A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns

Dario Di Nucci[1] Hoang Son Pham[2] Johan Fabry[3] Coen De Roover[1]
Kim Mens[2] Tim Molderez[1] Siegfried Nijssen[2] Vadim Zaytsev[3]

[1]Vrije Universiteit Brussel, Belgium [2]Université catholique de Louvain, Belgium
[3]Raincode Labs, Belgium

Abstract

In an ongoing industry-university collaboration we are developing a language-parametric framework for mining code idioms in legacy systems. This modular framework has a pipeline architecture and a language-parametric meta representation of the artefacts used by each of its 5 components: source code importer, mining preprocessor, pattern miner, pattern matcher, and modernisation assistant. The pipeline enables reuse of its components across systems and languages, as well as for project partners to work on each of these components separately. An example is the exploration of novel pattern mining techniques independently of the languages on which they will be applied and the modernisation assistant in which they will be used. Our first results on mining Java and COBOL code are promising, even though challenges still lie ahead to make the framework and its constituting components truly scalable, customisable, and language independent.

1 Introduction

Legacy systems have been informally defined as "large software systems that we do not know how to cope with but that are vital to our organisation" [1, 2]. To keep their business value, legacy systems must evolve over time by being replaced, redeveloped, rearchitected, reengineered, reused, or by having their software components and platforms migrated when traditional maintenance practices can no longer achieve the desired system properties [3]. Technology consulting firms estimate that 180–200 billion lines of legacy code are still in active use today [4]. Since the potential benefits for legacy system modernisation are well recognised, these systems are being slowly replaced or retired in favour of alternatives.

This paper presents an initial framework that is being developed by two universities and a legacy modernisation company in the context of a code mining project. The company has been active since 1998, had a series of successful migration projects with a streak of satisfied customers, and has already won three migration-related technology excellence awards from Microsoft. The project's objectives, elaborated upon later in the paper, are to advance the state of the art in legacy modernisation by applying a novel merge of techniques from artificial intelligence, pattern mining, and program analysis.

Software systems that are regarded as legacy by their owners consist of more than just the old, obsolete, and soon to be retired artefacts written in 1960s languages like assembler [5] and COBOL [6]. As time went by, the circle of legacy has started to include systems that were built with 4^{th} generation languages (4GLs) of the 1980s [7], developed using model-driven architecture (MDA) of the 1990s [8], or created using domain-specific languages (DSLs) of the 2000s [9]. It is thus crucial for legacy software modernisation companies to be able to adapt to new languages and previously unknown idioms.

Conquering even one legacy ecosystem with all its languages, dialects, configurations and preprocessors, is a substantial effort for a company. It is beyond trivial to reuse knowledge about prior successful mi-

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.org

gration projects to cope with the next one, for each of them is unique in some way. The patterns to solve the Y2K problem [10] are drastically different from patterns for database migration or turning procedural to object-oriented code, and renovation patterns, working effectively in one 4GL, are often inapplicable to another 4GL. In this context, any degree of automation in the discovery and detection of coding idioms and modernisation patterns and their corresponding code transformation actions is worthwhile to pursue. However, traditional software analysis and analytics tools are usually geared towards detecting precise matches for *known* patterns, such as a particular combination of conditions and GO TO jumps that can be refactored into a WHILE loop. What is really needed instead, and what we are aiming to achieve, is the ability to find and act upon *unknown* patterns that are perhaps only adhered to a limited extent.

After having introduced the context of our work, the rest of this paper is organised as follows: section 2 explains our objectives in sufficient detail to appreciate the rest; section 3 dives into prior related work around *code idioms*—patterns that we are mining for; section 4 visualises the pipeline of our framework (Figure 1) and explains its components; section 5 reports on preliminary results and concludes the paper.

2 Project Goals

The goal of our work is to design and implement a framework to explore novel pattern mining algorithms for source code and to incorporate them in an intelligent software modernisation assistant tool set. Ideally, at the end of the project (end 2020), we should have a tool set powerful enough to help legacy software engineers analyse a previously unseen codebase in some software language for previously unknown patterns. With these tools, it should be possible to analyse the available data (often just source code) quickly and efficiently, recognise frequently occurring patterns, confront domain experts with them and annotate them with modernisation actions to produce a mature modernisation solution within weeks, not decades.

The framework being developed is language-parametric thanks to a metamodel representation that is able to support a variety of software languages. The modernisation assistant will pro-actively recommend source code modernisation actions [11] by comparing the code being renovated with insights gained by treating the source code and development history as data. The assistant will continuously mine for previously unknown patterns within the system's source code and structure. Thus, the modernisation recommendations made by the assistant can improve over time as it refines or uncovers more previously unknown patterns.

The three main goals of our framework are to:

1. **Discover syntactic patterns** to replace large, repeated, error-prone programming idioms [12] by more succinct macros or proven programming language built-ins, with the purpose of improving code reliability, understandability, and maintainability.

2. **Discover code deviating from expected patterns** which may be indicative of dissimilarities and dormant errors.

3. **Propose actions to improve respect of idioms** such as rewriting old-style FOR loops to functional alternatives in Java 8+ or replacing ad hoc string manipulations in older COBOL versions with modern equivalents from the standard library.

3 Idiomatic Code Patterns

Coding conventions and idioms are syntactic patterns in the source code. Conventions describe an overall syntactic style that is meant to foster readability and maintainability of source code [13]. Idioms are fragments of code that recur frequently across different projects, and play one semantic role [12]. A piece of code is often termed idiomatic if experienced developers consider it to be written in an intuitive, natural way. An idiom can be described in the form of a code template, *i.e.,* a snippet of code where parts can be abstracted away with meta-variables. Examples of scenarios that can be described with idioms include iteration over a data structure, manipulating resources (open, close, lock, *etc.*), handling errors, or executing database transactions.

IDEs often offer facilities to manually define idioms and insert them whenever needed. However, these do not help programmers if they are using a language or library the IDE is not familiar with. To assist programmers, Allamanis *et al.* [12] describe an approach that mines for code idioms in a corpus of idiomatic code. These idioms are represented as a syntactic probabilistic model that uses probabilities to measure the quality of a proposed idiom. Similar approaches have been used for measuring how natural/idiomatic code is, or how it changes when bugs are fixed [14, 15, 16, 17]. Based on such measures, these approaches have all found that software is repetitive—in other words, that idioms are often used.

Allamanis *et al.* created the *Naturalize* tool [18], which learns the coding convention style of a program and suggests changes to improve code consistency. It uses statistical natural language processing to suggest natural identifier names and formatting conventions.

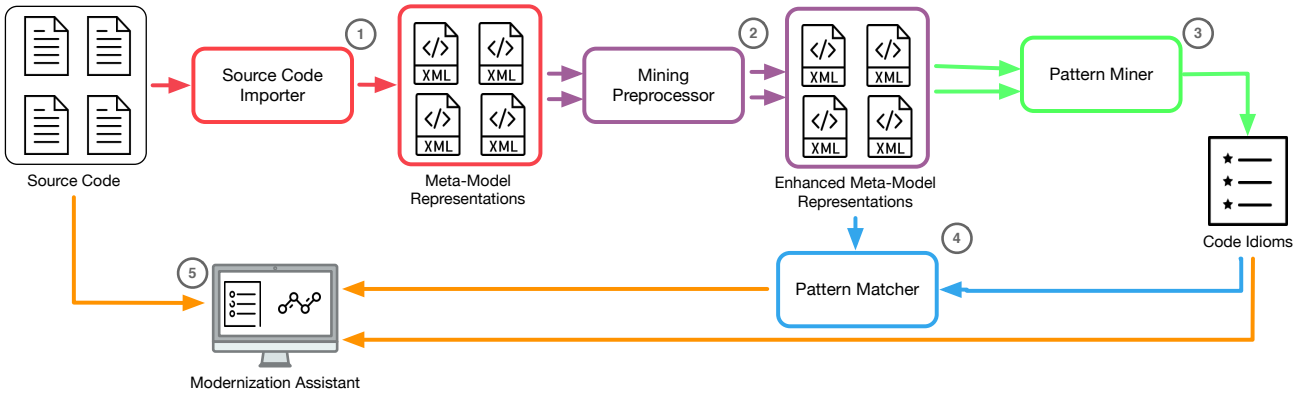

Figure 1: Our Language-Parametric Modular Framework for Mining Idiomatic Code Patterns

A follow-up project [19] focused on suggesting appropriate method and class names from their bodies by using a neural network and an n-gram language model.

As idioms and coding conventions directly relate to a programming language's syntax, most existing work in this area focuses on tools targeted at one specific language. Our work goes beyond this through the use of metamodels to provide a language-parametric representation for idioms and conventions. Our goal is to demonstrate that patterns can be mined across multiple languages with relatively small tooling effort.

Considering language-parametric or language-independent representations of source code, there have been multiple efforts in this area. An arguably well-known example is MOOSE [20] and its FAMIX [21] metamodel. Their focus is however different to our work. Firstly, MOOSE was originally created for the re-engineering of object-oriented systems, whereas we do not have any restriction at all on the paradigm of the programming language. Secondly, the FAMIX metamodel allows for its instances (i.e. models of programming languages) to abstract over certain parts of the programs being modeled. Typically, such models do not contain any information at a granularity finer than method invocations. For pattern mining, we however require the complete abstract syntax tree of a program to be present.

Alternatively, Rakič et.al. have worked on language-independent static code analysis [22], based on concrete syntax trees that are enriched with universal nodes: nodes that are considered to be semantically equivalent in all programming languages. However the presence of such nodes does not provide any additional information that is relevant for our work. We search for patterns in the source code, without regarding the semantics of the nodes in a pattern tree. This is because we do not have language-independent patterns as a goal, instead our patterns are specific to the language being mined.

4 The Framework

As depicted in Figure 1, our framework is structured as a pipeline, comprising five main components:

4.1 Source Code Importer

A first challenge of the metamodel for our modernisation assistant is to accommodate multiple (legacy or other) programming languages. Indeed, it would not be economical if a new version of the metamodel had to be re-implemented for every language or even language dialect it is applied to. To address this issue within our framework, the metamodel defines a language-agnostic abstract syntax tree format (AST) for source code.

The format is an XML form of the AST: each AST node is an XML element that has as content the child relationships of the node. Begin and end-tags of AST nodes identify the type of AST node, *e.g.,* `<ForStatement>` is a Java `for` statement node. The relationships inside of a such an element are again XML elements, with as tags the kind of relationship, *e.g.,* in a Java `for` statement node these would be `<initializers>`, `<expression>`, `<updaters>` and `<body>`. Each of these elements again contains a (list of) AST nodes, in XML form.

The purpose of the source code importers is thus to transform programs in a given language to their representation in this format. Fundamentally, the only language-dependent part of the framework is this first step. Once an importer for a language has been created, the remainder of the framework is used as-is.

4.2 Mining Preprocessor

Before they are passed to the pattern miner, the ASTs may be preprocessed in order to enhance the mining process. Different preprocessing steps may be applied, depending on what is being mined for. For example, when considering naming conventions as part of the mining, one preprocessor can split identifiers into a

subtree based on camelcase or based on underscores. Another example would be mining at a granularity of procedure-level entities and hence first removing elements at finer granularities like statements or (module-level) variable declarations.

4.3 Pattern Miner

The pattern miner is responsible for extracting idiomatic code patterns, taking the preprocessed ASTs as input. A concrete example of an idiomatic pattern we found in the project *JHotDraw* is given in Fig. 2. In several instances, a method is defined that instantiates an `AbstractUndoableEdit` object with specific implementations for undo and redo functionality. Note that the ellipses (...) in the pattern are wildcards that can represent any amount of code, illustrating that the miner is able to capture complex patterns that cannot be found otherwise via e.g. clone detection tools.

We are currently exploring the use of frequent graph mining algorithms, though other mining algorithms may be tried in the future. The most popular frequent graph mining algorithms are developed for trees [23] and undirected graphs [24, 25, 26], although standard algorithms produce a (too) large amount of patterns (as discussed in section 5). Thus, an important component of our pattern miner is the definition of the heuristics and constraints used during the mining process, so as to avoid discovering redundant or useless patterns [12]. In particular, our pattern mining algorithm relies on two ideas:

1. maximal frequent subtree mining to ensure that a condensed representation of only large patterns is found

2. constraint-based data mining, in which additional constraints are imposed on the patterns to be found.

The key benefit of constraint-based mining is that it allows developers to specify easy to interpret constraints on the patterns to include in the output of the algorithm.

We are currently exploring what heuristics work best for different kinds of idioms, and how to represent these heuristics in an idiom- and language-agnostic way, so that they can easily be adapted when looking for other kinds of idioms, or when mining other languages.

4.4 Pattern Matcher

The pattern matcher is responsible for finding all AST subtrees that match the patterns extracted by the miner. While these ASTs are already known to the pattern miner, we may want to apply postprocessing steps to the patterns that are found, *e.g.,* to further generalise them such that the patterns are more widely applicable. The pattern matcher is then needed to find matches of these modified patterns. Another application of the pattern matcher is that, when a pattern was mined in one project, the pattern matcher can now match this pattern against any other project. The tool is designed to be language-parametric and is based on code templates [27, 28]. A template is a concrete snippet of source code, in which some parts can be replaced by wildcards or metavariables. It is also possible to attach so-called "directives" to parts of the snippet, which can affect the semantics of the pattern to match in various ways.

```
protected void ...() {
   ...
   final ArrayList<Object> restoreData =
      new ArrayList<Object>(...);
   ...
   UndoableEdit edit = new AbstractUndoableEdit() {
      ...
      @Override
      public String getPresentationName() { ... }
      ...
      @Override
      public void undo() {
         super.undo();
         Iterator<Object> iRestore =
            restoreData.iterator();
         ...
      }
      ...
      @Override
      public void redo() {
         super.redo();
         ...
      }
   };
   fireUndoableEditHappened(edit);
}
```

Figure 2: Undo/redo pattern in JHotDraw

4.5 Modernisation Assistant

The modernisation assistant provides a GUI that allows a user to inspect all patterns uncovered by the pattern miner, and their matches, both as text and as graphs. The screenshots in Fig. 3 and Fig. 4 respectively show a match of the JHotdraw undo/redo pattern in a specific source file, and the graph representation of this pattern. The engineer is presented a list of patterns with their pattern size, support, confidence, and type of root AST node. A specific pattern can be selected for inspection showing an overview of pattern matches in the source code as well as concrete source code snippets highlighted according to the structure of the pattern. The graph representation of the pattern essentially is an AST, where certain nodes are annotated with the directives mentioned in Sec. 4.4. For instance, in Fig. 4, a "match-set" directive is attached to an `AnonClassDecl`, which indicates that this

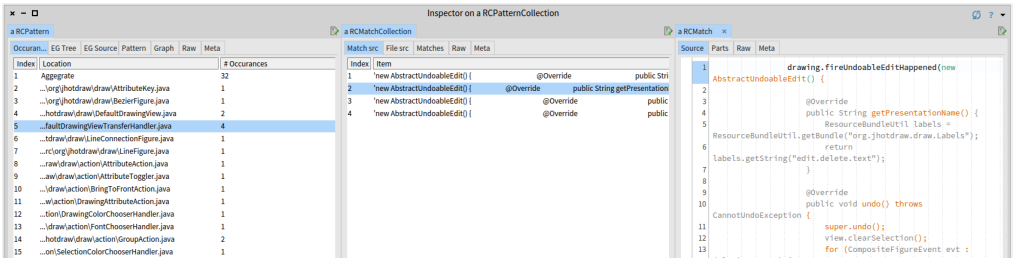

Figure 3: Modernisation assistant showing a pattern match

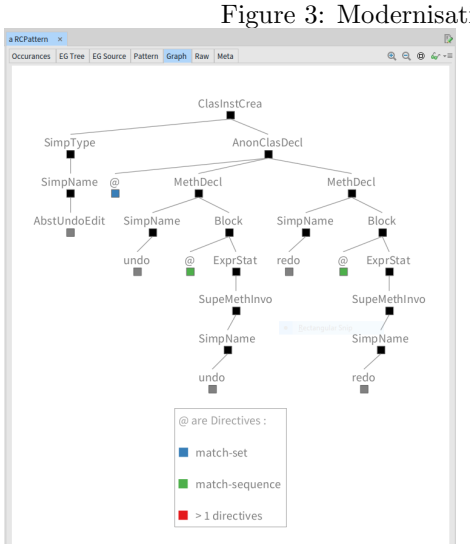

Figure 4: Graph representation of a pattern

`AnonClassDecl` node will match as long as its children (two method declarations) can be found, even if the actual matching node contains additional children or they appear in a different order.

The modular architecture of our framework is key to achieve our research objectives. For example, given a new programming language we mainly need to provide a new *Source Code Importer*. However, we may also define or configure a *Preprocessor* specific to the kind of idioms we want to mine for in that language, and that we need to adapt the heuristics and constraints used by the *Pattern Miner*. But the general pipeline and algorithms would remain the same. Similarly, if we would like to explore alternative or more advanced pattern mining algorithms, in a language-agnostic way, this could be done mostly by replacing the *Pattern Miner*.

5 Preliminary Results & Challenges

In this section, we report on the current state of the implementation of our framework, some preliminary results, as well as some of the challenges we have faced:

5.1 Source Code Importer

We currently have importers for COBOL and Java. [1] The former is pragmatic custom code that is able to process the entire NIST COBOL 85 compliance test suite[2] as well as the code for a variety of industrial legacy systems. The latter uses the Eclipse Java meta-model and is able to successfully produce ASTs for all source code in `QUAATLAS` [29]: a refined subset of the Qualitas Corpus [30] of Java programs. The importers also produce a description of the grammar of the language that is used by the miner. Again, the Java importer uses the Eclipse Java meta-model to produce this grammar, whereas for the COBOL importer this is custom code.

5.2 Mining Preprocessor

For the moment, we have only implemented a preprocessing component that is able to split the identifiers contained in a node into a subtree based on camelcase or the dash/underscore convention. When using that preprocessor, instead of considering identifiers as similar only when they are equal, identifiers can be matched at a finer-grained level based on the similar keywords they contain.

5.3 Pattern Miner

Our pattern miner implements an extended and adapted version of the *FreqT* [31] frequent subtree mining algorithm. Although we have found that pure FreqT can indeed be used for mining idiomatic code patterns, it does have some limitations such as being highly time consuming and generating a large amount of patterns as well as redundant patterns. To tackle these problems, we have been exploring various customizations of the FreqT algorithm. As a result, we have managed to reduce the execution time of FreqT significantly, and to limit the number of discovered patterns. Although we have not completed a full empirical study yet, many of the discovered patterns seem to correspond to relevant code idioms.

[1] We are currently working on an importer for C# as well.
[2] https://www.itl.nist.gov/div897/ctg/cobol_form.htm

To achieve these results, we had to use a variety of heuristics and constraints. However, selecting the appropriate constraints to apply is not a trivial task since it seems to depend partly on the language and on the kinds of patterns one wants to find. Even though those constraints can easily be configured for other languages and other kinds of patterns, it is less obvious how to choose the appropriate constraints for legacy languages that are less well-known, or when we do not know upfront what kind of patterns we are looking for. A particular challenge of our current research therefore remains how to efficiently search for and evaluate interesting and surprising patterns. As it is difficult, nor is this the focus of our work, to measure how exhaustive our approach is, we believe our framework's value lies in uncovering any new interesting patterns that would be difficult to find otherwise. As such, aside from measuring the miner's scalability towards larger projects, our evaluation will mainly consider qualitative aspects, *e.g.,* how many patterns are genuinely useful? ; do patterns tend to be project-specific, or general-purpose? ; can these patterns be classified in a number of categories? ; given different configurations, what is the ratio of interesting/non-interesting patterns?

5.4 Pattern Matcher

Currently, our pattern matcher is able to match precise syntactic patterns. In the future, we plan to support anomaly detection including the on-demand detection of partial matches for a given mined pattern. To facilitate inspection by a software engineer, the pattern matching algorithm should also quantify its results by indicating the extent to which a partial match corresponds to a given pattern.

5.5 Modernisation Assistant

Based on the output of the miner and pattern matcher, the modernisation assistant is able to visualise patterns, matches and their corresponding source code. Despite its seemingly summarising role, it was useful from very early on in the project to explore mining results and let human users interpret them. It has consequently been a driving force in customising the miner and matcher to provide results that are more straightforwardly interpretable by a modernisation engineer. For example, we found that since patterns are subtrees with parts that are left unspecified, it is important for highlighted source code to show which part of the source code is specified by the pattern and which part is not. Hence, the pattern matcher should include this information in each match.

6 Conclusions and Future Work

In this paper we have outlined our language-parametric modular framework for mining idiomatic code patterns whose goal is to assist software modernization engineers in their work of migrating legacy systems. We reported some preliminary results, as well as some challenges we faced.

The most notable challenges lie in configuring and selecting the appropriate heuristics and constraints when mining to guide the algorithm towards the kinds of patterns one wants to find. This is particularly relevant since the modernisation engineer will face languages that are unknown to us and will not know upfront what kind of patterns to look for. In light of this, our focus is currently on establishing how to efficiently search for and evaluate interesting and surprising patterns. This would allow for easier experimentation with heuristics and constraints.

Obviously, more challenges still remain to make our framework truly scalable and language independent, but our promising first results make us confident that our goals will be reached.

Acknowledgments

The project is funded by the Belgian Innoviris TeamUp project INTiMALS (2017-TEAM-UP-7).

References

[1] K. Bennett, "Legacy Systems: Coping with Success," *IEEE Software*, vol. 12, no. 1, pp. 19–23, 1995.

[2] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "Legacy Information Systems: Issues and Directions," *IEEE Software*, vol. 16, no. 5, pp. 103–111, 1999.

[3] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage, "How Do Professionals Perceive Legacy Systems and Software Modernization?" in *ICSE'14*. ACM, 2014, pp. 36–47.

[4] N. Veerman, "Revitalizing modifiability of legacy assets," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 4-5, pp. 219–254, 2004.

[5] V. Blagodarov, Y. Jaradin, and V. Zaytsev, "Raincode Assembler Compiler," in *SLE'16*, 2016, pp. 221–225.

[6] M. P. A. Sellink, H. M. Sneed, and C. Verhoef, "Restructuring of COBOL/CICS Legacy Systems," in *CSMR'99*. IEEE, 1999, pp. 72–82.

[7] V. Zaytsev, "Open Challenges in Incremental Coverage of Legacy Software Languages," in *PX/17.2*, 2017, pp. 1–6.

[8] S. J. Mellor, K. Scott, A. Uhl, D. Weise, and R. M. Soley, *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[9] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*, 2013.

[10] C. Jones, *The Year 2000 Software Problem: Quantifying the Costs and Assessing the Consequences*. ACM Press/Addison-Wesley, 1997.

[11] A. F. Iosif-Lazar, A. S. Al-Sibahi, A. S. Dimovski, J. E. Savolainen, K. Sierszecki, and A. Wasowski, "Experiences from Designing and Validating a Software Modernization Transformation," in *ASE'15*. IEEE, 2015, pp. 597–607.

[12] M. Allamanis and C. Sutton, "Mining Idioms from Source Code," in *FSE'14*. ACM, 2014, pp. 472–483.

[13] B. Goncharenko and V. Zaytsev, "Language Design and Implementation for the Domain of Coding Conventions," in *SLE'16*, 2016, pp. 90–104.

[14] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the Naturalness of Buggy Code," in *ICSE'16*. IEEE, 2016, pp. 428–439.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the Naturalness of Software," in *ICSE'12*. IEEE, 2012, pp. 837–847.

[16] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza, "On the Uniqueness of Code Redundancies," in *ICPC'17*, 2017, pp. 121–131.

[17] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax Errors Just aren't Natural: Improving Error Reporting with Language Models," in *MSR'14*. ACM, 2014, pp. 252–261.

[18] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning Natural Coding Conventions," in *FSE'14*. ACM, 2014, pp. 281–293.

[19] ——, "Suggesting Accurate Method and Class Names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[20] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli, "Meta-environment and executable meta-language using smalltalk: an experience report," *Software & Systems Modeling*, vol. 8, no. 1, pp. 5–19, 2009.

[21] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings International Symposium on Principles of Software Evolution*. IEEE, 2000, pp. 154–164.

[22] G. Rakić, Z. Budimac, and M. Savić, "Language independent framework for static code analysis," in *Proceedings of the 6th Balkan Conference in Informatics*, ser. BCI '13. New York, NY, USA: ACM, 2013, pp. 236–243. [Online]. Available: http://doi.acm.org/10.1145/2490257.2490273

[23] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent Subtree Mining—An Overview," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2005.

[24] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," in *ICDM'01*. IEEE, 2001, pp. 313–320.

[25] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *ICDM'02*. IEEE, 2002, pp. 721–724.

[26] S. Nijssen and J. N. Kok, "A Quickstart in Frequent Structure Mining Can Make a Difference," in *KDDM'04*. ACM, 2004, pp. 647–652.

[27] C. De Roover and K. Inoue, "The ekeko/x Program Transformation Tool," in *SCAM'14*. IEEE, 2014, pp. 53–58.

[28] T. Molderez and C. De Roover, "Automated Generalization and Refinement of Code Templates with ekeko/x," in *SANER'16*, vol. 1. IEEE, 2016, pp. 669–672.

[29] C. De Roover, R. Lammel, and E. Pek, "Multi-dimensional Exploration of API Usage," in *ICPC'13*. IEEE, 2013, pp. 152–161.

[30] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," in *APSEC'10*. IEEE, 2010, pp. 336–345.

[31] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, *Efficient Substructure Discovery from Large Semi-structured Data*, 2002, pp. 158–174.