Inferring Behavioral Specifications from Large-scale Repositories by Leveraging Collective Intelligence

Hridesh Rajan[1], Tien N. Nguyen[2], Gary T. Leavens[3], Robert Dyer[4]
[1,2]Iowa State University, [3]University of Central Florida, [4]Bowling Green State University
[1,2]Ames,Iowa,[3]Orlando, Florida [4]Bowling Green, Ohio
Email: [1]hridesh@iastate.edu,[2]tien@iastate.edu,[3]leavens@eecs.ucf.edu, [4]rdyer@bgsu.edu

***Abstract*—Despite[1] their proven benefits, useful, comprehensible, and efficiently checkable specifications are not widely available. This is primarily because writing useful, non-trivial specifications from scratch is too hard, time consuming, and requires expertise that is not broadly available. Furthermore, the lack of specifications for widely-used libraries and frameworks, caused by the high cost of writing specifications, tends to have a snowball effect. Core libraries lack specifications, which makes specifying applications that use them expensive. To contain the skyrocketing development and maintenance costs of high assurance systems, this self-perpetuating cycle must be broken. The labor cost of specifying programs can be significantly decreased via advances in specification inference and synthesis, and this has been attempted several times, but with limited success. We believe that practical specification inference and synthesis is an idea whose time has come. Fundamental breakthroughs in this area can be achieved by leveraging the collective intelligence available in software artifacts from millions of open source projects. Fine-grained access to such data sets has been unprecedented, but is now easily available. We identify research directions and report our preliminary results on advances in specification inference that can be had by using such data sets to infer specifications.**

I. Overview

Specifications tell humans and tools about intended behavior of software and make verification possible. Useful, comprehensible, and efficiently checkable formal specifications can help contain the cost of developing high assurance, reliable, and secure software systems [1]. Despite these benefits, formal specifications are not widely available.

A. The Problem

Currently the effort needed for humans to formally specify code is similar to that required to write the code itself [2]. Although the process of writing specifications can help clarify a project's requirements, which can help reduce costs [1], the extra effort stops many developers.

There have been several practical successes in the use of formal methods to ensure safety and security. A famous safety example is the Paris Metro system, in which the safety of automated trains was verified [3]. Another example is the London air traffic control center's central command function, which, while increasing throughput, was also proved safe [4]. In the area of security, the verified operating system kernel seL4 allows users to avoid security problems (including code injection attacks and buffer overflows) and makes it clear how to use the kernel properly [5]. In each case crucial tool support was made possible by formal specifications.

Our focus is on *behavioral interface specifications*, which are specifications of the behavior[2] of methods (using pre- and postconditions) and modules (e.g., using invariants in classes and interfaces). Methods and modules together constitute application programming interfaces (APIs).

Since existing APIs are very frequently used when writing new software, projects that need to verify or analyze code are hampered by the lack of formal specifications for APIs that their code uses, e.g. only 7% of JDK APIs are currently specified. If a project needs to write its own formal specifications for the JDK or other common APIs, then they will face a large task that will overwhelm their resolve and budget.

B. Solution

We are attacking this longstanding challenge in software engineering of high assurance systems. Building on our prior success in establishing Boa [6]–[10], the first end-to-end infrastructure for mining code and its evolution in open source repositories at a large-scale, we are making advances in specification inference and synthesis, and utilizing them to specify the JDK and other common APIs that will drastically reduce the cost of specifying programs. Such fundamental breakthroughs can be achieved by leveraging the collective intelligence available in software artifacts from millions of open source projects. Henceforth, by *big code* we mean a collection of over a million open source projects with their software evolution history.[3] In this work, we use the term *widely-used code* to refer to types and methods that are used hundreds of thousands of times.

In our view, the overall problem of inferring specifications needs a three-pronged approach:

1) For widely-used code, create specifications by generalizing from uses of the code (exploiting programmers' knowledge). We will use these inferred specifications to

[1]The text of this paper is taken from a recent NSF proposal submitted by the authors.

[2]For us, the term *behavior* is primarily the relation between the two states of the program (e.g., the states before and after a method call). However, we also intend "behavior" to encompass performance, such as constraints on time and other resources, and properties that constrain sequences of states.

[3]*Software evolution history* consists of changes stored in the version control system of each project, extracted from open source repositories such as the Apache foundation, SourceForge, and GitHub.

build the largest, openly-available corpus of formally-specified code that has ever existed.
2) For rarely-used code that is similar to already-specified code, leverage (code, specification) pairs available in the formally-specified corpus, and create specifications by developing and exploiting code similarity and extrapolation techniques.
3) For rarely-used code that is not similar to already-specified code, develop techniques that use the code's revision history to decompose it into fragments, infer the specifications of those fragments, and compose an overall specification from those specifications.

1) Consensus-based Inference: The first part of our approach is essential for bootstrapping the other parts. Fortunately, our preliminary results [11] suggest that this hurdle can be crossed by exploiting our insights that: (a) there are many example clients for widely-used code in big code, (b) most, but not all, such clients work, and (c) broken clients tend to get fixed eventually. Using these insights we will infer specifications for widely-used code. We call our technique that uses these insights "consensus-based inference". For example, developers commonly check preconditions of methods before calling them. This style of programming makes the software more resilient to the unexpected inputs, thus, avoids unexpected program behaviors and bugs. Using consensus-based inference we have begun to specify a large portion of widely-used code (referred to as Boa's knowledge-base), which (when complete) in itself will be a major contribution.

2) Similarity and Differential-based Inference: The second part of our approach is driven by our insight that: (a) programmers often reuse and clone good patterns of program design, (b) programmers of widely-used code also clone and use the same good patterns of program design, and (c) similar code should have similar specifications. Our ongoing work is developing two techniques that uses these insights: "similarity-based inference" and "differential-based inference". We expect that code that is rarely-used but similar to widely-used code will be the second most common class of code.

The general goal of this effort is to develop similarity-based techniques to infer specifications for rarely-used source code. Our key insight is that similar code would likely have similar or the same specification.

First, using the specification mining techniques developed in the previous steps, the idea is to perform mining for all projects in the big code. The result is the collection of (code, specification) pairs. Initially, the result only contains code and specifications for widely-used code. Then these pairs are fed into a clustering engine based on a chosen abstract representation. We then produce clusters of similar (code, specification) pairs. In particular, we will use the similarity measures for finding similar code, code with similar usages, and code with same/similar specifications. This process will give us "similar" code with the same/similar specifications from widely-used code. Such a result will be verified using the techniques developed in the early phase. The verified specifications and code will also be used to enhance (code, specification) pairs contained in Boa's knowledge-base. For the given new code, we will compare it against our corpus of (code, specification) pairs and use those similarity measures to derive its specification.

3) Decomposition-based Inference: The third part of our approach is driven by our insight that function signatures in widely-used code, if treated as an alphabet, can help quickly narrow down (code,specification) pairs that may be composed to synthesize target code. Our technique that uses these insights is called "decomposition-based inference".

C. Context for Technical Development

All of the technical development in this project is being carried out in the context of the Boa project (boa.cs.iastate.edu) [6]–[10]. Since 2012, the Boa project has developed a domain-specific language, also called Boa, an ultra-large corpus of open source projects, and an infrastructure for mining and understanding software repositories [7]. The Boa language abstracts away details and simplifies mining tasks. The Boa corpus contains about 700K open source projects with their full evolutionary history, and will have a million+ open source projects by the end of 2015. Boa has support for mining source code [12] and this support has been utilized to conduct large-scale studies, e.g. on the evolution and usage of Java language features over the last decade [9]. Our work on Boa has positioned us well for this project, but new components are also being added to the infrastructure as needed.

II. Potential Impacts of Proposed Research

By helping solve the problem of specification inference we envision creating a world in which a programmer selects a piece of code, pushes a button, and the integrated development environment (IDE) connected to a specification inference infrastructure synthesizes a specification for that code (see Figure 1).

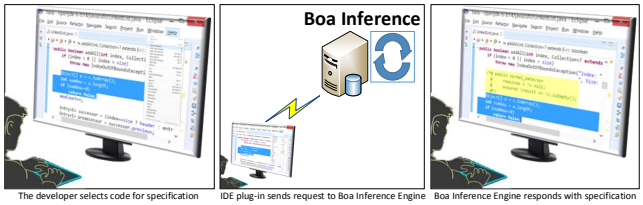

Fig. 1. Overview of Envisioned Capabilities from a Developer's View

In this world, the specification inference infrastructure would synthesize new specifications for libraries and frameworks while progressively improving existing specifications. As a result, libraries and frameworks come pre-specified. In this world, formal methods are widely applied because they can rely upon broadly-available specifications. Most importantly, in this world, the costs of producing and maintaining high assurance software systems that the nation can rely upon are more predictable.

Effective specification inference capabilities will have several substantial benefits. First, having formal specifications for

widely-used APIs will allow critical code that uses these APIs to be formally analyzed (or verified) to establish important properties in a modular fashion. Modularity will be enabled as the specification of an API's method can be used whenever that method is called. Modularity is the key to scalability of such an effort and thus permits the analysis (or verification) of very large code bases. Second, newly inferred specifications will enable a new kind of software infrastructure that can be used to maintain, optimize, and synthesize programs. This will greatly increase programmer productivity. In particular:

- Maintenance of code will be easier, because engineers will not need to spend time reverse engineering code. The use of specifications will reduce the amount of code that an engineer needs to look at to understand any given piece of code, since specifications can be consulted instead of trying to understand the code of APIs that are used. This will result in a lower total lifecycle cost.
- Optimization of code will be greatly facilitated, because programmers will know what behavior they must implement when changing data structures and algorithms.
- The introduction of new bugs during maintenance and optimization can be prevented by checking that the refactored code meets the original specification.
- Retrieval of code using specifications as queries will promote software reuse [13].
- Synthesis of code will become practical by combining modules and calls to different methods to achieve a given specification. This will lead to enormous productivity gains.

More broadly, because specifications will enable modular analysis and verification, modular and (thus) scalable tools that ensure safety and security of critical systems code will become practical.

III. Related Work

In general, other efforts to automatically create specifications have not used the large body of code and associated information in open source repositories that our project will exploit [11]. Existing approaches can be broadly classified into: *program analysis-based* and *data mining-based* approaches. We discuss these approaches, which compliment each other, in turn below.

Program Analysis Approaches. Several authors have proposed approaches based on dynamic analysis [14]–[18]. These dynamic approaches all run a large number of test cases on a program and then mine the resulting execution traces for data and temporal invariants (which can include pre- and postconditions). However, their results may be incomplete, since they depend on the existence of a test suite that adequately explores the program's state space.

On the other hand, static analysis approaches [19]–[25] do not require dynamic instrumentation but often will create behavioral specifications that are only true for that particular program (and are thus not general enough to permit intended evolution or maintenance). Such approaches are most successful with small programs as Wei *et al.* [25] show, however, their focus has been on inferring program-specific invariants till date.

All the program analysis techniques mentioned above have only been used on *individual* programs or projects, and thus they only deal with a small number of call sites for a small number of APIs. These techniques are not designed to combine analysis results from many projects.

Software Repository Mining Approaches. In contrast to program analysis-based approaches, other techniques in the mining software repositories (MSR) area have applied data mining techniques to derive API specifications from existing code repositories [27]–[34]. The key difference between MSR approaches and program-analysis based approaches is that MSR approaches consider *the usage of the APIs* at the call sites in the client programs of the APIs to derive the conditions *regarding only the usage and temporal order* among the API calls.

While some approaches detect such orders as pairs of method calls [29], [32], [35] (e.g., `p` must be called before `q`), other approaches mine the sequences of calls [30], [36] or even a graph or finite state diagram of method calls [31], [34], [37], [38]. Other MSR approaches focus on associations of API entities [27], [28]. However, those MSR approaches do not aim to recover behavioral interface specifications. Moreover, except for a few methods [24], they mainly rely on mining techniques without in-depth program analysis of data and control properties in the mined code.

There is also some work going on software repository mining under the DARPA MUSE program, but those projects are focussing on program synthesis (whereas we focus on specification inference), and do not consider evolution history. *As of this writing, ours is the first result in this area [11].*

IV. Conclusion

We have set out to address the challenge that useful, comprehensible, and efficiently checkable specifications are not widely available. To that end, we are creating novel techniques for inference and synthesis of behavioral specifications. Our work deviates from commonly held belief in the area of behavioral specification inference by making use of an ultra-large corpus of software repositories as a central piece of our solution. We are first targeting widely-used code, i.e. libraries and frameworks, both to bootstrap our inference engine as well as to leverage the collective intelligence available in millions of SLOC available in open source repositories. From the usage of the widely-used code, i.e. by program analysis of millions of call sites, we are deriving the behavioral specifications of widely-used code. Our ongoing work plan is to exploit these inferred specifications for widely-used code as a basis to apply similarity, differential and decomposition-based techniques.

Acknowledgment

Authors work on this idea was supported in part by the NSF under grants CCF-08-46059, CCF-11-17937, CCF-13-49153, CCF-14-23370, CCF-0916715, CCF-1017262, CNS-1228695, CCF-10-18600, CNS-12-23828, CCF-1320578, CCF- 1349153, and CCF-1320578.

REFERENCES

[1] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, no. 5, pp. 11–19, Sep. 1990.

[2] G. T. Leavens and C. Clifton, "Lessons from the JML project," in *Verified Software: Theories, Tools, Experiments, Zurich, Switzerland*, ser. Lecture Notes in Computer Science, B. Meyer and J. Woodcock, Eds., vol. 4171. Springer-Verlag, 2008, pp. 134–143. [Online]. Available: \url{http://dx.doi.org/10.1007/978-3-540-69149-5_15}

[3] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier, "Météor: A successful application of B in a large project," in *FM'99: Formal Methods*, ser. Lecture Notes in Computer Science, J. M. Wing, J. Woodcock, and J. Davies, Eds. Springer Berlin Heidelberg, 1999, vol. 1708, pp. 369–387. [Online]. Available: http://dx.doi.org/10.1007/3-540-48119-2_22

[4] A. Hall and D. Isaac, "Formal methods in a real air traffic control project," in *Software in Air Traffic Control Systems - The Future, IEE Colloquium on*, Jun. 1992, pp. 7/1–7/4.

[5] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1743546.1743574

[6] R. Dyer, "Task fusion: Improving utilization of multi-user clusters," in *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, ser. SPLASH SRC, 2013, pp. 117–118.

[7] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 35th International Conference on Software Engineering*, ser. ICSE'13, 2013, pp. 422–431.

[8] R. Dyer, "Bringing ultra-large-scale software repository mining to the masses with boa," Ph.D. dissertation, Iowa State University, 2013.

[9] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of Java language features," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE'14, 2014.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, *The Art and Science of Analyzing Software Data*. Morgan-Kaufmann, 2015, ch. Boa: an Enabling Language and Infrastructure for Ultra-large Scale MSR Studies.

[11] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *22nd International Symposium on Foundations of Software Engineering*, ser. FSE'14, November 2014, pp. 166–177.

[12] R. Dyer, H. Rajan, and T. N. Nguyen, "Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes," in *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE, 2013, pp. 23–32.

[13] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 4, pp. 333–369, Oct. 1997.

[14] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2002, pp. 4–16.

[15] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Foundations of Software Engineering (FSE)*. ACM, 2011, pp. 267–277.

[16] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *International conference on Software engineering*, ser. ICSE'99. ACM, 1999, pp. 213–224.

[17] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*. IEEE CS, 2008, pp. 117–126.

[18] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *TACAS*. Springer-Verlag, 2005, pp. 461–476.

[19] D. Distefano and M. J. Parkinson, "jStar: towards practical verification for Java," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, G. E. Harris, Ed. New York, NY: ACM, 2008, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449782

[20] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM, 2001, pp. 57–72.

[21] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *Formal Verification of Object-Oriented Software*, ser. Lecture Notes in Computer Science, B. Beckert and C. Marché, Eds. Springer-Verlag, 2011, vol. 6528, pp. 10–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18070-5_2

[22] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity*, ser. Lecture Notes in Computer Science, J. N. Oliveira and P. Zave, Eds., vol. 2021. Springer-Verlag, Mar. 2001, pp. 500–517. [Online]. Available: http://www.springerlink.com/content/nxukfdgg7623q3a9

[23] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: inferring the specification within," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. USENIX Association, 2006, pp. 161–176.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. ACM, 2007, pp. 123–134.

[25] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 191–200.

[26] D. McAllester, "On the complexity of static analysis," *Journal of the ACM*, vol. 49, no. 4, pp. 512–537, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1145/581771.581774

[27] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 296–305, 2005.

[28] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ESEC/FSE-13: Symposium on Foundations of software engineering*. ACM, 2005, pp. 306–315.

[29] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 35–44.

[30] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Springer-Verlag, 2009, pp. 318–343.

[31] 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 *Symposium on The foundations of software engineering*, ser. ESEC/FSE '09. ACM, 2009, pp. 383–392.

[32] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 339–349.

[33] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 282–291.

[34] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE'09: Conference on Automated Software Engineering*. IEEE CS, 2009, pp. 295–306.

[35] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 466–480, 2005.

[36] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *ASE'09: Conference on Automated Software Engineering*. IEEE CS, 2009, pp. 283–294.

[37] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE'09: Conference on Automated Software Engineering*. IEEE CS, 2009, pp. 371–382.

[38] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 178–189.