of the Cyclomatic complexity or the Halstead complexity), inside classes (3 features, e.g., numbers of anonymous or inner classes), and method signatures (3 features, e.g., numbers of annotations or `throws`).

Dataset partition. The training set and the test set of MetPurity belong to the same project. As shown in Figure 1, the training set of pure methods and impure methods are automatically collected with the ReImInfer tool in Section 2.2 and the algorithm in Section 2.3. Then all the other methods with unknown purity are treated as the test set.

2.5 Implementation

Our tool MetPurity is implemented with Java JDK. The feature extraction for classification (Section 2.4) and the algorithm of impure method collection (Section 2.3) are implemented on top of a static analysis tool, Spoon.[1] Spoon[13] is a program analysis tool of source code analysis and transformation for Java code. The pure method collection (Section 2.2) is based on an off-the-shelf tool, ReImInfer [9]. Machine learning techniques, such as the random forest and the SMOTE, are implemented with Weka.[2] Weka [8] is a workbench for machine learning algorithms in Java. The components of MetPurity is linked with scripts. The prototype and evaluation data are available at http://cstar.whu.edu.cn/p/metpurity/.

3 PRELIMINARY EVALUATION

3.1 Evaluation Setup

Pure methods are widely used in automatic testing and debugging. In automatic test generation, assertions based on pure methods cannot change existing object states. Such assertions do not interrupt correct program states and can be used to detect potential faults. As mentioned in Section 1, in EvoSuite [6], a pure method with no parameters and returning primitive values is converted into an

[1]Spoon, http://spoon.gforge.inria.fr/.
[2]Weka 3.8, http://www.cs.waikato.ac.nz/ml/weka/.

Table 1: Error rate of pure method identification in MetPurity

Project	Joda Time 2.9.6	Http Client 4.1.2	Eclipse Core 3.2.0	Apache Commons Math 3.6.1
# Collected pure methods	514	184	721	1248
# Collected impure methods	241	61	357	207
# Unknown methods (i.e., the test set)	144	50	135	344
# Identified pure methods by MetPurity	128	32	70	306
# Incorrectly identified pure methods	0	4	9	2
Error rate of pure method identification	0.00% (0/128)	12.50% (4/32)	12.86% (9/70)	0.65% (2/306)

Table 2: Number of generated assertions in test cases for all classes under testing when applying MetPurity to EvoSuite

Project	Joda Time	Http Client	Eclipse Core	Apache Commons Math
# Assertions without purity analysis in EvoSuite	17803	1411	325488	152747
# New assertions by purity analysis in EvoSuite	4495	493	55635	34164
# New assertions via pure methods in the training set	1964	151	6627	23123
# New assertions via prediction by MetPurity in the test set	779	35	2454	34607
Rate of newly added assertions by MetPurity	12.30%	9.77%	2.38%	30.89%

inspector assertion. An automatically-generated assertion based on a pure method can be directly added into test cases without changing the current program states. The original purity analysis in EvoSuite is intra-method static analysis, which only considers source code inside one method [7]. This makes EvoSuite identify only a small subset of pure methods and skip the others.

In this paper, we evaluate the proposed tool MetPurity on four widely-used open-source Java projects, including Joda Time, Http Client, Eclipse Core, and Apache Commons Math. We investigate the benefit of applying MetPurity to the off-the-shelf test generation tool, EvoSuite.

In EvoSuite, assertions are generated based on pure methods that contain no parameters and return primitive values. To keep consistency, the following evaluation only shows the numbers of pure methods without parameters and returning primitive values.

3.2 Effectiveness of Pure Method Identification

Table 1 shows the number of identified pure methods by MetPurity and the error rate of identification. For instance, in Project Apache Commons Math, the number of collected pure methods and collected impure methods in the training set are 1248 and 207, respectively. This leaves 344 methods with unknown purity. After MetPurity is applied to these unknown methods, 306 methods are identified as pure ones. Among these 306 identified methods, 304 methods are correct identification and 2 methods are incorrect identification, i.e., 2 false positives. The error rate of pure method identification is 2/306 = 0.65%. Among four projects under evaluation, the error rate ranges from 0.00% to 12.86%.

3.3 Effectiveness of Application to EvoSuite

The test generation tool, EvoSuite [5], generates assertions to detect potential violations in regression testing. We evaluate the results of applying pure methods by MetPurity to EvoSuite via counting the number of generated assertions. In general, a high number of generated assertions indicates the strong capability of detecting requirements violations [7, 11].

Table 2 shows the numbers of generated assertions in test cases when applying MetPurity to EvoSuite. The first four rows are mutual: only newly generated assertions are counted. We calculate the rate of newly generated assertions by MetPurity. For example, in Project Apache Commons Math, 23123 assertions are generated based on pure methods in the training set of MetPurity and 34607 assertions are generated based on the identification of pure methods in the test set; in EvoSuite, 152747 assertions and 34164 assertions are generated with and without purity analysis, respectively. Then the rate of newly generated assertions by MetPurity is 30.89%, i.e., $\frac{23123+34607}{152747+34164}$. As shown in Table 2, the rate of newly generated assertions by MetPurity is between 2.38% to 30.89%. Pure methods that are identified by MetPurity can add new assertions to the result of EvoSuite. This suggests that test cases with these assertions can enhance the detection of requirements violations by EvoSuite.

Summary of evaluation. Evaluation on four projects shows that MetPurity can newly identify 32 to 306 pure methods with 0 to 9 false-positives. Applying MetPurity to EvoSuite shows that applying MetPurity can add 2.38% to 30.89% newly generated assertions.

4 CONCLUSION

We present MetPurity, a learning-based tool of pure method identification. The training data of MetPurity are labeled via automatic program analysis and are not manually labeled by developers. The tool provides a list of identified pure methods with a low error rate. MetPurity is applied to a widely-studied tool of test generation, EvoSuite. Applying MetPurity to EvoSuite can increase by 2.38% to 30.89% newly generated assertions for regression testing. Future work of developing MetPurity is to further reduce the error rate of pure method identification.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China under Grant No. 2018YFB1003901 and the National Natural Science Foundation of China under Grant No. 61872273.

REFERENCES

[1] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[2] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Intell. Res.* 16 (2002), 321–357. https://doi.org/10.1613/jair.953

[3] Lars R Clausen. 1997. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience* 9, 11 (1997), 1031–1045.

[4] Werner Dietl, Michael D. Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science, Vol. 6813)*, Mira Mezini (Ed.). Springer, 333–357. https://doi.org/10.1007/978-3-642-22655-7_16

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 416–419. https://doi.org/10.1145/2025113.2025179

[6] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 8:1–8:42. https://doi.org/10.1145/2685612

[7] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Software Eng.* 38, 2 (2012), 278–292. https://doi.org/10.1109/TSE.2011.93

[8] Mark A. Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11, 1 (2009), 10–18.

[9] Wei Huang and Ana Milanova. 2012. ReImInfer: method purity inference for Java. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 38. https://doi.org/10.1145/2393596.2393640

[10] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 879–896. https://doi.org/10.1145/2384616.2384680

[11] Ping Ma, Hangyuan Cheng, Jingxuan Zhang, and Jifeng Xuan. 2020. Can This Fault Be Detected: A Study on Fault Detection via Automated Test Generation. *Journal of Systems and Software* (2020).

[12] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[13] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A library for implementing analyses and transformations of Java source code. *Softw., Pract. Exper.* 46, 9 (2016), 1155–1179.

[14] David J. Pearce. 2011. JPure: A Modular Purity System for Java. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6601)*, Jens Knoop (Ed.). Springer, 104–123. https://doi.org/10.1007/978-3-642-19861-8_7

[15] Alexandru Salcianu and Martin C. Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3385)*, Radhia Cousot (Ed.). Springer, 199–215. https://doi.org/10.1007/978-3-540-30579-8_14

[16] Oksana Tkachuk and Matthew B. Dwyer. 2003. Adapting side effects analysis for modular program model checking. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, Jukka Paakki and Paola Inverardi (Eds.). ACM, 188–197. https://doi.org/10.1145/940071.940097

[17] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[18] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2015. Revealing Purity and Side Effects on Functions for Reusing Java Libraries. In *Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 8919)*, Ina Schaefer and Ioannis Stamelos (Eds.). Springer, 314–329. https://doi.org/10.1007/978-3-319-14130-5_22