DynaDoc: Automated On-Demand Context-Specific Documentation*

Ahmed Tamrawi†, Sharwan Ram§, Payas Awadhutkar§, Benjamin Holland§, Ganesh Ram Santhanam†, Suresh Kothari§
§Electrical and Computer Engineering Department
Iowa State University, Ames, Iowa 50011
{sharwan§,payas§,bholland§,kothari§}@iastate.edu
†Ensoft Corp., Ames, Iowa 50010
{ahmedtamrawi†,ganeshramsanthanam†}@ensoftcorp.com

***Abstract*—This 2018 DOCGEN Challenge paper describes DynaDoc, an automated documentation system for on-demand context-specific documentation. A key novelty is the use of graph database technology with an eXtensible Common Software Graph Schema (XCSG). Using XCSG-based query language, DynaDoc can mine efficiently and accurately a variety of program artifacts and graph abstractions from millions of lines of code to provide semantically rich documentation. DynaDoc leverages the extensibility of XCSG to link program artifacts to supplementary information such as commits and issues.**
Sample documentation generated by DynaDoc is available at: `https://ensoftcorp.github.io/DynaDoc/`

***Index Terms*—Software Documentation, Graph Database, Graph Abstractions**

I. Introduction

Documentation is an essential resource for creating and maintaining software systems. A recent paper [8] describes challenges for developing On-Demand Developer Documentation (OD3) systems.

This paper is about the OD3 system we are developing. A key novelty of our OD3 system lies in the use of graph database technology with an eXtensible Common Software Graph Schema (XCSG) [4] for linking program and supplementary artifacts at different levels of granularity. The graph schema serves as a common foundation for developing documentation systems that can work for a variety of programming languages. Using the XCSG-based query language provided by our Atlas platform [5], we have developed an engine for mining program artifacts accurately and efficiently from millions of lines of code [2], [9].

Besides textual and tabular information, DynaDoc generates different kinds of program graphs as a part of the documentation. We shall illustrate how the program graphs serve as valuable context-specific documentation.

The automatically generated sample documentation page [1] incorporates search and interaction capabilities to navigate through the documentation and access program graphs. A guide to DynaDoc can be viewed by clicking the *DynaDoc Guide* button at the top of the sample page.

* This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

II. DynaDoc System Summary

Developing a system like the DynaDoc from scratch would be many years of engineering effort. Fortunately, we have been able to cut down the effort significantly by leveraging our Atlas platform [5] for building software analysis and transformation tools. Atlas stores program artifacts using its graph database with various types of nodes and edges. Tags for nodes and edges represent program semantics. Results from a number of control and dataflow analyses are incorporated in the graph database. Program analyzers can use the *tagging mechanism* to add new semantics (e.g., call sites in C programs are tagged by an analyzer to denote use of function pointers). The tagging has multiple uses including its use for storing supplementary artifacts such as nodes that represent issues.

The generated documentation includes the standard information that Javadoc [7] provides. With the Atlas graph database, DynaDoc has access to program semantics that is significantly richer than the information produced by the Javadoc. Moreover, the XCSG-based query language and query-embedded scripts that we provide with Atlas can be invoked by DynaDoc to produce context-specific documentation.

The DynaDoc baseline documentation is shown in the sample documentation page [1]. Beyond the Javadoc information, the DynaDoc baseline documentation has program graphs. These include: the *control flow graph* (CFG) the *call graph* (CG), and the *data flow graph* (DFG). We shall later discuss a case of how the program graphs can be valuable as context-specific documentation.

Context-specific documentation is needed to work with a particular class of programs. For example, while Android applications are written in Java, the baseline Java documentation is not enough. Developers need Android-specific documentation, including documentation about how the Android permissions and intents are used by the application. With the extensible graph schema, context-specific information (e.g., information gleaned from the App manifest) can be captured in the graph database of Atlas [6]. The specificity of documentation is also important for working with a particular class of problems. For example, the documentation about loops (e.g., inter-procedural nesting of loops) is needed for addressing the

execution time performance problems [3]. Generating context-specific documentation poses the challenge of developing accurate and scalable program analyzers for mining context-specific information.

The burden of developing program analyzers is significantly alleviated by the high-level query-embedded programming enabled by the Atlas platform [5]. Let us illustrate it by the following queries we have implemented to generate the data flow graph included in the baseline DynaDoc documentation.

```
var dfEdges = edges(XCSG.DataFlow_Edge);
dfForward = dfEdges.forwardStep(field);
dfReverse = dfEdges.reverseStep(field);
fieldUse = dfForward.union(dfReverse);
```

The query uses the pre-computed dataflow edges stored in the Atlas database. Starting at a field, it goes one step forward to mine the methods that use (read from) the selected field and one step backward to mine the methods that define (write to) that field.

By implementing graph traversal, the query provides a versatile mechanism to generate a variety of program graphs. For example, substitute `method` for `field` and `Call` for `DataFlow_Edge` and we get the following queries for generating the *call graph* and the *reverse call graph* starting at the selected method. One other change is to use `forward` and `reverse` instead of `forwardStep` and `reverseStep` to fully traverse both ways to get the entire graph.

```
var callEdges = edges(XCSG.Call);
callForward = callEdges.forward(method);
callReverse = callEdges.reverse(method);
```

To incorporate supplementary artifacts into the generated documentation, DynaDoc mines the given datasets for Bugzilla issues and the commits history. The goal is to link each program artifact (e.g., Java class) to corresponding issues and commit records.

DynaDoc leverages the modified program artifacts mentioned in commit records to link commit records with program artifacts. It also leverages the commit record messages that include references to Bugzilla issue IDs - to associate commit records with issues. DynaDoc links Bugzilla issues to program artifacts if those artifacts are mentioned in the summary fields of the issues.

Linking the issues and other supplementary artifacts to program artifacts can be challenging in general and it is an area of active research. Currently, DynaDoc implements a proof-of-concept approach for linking program artifacts to supplementary artifacts. Moving forward, we plan to research different techniques and heuristics.

Atlas takes 3 minutes[1] to generate the graph database from the Java source code for the Apache POI project[2] (commit 219dff00e6) with 246401 LOC. The graph database contains 1082829 nodes and 4603117 edges. DynaDoc generates, in 40 seconds, the documentation page [1] for the competition.

III. An Illustration of Practical Use

We abstract the Bugzilla issues in categories based on the types of constraints that are not documented by Javadoc. One category of issues is related to undocumented constraints on parameters or field values. Developers encounter problems because their code does not follow the constraints.

Let us use the Bugzilla issue 51415[3] as an example. The class `XSSFWorkbook` has a method `setSheetName(int sheetIndex, String sheetname)`. This issue arises because there are constraints on the parameter `sheetname` that are not documented by Javadoc. Developers, unaware of the constraints, encounter problems because they use parameter values that do not satisfy the constraints. The DFG shown in the sample documentation [1] for the method `setSheetName` reveals these constraints (e.g., the length of the `sheetname` should be less than 31). The constraint information serves as valuable documentation for developers to avoid this issue.

The issue 51415 is fixed (revision 1138819). By examining the CFG and the DFG from the DynaDoc documentation, we found and reported a similar but new issue[4] that is, currently, not fixed.

Acknowledgment

We thank our colleagues at EnSoft and the Knowledge-Centric Software Engineering Lab at Iowa State University for helping us with our research. We are grateful to the reviewers for their suggestions that have helped us in improving the paper. Dr. Kothari is the founding President of EnSoft.

References

[1] Dynadoc sample documentation. https://ensoftcorp.github.io/DynaDoc/.

[2] Linux Kernel Verification Results. https://lsap.knowledgecentricsoftwarelab.com/.

[3] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 249–258. IEEE, 2017.

[4] EnSoft Corp. Extensible common software graph. https://ensoftatlas.com/wiki/Extensible_Common_Software_Graph. Accessed on: July 18, 2018.

[5] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. Atlas: a new way to explore software, build analysis tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 588–591. ACM, 2014.

[6] Benjamin Holland, Tom Deering, Suresh Kothari, Jon Mathews, and Nikhil Ranade. Security toolbox for detecting novel and sophisticated android malware. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 733–736. IEEE Press, 2015.

[7] Oracle. Javadoc tool home page. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-135444.html. Accessed on: July 18, 2018.

[8] Martin P Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, et al. On-demand developer documentation. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 479–483. IEEE, 2017.

[9] Ahmed Tamrawi and Suresh Kothari. Projected control graph for computing relevant program behaviors. *Science of Computer Programming*, 163:93 – 114, 2018.

[1] Timings are based on a MacBook Pro machine with 2.5 GHz Intel Core i7 and 16 GB of RAM.

[2] https://github.com/apache/poi/tree/REL_3_17_FINAL

[3] https://bz.apache.org/bugzilla/show_bug.cgi?id=51415

[4] The reported issue: https://bz.apache.org/bugzilla/show_bug.cgi?id=62551