Language-agnostic multi-paradigm code quality assurance framework

Marnick Q.T.P. van der Arend[1]

[1]University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands

Abstract

Several object-oriented programming (OOP) languages have implemented constructs from the functional programming (FP) paradigm. We call them multi-paradigm (MP) programming languages. To measure code quality, the use of metrics is often included. While metrics for a specific paradigm are well-known, there are few multi-paradigm metrics. Such metrics are only implemented for a specific MP language. We explore the design of a language-agnostic multi-paradigm code quality assurance framework. This framework transforms MP languages to a generic model to extract metrics for OOP, FP and MP. We conclude, by using transformations and analysis, that we can do code quality assurance for every MP language.

Keywords

multi-paradigm, language-agnostic, code quality assurance, modelling, parsing, metrics

1. Introduction

Programming languages come in many shapes and sizes. They are categorised by their programming paradigm. Every programming paradigm has its purpose, solving a certain problem with the best solution. A large number of popular general purpose programming languages have adopted multiple programming paradigms within their language. While using multiple paradigms can help solve a problem in different ways, on the other hand, it can produce new combinations that lead to unintended behaviour of the written code. Code quality can be measured to decrease the possibility for unintended behaviour. Code quality for individual programming languages can be measured and improved using code quality assurance tools. These tools try to locate areas of improvement for a source code with e.g. the use of static code analysis. Quality assurance tools can analyse source code before it is being tested and therefore, potential problems can be addressed early when it is relatively cheap to fix [1].

With code quality assurance, we use metrics to measure the quality of source code. While there have been decades of research on paradigm-specific metrics, e.g. object-oriented programming (OOP) [2, 3] or functional programming (FP) [4], there has been less research into metrics that capture the mixed use of these programming paradigms. In this paper, we refer to the mixed use of OOP and FP as multi-paradigm (MP) programming.

Research into MP metrics has focused on detecting fault proneness in software programs [5, 6,

Belgium-Netherlands Software Evolution Workshop, 2022

m.q.t.p.vanderarend@student.utwente.nl (M. Q.T.P. v. d. Arend)
0000-0001-6689-1596 (M. Q.T.P. v. d. Arend)

7]. These research projects identified and validated MP metrics for use in the MP programming languages C# and Scala. Due to data constraints, the accuracy of the fault proneness detection was poor. Software projects are built in different ways and implement concepts of paradigms differently. Therefore, a large data set is crucial for accurate predictions of fault proneness.

We see many similarities in the MP constructs of C# and Scala. Code quality metrics are not bound by any specific programming language but rather by programming paradigm. Therefore, we hypothesise that code quality can be measured on a language-agnostic level. We hypothesise that we can abstract from language-specific syntactical context and extract the concepts and constructs of these MP programming languages to measure code quality. If our hypothesis is confirmed, we can use open source project codebases for each MP programming language to measure code quality in a language-agnostic manner. The number of codebases that can be measured with our framework increases and therefore, increasing the accuracy of predicting external quality attributes such as fault proneness.

Besides solving the scarcity of data, performance is an important challenge. The speed and accuracy of measurements executed by a language-agnostic code quality assurance approach must be comparable to a language-specific approach. Therefore, this research explores the design of a performant language-agnostic code quality assurance framework that can assess the quality of source code for MP languages. If the framework is as performant as a language-specific approach, we can decrease the time required for developing new code quality assurance tools for MP languages.

The contribution of this paper is providing a theoretical background and an approach for a framework with which MP source code can be assessed on its code quality in a language-agnostic setting.

2. Background

In this section we walk through the context in which the language-agnostic multi-paradigm code quality assurance framework will be designed.

2.1. Multi-Paradigm Languages

To understand what type of framework should be designed, it is important to clarify the concepts of MP programming languages.

There is a large number of popular programming languages that are MP. Python, Java, JavaScript and C# account for more than 50% of share in popularity [8]. Three of these languages are primarily object-oriented with functional constructs integrated at a later stage. E.g., Java integrated lambda expressions in Java 8 that allowed for functional constructs like `map`, `filter` and `count` on collections [9].

OOP is considered a part of imperative programming paradigm. The term was originally coined by Dr. Alan Kay in 1966 describing an architecture for messaging where objects pass on messages using procedures (i.e. methods) [10]. It strongly focuses on states and modifiable data.

FP is part of the declarative programming paradigm and originates from Lambda Calculus [11], a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution [12].

Within MP languages, the data encapsulation and the type system of the OOP paradigm are often used. There are improvements to make the language more suitable to adopting the FP paradigm by the addition of (anonymous) function types.

2.1.1. Object-Oriented concepts

- *Encapsulation*: data and the methods that operate on that data are encapsulated within the context of OOP with classes. It is used to protect the private information of that class and only expose functionality that is made public [13].
- *Inheritance*: a class within OOP can inherit (much of) the functionality from another class. This hierarchy can exist on multiple levels where a class has one or more children or parents [13].
- *Polymorphism*: it loosely translates to 'different forms' from Latin. In OOP, it means that one class at the root of the hierarchy lays the blueprint for a set of classes with similar behaviour [13]. E.g., the `Bus` class and `Car` class both inherit from `Vehicle` and this root class can contain the `drive()` method which applies to both `Bus` and `Car`.

2.1.2. Functional concepts

- *Functions as First-Class citizens*: functions are seen as first-class citizens which means that they can be used as variables within the source code. Functions can be used as parameters, passed as return values or stored in a data structure. In FP, the whole program is a function that can contain other functions to make the program more modular but it outputs a result in the end [12].
- *Lambda functions*: lambda functions are functions that are based on Lambda Calculus [11]. Closures within FP are related to Lambda Calculus since lambda functions take some input, perform some mathematical computation and output a value [14]. Within FP, there are several types of functions that can be used solve problems with an optimal solution:
  - *Anonymous functions*: an anonymous function is a function without a name. Therefore, it cannot be referenced from another part of the code. It is a way to write a concise solution for a problem that is often only required in one part of your program.
  - *Higher-order functions*: as we have learnt, functions are first-class citizens within FP. With this feature, we can create functions that take as arguments one or more other functions. This type of function is called a higher-order function. The benefit of this type of function is that its behaviour can change depending on its arguments [6].
  - *Nested functions*: nested functions are a common tool for dividing a function into readable chunks and increase reusability of source code. Functions are first-class citizens, which allows them to be created within the scope of another function to only be referenced within that scope.
- *Referential transparency*: functions are expressions that always output the same value when the same input values are provided. Therefore, when you refer back to an expression

in a later part of the program, you can be sure that the value of the expression has not changed. You could change the expression with its value or vice versa and it would not affect the behaviour of the program [12]. This removes any occurrence of side effects [15].

- *Recursion*: looping over collections in FP is performed via recursion instead of sequencing. In recursive functions, the data state is explicitly passed through via argument(s) of the function until the base case is reached. Thereafter, this state is passed as output to the function caller until the start of the recursive operation is reached, thereby completing the chain [16]. Recursion is used for well-known operations on collections such as `map`, `reduce` and `filter`.
- *Lazy evaluation*: non-strict semantics or lazy evaluation, often also called *call by need* has a key feature that it only evaluates arguments in functions once. Lazy evaluation allows the developer to be more expressive and relieves them from concerns about evaluation order. It is also used for dealing with data structures that have an unbounded size [16].
- *Pattern matching*: it is a form of syntactic sugar within the FP paradigm that allows a developer to write multiple equations within a function. In this type of function, only one of the equations is applicable in a given situation [16].
- *Currying*: currying can be done by splitting a function that takes multiple arguments into separate functions with one argument. With a FP function as first-class citizen, functions can be returned as results in another function. Therefore, function calls can be chained. The transformation of creating single argument functions from a multi-argument function is therefore called currying [17], named after the mathematician Haskell Curry.

2.2. Code Quality

Source code comes in many forms, it can be categorised into one or more paradigms, and it is most often written by humans. Poor source code can have catastrophic effects [18]. Therefore, it is important to measure the quality of code and fix poor quality code. Code quality describes the state of a code as being good (high quality) or bad (low quality). ISO and IEC have created the ISO/IEC 25010:2011 standard [19] which improves upon the ISO/IEC 9126:1991 standard [20]. The improved standard contains the product quality model that measures code quality with 8 characteristics:

- functional suitability
- performance efficiency
- usability
- reliability
- maintainability
- portability
- compatibility
- security

The standard provides consistency for measuring and evaluating software product quality. The 8 characteristics contain sub-characteristics to provide more context to the characteristic.

The Software Improvement Group has used definitions from the standard to create the Maintainability Model [21]. This model measures the maintainability characteristic in a language-agnostic manner.

2.3. Source Code Metrics

In Section 2.2 we explained the characteristics of code quality. To measure source code, we can use metrics. Every programming paradigm has specific metrics tailored to their concepts. For OOP, a well-known set of metrics was formally defined by Chidamber and Kemerer [22]. These are:

- Weighted Method per Class (WMC)
- Number of Ancestor Classes (NAC) [23]
- Number of Descendent Classes (NDC) [23]
- Coupling Between Objects (CBO)
- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

In a later mapping study by Nuñez-Varela et al. [24], these metrics were found used by the larger portion of papers. Other frequently used metrics from this study that were paradigm agnostic are:

- Source Lines of Code (SLOC)
- Comment Density (CD)
- McCabe's Cyclomatic Complexity (CC) [25]

There is less research into FP metrics. A study Ryder & Thompson [4] has defined a collection of metrics for measuring Haskell source code. The metrics discussed in this paper have been adopted from OOP metrics with additional metrics for a strong type-system, higher-order and polynomial functions. These metrics include:

- Pattern Size (PSIZ)
- Number of Pattern Variables (NPVS)
- Strongly Connected Component Size (SCCS)
- Pathcount of a Function (PATH)

With our definition of MP languages, we also have to include metrics for the combination of OOP with FP. A study by Zuilhof [6] defined several metrics:

- Number Of Lambda Functions Used In A Class (LC)
- Source Lines of Lambda (SLOL)
- Lambda Score (LSc)
- Number of Lambda Functions using Outer Variables (LMFV)
- Number of Lambda Functions using Local Variables (LMLV)
- Number of Lambda Functions with Side Effects (LSE)

2.4. Static Code Analysis

As mentioned in Section 1, the quality of source code can be measured using a technique called static code analysis. Static code analysis is a way to automatically examine a source code without executing the program it describes [26]. There are several data structures that can be used to aid static code analysis [27]. In our research, we focus on source code, abstract syntax trees and control flow graphs. The preferred data structure of each metric is visualised in Table 1. The metrics described in Section 2.3 can be extracted using the techniques described in the following sections.

2.4.1. Source Code

Source code is human-readable text that defines a set of instructions for what a software program can do. Our definition of source code includes all physical lines of code, which entails lines of logical code, comments and blank lines [28]. All of these lines provide structure and domain-specific context to a user who want to understand that source code, while only executable lines of code will be compiled.

Source code is translated by a compiler to transform these human-readable instructions into executable machine code. The first step of the compiler is lexical analysis (i.e. scanning), where the stream of characters from a source code is read and a lexical token stream is created. These tokens are stored in a symbol table in the format `<token-name, attribute-value>`. This token stream is used as input for syntax analysis (i.e. parsing), where a concrete syntax tree (CST) (i.e. parse tree) is created that depicts the grammatical structure of the token stream. This CST is used for semantic analysis to create an AST, which we will cover in Section 2.4.2. After all analyses phases are completed, machine-executable code is generated. This complex process often includes code optimisation to make the source code more efficient for the machine to execute [27, 29].

2.4.2. Abstract Syntax Tree

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of a source code. During semantic analysis, an AST is created from a CST and its corresponding symbol table. Semantic analysis is used to check if the source code is semantically consistent with its language definition (i.e. grammar) [29].

2.4.3. Control Flow Graph

A control flow graph (CFG) is a way in which the flow of a software program can be represented. It is a graph-based representation of control flow relationships within a software program. The technique was created by Frances E. Allen in 1970 [30]. It is a widely used technique for static analysis of a program as it can accurately describe the flow of (a part of) a program.

2.4.4. Metric Mapping

In Section 2.3, we covered the code quality metrics. With the data structures covered in the previous subsections, we have an approach to measure them. To provide an overview of the preferred choice of a data structure to measure a metric, we have created Table 1.

While it is possible to calculate some metrics with multiple data structures, we have chosen to show only our preferred choice of data structure for each metric. Some of these metrics do require multiple data structures to measure a result. E.g., SLOL requires the domain knowledge retrieved from the AST to extract lambda functions and the source code to count lines of code. When it is known which data structure is required for a metric, it can easily be used in a code quality assurance framework.

Table 1
Metric Data Structure Mapping

	Abstract Syntax Tree	**Control Flow Graph**	**Source Code**
Generic	CC		SLOC, CD
OOP	WMC, NAC, NDC, CBO, RFC, LCOM		
FP	PSIZ, NPVS	SCCS, PATH	
MP	LC, SLOL*, LSc*, LMFV, LMLV, LSE		SLOL*, LSc*

*Requires combined measurement technique

2.5. Models and Transformations

For a framework that aims to analyse MP languages in a language-agnostic manner, we can use model-driven engineering (MDE) to generalise these MP languages into one analysable model.

Models are abstractions from phenomena in the real world. Programming languages can also be modelled. Metamodels can abstract away from language-dependent syntax to create an overarching model (i.e., modelling a programming paradigm). A metamodel is merely a higher form of abstraction from a model, a blueprint of what the model itself should conform to [31].

To transform a model to another model, model-to-model transformation can be used. This type of transformation provides a mapping between a source and target model. The transformation is defined upon the metamodels of the respective models on which the transformation is performed. Therefore, with several transformations, the model of one MP language can be transformed to a model of another MP language [31].

We argue that MDE can give us an advantage by being able to transform any MP language into one abstract model, using transformations. Due to the rapid updates of programming languages, we argue that the MDE approach will also give us the ability to quickly adapt to changes and include new versions of programming languages.

3. Related Work

To measure the maintainability of source code, the Software Improvement Group (SIG) has designed the language-agnostic SIG Maintainability Model [21]. This model measures all sub-characteristics of maintainability by mapping them to language-agnostic source code metrics. Using this mapping, they determine the overall maintainability of a source code.

To apply source code analysis, it is useful to use an AST or CFG. There are other tools that can help for specific languages, such as srcML. srcML is an XML format for the representation of C/C++/Java source code that wraps source code with information from the AST into a single XML document. It can be used to perform scalable lightweight fact-extraction and transformation since other XML tools can be used once it is in the XML form (e.g., XPath and XSLT) [32]. Another tool for source code analysis is the meta-programming language Rascal [33]. It can be used for extracting metrics from programming languages and is extendable with libraries like M^3 [34, 35]. The M3 model is a simple and extensible language-agnostic model for capturing facts about source code for future analysis [34].

With regards to multi-paradigm modelling, Owens [36] describes an extension of GASTM for the FP paradigm. Functional operations on lists, lambda functions and support for container-less list of functions and constants are included in the extension. Merging the FP paradigm within the metamodel will allow a broader set of metrics to be analysed.

4. Proposed Approach

Programming languages have improved over the years and have added a vast amount of new features. Due to the number of supported features and the fast pace of new developments in MP languages, it is reasonable to create a language-agnostic code quality assurance framework based on an iterative approach.

A code quality assurance framework that can apply to more than one multi-paradigm programming language has many requirements. It is important to find out what constructs are present in MP languages to be able to abstract from language-specific implementations. With extensive knowledge of MP languages and their constructs, we can select the quality characteristics for our quality assurance framework. With these characteristics known, we can discover what is required to measure them.

With the metrics and their requirements, we can research how we can use static code analysis techniques for our framework. It is essential to know what metrics can be used on an AST, which metrics require the CFG and which metrics require plain source code. A category mapping of the metrics has been shown in Table 1.

Figure 1 describes the key points in the flow of the quality assurance framework. Every MP language has their own grammar and way of representing MP concepts. Therefore, every language will require their own tooling to transform the source code to its language specific AST. Another transformation is required for transforming the language AST (L-AST) to a generic AST (G-AST). This transformation maps the relations between the two AST models. To calculate all metrics in the G-AST, we must know what parts of the AST are required for every individual metric. When this is known, the code quality metrics can be extracted.

With the extracted metrics, we can evaluate how the framework performs in comparison to language-specific analyses to determine the feasibility of a language-agnostic quality assurance framework.

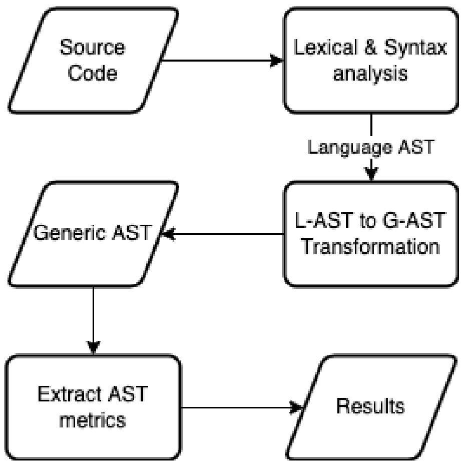

Figure 1: Quality Assurance Framework Flow for any MP language

4.1. Research Questions

While the key points of the framework were captured in Figure 1, much of the details are to be researched. Therefore, our main research question is:

RQ1: To what extent can a language-agnostic code quality assurance framework be designed for multi-paradigm languages?

> *RQ1.1: What are the constructs of multi-paradigm programming languages?*
>
> *RQ1.2: Which code quality characteristics must the framework capture, and how?*
>
> *RQ1.3: How can we use static code analysis techniques to measure our selected metrics in a language-agnostic multi-paradigm manner?*
>
> *RQ1.4: To what extent can language-specific context be preserved within this framework?*
>
> *RQ1.5: How does this framework perform in comparison to language-specific analyses?*

5. Discussion

In this section, we speculate about possible obstacles that could impede the effectiveness of our framework design.

One of the goals of the proposed framework is to minimise the amount of effort needed to develop tools in the future. The amount of effort that is required to create a specific transformation from a MP language to the generic model is unknown. If the creation of such transformations requires more time than using a language-specific tool, it might defeat the purpose of this framework.

Besides effort, validity and accuracy is a major concern. The transformation of a specific MP language to the generic model might lead to many details about the context getting lost in translation. This will affect the validity and accuracy of the framework.

Every language is designed in a slightly different way and has users that each incorporate certain paradigms in a different way. One can argue that generalising MP languages as being equal is not reliable for every specific language. Therefore, comparison with benchmark data of specific languages will be necessary.

6. Conclusion

The combination of OOP with FP allows developers to be more expressive in their practice but it can also be a pitfall. Code quality assurance can help monitor the well-being of a source code. For multi-paradigm languages, it is important to capture the code quality with the combination of OOP and FP. We have shown an approach for the design of a multi-paradigm code quality assurance framework. It transforms MP languages into a generic model. This generic model is used to extract metrics. The metrics need only to be implemented on the generic model, thereby possibly reducing the amount of effort needed to measure the code quality of any MP language.

References

[1] A. Khanjani, R. Sulaiman, The process of quality assurance under open source software development, in: 2011 IEEE Symposium on Computers & Informatics, 2011, pp. 548–552. doi:10.1109/ISCI.2011.5958975.

[2] M. Jureczko, D. Spinellis, Using Object-Oriented Design Metrics to Predict Software Defects, volume Models and Methodology of System Dependability of *Monographs of System Dependability*, Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland, 2010, pp. 69–81.

[3] D. de Champeaux, D. Lea, P. Faure, The process of object-oriented design, in: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '92, Association for Computing Machinery, New York, NY, USA, 1992, p. 45–62. URL: https://doi.org/10.1145/141936.141942. doi:10.1145/141936.141942.

[4] C. Ryder, S. Thompson, Software Metrics: Measuring Haskell, in: M. van Eekelen (Ed.), Trends in Functional Programming, Trends in Functional Programming, Intellect Books, Bristol, UK, 2005, pp. 1–17. URL: https://kar.kent.ac.uk/14265/.

[5] E. Landkroon, Code Quality Evaluation for the Multi-paradigm Programming Language Scala, Master's thesis, Universiteit van Amsterdam, 2017.

[6] B. Zuilhof, R. van Hees, C. Grelck, Code Quality Metrics for the Functional Side of the Object-Oriented Language C#, in: SATToSE, 2019, pp. 1–10.

[7] S. Konings, Source code metrics for combined functional and Object-Oriented Programming in Scala, Master's thesis, University of Twente, 2020. URL: http://essay.utwente.nl/85223/.

[8] P. Carbonnelle, PYPL popularity of Programming Language index, https://pypl.github.io/PYPL.html, 2022. Accessed: 04-07-2022.

[9] Oracle, What's New in JDK 8, https://www.oracle.com/java/technologies/javase/8-whats-new.html, 2022. Accessed: 04-07-2022.

[10] B. Stefan L. Ram, Dr. alan kay on the meaning of ”object-oriented programming” (document), 2003. URL: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en.

[11] A. Church, A set of postulates for the foundation of logic, Annals of Mathematics 33 (1932) 346–366. URL: http://www.jstor.org/stable/1968337.

[12] J. Hughes, Why Functional Programming Matters, Computer Journal 32 (1989) 98–107. URL: http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf.

[13] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, K. Houston, Object-Oriented Analysis and Design with Applications, Third Edition, third ed., Addison-Wesley Professional, 2007.

[14] P. Van Roy, et al., Programming paradigms for dummies: What every programmer should know, New computational paradigms for computer music 104 (2009) 616–621.

[15] D. A. Spuler, A. S. M. Sajeev, D. A. Spuler, A. S. M. Sajeev, Abstract Compiler Detection of Function Call Side Effects, 1994.

[16] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Comput. Surv. 21 (1989) 359–411. URL: https://doi.org/10.1145/72551.72554. doi:10.1145/72551.72554.

[17] H. H. Barendregt, E. Barendsen, Introduction to lambda calculus, Nieuw archief voor wisenkunde 4 (1984) 337–372.

[18] S. Pedersen, The impact of Poor Software Quality, 2021. URL: https://www.buildingbettersoftware.com/blog/the-impact-of-poor-software-quality/.

[19] ISO/IEC 25010, ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, Technical Report, ISO/IEC, 2011.

[20] ISO/IEC, ISO/IEC 9126. Software engineering – Product quality, ISO/IEC, 2001.

[21] I. Heitlager, T. Kuipers, J. Visser, A practical model for measuring maintainability, in: 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), 2007, pp. 30–39. doi:10.1109/QUATIC.2007.8.

[22] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493. doi:10.1109/32.295895.

[23] W. Li, Another metric suite for object-oriented programming, Journal of Systems and Software 44 (1998) 155–162. URL: https://www.sciencedirect.com/science/article/pii/S0164121298100523. doi:https://doi.org/10.1016/S0164-1212(98)10052-3.

[24] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, Journal of Systems and Software 128 (2017) 164–197. URL: https://www.sciencedirect.com/science/article/pii/S0164121217300663. doi:https://doi.org/10.1016/j.jss.2017.03.044.

[25] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (1976) 308–320. doi:10.1109/TSE.1976.233837.

[26] X. Rival, K. Yi, Introduction to Static Analysis: An Abstract Interpretation Perspective, MIT Press, 2020. URL: https://books.google.nl/books?id=96LLDwAAQBAJ.

[27] V. Zaytsev, A. H. Bagge, Parsing in a Broad Sense, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahão, E. Insfran (Eds.), Proceedings of the 17th International Conference on Model

Driven Engineering Languages and Systems (MoDELS 2014), volume 8767 of *LNCS*, Springer, 2014, pp. 50–67. doi:10.1007/978-3-319-11653-2_4.

[28] K. Bhatt, V. Tarey, P. Patel, K. B. Mits, D. Ujjain, Analysis of source lines of code (SLOC) metric, International Journal of Emerging Technology and Advanced Engineering 2 (2012) 150–154.

[29] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[30] F. E. Allen, Control flow analysis, SIGPLAN Not. 5 (1970) 1–19. URL: https://doi.org/10.1145/390013.808479. doi:10.1145/390013.808479.

[31] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, second edition, Synthesis Lectures on Software Engineering 3 (2017) 1–207. URL: https://doi.org/10.2200/S00751ED2V01Y201701SWE004. doi:10.2200/S00751ED2V01Y201701SWE004. arXiv:https://doi.org/10.2200/S00751ED2V01Y201701SWE004.

[32] M. L. Collard, M. J. Decker, J. I. Maletic, Lightweight Transformation and Fact Extraction with the srcML Toolkit, in: Proceedings of the 11th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society, 2011, pp. 173–184. URL: https://doi.org/10.1109/SCAM.2011.19. doi:10.1109/SCAM.2011.19.

[33] P. Klint, T. van der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.

[34] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, J. J. Vinju, M3: A general model for code analytics in rascal, in: O. Baysal, L. Guerrouj (Eds.), 1st IEEE International Workshop on Software Analytics, SWAN 2015, Montreal, QC, Canada, March 2, 2015, IEEE Computer Society, 2015, pp. 25–28. URL: https://doi.org/10.1109/SWAN.2015.7070485. doi:10.1109/SWAN.2015.7070485.

[35] N. de Groot, Analysing and Manipulating CSS using the M^3 Model, Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, 2016. URL: http://www.scriptiesonline.uba.uva.nl/613750.

[36] D. Owens, A generic framework facilitating automated quality assurance across programming languages of disparate paradigms, Ph.D. thesis, Edge Hill University, Ormskirk, UK, 2016. URL: http://repository.edgehill.ac.uk/7778/.