XScan: An Integrated Tool for Understanding Open Source Community-based Scientific Code *

Weijian Zheng[1][0000−0003−2791−0031], Dali Wang[2][0000−0001−6806−5108] **, and Fengguang Song[1][0000−0001−7382−093X]

[1] Indiana University-Purdue University, Indianapolis, IN 46202, USA
zheng273@purdue.edu, fgsong@iupui.edu
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, United States
wangd@ornl.gov

Abstract. Many scientific communities have adopted community-based models that integrate multiple components to simulate whole system dynamics. The community software projects' complexity, stems from the integration of multiple individual software components that were developed under different application requirements and various machine architectures, has become a challenge for effective software system understanding and continuous software development. The paper presents an integrated software toolkit called X-ray Software Scanner (in abbreviation, *XScan*) for a better understanding of large-scale community-based scientific codes. Our software tool provides support to quickly summarize the overall information of scientific codes, including the number of lines of code, programming languages, external library dependencies, as well as architecture-dependent parallel software features. The XScan toolkit also realizes a static software analysis component to collect detailed structural information and provides an interactive visualization and analysis of the functions. We use a large-scale community-based Earth System Model to demonstrate the workflow, functions and visualization of the toolkit. We also discuss the application of advanced graph analytics techniques to assist software modular design and component refactoring.

Keywords: Application software analysis · Community-based code · Code modulation · Code refactoring

1 Introduction

Many scientific communities have employed community-based models to simulate complex dynamics [8, 22]. These community-based models usually adopted open modeling and open coupling infrastructure, and integrated many individual components to address community-driven scientific questions. Since these

* This research was funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (Interoperable Design of Extreme-scale Application Software).

** Corresponding author, POBox 2008, MS 6301, Oak Ridge National Lab, Oak Ridge, TN 37831, USA. Tel: 8652418679

Zheng, W., Wang, D., & Song, F. (2019). XScan: An Integrated Tool for Understanding Open Source Community-Based Scientific Code. In J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, & P. M. A. Sloot (Eds.), Computational Science – ICCS 2019 (pp. 226–237). Springer International Publishing. https://doi.org/10.1007/978-3-030-22734-0_17

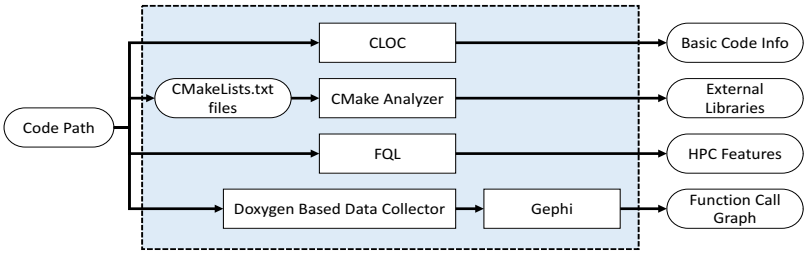


Fig. 1: Architecture of the XScan toolkit.

models are developed by multidisciplinary communities over a variety of high-performance computational facilities [8], close collaborations and continuous communications among many domain science groups and computational science groups are required for improving model understanding and development. To this end, we present a software toolkit to facilitate the understanding of these complex software systems and layout some considerations on further code migration and refactorization. We design and develop a static software analysis tool, which is named as X-ray Software Scanner (i.e., *XScan*). The XScan toolkit can collect an array of software specific information related to overall source code meta data, third-party library requirements/dependencies, major HPC software features, and detailed function relationship. XScan consists of four components (see Fig. 1): 1) an integration with CLOC [3] to provide information about the used high level programming languages and number of lines of code in each programming language (Section 2.1.1); 2) a CMake [13] based analysis component to reveal what external third-party libraries are required by an open-source community project and their dependency relationship (Section 2.1.2); 3) an HPC-specific query language (Feature Query Language FQL) and component to search for HPC hardware and architecture features that are required by an open-source project (Section 2.1.3); and 4) a Doxygen [24] based data collector to collect function-relevant information and build graphs to facilitate big graphs and networking analysis targeting software engineering problems (Section 2.2).

Finally, for the demonstration purpose, we apply the XScan toolkit to collect the overall information and structural relationship of an open-source community Earth System Modeling system, called Exascale Energy Earth System Model (E3SM). Mainly funded by US Department of Energy, E3SM is a computationally advanced coupled climate-energy model to investigate the challenges posed by the interactions of weather-climate scale variability with energy and related sectors.

2 Methodology

Fig. 1 shows the architecture of our XScan toolkit, whose four components can be classified into two categories: 1) Components to show overall information of

source code, and 2) Components to show function relationship information. The two categories will be described in the following two subsections, respectively.

2.1 Category 1: Overall information of source code

2.1.1 Language and size The very first step of software system understanding is to present the information about a community project's size and programming languages. There are several existing tools available for code size evaluations, such as CLOC (Counting Lines Of Code) [3], Sonar [19] and SLOC [25]). We incorporates CLOC to XScan to provide the language and size information because CLOC consists of a single Perl file and can be executed on any machine. With CLOC, XScan can measure the lines of source code for a variety of languages, and can differentiate between the actual code, blank lines, code crossing multiple lines, and comment lines for each programming language. We consider it a convenient feature for XScan users.

2.1.2 External library dependency In addition to the programming language and size information, it is important for community users to know what third-party libraries are needed by the open-source code. Since physically installing open-source projects is often challenging and time-consuming, we target designing a module (inside XScan) to parse and analyze the project's build systems (e.g., Makefile [20], CMake [13], Autoconf [12]) to automatically collect the library dependency information. This quick scan functionality is particularly useful when users are evaluating many choices of open source software and considering distinct computer architectures.

In this paper, we choose CMake as an example to achieve the objective without any physical software installation. CMake is an operating system independent tool for building and testing software. By modifying a configuration file (commonly named as *CMakeLists.txt*), developers are able to control the whole building process on most systems.

We design and develop a "CMake Analyzer" module in XScan, which scans all the *CMakeLists.txt* files in an open-source software project, and finds all the

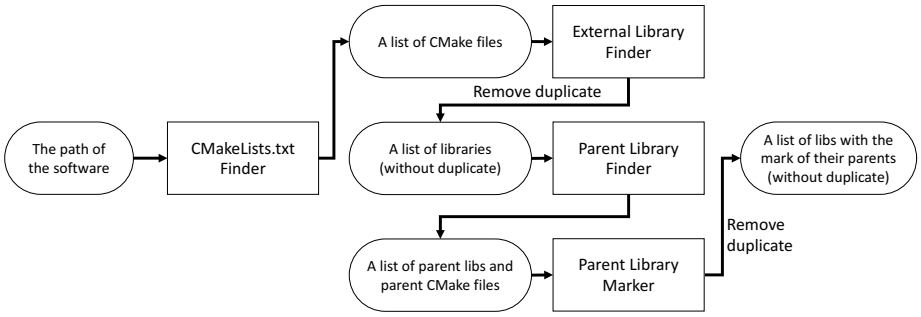


Fig. 2: Workflow of the CMake Analyzer.

needed external libraries and identifies the dependency relationship among the detected libraries.

The design of CMake Analyzer is shown in Fig. 2, in which ellipses represent the data exchanged between software components meanwhile rectangles represent the software components. CMake Analyzer works in the following three steps: 1) Search for CMake files by using the *CMakeLists.txt Finder* component (see Fig. 2); 2) Scan the CMake files and identify the needed external libraries by the *External Library Finder* component; And 3) Use the *Parent Library Finder* and *Parent Library Marker* components to search and mark the parent of each needed library. The final output is a dependency graph for all the libraries.

2.1.3 Query software features User community may want to understand special characteristics of the complex community projects. We defined these special characteristics as "features" of these community projects. For example, the user who are interested in high performance computing (HPC), may ask whether MPI or GPU is used by the code. To answer these questions, we created a new language, called *Feature Query Language* (FQL) [26], to describe the user's questions. It is an extensible and flexible language. Users can translate their questions to a query quickly when they know the keywords of a feature. Next, we design and implement a program parser to parse and execute the FQL queries. We also provide a number of predefined queries for commonly asked HPC related feature questions. By executing those queries, the FQL component can provide users with an overview of the HPC features in the community project.

2.2 Category 2: Function relationship of the source code

It is important to gather function relationship information, from which users can gain insights into the internal structure of a community project.

XScan uses Doxygen [24] as a lower-level internal library to collect detailed information including code structure, function dependency, class inheritance relations as well as collaboration diagrams. Instead of using Doxygen as a GUI tool that generates documentation for projects, we call the Doxygen library and API directly to create an in-memory software-data repository, and perform in-situ graph partitioning, coloring, and data analysis.

Fig. 3 shows an overview of the in-memory Doxygen-enhanced Analyzer, which consists of three major parts: 1) *Data Collection* (i.e., the large blue rectangle on the upper left in Fig. 3), 2) *Doxygen API* (the large yellow rectangle at the bottom), and 3) *Data Presentation and Analysis* (the blue rectangle on the upper right). The following subsections will describe them in more details.

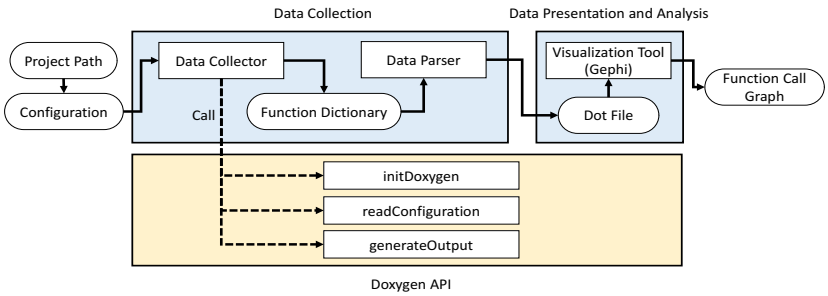


Fig. 3: Workflow of Doxygen-based Software Analyzer

2.2.1 Data Collection & The Doxygen API :

Data Collection: The input is a configuration file to configure Doxygen. Next, *Data Collector* calls the Doxygen APIs (shown in the yellow box) to obtain information of the code. In our implementation, *Data Collector* first calls *initDoxygen()*, then calls *readConfiguration* to read the configuration file, and finally calls *generateOutput* to generate the output.

Function Dictionary: The *generateOutput* function yields a *Function Dictionary*, which is stored in an in-memory data structure. In XScan, we need to know each function's function name, caller, module name and location. All of them are stored in this data structure. Because any information can be found by looking up the in-memory *Function Dictionary* data structure, XScan can perform fast graph and network data analytics efficiently in an in-situ manner.

Data Parser: Finally, XScan converts the code information stored in *Function Dictionary* to different graphs. The output graphs are stored in the Dot format [5]. Note that a Dot file can be read by many graph visualization tools.

2.2.2 Data Presentation and Analysis :

This last part is responsible for reading the graphs generated from *Data Parser*, and using the open-source graph visualization tool Gephi [2] to visualize them with different layouts. These graphs will also be processed by using big graph/network analysis tools to facilitate software engineering code optimization, redesign and refactoring.

3 Case Study

In this section, we first introduce an Earth System Model. Then, we show the general code information of the model using XScan functions stated in section 2.1. After that, we present different function call graphs generated by the Doxygen-based Software Analyzer within XScan. Preliminary data analyses based on those graphs are also demonstrated at the end of the section.

3.1 The E3SM application

The Energy Exascale Earth System Model (E3SM) is a computationally advanced coupled climate-energy model to investigate the challenges posed by the interactions of weather-climate scale variability with energy and related sectors [8]. E3SM is a 3D (plus time) computer-based general-circulation model that uses mathematical formulas to simulate the chemical and physical processes that drive Earths climate. Being extraordinarily sophisticated, E3SM can be used to study phenomena ranging from the effect that an ocean surface temperature has on tropical cyclone patterns to the impact of land use on carbon dioxide concentration in the atmosphere.

Currently, E3SM contains several major community model components to simulate Earth systems: Atmosphere, Ocean, Land, Glacier, and Sea Ice. E3SM also provides a complicated system script tool, which allows science-oriented modelers to automatically reconfigure, compile, build, and submit jobs. It is an extremely valuable feature in helping E3SM users conduct computational experiments on various high-end computers.

3.2 Overall source code information of E3SM

3.2.1 Language and code size :

We use XScan/CLOC to check the overall information about E3SM's source code with respect to the source code size and programming languages. As listed in Table 1, the top ten programming languages used in E3SM (from the most to least) are FORTRAN 90, HTML, XML, C, Perl, Python, Tex, Fortran 77, Shell, and CMake. FORTRAN 90 is the most used programming language. There are 2184 FORTRAN files and nearly 1 million lines of FORTRAN code (excluding comments and empty lines), while HTML is mainly used for the documentation purpose.

Table 1: Top 10 programming languages inside the E3SM source code package

Detected language:	Number of files:	Number of lines of the code:
Fortran 90	2,184	934,296
HTML	429	158,640
XML	307	85,785
C	124	46,245
Perl	156	37,860
Python	240	34,786
Tex	171	23,596
Fortran 77	70	21,118
Bourne Shell	236	20,598
CMake	213	6,125

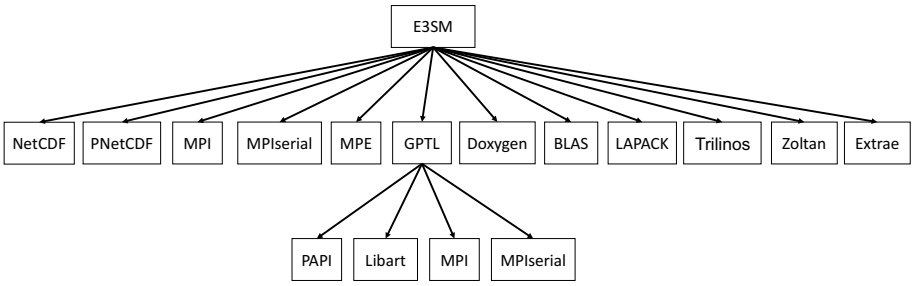


Fig. 4: CMake Analyzer results of E3SM via XScan.

3.2.2 External library dependencies :

XScan is applied to scan the required third-party libraries for E3SM. Fig. 4 shows the external libraries needed by E3SM. From the XScan-generated graph, we can see that E3SM needs more than ten external libraries, which include the data library NetCDF, message passing library MPI, numerical libraries of LAPACK, timing library GPTL, scientific application library Trillinos, scientific simulation library Zoltan, and program instrumentation library Extrae. We may also notice that MPI appears twice in the graph as it is used by both GPTL and the main program of E3SM.

3.2.3 HPC specific features in E3SM :

We apply XScan/FQL to extract HPC related features that are used and required by E3SM. The HPC results obtained by executing related FQL queries are listed in Table 2.

Table 2: FQL result of the E3SM.

MPI Yes	Min version required: 2.0	MPI one-sided communication: Yes
	MPI process topology: Cartesian	MPI I/O Yes
OpenMP Yes	Hybrid MPI/OpenMP: Yes	Task programming constructs: Yes
	OpenMP scheduling method: Dynamic	
CUDA No	Support multiple GPUs: –	Single/Double precision: –
OpenACC Yes	Atomic operation: No	Asynchronous operation: No
Language Support	Min required C compiler: C99	Fortran standard: Fortran 2003

From the table, we find that MPI, OpenMP and OpenACC are all employed by the E3SM project. We can also find further information about how E3SM uses MPI, OpenMP, and OpenACC. For instance, E3SM uses the MPI Cartesian

topology, one-sided communication and MPI I/O techniques. E3SM is written mainly in the FORTRAN language meanwhile using the C language for system level functions, such as parallel I/O and string handling. Also, please note that OpenMP is not listed in previous Fig. 4 since recent GCC compiler has the full support of OpenMP.

3.3 Presentation and exploration of code function relationship in E3SM

In this section, we present two sets of graph results generated by XScan for the E3SM project: Partitioned function graphs computed by Doxygen-enhanced Software Analyzer, and Preliminary data analysis results on investigating software modularity.

3.3.1 Different presentations of source code function graphs :

Fig. 5 presents the global function-call graph for the E3SM project. In this graph, different functions have different colors which are decided by their resident directories. As shown in the figure, the directory */components/cam* has the largest number of functions (i.e., the green nodes). Moreover, the functions located in directory */cime* — which are shown as the red nodes and working as the coupler to combine different models — have many connections with other colored nodes.

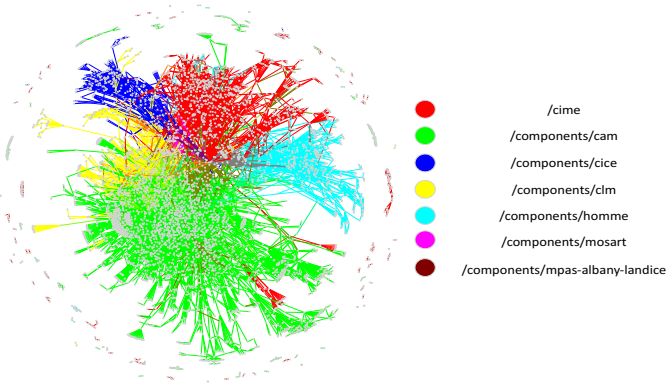
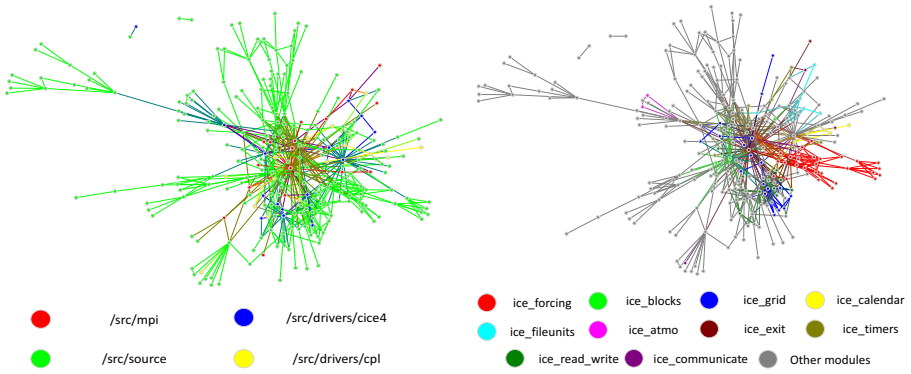


Fig. 5: Global function graph for the E3SM project.

Next, we want to dive into the specific */components/cice* directory to investigate the function relationship in the CICE component. For example, Fig. 6.a shows a colored function graph for the CICE component only, based on each function's subdirectoy location. Since there are four sub-directories in CICE, the function graph has four colors.



(a) CICE functions colored by locations. (b) CICE functions colored by modules.

Fig. 6: Different colored function graphs for the CICE component of E3SM.

Finally, we use XScan to study how a software module of interest can interact with other modules. By using the identical set of vertices and edges as Fig. 6.a, Fig. 6.b focuses on the module of *ice_forcing*, and reveals its interaction with other modules. In Fig. 6.b, all the nodes that have no connection with the module *ice_forcing* are colored in the grey color. The rest of the nodes (i.e., with colors) are colored separately based on their corresponding module names. As shown in the figure, nine modules have interactions with the *ice_forcing* module.

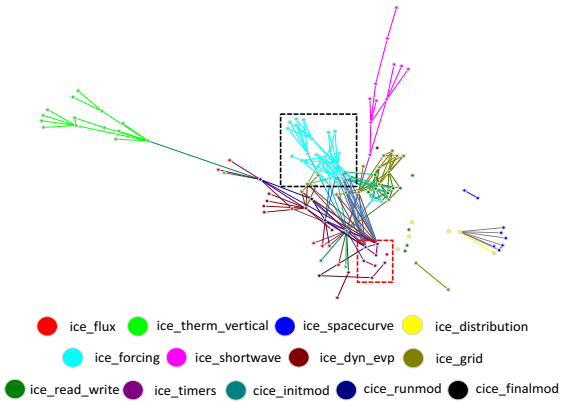


Fig. 7: Partial function graph for CICE before merging interface functions.

3.3.2 Preliminary graph analysis results :

Here, we present our preliminary big graph data analysis results based on the graph information collected by XScan. One purpose of the data analysis is to help users better understand the software modularity, and assist users to make good decision on software component refactoring. We use the CICE case again to illustrate our data analysis approach.

As shown in Fig. 6, the CICE component has more than 50 modules. For the purpose of demonstration, we abstract 13 modules as presented in Fig. 7. We propose and make a few modifications to those nodes, using common software engineering approaches, such as function encapsulation, merge, and function interface redesign. For all the modifications, we check how graph modularity score are changed. Graphs with higher modularity scores are more modular with denser connections inside the module and sparser connections between modules.

In Fig. 7, there are a certain number of functions (presented as nodes in the graph) that have edges connected with other modules. We call them *interface functions* of a module. We can merge the interface functions of the module *ice_timers* as shown in the red rectangle box. It is an action to mimic the function encapsulation and interface redesign. As a result, the new graph is more modular with its modularity score improved from 0.649 to 0.66. On top of that, we further combine the interface functions of module *ice_forcing*, which is shown in the big rectangle box on the top of the figure. The modified new graph is even more modular with its modularity score rising to 0.71. Therefore, we can conclude that merging the interface functions of a module is able to make open source software more modular.

Currently, we only apply the modularity metric to evaluate the impact of source code modifications. Our next research plans to design new metrics and apply graphs analysis techniques to guide our further code developments, such as using k-component analysis [15] to estimate the difficulties of module refactoring.

4 Related Work

A number of tools have been developed to collect various kinds of overall code information. For example, a few tools can analyze a projects code size and languages by scanning the source code [3, 25, 19]. Certain tools like the ScanCode toolkit [18] and fossology [6] can provide the software license, copyright, dependency and other kinds of information of the code. By integrating other third party package managers (e.g., MAVEN, PIP, NPM) and code scanners (e.g., Licensee [11], ScanCode), OSS Review toolkit can tell user dependencies of different open-source libraries of the project [21].

Function call graph is commonly used to represent the calling relationship between different functions [17]. Many tools such as Doxygen [24], CFlow [16] and Egypt [4], are developed to statically extract the relationship. There are also many different ways to use the function-call graph. For example, some researchers analyze the software change impacts by checking the transitive closure of the function-call graph [1, 9]. Function-call graph based reverse engineering work,

such as the work of Mitchell et al. [14] and Vassilios et al. [23], can abstract the structure of software by applying clustering based machine learning algorithms to function call graphs.

5 Future work

In the next phase, we will further enhance the in-memory software analyzer to collect more information (such as global variable, data context, data aliases) from static software analysis and to help users to evaluate the complexity of scientific software projects. We will apply graph algorithms and graph analysis tools (such as networkX [7] and SNAP [10]) to the software function graphs to estimate efforts to factorize a submodule as well as the workload to combine submodules from multiple different software to build an integrated software system. We will also look into the methods to estimate the effort of migrating software to a new platform. For instance, we can list modules, functions, and libraries (i.e., both internal and external) that need to be modified. Furthermore, with the vast collection of software information, we can transform the software understanding problem into software optimization problem. Then, we will apply machine learning approaches (such as reinforcement learning) to aid users in optimizing software structure and functional redesigns.

References

1. Arnold, R.S.: Software change impact analysis. IEEE Computer Society Press (1996)
2. Bastian, M., Heymann, S., Jacomy, M., et al.: Gephi: an open source software for exploring and manipulating networks. Icwsm **8**(2009), 361–362 (2009)
3. CLOC: https://github.com/AlDanial/cloc (2018)
4. Egpyt: http://www.gson.org/egypt (2018)
5. Gansner, E., Koutsofios, E., North, S.: Drawing graphs with dot (2006)
6. Gobeille, R.: The fossology project. In: Proceedings of the 2008 international working conference on Mining software repositories. pp. 47–50. ACM (2008)
7. Hagberg, A., Swart, P., S Chult, D.: Exploring network structure, dynamics, and function using networkx. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States) (2008)
8. Hurrell, J.W., Holland, M.M., Gent, P.R., Ghan, S., Kay, J.E., Kushner, P.J., Lamarque, J.F., Large, W.G., Lawrence, D., Lindsay, K., et al.: The community earth system model: a framework for collaborative research. Bulletin of the American Meteorological Society **94**(9), 1339–1360 (2013)
9. Law, J., Rothermel, G.: Whole program path-based dynamic impact analysis. In: Proceedings of the 25th International Conference on Software Engineering. pp. 308–318. IEEE Computer Society (2003)
10. Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. ACM Transactions on Intelligent Systems and Technology (TIST) **8**(1), 1 (2016)
11. Lisensee: https://ben.balter.com/licensee (2015)

12. MacKenzie, D., Elliston, B., Demaille, A.: Autoconf: Creating automatic configuration scripts. User Manual, Edition **2** (1996)
13. Martin, K., Hoffman, B.: Mastering CMake: a cross-platform build system. Kitware (2010)
14. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering **32**(3), 193–208 (2006)
15. Moody, J., White, D.R.: Structural cohesion and embeddedness: A hierarchical concept of social groups. American Sociological Review pp. 103–127 (2003)
16. Poznyakoff, S.: Gnu cflow (2005)
17. Ryder, B.G.: Constructing the call graph of a program. IEEE Transactions on Software Engineering (3), 216–226 (1979)
18. ScanCode: https://github.com/nexB/scancode-toolkit (2016)
19. Sonar: https://www.sonarqube.org/ (2018)
20. Stallman, R.M., McGrath, R., Smith, P.: GNU Make: A Program for Directed Compilation. Free Software Foundation (2002)
21. oss-review toolkit: https://github.com/heremaps/oss-review-toolkit (2017)
22. Trolle, D., Hamilton, D.P., Hipsey, M.R., Bolding, K., Bruggeman, J., Mooij, W.M., Janse, J.H., Nielsen, A., Jeppesen, E., Elliott, J.A., et al.: A community-based framework for aquatic ecosystem models. Hydrobiologia **683**(1), 25–34 (2012)
23. Tzerpos, V., Holt, R.C.: Accd: an algorithm for comprehension-driven clustering. In: Reverse Engineering, 2000. Proceedings. Seventh Working Conference on. pp. 258–267. IEEE (2000)
24. Van Heesch, D.: Doxygen: Source code documentation generator tool. URL: http://www.doxygen.org (2019)
25. Wheeler, D.A.: Sloc count users guide (2004)
26. Zheng, W., Song, F., Wang, D.: FQL: An extensible feature query language and toolkit on searching software characteristics for HPC applications. arXiv preprint arXiv:1905.09364 (2019)