SNIPR: Complementing Code Search with Code Retargeting Capabilities

Huascar A. Sanchez
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95060
hsanchez@cs.ucsc.edu

***Abstract*—This paper sketches a research path that seeks to examine the search for suitable code problem, based on the observation that when code retargeting is included within a code search activity, developers can justify the suitability of these results upfront and thus reduce their searching efforts looking for suitable code. To support this observation, this paper introduces the Snippet Retargeting Approach, or simply SNIPR. SNIPR complements code search with code retargeting capabilities. These capabilities' intent is to help expedite the process of determining if a found example is a best fit. They do that by allowing developers to explore code modification ideas in place, without requiring to leave the search interface. With SNIPR, developers engage in a virtuous loop where they find code, retarget code, and select only code choices they can justify as suitable. This assures immediate feedback on retargeted examples and thus saves valuable time searching for appropriate code.**

I. Research Problem and Solution Outline

The recent rise of Internet-scale code search engines and Q&A sites—e.g., Portfolio [1], Sourcerer [2], Ohloh code [1], StackOverflow [2]—has catapulted search-driven development from backwater to ubiquity, and given rise to an active research community focused on this phenomenon [3]–[5]. Clearly, these engines and sites have changed how developers coordinate their work, and where they find information. This new condition has enabled developers to build applications opportunistically by iteratively finding, and reusing online source code [6]–[9]. This opportunistic style of development is not easy because searched sources are large, in most cases unsuitable, and quite often unrelated [10]. Consequently, if search-driven development were to be established as best practice, then the time involved in deciding a best search result to reuse must be minimized.

Obviously, code search all by itself won't solve the whole problem. In fact, code-only searching misses out on certain human abilities that are important in search-driven development, such as the ability to quickly identify—based on knowledge and experience—a better result among many, and to change the any found code to better reflect what existed only in the human mind. Previously published work has started tackling these issues from different directions[3]; each with unique strengths and weaknesses. These directions have one thing in common, which is that developers have to try the code examples first, before they know the examples' suitability [21]. In fact, developers are given no guidance as to which result may be a best fit for their code in progress, beyond relative ranking values. This limitation is one of the reasons why search-driven development is so cumbersome, and ultimately a time drain.

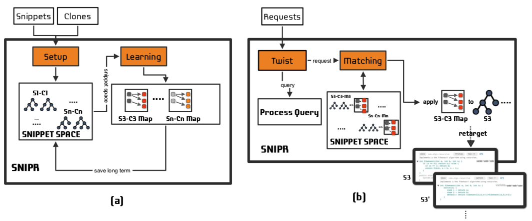


Fig. 1. SNIPR's conceptual architecture.

To alleviate the problem imposed by the above limitation, I propose the *Snippet Retargeting Approach* or simply SNIPR. SNIPR complements code search with code retargeting capabilities. These capabilities are mapping-based transformations that convert the structure of one snippet into the structure of another—i.e., code retargeting. They are learned from code examples and encapsulated in a representation that could be accessed algorithmically—i.e., a code mapping [22]. These capabilities' intent is to narrow the search for suitable code. This is done by allowing developers to explore code modification ideas in place, without requiring to leave the search interface. With SNIPR, developers could dynamically evaluate code retargeting ideas as they search, selecting only matches they can justify as suitable: the code is retargetable and its cognitive distance [23] is minimal. Figure 1 illustrates the SNIPR conceptual architecture.

The SNIPR approach will be examined by addressing the following research questions:

RQ1 What kind of SNIPR operations could be invoked directly from the search box?

Advisor: Prof. Jim Whitehead, University of California, Santa Cruz

[1]http://code.ohloh.com

[2]http://www.stackoverflow.com

[3]Two of these directions are the most popular: enhancing search technology [9], [11]–[14], and coupling Web search and crowdsourced input with IDEs [15]–[20]

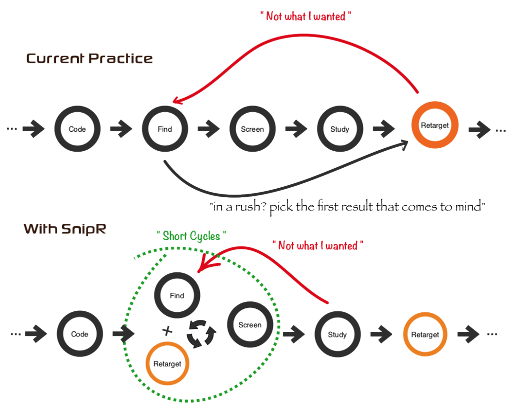
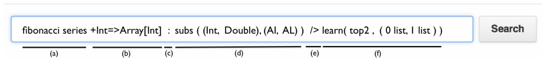


Fig. 2. Search Process before and with SNIPR. A pre–SNIPR scenario requires developers to find and select the most promising examples (i.e., screening), and then to peruse them before eventually consuming them in the IDE. If the resulting code is not the desired code, then the search process is restarted. With SNIPR, developers engage in a virtuous loop where they find and select only choices they can justify as suitable.



```
fibonacci series +Int=>Array[Int]  :  subs ( (Int,  Double), (AI, AL) )  /> learn( top2 ,  ( 0 list, 1 list ) )
```

Fig. 3. An expression in Twist. A code search can be textual (Part (a)) or by type signature (Part (b)) or both. Anything after : (Part (c)) represents a command. The command in Part (d) replaces the type Int with type Double, and replaces type Array[Int] with type ArrayList[Int]. The *retargetable* outcome of Part (d) is turned (Part (e)) as input of the command in Part (f). Part (f) gives a name to the top-2 results in the outcome of Part (d).

RQ2 How does SNIPR learn to translate structure between related snippets?

RQ3 How computationally expensive is it to retarget (part of) examples each time?

RQ4 How does the time needed to perform a more complete code search task of the SnipR approach compare to the current approaches of code search?

For RQ1, Twist, a command line language for interacting with search results and applying retargeting operations to those results is proposed. Twist allows requests for results and retargeting commands to be intermixed or used separately. Therefore, Twist must support textual queries and a set of code retargeting operators well-suited for manipulating code examples. Figure 3 gives an example of Twist's expression. For RQ2, a set of algorithms that allow SNIPR to learn mapping-based code transformations—hereafter called code mappings—from example code is proposed. These algorithms are: Setup, Learning, and Mapping (See Figure 1).

RQ3 is about the performance (in terms of computational throughput) of the code retargeting algorithms. In other words, this is about how long it will take to find the right code mapping and apply it to a particular code snippet. Code retargeting is an operation that can operate on a single result or an entire result set. Therefore, this operation requires that those cases where code mappings can be applied are carefully identified, to prevent any unnecessary work. For RQ4, an evaluation of the efficiency of the SNIPR approach vs. existing approaches for code search is performed.

These solutions are the expected contributions in search-driven development. For the rest of the paper, the focus will shift to the proposed solution to RQ1 (Section II-A), and to a brief description of RQ2, RQ3, and RQ4 in Section II-B. Section III briefly describes a plan for evaluating this work. Section IV outlines the progress to date and Section V concludes this paper.

II. Proposed Solutions

A. Designing SNIPR operations that could be invoked directly from the search box (RQ1)

The key idea is to design a set of retargeting operations that could be invoked from the search box, without losing any sight of the current search goal. This design should balance two inter-related principles: ease of use and flexibility. To understand what's needed to produce this type of design, I'll turn to the seminal work on sloppy command lines for the Web [24], [25] and on platform-specific linguistic command lines [26]. Similar to both types of work, SNIPR's Twist will have flexible syntax requirements. This flexibility allows requests for results and retargeting operations to be intermixed—or used separately. The focus of these operations will be on performing code modifications to modules (methods that properly implement an API). Consequently, for ease of use, SNIPR will provide most of the mechanisms for expression and control of changes that can be made to modules. For flexibility, a simple language for combining, and executing code retargeting commands will be developed. Twist's syntax is influenced by the Io programming language[4].

Previous work in search interfaces for programming exist [2], [11], [16], [27]–[29]. Undeniably, these types of search interfaces have improved how developers locate code snippets. Large sets of code examples are just one search away. Clearly, this easy access to such an amazing treasure trove code has a value; however, it also has a significant limitation: they are less effective in dynamic and exploratory scenarios; i.e., when developers are working in an unfamiliar domain. In such a scenario the type and number of examples may quickly change as further searchers are performed. Therefore, locating snippets can easily become overwhelming for any developer. SNIPR differs from these tools by providing a platform that permits rapid and dynamic exploration of search results.

B. Performing Code Retargeting (RQ2, RQ3, and RQ4)

Code retargeting is an operation that can work on a single search result or an entire result set. Therefore, this operation requires that those cases where code mappings can be learned and/or applied are carefully identified. This will prevent any unnecessary work from happening as matched code examples are being returned by the query engine. This requirement

[4]http://iolanguage.org/

will be addressed by developing a set of reliable and cache conscious SNIPR algorithms (RQ2). They should be reliable by consistently learning code mappings from example code. They should be cache conscious by exploiting recently read example code if this code must be read again in the future. These algorithms represent the modules required to implement SNIPR functionality. They are the Setup, Learning, Matching steps (See Figure 1).

The Setup and Learning steps occur at indexing time. Given a set of indexed snippets, the Setup step populates the Snippet Space (cached snippets) by invoking a code clone searching tool. Then it considers all the possible combinations of (original-clone) pairs and saves them into the Snippet Space. Code clones are needed to realize how SNIPRs learning algorithm operates: they are related implementations and thus similar. They are also syntactically different and are indeed interchangeable. This knowledge is used by the learning algorithm to hypothesize and map together related code elements between snippets. The output of the Setup step is fed into the Learning step. The Learning step then builds mappings for all the (original-clone) pairs in the Snippet Space. To do that, it induces a translation table over the two snippets' similar structures. This table specifies how two code snippets correspond. Once this table is created, the algorithm uses this table to determine structure translation candidates and then uses structured prediction [30] to find an optimal code mapping between the two snippets.

Given a request (See Figure 3), Twist interprets this input by searching for a mapping that will make sense of this input. At this point, the Matching step maps the retargeting request to its corresponding optimal mapping and then applies this optimal mapping to the original snippet. This should happen with minimal or no cost to the developer. Therefore, these operators should be reliable in the sense of efficiently applying learned mappings (RQ3).

In a dynamic and exploratory scenario, developers using SNIPR will engage in a virtuous loop where they find and select only suitable choices (See Figure 2). This results in less time spent on unsuitable, difficult-to-retarget results. This minimizes the number of iterations and thus time in the search process since the developer is only dealing with suitable choices (RQ4). This will be possible only if SNIPR's retargeting operations are efficient, which will be demonstrated by answering RQ3.

Previous work in adapting code to alternate contexts and/or to new APIs exist. Such work has helped developers with many development tasks, such as adapting example code to new contexts [20] or to new APIs [22]. This also includes resolving many simple coding errors quickly[5], or suggesting ways for correcting compiler and runtime errors [17]. SNIPR differs from these tools in focus and approach. SNIPR focuses on helping developers justify the suitability of code examples during their search-driven development activities. The other solutions focus on either specifying and applying a class of program transformations, or creating code integration templates which will assist developers in integrating a snippet into their projects.

[5]Quick Fix Scout: https://code.google.com/p/quick-fix-scout/, EUKLAS: http://www.cs.cmu.edu/~euklas

III. Evaluation

The evaluation methodology to be followed is to validate the SNIPR approach and results through user and lab studies. The tests will be run on SNIPR prototypes in a working code search system. The user studies will involve a list of subjects, solicited from a public mailing list at a college campus. The subjects will be experienced Web users, have some programming experience, and could type reasonably well. The use and test of the SNIPR prototype will be done on the basis of a programming problem to be solved; aiming to answer the research questions listed in Section I. For instance,

- A user study will be performed to test for the flexibility and ease of use of Twist. The test attempts to determine how intuitive this language is for end-users and how easy is for the end-users to translate their ideas into expressions in Twist. The test is also used to determine how this language might be used in daily development activities and to evaluate some of the decisions made on the design of the language's "intuitively readable commands." Each subject will be instructed on using Twist and the instructed to use only the search box to do any of the assigned tasks. Subjects will be asked to solve a programming problem. Once they have completed the assignment, they will be asked to complete survey about their experience with Twist. The data gathered from this test and survey will be used to answer RQ1.
- Besides creating and executing a set of microbenchmarks, the same user study will be used to provide a feel for the speed and accuracy of the SNIPR algorithms. Inputs from the user study will be used to derive an average processing time—in terms of applying code mappings from/to code examples. It is expected that the processing time varies for input sizes of different lengths. From this, one could guess that the average-case running time is polynomial, but that is reasonable for relatively small source code (e.g., less than 40 lines of code [31]), such is the case of example code (RQ3).
- To evaluate the effectiveness of the learning algorithm, a hold-out test will be run. A set of mappings extracted from a collected corpus will be used as training data, and another different set of mappings—randomly chosen—as a test data. Then a machine learning algorithm will be run for a given number of iterations. The output of this will be a set of mapping-based program transformations. Then, we will use different metrics—e.g., average similarity—to compare the learned and reference mappings.
- Another user study will be conducted comparing SNIPR to a general Web code search engine (e.g., Ohloh code) and an example-centric programming tool (e.g., SnipMatch or Blueprint). This study attempts to examine how quickly developers perform a more complete code search

task (with or without SNIPR). Subjects will be asked to solve a programming problem. Subjects are instructed not to write any code from scratch, but instead to use these tools to find code examples. The order in which search interfaces are presented is controlled by a Latin Square design. The data gathered from this study will be used to answer RQ4.

IV. Progress to Date

This work is still at early stages. Work in progress includes my advancement to candidacy by the end of next month, and an early design of the command line language for retargeting source code. The development of code retargeting algorithms, the realization of the SNIPR conceptual architecture, and the implementation of SNIPR's Twist language are work that remains.

V. Conclusion

This paper has introduced SNIPR, an approach that complements code search with code retargeting capabilities. Unlike prior work, the SNIPR enables developers to engage in a virtuous loop where they find and select only the code examples they can justify as suitable. This will minimize the number of iterations in the search process that developer has to go through, since the evaluation step is only dealing with suitable choices. It is envisioned that SNIPR will not be seen as a competitor to any code search systems, but more like a platform for searching for suitable code.

References

[1] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Portfolio: finding relevant functions and their usage," in *ICSE '11 Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Baldi, and C. V. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006.

[3] S. Bajracharya, A. Kuhn, and Y. Ye, "SUITE 2009: First International Workshop on Search-Driven Development – Users, Infrastructure, Tools and Evaluation," in *2009 31st International Conference on Software Engineering - Companion Volume*. IEEE, May 2009, pp. 445–446.

[4] ——, "SUITE 2010: 2nd International Workshop on Search-Driven Development - Users, Infrastructure, Tools & Evaluation," in *ICSE '10*. New York, New York, USA: ACM Press, 2010, pp. 427–428.

[5] ——, "Third International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE 2011)," in *ICSE '11*. New York, New York, USA: ACM Press, 2011, pp. 1228–1229.

[6] J. Brandt, P. J. Guo, and J. Lewenstein, "Opportunistic programming: how rapid ideation and prototyping occur in practice," in *WEUSE '08 Proceedings of the 4th international workshop on End-user software engineering*, 2008.

[7] C. Ncube, P. Oberndorf, and A. W. Kark, "Opportunistic Software Systems Development: Making Systems from What's Available," *IEEE Softw.*, vol. 25, no. 6, pp. 38–41, 2008.

[8] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code," in *CHI '09*. New York, New York, USA: ACM Press, 2009, p. 1589.

[9] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending Source Code for Use in Rapid Software Prototypes," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 848–858.

[10] R. E. Gallardo-Valencia and S. Elliott Sim, "Internet-Scale Code Search," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*. IEEE, May 2009, pp. 49–52.

[11] S. Bajracharya, J. Ossher, and C. V. Lopes, "Searching API usage examples in code repositories with sourcerer API search," *SUITE '10 Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, 2010.

[12] F. S. Gysin and A. Kuhn, "A trustability metric for code search based on developer karma," in *SUITE '10*. New York, New York, USA: ACM Press, 2010, pp. 41–44.

[13] C. McMillan, "Finding Relevant Functions in Millions of Lines of Code," in *ICSE '11*. New York, New York, USA: ACM Press, 2011, pp. 1170–1172.

[14] A. T. T. Ying, "Facilitating code example search on the web through expertise personalization," in *UMAP'12: Proceedings of the 20th international conference on User Modeling, Adaptation, and Personalization*. Springer-Verlag, Jul. 2012.

[15] A. Bacchelli, L. Ponzanelli, and M. Lanza, "Harnessing Stack Overflow for the IDE," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. IEEE, 2012, pp. 26–30.

[16] J. Brandt, M. Dontcheva, and M. Weskamp, "Example-centric programming: integrating web search into the development environment," in *CHI '10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.

[17] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What Would Other Programmers Do? Suggesting Solutions to Error Messages," in *CHI '10*. New York, New York, USA: ACM Press, 2010, p. 1019.

[18] R. Hoffmann, J. Fogarty, and D. S. Weld, "Assieme: finding and leveraging implicit references in a web search interface for programmers," in *UIST '07 Proceedings of the 20th annual ACM symposium on User interface software and technology*, 2007.

[19] S. Oney and J. Brandt, "Codelets: linking interactive documentation and example code in the editor," in *CHI '12: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, May 2012.

[20] D. Wightman, Z. Ye, J. Brandt, and R. Vertegaal, "SnipMatch: Using Source Code Context to Enhance Snippet Retrieval and Parameterization," in *UIST '12*. New York, New York, USA: ACM Press, 2012, p. 219.

[21] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, pp. 4:1–4:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2063239.2063243

[22] M. Nita and D. Notkin, "Using Twinning to Adapt Programs to Alternative APIs," in *ICSE '10*. New York, New York, USA: ACM Press, 2010, pp. 205–214.

[23] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, 1992.

[24] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: Capture, Share, Automate, Personalize Business Processes on the Web." in *CHI '07*. New York, New York, USA: ACM Press, 2007, p. 943.

[25] R. C. Miller, V. H. Chou, M. Bernstein, G. Little, M. Van Kleek, D. Karger, and m. schraefel, "Inky: a sloppy command line for the web with rich visual feedback," in *UIST '08*. New York, New York, USA: ACM Press, 2008, p. 131.

[26] A. Raskin, "The linguistic command line," *interactions*, 2008.

[27] N. Sahavechaphan and K. Claypool, "XSnippet: mining For sample code," *ACM SIGPLAN Notices*, 2006.

[28] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling Reusable Software out of Thin Air," *IEEE Softw.*, vol. 25, no. 5, pp. 45–52, 2008.

[29] S. P. Reiss, "Semantics-Based Code Search," in *2009 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sep. 2009, pp. 385–386.

[30] M. Collins, "Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms," … *conference on Empirical methods in natural language* …, 2002.

[31] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Opportunistic Programming: Writing code to prototype ideate and discover - Google Search," *IEEE Softw.*, vol. 26, no. 5, pp. 18–24, 2009.