Hunter: Next-Generation Code Reuse for Java*

Yuepeng Wang
UT Austin, USA
ypwang@cs.utexas.edu

Yu Feng
UT Austin, USA
yufeng@cs.utexas.edu

Ruben Martins
UT Austin, USA
rmartins@cs.utexas.edu

Arati Kaushik
UT Austin, USA
arati@cs.utexas.edu

Isil Dillig
UT Austin, USA
isil@cs.utexas.edu

Steven P. Reiss
Brown University, USA
spr@cs.brown.edu

ABSTRACT

In many common scenarios, programmers need to implement functionality that is already provided by some third party library. This paper presents a tool called HUNTER that facilitates code reuse by finding relevant methods in large code bases and *automatically synthesizing* any necessary wrapper code. Since HUNTER internally uses advanced program synthesis technology, it can automatically reuse existing methods even when code adaptation is necessary. We have implemented HUNTER as an Eclipse plug-in and evaluate it by (a) comparing it against S^6, a state-of-the-art code reuse tool, and (b) performing a user study. Our evaluation shows that HUNTER compares favorably with S^6 and increases programmer productivity.

CCS Concepts

•Software and its engineering → Programming by example; Software libraries and repositories;

Keywords

code reuse, code adaptation, program synthesis

1. INTRODUCTION

In many common scenarios, programmers need to implement functionality that is already provided by some third party library. In such cases, accepted wisdom dictates that it is preferable to reuse existing code rather than reimplementing the desired functionality from scratch. Indeed, there are at least two benefits to software reuse: First, it increases programmer productivity, allowing developers to focus on more creative tasks. Second, implementations provided by existing APIs are typically well-tested; hence, code reuse decreases the likelihood that the implementation is buggy.

*This work is supported by the Air Force Research Laboratory under agreement number FA8750-14-2-0270.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983934

Unfortunately, there is often a wide gap between the adage "software reuse is good" and the reality of software development: In many cases, programmers end up reimplementing functionality that already exists in a library. This duplicated effort is sometimes inadvertent because programmers may not know whether the desired functionality exists or which third-party library provides it. In other cases, the exposed interface may not quite fit the developer's needs, tempting programmers to re-implement the desired functionality.

This paper seeks to address both of these problems by presenting a tool called HUNTER for automatically reusing existing code. To use the HUNTER system, the user provides a *search query*, comprised of the signature of the desired method, together with an English description of its functionality. In addition, the user also provides a few test cases that HUNTER can use to check the correctness of the synthesized code. Given this input, HUNTER finds existing methods that fit the English description and *automatically synthesizes* any necessary wrapper code. A key novelty of HUNTER is that it allows code reuse even when the signature of the desired method differs substantially from that of an existing method in the corpus. In particular, HUNTER uses *type similarity metrics* to "align" different method signatures and employs type-directed program synthesis to generate adapter code.

As shown schematically in Figure 1, the HUNTER tool consists of three components, namely *code search*, *interface alignment*, and *synthesis*. HUNTER's *code search* component is responsible for finding a ranked list of *candidate adaptees* that fit the English description. For each candidate adaptee, HUNTER invokes the *interface alignment* module to infer a suitable mapping between the parameters of the adaptee $\mathcal{E}$ and those in the desired method $\mathcal{R}$ (i.e., adapter). In particular, since the signature of $\mathcal{R}$ may differ from that of the adaptee, HUNTER uses *type similarity metrics* to infer which parameters of $\mathcal{E}$ are likely to correspond to which parameters of $\mathcal{R}$. For instance, the interface alignment module might infer that parameters $p_1 : \tau_1, \ldots, p_n : \tau_n$ of $\mathcal{R}$ are most likely to correspond to parameter $p : \tau$ of $\mathcal{E}$. Given such a solution to the interface alignment problem, HUNTER invokes the *code synthesis* module to generate code that "produces" an object of type τ by consuming objects of type $\tau_1, \ldots, \tau_n$.

For any given code reuse task, HUNTER typically generates many candidate implementations of the desired method and runs it on the user-provided test cases. If any test case fails, HUNTER *backtracks* by either finding a different solution to the interface alignment problem or trying a different candidate adaptee identified by the code search module. When HUNTER terminates, the synthesized code is always guaran-

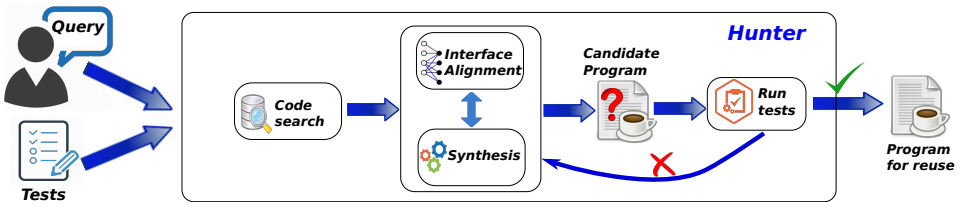

Figure 1: Workflow of the HUNTER tool

```
@Test public void test() {
    Vector<MyPoint> v1 = new Vector<>();
    v1.add(new MyPoint(0, 0)); v1.add(new MyPoint(1, 1));
    v1.add(new MyPoint(2, 1)); v1.add(new MyPoint(3, 2));
    v1.add(new MyPoint(4, 2)); v1.add(new MyPoint(5, 3));
    Vector<MyPoint> res = new Vector<>();
    Bresenham.drawLine(new MyPoint(5, 3), res);
    assertEquals(v1, res); }
```

Figure 2: Test cases for the desired `drawLine` method

teed to compile and pass the user-provided test cases.

We have implemented HUNTER as an Eclipse plug-in and made it publicly available in the Eclipse marketplace [3]. To assess the effectiveness of HUNTER, we compare it against S^6, a state-of-the-art code reuse tool, and show that HUNTER can reuse more code compared to S^6. We have also performed a user study and show that HUNTER allows users to finish programming tasks quicker and with fewer bugs.

2. MOTIVATING EXAMPLE

We now give an example of a programming task in which HUNTER can be useful to programmers. Consider a user, Alice, who needs to implement a procedure to draw a straight line from the origin to a specified point. Specifically, Alice wants to implement the following `drawLine` method:

```
void drawLine(MyPoint pt, Vector<MyPoint> res)
```

Here, `pt` is the point specified by the user, and `res` is a vector of points on the raster that should be selected to approximate a straight line between the origin and `pt`. Alice knows about Bresenham's line drawing algorithm that could be used to implement this functionality, but she does not know exactly how it works. While she would like to reuse an existing implementation of Bresenham's algorithm, she cannot find an implementation that quite fits her needs.

In order to use HUNTER, Alice needs to provide the method signature shown above as well as a brief natural language description, such as *"Bresenham's line drawing"*. Alice also needs to provide a test suite (e.g., the one shown in Figure 2) that HUNTER can use to validate the synthesized code.

HUNTER first queries a code search engine using Alice's description and retrieves the top k results. To simplify the example, suppose that the search engine yields a single function with the following signature:

```
Point[] bresenham(int x0, int y0, int x1, int y1)
```

Specifically, this function returns an array of `Points` to approximate a line starting from $(x0, y0)$ to $(x1, y1)$. Note that there are several differences between Alice's desired `drawLine` interface and the existing `bresenham` function:

- Alice's interface uses `MyPoint` to represent the user-specified point, while the existing function uses two integers (namely, `x1` and `y1`) to represent the end point.
- Alice's interface assumes the origin as a starting point, whereas `bresenham` takes `x0` and `y0` as input.
- The existing `bresenham` function returns an array of `Point`'s, whereas Alice's interface "returns" the line by storing the result in `res`. Furthermore, Alice represents points using a custom type called `MyPoint`, whereas `bresenham` uses a different type called `Point`.

Despite these significant differences, HUNTER is able to automatically generate the following implementation of Alice's `drawLine` interface:

```
void drawLine (MyPoint pt, Vector<MyPoint> res) {
    int v1 = pt.getX();  int v2 = pt.getY();
    Point[] v3 = external.bresenham(0, 0, v1, v2);
    for (Point v4 : v3) {
        int v5 = v4.getX(); int v6 = v4.getY();
        MyPoint v7 = new MyPoint(v5, v6);
        res.add(v7); }
}
```

Observe that the code generated by HUNTER first deconstructs the `MyPoint` object `pt` into a pair of integers by invoking the appropriate getter methods. Also note that HUNTER can supply default values for `x0` and `y0` even though there are no corresponding parameters in the `drawLine` interface. Finally, after invoking the existing `bresenham` method, HUNTER can synthesize code to convert the array of `Points` into a vector of `MyPoints`.

3. SYSTEM OVERVIEW

We now give a high-level overview of HUNTER's code reuse algorithm (shown in Algorithm 1) as well as the various components it uses. For details, we refer the interested reader to the extended version of the paper [26].

Code search. HUNTER can use any code search engine that yields results at the granularity of methods. In our current implementation, we integrated HUNTER with the Pliny code search engine [6] and use a database of over 12 million Java methods collected from open source repositories, such as Github [2] and Bitbucket [1].

Type similarity metrics. After performing code search, HUNTER computes a *type similarity matrix* between the types used in the desired signature and those appearing in the search results. Specifically, the COMPUTETYPEDISTANCE procedure used in line 5 of Algorithm 1 yields a matrix Λ that maps each pair of types (τ, τ') to a distance. The larger the distance, the less similar τ is to τ' and the less likely it is that an argument of type τ will be mapped to an argument of type τ'. To compute type similarity metrics, we represent each Java type τ using a *multiset (bag)* representation, denoted as $\psi(\tau)$. For instance, given a class `Point` with two integer fields, $\psi(Point)$ is the multiset $\{Point, int, int\}$. We

Algorithm 1 Code Reuse Algorithm

```
1: procedure CodeReuse(S, D, T)
2:     Input: signature S, description D, and tests T
3:     Output: adaptee E with adapter R or failure ⊥
4:     [E] := CodeSearch(S, D)
5:     Λ := ComputeTypeDistance(S, [E])
6:     for all E ∈ [E] do
7:         do
8:             M := GetBestAlign(S, E, Λ)
9:             R := AdapterGen(M, E, S)
10:            if RunTests(R, E, T) then return (R, E)
11:        while M ≠ ∅
12:    return ⊥
```

then define the type distance $\delta(\tau, \tau')$ to be:

$$\delta(\tau, \tau') = \frac{|(\psi(\tau) - \psi(\tau')) \cup (\psi(\tau') - \psi(\tau))|}{|\psi(\tau) \cup \psi(\tau')|}$$

Note that $\delta(\tau, \tau')$ is a real number in the range $[0, 1]$ and represents the fraction of elements in $\psi(\tau) \cup \psi(\tau')$ that are not shared between $\psi(\tau)$ and $\psi(\tau')$.

Interface alignment. After performing code search and computing type similarity metrics, Hunter tries to generate an implementation of the desired method using each candidate adaptee $\mathcal{E}$ among the code search results. Towards this goal, Hunter uses the type similarity matrix Λ to find an *optimal alignment* between the desired signature $\mathcal{S}$ and that of candidate adaptee $\mathcal{E}$. Specifically, the GetBestAlign procedure used in line 8 of Algorithm 1 yields a minimum-cost mapping $\mathcal{M}$ where the cost of $\mathcal{M}$ is computed using a heuristic cost function based on the type similarity matrix Λ. In order to maximize opportunities for code reuse, note that we do not require the mapping $\mathcal{M}$ to be one-to-one. Furthermore, to find the *minimum cost mapping*, we solve a minimization problem using *integer linear programming (ILP)*. For example, the optimal alignment between `drawLine` and `bresenham` is given by the following mapping:

```
x1: int -> pt: MyPoint ; y1: int -> pt: MyPoint
v3: Point[] -> res: Vector<MyPoint>
```

Code synthesis. Given an optimal alignment $\mathcal{M}$ between the adapter and the adaptee, Hunter's code reuse algorithm invokes the AdapterGen method (line 9 of Algorithm 1) to automatically generate wrapper code based on this alignment. For instance, for our running example from Section 2, AdapterGen synthesizes code to "convert" the `pt` object of type `MyPoint` to an integer `x1` of type `int`.

In principle, Hunter can be integrated with any type-directed code synthesis engine, such as Prospector [19], InSynth [14], or CodeHint [13]. In particular, suppose that the solution $\mathcal{M}$ returned by GetBestAlign maps parameters $p_1 : \tau_1, \ldots, p_n : \tau_n$ to parameter p of type τ, Hunter uses a code synthesis engine to synthesize a function that takes arguments of type $\tau_1, \ldots, \tau_n$ and returns a value of type τ.

In the current version of Hunter, the AdapterGen procedure is implemented by a new synthesis tool called SyPet, which employs a novel type-directed synthesis algorithm based on Petri net reachability analysis [12].

Running test cases. After synthesizing wrapper code for a given candidate adaptee $\mathcal{E}$ and alignment $\mathcal{M}$, Hunter automatically resolves all dependencies and runs the synthesized program on the provided test cases. If any test fails, Hunter backtracks by adding an additional constraint to the corresponding ILP problem and asking the ILP solver for a different solution. If no such solution exists, it means that Hunter has exhausted all valid alignments for candidate adaptee $\mathcal{E}$. In this case, it backtracks by considering a different candidate adaptee. The CodeReuse algorithm terminates when Hunter finds an implementation that passes all test cases or it runs out of possible adaptees.

4. EVALUATION

To evaluate the usefulness of Hunter, we compare Hunter against S^6 [23]. We also perform a user study and evaluate how long participants take to complete various algorithmic tasks with and without Hunter. All experiments are conducted on a Lenovo laptop with an Intel i7-5600U CPU and 8G of memory running Ubuntu 14.04.

4.1 Comparison with S^6

S^6 [23] is a state-of-the-art code reuse tool with a web interface that takes the same set of inputs as Hunter (i.e., method signature, natural language description, and test cases). We compare Hunter against S^6 on a variety of benchmarks collected from three different sources: (i) examples used to evaluate S^6 [23], (ii) all challenge problems taken from the MUSE project [5] and (iii) linked list and binary tree benchmarks from Leetcode [4], which is an online resource containing programming problems.

Table 1 describes the 40 benchmarks used in our evaluation in more detail. In this table, a ✓ indicates that the tool was able to successfully synthesize the desired code, and an ✗ indicates that synthesis failed. When the synthesis is successful we also report the time taken by the corresponding tool. As summarized in Table 1, Hunter is able to solve *all* benchmarks with an average running time of 14 seconds. In contrast, S^6 can only solve 37.5% of the benchmarks. Since both tools use the same underlying code corpus, these results indicate that Hunter is more successful at reusing existing code compared to S^6.

4.2 User Study

To evaluate the impact of Hunter on programmer productivity, we conducted a user study involving 16 graduate students, 2 professional programmers, and 3 undergraduate students. Specifically, we asked the participants to produce a Java implementation for three different programming tasks, namely matrix multiplication (T1), longest common subsequence (T2), and Bresenham's line drawing algorithm (T3). For each problem, we provided a detailed description of the task to be performed, including a simple input-output example. Since our goal is to compare programmer productivity with and without Hunter, we instructed participants to complete each task using two different methods:

- ***Manual:*** In this scenario, participants were asked to complete the programming task without using Hunter. However, participants were explicitly told that they can use any existing code search tool and adapt the search results to their needs.
- ***Using Hunter:*** In this scenario, participants were asked to use Hunter to automatically synthesize the desired code by providing a natural language query and implementing appropriate JUnit tests.

Table 1: Comparison between HUNTER and S^6

		HUNTER		S^6	
Id	Benchmark	Solved	Time(s)	Solved	Time(s)
1	SimpleTokenizer	✓	19	✓	23
2	QuoteTokenizer	✓	10	✓	19
3	CheckRobots	✓	12	✓	25
4	LogBase	✓	13	✓	224
5	RomanNumeral	✓	14	✓	99
6	RomanToInt	✓	11	✓	120
7	PrimeNumber	✓	7	✓	27
8	PerfectNumber	✓	7	✓	24
9	DayOfWeek	✓	13	✓	58
10	EasterDate	✓	10	✓	180
11	MatrixMultiplication	✓	8	✗	—
12	LcsInteger	✓	7	✗	—
13	RemoveDuplicates	✓	13	✓	112
14	TransposeMatrix	✓	10	✗	—
15	InvertMatrix	✓	12	✗	—
16	MatrixPower	✓	8	✗	—
17	DotProduct	✓	9	✗	—
18	MatrixDeterminant	✓	10	✓	200
19	CountingSort	✓	13	✗	—
20	MatrixAddition	✓	9	✗	—
21	FloodFill	✓	26	✗	—
22	FindMedian	✓	11	✗	—
23	ListAverage	✓	8	✗	—
24	BresenhamLine	✓	62	✗	—
25	BresenhamCircle	✓	11	✗	—
26	Distance	✓	18	✗	—
27	Slope	✓	32	✗	—
28	PrimeSieve	✓	7	✓	27
29	Anagram	✓	10	✓	72
30	Palindrome	✓	8	✓	22
31	PartitionList	✓	16	✗	—
32	RotateList	✓	15	✗	—
33	ListInsertion	✓	12	✗	—
34	PalindromeList	✓	10	✗	—
35	SwapNodesPairs	✓	12	✗	—
36	InvertBinaryTree	✓	14	✗	—
37	MinDepthBinaryTree	✓	13	✗	—
38	BinaryTreePostorder	✓	21	✗	—
39	BinaryTreeInorder	✓	11	✗	—
40	SumRootLeafNumbers	✓	8	✗	—

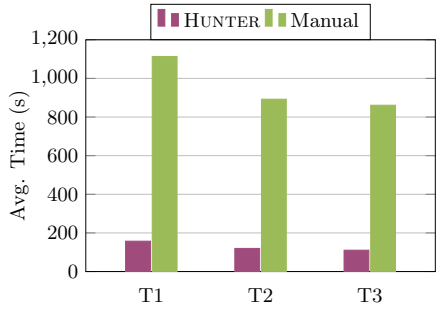

Figure 3: Comparison between HUNTER and manual

For both scenarios, we asked participants to stop working on a given task after 30 minutes. Hence, any task that the users could not complete within 30 minutes was considered as a "failure". For the manual implementation case, we did not require participants to write test cases for their code. Hence, there was no overlap between the tasks that the users needed to complete for the two different usage scenarios.

Programmer productivity. Figure 3 shows the average time for completing each of the three tasks with and without HUNTER. Overall, participants took an average of 130 seconds using HUNTER and an average of 948 seconds when writing the code manually. Furthermore, while participants were able to successfully complete 100% of the tasks using HUNTER, success rate was only 85% without HUNTER.

To validate that these results are statistically significant, we also performed a two-tailed paired t-test [11] for each task, ignoring samples that were not completed within the 30 minute time-limit. Since the t-test for each task returned p-values smaller than 0.0001, this evaluation shows that there is a significant difference in average task completion time with and without HUNTER.

Quality of solutions. To compare the *quality* of the solutions implemented manually vs. using HUNTER, we also manually inspected the solution and ran the programs on a large test suite. While programs synthesized by HUNTER pass all test cases, 19% of the manually written programs are actually buggy and fail at least one of our test cases. Among those buggy programs, ten of them contain off-by-one bugs, one program throws an exception for matrices that are not square, and one program produces the wrong result due to a typo in the copy-pasted code. Hence, these results show that programs synthesized by HUNTER are less error-prone compared to their manually written versions.

5. RELATED WORK

HUNTER is related to a line of work on automated code reuse [23, 17, 16, 25, 29, 28, 7, 21, 15, 20, 27, 10, 22, 9, 18, 8, 24]. Among these tools, S^6 [23] is the most similar to HUNTER. Specifically, S^6 uses a combination of test cases, method signatures, and natural language description to find relevant methods. Furthermore, similar to HUNTER, S^6 can adapt existing code to fit the desired interface by performing various kinds of transformations. However, there are two key differences between S^6 and HUNTER: First, S^6 directly modifies the search result instead of synthesizing wrapper code. Second, HUNTER uses integer linear programming to measure similarity between type signatures and integrates this information with program synthesis tools. As demonstrated in our experiments, HUNTER outperforms S^6, both in terms of running time and in the amount of reusable code.

Another test-driven tool similar to HUNTER is CodeConjurer [16], which allows users to specify class components using test cases and UML-like interface description. Similar to HUNTER, CodeConjurer can find a suitable mapping between the methods of the candidate class and those of the desired class. However, CodeConjurer tries all possible method-mapping permutations and cannot synthesize adapter code. Since the mapping technique is based on brute-force search, CodeConjurer's method matching procedure can take several hours. In contrast, HUNTER solves a different kind of interface alignment problem and finds an optimal solution through integer linear programming.

Similar to HUNTER, the CodeGenie tool [17] also uses test cases to partially specify the behavior of the desired method. Specifically, CodeGenie extracts the method signature and search keywords from JUnit tests and queries Sourcerer [7] to find relevant methods. It then uses the provided test cases to validate the search result and merges the code into development environment. However, unlike HUNTER, CodeGenie cannot synthesize adapter code.

6. REFERENCES

[1] Bitbucket. https://bitbucket.org.

[2] GitHub. https://github.com.

[3] Hunter. http://fredfeng.github.io/Hunter/.

[4] Leetcode Online Judge. https://leetcode.com.

[5] Mining and understanding software enclaves. http://www.darpa.mil/program/mining-and-understanding-software-enclaves.

[6] Pliny Project. http://pliny.rice.edu.

[7] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.*, 79:241–259, 2014.

[8] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proc. International Symposium on Software Testing and Analysis (ISSTA), Baltimore, USA*, pages 257–269, July 2015.

[9] T. T. Bartolomei, K. Czarnecki, and R. Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *Proc. IEEE International Conference on Software Maintenance (ICSM), Timisoara, Romania*, pages 1–10, September 2010.

[10] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for java using free-form queries. In *Proc. Fundamental Approaches to Software Engineering (FASE), York, UK*, pages 385–400, March 2009.

[11] W. J. Dixon and F. J. Massey Jr. *Introduction to statistical analysis*. McGraw-Hill, 1957.

[12] Y. Feng, Y. Wang, R. Martins, A. A. Kaushik, and I. Dillig. Type-directed Component-based Synthesis using Petri Nets. Technical Report TR-16-01, Department of Computer Science, UT-Austin, 2016.

[13] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. Codehint: dynamic and interactive synthesis of code snippets. In *Proc. International Conference on Software Engineering (ICSE), Hyderabad, India*, pages 653–663, May 2014.

[14] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, USA*, pages 27–38, June 2013.

[15] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. International Conference on Software Engineering (ICSE), St. Louis, USA*, pages 117–125, May 2005.

[16] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[17] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, P. C. Masiero, and C. V. Lopes. Applying test-driven code search to the reuse of auxiliary functionality. In *Proc. ACM Symposium on Applied Computing (SAC), Honolulu, USA*, pages 476–482, March 2009.

[18] J. Li, C. Wang, Y. Xiong, and Z. Hu. SWIN: towards type-safe java program adaptation between apis. In *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM), Mumbai, India*, pages 91–102, January 2015.

[19] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, USA*, pages 48–61, June 2005.

[20] L. Martie, T. D. LaToza, and A. van der Hoek. Codeexchange: Supporting reformulation of internet-scale code queries in context (T). In *Proc. IEEE/ACM International Conference on Automated Software Engineering, (ASE), Lincoln, USA*, pages 24–35, November 2015.

[21] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proc. International Conference on Software Engineering (ICSE), Waikiki, Honolulu, USA*, pages 111–120, May 2011.

[22] M. Nita and D. Notkin. Using twinning to adapt programs to alternative apis. In *Proc. ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa*, pages 205–214, May 2010.

[23] S. P. Reiss. Semantics-based code search. In *Proc. International Conference on Software Engineering (ICSE), Vancouver, Canada*, pages 243–253, May 2009.

[24] K. T. Stolee, S. G. Elbaum, and D. Dobos. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26, 2014.

[25] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, USA*, pages 204–213, November 2007.

[26] Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, and S. P. Reiss. Type-directed code reuse using integer linear programming. http://arxiv.org/abs/1608.07745, 2016.

[27] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proc. International Conference on Software Engineering (ICSE), Orlando, USA*, pages 513–523, May 2002.

[28] A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, 4(2):146–170, 1995.

[29] A. M. Zaremski and J. M. Wing. Specification matching of software components. In *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington DC, USA*, pages 6–17, October 1995.