RICE UNIVERSITY

Corpus-Driven Programming Systems for Program Synthesis and Refactoring

by

Yanxin Lu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Swarat Chaudhuri, Chair
Associate Professor of Computer Science

Christopher Jermaine
Professor of Computer Science

Ankit B. Patel
Assistant Professor of Electrical and
Computer Engineering

Houston, Texas

August, 2018

Contents

Illustrations

Tables

Chapter 1

Introduction

With the advancement in technologies such as artificial intelligence and also the expansions of high-tech companies, computer programming starts to become an important skill, and the demand for programmers has been growing dramatically in the past few years. The overall productivity has been boosted significantly thanks to the increasing number of programmers, but we still have not witnessed any boost in individual programming productivity.

The most important reason is that programming is a difficult task. It requires programmers to deal with extremely low-level details in complex software projects, and it is almost inevitable for programmers to make small mistakes. People tend to assume that a piece of untested software does not function properly. To deal this the problem, software engineering techniques and formal method based techniques have been proposed to help facilitate programming. These techniques include various software engineering methodologies, design patterns, sophisticated testing methods, program repair algorithms, model checking algorithms and program synthesis methods. Some techniques such as software engineering methodologies, design patterns and unit testing have been practical and useful in boosting programming productivity and the industry has been adopting these techniques for more than a decade. The main reason for its popularity and longevity is that these techniques are quite easy to execute for average programmers. However, one dominant problem with these software engineering approaches is that they are not rigorous enough. If the specification

of a method is not followed strictly, its benefits will tend to be hindered. Advance methods with more rules have been proposed, but the specification tend to be vague sometimes, which results in execution difficulties.

Some researchers switched their attention to applying formal methods to tackle the difficulties in programming. Methods such as model checking and program synthesis are much more rigorous than traditional software engineering techniques, and its performance and benefit is guaranteed once everything works accordingly. However, the impact of these formal methods technique is much less compared to the influence brought by the software engineering techniques, and the reason is that it is very likely that a formal method based approach will not work when large input is provided, because it will not terminate and produce any useful result due to its large search space. These large search spaces are inevitable, since formal methods techniques typically deal with extremely complex problems in theory. However, people have been trying to make formal method approaches practical by introducing additional hints [1] or by restricting the problem domain [2, 3, 4].

With the advent of "big data", researchers started to pay attention to the problems that were considered difficult or impossible, and this has led to a significant advancement in the area of machine learning. Similarly, as more and more open source repositories such as `Google Code`, `Github` and `SourceForge` have come online where thousands of software projects and their source code become available, researchers from the programming language community also started to consider using "big code" to tackle the problems that were considered difficult. With the help of "big code", many new techniques that use formal methods and aim to facilitate programming have been proposed. These techniques include program property prediction [5, 6], API sequence prediction [7, 8, 9] and small program generations [10],

Researchers have showed that using data can indeed make the problem of synthesis feasible [10] and practical tools that can help human developers have also started to appear and programmers have started to use those in practice [6, 8].

Two major types of algorithms were used in the current literature of applying formal methods to software engineering. The first type of algorithms is based on combinatorial search. Combinatorial search plays an important role in model checking and traditional program synthesis problems [11, 12, 13, 14, 15, 16, 17, 18, 4]. The main idea is to first define a goal and also the steps for reaching the goal. Programmers can then let the computer to search for an solution. Typically heuristics are defined to reduce the search space and to speed up the search time. The advantages of search-based methods include (1) it is relatively easy to implement and it can be used to solve problems where no efficient solutions exist, (2) sometimes the algorithms can discover results that are hard to think about as humans because computers can easily discover solutions in a large search space quickly compared to humans, and (3) search-based methods can solve problems that requires precision and precision is typically required for analyzing computer programs. As SAT solvers and SMT solvers became sophisticated, people have been able to use those fast solvers to gain significant performance boost. The biggest drawback of search-based methods is its high algorithmic complexity. The search space grows indefinitely as the input size graduately increases and this is the main reason why most traditional model checking methods and program synthesis algorithms cannot deal with large programs [4]. Another drawback that is worth mentioning is that search-based methods tend to be quite fragile. Those methods typically require inputs at every step to be extremely precise, or the algorithm would not perform as expected.

The second type of algorithms is based on learning. The idea of learning is to let

machine improve its performance using data in solving a task and during the process learning-based methods are able to capture idioms that are essential in solving the problem. These idioms are typically hard to express or discover for humans. The large amount of data was not available online until around 2012 and after that researchers started applying learning-based methods to programming systems [5, 6, 7, 8, 9, 10]. The biggest advantage brought by "big data" or "big code" is that it allows researchers to find idioms that reduce the search space significantly by using machine learning techniques. Examples include relationships between variable names and their semantics information and API call sequence idioms. These idioms cannot be made available without people analyzing a large amount of data. Another advantage compared to search-based method is its robustness and this is because machine learning algorithms tend to use a large amount of data where small noises are suppressed. Even though data-driven programming systems are quite impactful, learning-based methods are not as accessible as search-based methods because learning-based methods tend to require data. In order to make learning-based algorithms perform well in practice, a large amount of data is typically required and this also leads to a large consumption on time and computation resources which might not be available for everyone.

In this thesis, we propose two additional corpus-driven systems that aim to automate the process of software reuse and software refactoring. In the current literature, the problem of software reuse and refactoring have been both considered, but no systems can fully automate software reuse and refactoring and some state-of-the-art tools [20, 21] still requires human to provide additional hints. By using a large code corpus, we claim that our systems can fully automate the process of software reuse and refactoring without human intervention, and our systems can accomplish the tasks efficiently and help human developers by boosting their program productivity.

1.1 Program reuse via splicing

We first introduce *program splicing*, a programming system that helps human developers by automating the process of software reuse. The most popular workflow nowadays consists of copying, pasting, and modifying code available online and the reason for its domination is that it is relatively easy to execute with the help of internet search. However, this process inherits the drawbacks from programming. This process requires extreme precision and carefulness from programmers similar to normal programming. When a software reuse task happens in a large and complicated software system, the cost of making mistakes and spending enormous time on repairing might exceed the benefit, let alone the fact that programmers sometimes do not even try to fully understand the code they bring in from the internet so long as it appears to work under their specific software environment. This might impose a threat to their future software development progress.

Existing techniques that inspire the idea of our method can be divided into two areas, search-based program synthesis techniques and data-driven methods. The problem of program synthesis has been studied for decades and researchers have been applying search-based methods to tackle the problem for several decades [16, 13, 1, 17, 18, 35]. The main benefit with respect to this work comes from the fact that search-based method can produce results that require precision. This is quite crucial when we aim to generate code snippets that needs to interact with pre-written software pieces and examples might include matching variables that are semantically similar or equivalent. However, the problem with search-based method is that it does not scale well into handling large inputs, which lead to large search spaces, due to the complexity of the problem, and this is the main reason why one of the competing system, μScalpel, is not as efficient as our splicing method. To alleviate the scalability

problem, people have proved that using "big data" can be quite effective [6, 7, 59, 10, 57]. Even though our splicing method does not use any statistical method, we still reduce our search space significantly and achieve high efficiency by relying on using natural language to search a big code corpus [33].

One of our novelty in this work is that we combine the ideas from search-based methods and data-driven methods. To use our programming system for program reuse, a programmer starts by writing a "draft" that mixes unfinished code, natural language comments, and correctness requirements. A program synthesizer that interacts with a large, searchable database of program snippets is used to automatically complete the draft into a program that meets the requirements. The synthesis process happens in two stages. First, the synthesizer identifies a small number of programs in the database [19] that are relevant to the synthesis task. Next it uses an enumerative search to systematically fill the draft with expressions and statements from these relevant programs. The resulting program is returned to the programmer, who can modify it and possibly invoke additional rounds of synthesis.

We present an implementation of program splicing, called Splicer, for the Java programming language. Splicer uses a corpus of over 3.5 million procedures from an open-source software repository. Our evaluation uses the system in a suite of everyday programming tasks, and includes a comparison with a state-of-the-art competing approach [20] as well as a user study. The results point to the broad scope and scalability of program splicing and indicate that the approach can significantly boost programmer productivity.

1.2 API refactoring using natural language and API synthesizer

Software refactoring typically involves reconstructing existing source code without modifying the functionality, and it is important and almost a daily routine that programmer will perform to keep their software projects clean and organized by constructing better abstractions, deleting duplicate codes, breaking down a big functionalities into small pieces that are universally applicable and etc. Software system maintenance is extremely crucial, because a software system can easily deteriorate and become obsolete and useless if maintenance is not done properly and regularly, especially when the external libraries it uses and the other underlying software systems it depends on evolve rapidly nowadays. After several decades of software development, most professional programmers have realized the importance of software refactoring, and software refactoring has been used heavily and regularly in the software industry. Similar to software reuse, software refactoring also inherits the drawbacks from programming. It again requires extreme accuracy from programmers, and programmers tend to make mistakes when they deal with large and complex software systems which typically involves keeping tracking of tens or even hundreds of variables and function components.

In this thesis, we focus on refactoring Application Programming Interface (API) call sequences. An API consists of all the definitions and usages of the resources available for external use from a software system, and almost all software systems are built using various APIs from other software systems nowadays. The process of API refactoring mainly consists of changing the API call sequence defined in one library into another sequence defined in another library. The benefit of performing

API refactoring is identical to general software refactoring, but API refactoring has its specific benefits. The first specific benefit allows programmers to reuse obsolete programs in which programmers can adopt an obsolete programs into the existing programming environment. Another benefit is that it can enhance the performance of existing programs by refactoring the existing program into another program that uses advanced libraries and platforms which typically have better performance.

The main difficulty of API refactoring comes from discovering semantically equivalent API calls between two libraries and how to instantiate the new API calls using the environment's variables so that the resulting API call sequence does not alter the functionality of the original API call sequence. One of the earliest work [21] that aims to help API refactoring requires human interventions. The user of the system needs to formally specify the mapping between the API calls in two libraries, and the system only focuses on refactoring *individual* API calls instead of refactoring sequences. Subsequent research in the area of API refactoring has been limited to the problem of API mapping or API translation. The goal is to discover two API calls that are semantically equivalent. Two types of methods were developed to solve the problem of API translation. The first one involves aligning two API call sequences using a statistical model and the translations can be extracted from the alignment results [23]. This alignment method allows people to find not only one-to-one API translations but also one-to-many API translations, but the downside is that this method requires a large amount of API call sequences to train the underlying statistical method. Another method relies on natural language features such as Javadoc to find semantically equivalent API calls [24, 25, 22]. Since Javadoc contains descriptions on the nature of API calls, correct translations can be found by calculating the similarities between the Javadoc texts of two API calls, and calculating similarities can easily be done

using a standard `Word2Vec` model which is able to calculate semantic similarities between words. The only drawback of using natural language features as the main glue is that it is difficult to discover one-to-many API translations.

In this thesis, we propose a new algorithm that automates the process of API refactoring by combining the natural language technique [24] and an state-of-the-art API call sequence synthesizer called `Bayou` [8]. The input to our algorithm includes an API call sequence and the name of the destination library, and our algorithm can produce another semantically equivalent sequence that uses only the API calls defined in the destination library. We solves the problem in two steps. We first translate the input API call sequences into a set of stand-alone API calls defined in the destination library using natural language features as the main driver [24, 25]. Then we feed the stand-alone API calls into a API sequence synthesizer called *Bayou* [8] which in turn synthesizes a complete sequence of API calls. We have designed a series of benchmark problems to evaluate the accuracy of our API refactoring algorithm, and here the accuracy is defined as the percentage of corrected generated API calls. The results show that our algorithm is able to refactor API call sequences accurately, given that the two involved libraries have similar coding practices and the input sequence is not rare in the training data.

Bibliography

[1] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 497–518, 2012.

[2] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 317–330, ACM, 2011.

[3] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 50–61, ACM, 2011.

[4] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, (New York, NY, USA), pp. 13–24, ACM, 2010.

[5] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based Semantic Code Search over Partial Programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 997–1016, ACM, 2012.

[6] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from

"big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, (New York, NY, USA), pp. 111–124, ACM, 2015.

[7] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 419–428, ACM, 2014.

[8] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," *arXiv preprint arXiv:1703.05698*, 2017.

[9] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for finding api usage errors," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 151–162, ACM, 2017.

[10] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[11] Z. Manna and R. Waldinger, "Fundamentals of deductive program synthesis," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 674–704, Aug. 1992.

[12] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 1–8, IEEE, Oct. 2013.

[13] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, (New York, NY, USA), pp. 404–415, ACM, 2006.

[14] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), pp. 166–178, ACM, 2015.

[15] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 43–54, ACM, 2015.

[16] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," in *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, ICALP '89, (London, UK, UK), pp. 652–671, Springer-Verlag, 1989.

[17] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, vol. 40, pp. 1–25, 2015.

[18] J. K. Feser, S. Chaudhuri, and I. Dillig, "Synthesizing data structure transformations from input-output examples," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, (New York, NY, USA), pp. 229–239, ACM, 2015.

[19] J. Zou, R. M. Barnett, T. Lorido-Botran, S. Luo, C. Monroy, S. Sikdar, K. Teymourian, B. Yuan, and C. Jermaine, "Plinycompute: A platform for high-performance, distributed, data-intensive tool development," in *Proceedings of the*

2018 International Conference on Management of Data, pp. 1189–1204, ACM, 2018.

[20] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, (New York, NY, USA), pp. 257–269, ACM, 2015.

[21] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *ACM SIGPLAN Notices*, vol. 40, pp. 265–279, ACM, 2005.

[22] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 307–318, IEEE Computer Society, 2009.

[23] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between apis," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 82–91, IEEE Press, 2013.

[24] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams, "Discovering likely mappings between apis using text mining," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pp. 231–240, IEEE, 2015.

[25] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 756–758, IEEE, 2016.

[26] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean," *arXiv preprint arXiv:1511.08497*, 2015.

[27] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pp. 83–92, IEEE, 2004.

[28] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 485–495, IEEE Computer Society, 2009.

[29] H. Sajnani, V. Saini, J. Ossher, and C. Lopes, "Is popularity a measure of quality? an analysis of maven components," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pp. 231–240, Sept 2014.

[30] J. Ossher, H. Sajnani, and C. Lopes, "Astra: Bottom-up construction of structured artifact repositories," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 41–50, 2012.

[31] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241 – 259, 2014.

[32] A. Solar-Lezama, "The sketching approach to program synthesis," in *Asian Symposium on Programming Languages and Systems*, pp. 4–13, Springer, 2009.

[33] V. Kashyap, D. B. Brown, B. Liblit, D. Melski, and T. W. Reps, "Source forager: A search engine for similar source code," *CoRR*, vol. abs/1706.02769, 2017.

[34] H. Feild, D. Binkley, and D. Lawrie, "An empirical comparison of techniques for extracting concept abbreviations from identifiers," in *Proceedings of IASTED International Conference on Software Engineering and Applications (SEA06)*, Citeseer, 2006.

[35] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri, "Synthesizing transformations on hierarchically structured data," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 508–521, ACM, 2016.

[36] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 599–612, ACM, 2017.

[37] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," *CoRR*, vol. abs/1611.07502, 2016.

[38] O. Polozov and S. Gulwani, "Flashmeta: A framework for inductive program synthesis," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 107–126, 2015.

[39] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013.

[40] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," *Dependable Software Systems Engineering*, vol. 40, pp. 1–25, 2015.

[41] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 243–253, IEEE Computer Society, 2009.

[42] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE software*, vol. 25, no. 5, 2008.

[43] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "Applying test-driven code search to the reuse of auxiliary functionality," in *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 476–482, ACM, 2009.

[44] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss, "Hunter: Next-generation code reuse for java," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 1028–1032, ACM, 2016.

[45] T. Gvero and V. Kuncak, "Interactive synthesis using free-form queries," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, pp. 689–692, IEEE, 2015.

[46] "Beanshell," 2017. Accessed: 2017-04-04.

[47] "Nailgun," 2017. Accessed: 2017-04-04.

[48] B. Efron, *The jackknife, the bootstrap and other resampling plans*. SIAM, 1982.

[49] "Jsoup," 2017. Accessed: 2017-04-02.

[50] "Stackoverflow," 2017. Accessed: 2017-04-02.

[51] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, pp. 20:1–20:44, Feb. 2013.

[52] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *European Conference on Genetic Programming*, pp. 137–149, Springer, 2014.

[53] M. Harman, Y. Jia, and W. B. Langdon, "Babel pidgin: Sbse can grow and graft entirely new functionality into a real world system," in *International Symposium on Search Based Software Engineering*, pp. 247–252, Springer, 2014.

[54] Y. Jia, M. Harman, W. B. Langdon, and A. Marginean, "Grow and serve: Growing django citation services using sbse," in *International Symposium on Search Based Software Engineering*, pp. 269–275, Springer, 2015.

[55] A. Marginean, E. T. Barr, M. Harman, and Y. Jia, "Automated transplantation of call graph and layout features into kate," in *International Symposium on Search Based Software Engineering*, pp. 262–268, Springer, 2015.

[56] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *ACM SIGPLAN Notices*, vol. 50, pp. 43–54, ACM, 2015.

[57] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, IEEE, 2012.

[58] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 858–868, IEEE Press, 2015.

[59] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *ACM SIGPLAN Notices*, vol. 51, pp. 761–774, ACM, 2016.

[60] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 630–640, IEEE, 2015.

[61] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[62] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105, IEEE Computer Society, 2007.

[63] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 664–675, ACM, 2014.

[64] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Notices*, vol. 49, pp. 349–360, ACM, 2014.

[65] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ACM/IEEE International Conference on Software Engineering*, (New York, New York, USA), ACM Press, May 2010.

[66] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *International conference on Software engineering*, (New York, New York, USA), ACM Press, May 2011.

[67] K. T. Stolee and S. Elbaum, "Toward semantic search via smt solver," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 25, ACM, 2012.

[68] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 295–306, IEEE, 2015.

[69] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1, pp. 325–334, IEEE, 2010.

[70] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 457–468, ACM, 2014.

[71] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 195–204, ACM, 2010.

[72] A. Gokhale, D. Kim, and V. Ganapathy, "Data-driven inference of api mappings," in *Proceedings of the 2nd Workshop on Programming for Mobile & Touch*, pp. 29–32, ACM, 2014.

[73] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *ACM Sigplan Notices*, vol. 45, pp. 302–321, ACM, 2010.

[74] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pp. 289–298, 2012.

[75] H. D. Phan, A. T. Nguyen, T. D. Nguyen, and T. N. Nguyen, "Statistical migration of api usages," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pp. 47–50, IEEE, 2017.

[76] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 383–392, ACM, 2009.