






# FQL: An Extensible Feature Query Language and Toolkit on Searching Software Characteristics for HPC Applications

Weijian Zheng<sup>1</sup> , Dali Wang<sup>2</sup> , and Fengguang Song<sup>1</sup> 

<sup>1</sup> Indiana University-Purdue University, Indianapolis, IN 46202, USA  
zheng273@purdue.edu, fgsong@iupui.edu

<sup>2</sup> Oak Ridge National Laboratory, P.O. Box 2008, MS 6301,  
Oak Ridge, TN 37831, USA  
wangd@ornl.gov

**Abstract.** The amount of large-scale scientific computing software is dramatically increasing. In this work, we designed a new query language, named Feature Query Language (FQL), to collect and extract HPC-related software features or metadata from a quick static code analysis. We also designed and implemented an FQL-based toolkit to automatically detect and present software features using an extensible query repository. A number of large-scale, high performance computing (HPC) scientific applications have been studied in the paper with the FQL toolkit to demonstrate the HPC-related feature extraction and information/metadata collection. Different from the existing static software analysis and refactoring tools which focus on software debug, development and code transformation, the FQL toolkit is simpler, significantly lightweight and strives to collect various and diverse software metadata with ease and rapidly.

**Keywords:** Feature Query Language · Static code analysis · High-performance computing

## 1 Introduction

Open source scientific software projects are growing explosively in number and size. Many companies, universities, and national laboratories build their software ecosystems around the open-source software projects. There are also a lot of ongoing efforts to combine different software modules to create a larger scale software system (e.g., climate modeling and simulation [1], fluid/solid dynamics computations [20], material science [17], etc.). The complexity of large-scale scientific

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).

© This is a U.S. government work and not under copyright protection in the US.; foreign copyright protection may apply 2020

G. Juckeland and S. Chandrasekaran (Eds.): HUST 2019/SE-HER 2019/WIHPC 2019, CCIS 1190, pp. 129–142, 2020.

[https://doi.org/10.1007/978-3-030-44728-1\\_8](https://doi.org/10.1007/978-3-030-44728-1_8)

models developed for specific machine architectures and application requirements has become a barrier that impedes continuous software development. Furthermore, more and more scientific codes have incorporated high-performance computing (HPC) features that, in turn, create machine configuration, computer architecture, user and system library dependency issues.

Hence, given a large number of open source software projects, it is critical to provide an efficient way for decision makers (such as users, administrators, customers, developers, investors, and software managers) to quickly evaluate the software and understand its structure and characteristics [18, 32]. Also, as numerous codes have been released and published every day in the open repositories (such as GitHub and bitbucket) as well as institution-owned repositories (such as DOECode at the Office of Scientific and Technical Information ([www.osti.gov/doecode](http://www.osti.gov/doecode))), we need to develop a portable tool that can automatically extract and collect essential features from these scientific codes.

In this paper, we target at creating a software toolkit to discover open source software projects' features. Here, "features" refer to any characteristic and metadata related to the software, including programming languages, third-party library dependency, special hardware requirement, particular tools, adopted programming models, and so on.

We use open source science and engineering application software on high performance computing (HPC) systems as examples to drive the design and development of our toolkit due to the science and engineering software's large scale, high complexity, and utilization of a wide variety of computer hardware. For instance, we experiment with a number of science codes from several large-scale DOE programs to harvest HPC features for code archive purpose and beyond.

In order to handle nearly "arbitrary" queries of interest from users, we need a flexible and extensible solution that can process any number/type of features in any open source software and can also efficiently answer these feature-related questions. Our proposed solution is based upon a new language called *Feature Query Language* (FQL) that lets users describe their queries (or questions) in the FQL language. Given an FQL query, we then design a new software toolkit, which can parse the user input, execute the query, scan open source software, and present the final results. Our design shares the same philosophy with the popular Structured Query Language (SQL) to support users' arbitrary queries about databases [11]. The distinction between FQL and SQL is that FQL is designed to query open source code repository, which is being viewed as another type of database, meanwhile achieving SQL's portable, regular, structured, and simple characteristics. We expect that this new toolkit will significantly benefit broader scientific computing communities who are facing similar challenges.

The rest of the paper is organized as follows. Next section presents the related work. Section 3 describes our toolkit from three perspectives: (1) FQL language, (2) overall software workflow, and (3) FQL toolkit implementation. Section 4 presents the results obtained by executing predefined queries. Section 5 concludes the impact and the possible future directions for our work.

## 2 Related Work

This paper introduces a SQL-like query language and supporting toolkit, called Feature Query Language (FQL), to collect HPC-related software metadata from a quick static code analysis. We present the related work in two categories: (1) software analysis tools without using domain-specific languages; and (2) software analysis tools using domain-specific languages.

### 2.1 Software Analysis Tools Without Using Domain-Specific Languages

There is a lot of software engineering work on code analysis that does not use any domain specific language. A few software analysis tools are designed to obtain low level code information. For instance, security flaws may be detected effectively [35]. In the work of Bush et al., an analyzer for program errors is created [5]. Buffer overflow and function dependencies can be found in the work [14] and [33] respectively. On the other hand, a number of tools focus on providing a higher level overview of the code. For example, the open source toolkits ScanCode [26] and Fossology [16] are used to extract the license, copyright, package dependency and other information. Oss-review-toolkit is designed to provide the dependencies of different open source libraries for software [25].

Although these software tools can detect specific software information, they are not generic enough to query any type of features that may be interesting to different users. Our FQL is designed to provide an abstract query interface to users so that one can define any new queries in FQL and get answers quickly.

### 2.2 Software Analysis Tools Using Domain-Specific Languages

On the other hand, many software engineering tools define new programming languages that are of domain purpose only (i.e., DSL) [21]. As described by Deursen et al. [29], DSL has the following advantages: (1) more expressiveness in the specific domain, (2) more friendly to domain experts, and (3) more verification and optimization can be performed at the domain level. Hence, DSL has been widely used in various static code analysis and refactoring tools. In this subsection, we present software tools that use DSL in two classes: (1) static code analysis and (2) code transformation.

**Static Code Analysis:** Static analysis tools such as crocopat [3], JRelCal [23], JTL [6], SOUL [13], .QL and SemmleCode [12,30] use their own languages and patterns to represent the desired features in the target code. For instance, crocoPat uses patterns that are described by binary decision diagrams for detecting inter-class structures, which consist of good object-oriented (OO) design patterns, weak anti-patterns, etc. [3]. JRelCal is a library to obtain different kinds of relations in the source code based on the binary relational calculus [23]. JTL (Java Tools Language) is a Java programming language extension to represent

Java code patterns by providing native and predefined first-order logical predicates [6]. By using the new JTL representation, a JTL processor can analyze queries for object-oriented Java programs. SOUL is a Prolog-like language to query a program’s structure and is mainly used for detecting source code structures of Java and Smalltalk programs [13]. .QL is an object-oriented query language for measuring code quality, observing bugs and other analysis tasks [12], meanwhile SemmlCode [30] is a free Eclipse plugin, which adopts the .QL language.

However, the above programming languages or tools have different goals from ours, and they are intended to query software development related questions (e.g., design defect, implementation bug, suboptimal code structure, inter-class relationship, design violation, etc.), while our work targets at collecting software’s meta-data, parallel library requirements, architecture or device dependency, and other HPC-related questions.

**Code Transformation:** Another influential line of work aims to support automatic code transformation or refactoring instead of static code analysis only [2, 4, 7, 9, 18, 22, 31]. SrcML is a platform for both code analysis and code manipulation via representing the code in a language called XPath that is similar to XML [7]. Rascal is a programming language designed to integrate program analysis with code manipulation [18]. By defining new basic data types and statically checking the types, Rascal allows programmers to represent a program then transform the program automatically under a number of constraints. DMS is a commercial code transformation tool, which has different performance optimizations [2]. The TXL (Tree Transformation Language) and ELAN [4] languages are used for rule-based code refactoring [9]. Unlike TXL/ELAN, Stratego is able to describe both refactoring strategies and refactoring rules [31]. Moreover, Coccinelle is a special tool to automatically transform Linux kernel drivers by using a language based on the *patch* syntax [22].

Nevertheless, the above code transformation tools require users to write new programs or representations by using the new languages provided by the tools. While they are efficient in code refactoring, they are overly complicated and time-consuming for one’s simple goal of software metadata collection.

### 3 The FQL Language and Toolkit

In this section, we introduce the Feature Query Language (FQL) definition in Subsect. 3.1, then describe the overall workflow of using the FQL toolkit in Subsect. 3.2 followed by the design and implementation of our software toolkit in Subsect. 3.3.

#### 3.1 Feature Query Language (FQL)

The Feature Query Language (FQL) is designed for users to ask any software feature or metadata related questions such as “Does the software require MPI-2?”, “Does the software need GPUs”, “Does the software depend on a special

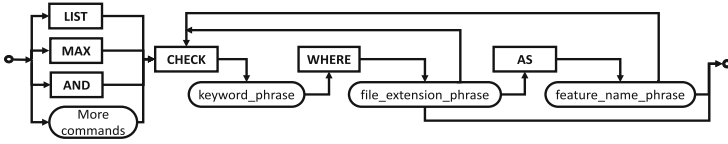


Fig. 1. FQL syntax diagram.

compiler or parallel system library?”, “Does the software take advantage parallel I/O”, etc. Since a user’s questions could be greatly diverse, our FQL must have an extensible architecture and can incorporate any questions of interest. As long as the user knows the keywords of targeted software features, he or she can write a corresponding query in FQL (i.e., an FQL *sentence*) quickly.

### 3.1.1 FQL Syntax

FQL is defined by the FQL syntax, which is comprised of one or more *clauses* (defined in the next paragraph).

If there is one clause, we return the query result of the specific clause. If there are multiple clauses, results from the various clauses will be summarized by an *FQL command*. An FQL *sentence* with multiple clauses can be expressed in the following form:

$$FQL\_command (Clause1, Clause2, \dots) \tag{1}$$

An FQL *Clause* is defined as a combination of *phrases* and FQL *reserved keywords*. An example of *clause* is provided below:

$$CHECK (keyword\_phrase) WHERE (file\_extension\_phrase) AS (feature\_name\_phrase) \tag{2}$$

In the above syntax, *CHECK*, *WHERE* and *AS* are reserved keywords in FQL. They are not case sensitive. Here, a *phrase* is just a set of strings. The current version of FQL has three types of phrases: (i) *keyword\_phrase*, (ii) *file\_extension\_phrase*, and (iii) *feature\_name\_phrase*.

- (i) *keyword\_phrase*: A keyword phrase has a few keywords that describe a specific software feature. Each keyword is simply a string. Different keywords are concatenated by “|” or “&&”. Symbol “|” means that if one of the keywords is found, we claim that the feature is found. Symbol “&&” means that only if all keywords are found, we then claim that the feature is found.
- (ii) *file\_extension\_phrase*: This phrase is used to tell the FQL software toolkit where to search for the keywords. It consists of a list of file extensions connected by “,”. If it is specified as “\*”, the FQL toolkit will check all types of files.
- (iii) *feature\_name\_phrase*: This *feature\_name* phrase is optional and used to specify how to interpret (or name) the query result. For instance, if the *keyword\_phrase* is found to exist in the target software, the toolkit will return the specified meaningful *feature name*. Otherwise (i.e., if there is no

*feature\_name\_phrase* provided), the toolkit will only return True or False based upon the query result.

### 3.1.2 FQL Command

It is common that users may need more than one phrase to define a query. When there are several clauses in a query sentence, results from different clauses will be summarized by executing an FQL command. Currently, FQL provides three commands:

- **LIST:** The LIST command enumerates all the features whose query results are true.
- **MAX:** The MAX command returns the largest query value found in the available features. It can be used to check the software version required by a project.
- **AND:** The AND command returns True only if all clauses' features have been found.

The provided FQL commands can answer many frequently asked questions. In addition, both our FQL syntax and toolkit implementation are designed to be flexible and extensible. Such an extensible design allows new FQL commands to be added to the FQL language quickly whenever needed. We expect to add more commands as the types of queries increase.

To summarize the FQL grammar, we use Fig. 1 to illustrate the syntax of a valid FQL sentence containing more than one clause. FQL-provided commands and FQL-reserved keywords are written in bold uppercase inside rectangles. Phrases and FQL-provided commands are written in lowercase inside ovals. By following the arrows from left to right in Fig. 1, we can construct a valid FQL query.

### 3.1.3 FQL Query Examples

Here, we list five examples of HPC related questions that may be asked by users and the corresponding FQL queries as well as our remarks. For more examples, please refer to Table 1.

*Question 1:* Whether OpenMP is used in the code?

FQL: CHECK (!\$OMP || #pragma omp) WHERE (\*)  
AS (OpenMP)

Note: OpenMP is a widely used API for shared-memory programming [10] in HPC.

*Question 2:* Is one-sided MPI communication used?

FQL: CHECK (MPI\_Put || MPI\_RPut || MPI\_Get  
|| MPI\_RGet) WHERE (\*)

Note: This query is used to check the MPI one-side communication feature. Since no feature name is provided, our FQL toolkit will return True if one of those keywords is found.

**Table 1.** Examples of HPC-related asked questions and corresponding queries

Number	User's Interesting Question	Corresponding FQL Query
1	Is OpenACC used?	CHECK (!\$acc    #pragma acc) WHERE (*) AS (OpenACC)
2	Is OpenACC atomic operation used?	CHECK (acc atomic) WHERE (*) AS (atomicACC)
3	Is CUDA programming used?	CHECK (_device_    _global_    _host_    _noinline_    _forceinline_) WHERE (.cu,.cuh) AS (CUDA)
4	What OpenMP scheduling method is used?	LIST (CHECK (schedule(static) WHERE(*) AS (Static), CHECK (schedule(dynamic) WHERE(*) AS (Dynamic), CHECK (schedule(guided) WHERE(*) AS (Guided), CHECK (schedule(auto) WHERE(*) AS (Auto), CHECK (schedule(runtime) WHERE(*) AS (Runtime))
5	Does it use OpenMP Task programming constructs?	CHECK (omp task    end task    omp taskloop    omp taskloop simd    omp taskyield) WHERE (*)

*Question 3:* What is the minimum version requirement of MPI?

FQL: MAX (  
 CHECK (MPLAINT\_ADD || MPLAINT\_DIFF)  
 WHERE (\*) AS (3.1),  
 CHECK (MPLCOMM\_DUP\_WITH\_INFO ||  
 MPLCOMM\_SET\_INFO) WHERE (\*) AS (3.0),  
 CHECK (MPIDIST\_GRAPH\_CREATE\_ADJACENT  
 || MPIDIST\_GRAPH\_CREATE) WHERE (\*)  
 AS (2.2),  
 CHECK (mpi.h || use mpi || mpif.h) WHERE (\*)  
 AS (2.0))

Note: This query is used to search for the minimum version requirement of the MPI in the code. Please note that if our FQL toolkit finds that MPI is not used by the project, it will return “Not found”.

*Question 4:* What kind of MPI process topology (topologies) is (are) used?

FQL: LIST (  
 CHECK (MPI\_CART\_Create) WHERE(\*)  
 AS (Cartesian),

```

CHECK (MPI_GRAPH_Create) WHERE(*)
  AS (Graph),
CHECK (MPI_DIST_GRAPH_CREATE_Adjacent
  || MPI_DIST_GRAPH_Create) WHERE(*)
  AS (Distributed Graph)

```

Note: This query uses the command LIST, whose function is to list all the features found in the code. For this query, all the MPI process topologies used in the code will be listed by our toolkit.

*Question 5:* Does the project use a hybrid MPI/OpenMP programming model?

```

FQL: AND (
  CHECK (mpi.h || use mpi || mpif.h) WHERE (*)
  AS (MPI),
  CHECK (!$OMP || #pragma omp) WHERE (*)
  AS (OpenMP))

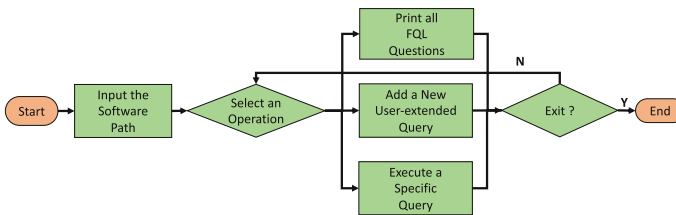
```

Note: This query uses the command AND, which is used to summarize whether all the features are found. As to this sentence, if both MPI and OpenMP are found, our toolkit will return True.

### 3.1.4 Predefined FQL Queries and User-Defined FQL Queries

Our software toolkit can support two types of FQL queries: predefined queries and user-defined queries. Predefined queries correspond to frequently asked questions, which are offered as a list of question choices by our software toolkit. User-defined FQL queries are written by a user based on his or her special questions. Both types of queries can be parsed and executed by our toolkit automatically. In our implementation, all the predefined FQL queries and their corresponding questions (in plain English) are stored in a text file. The user-defined queries can also be added to the text file for future use.

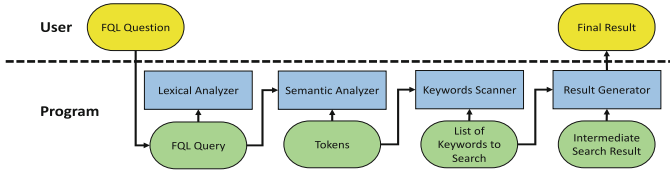
## 3.2 Overall Workflow of the FQL Software Toolkit



**Fig. 2.** Overall workflow of the software

Figure 2 shows the workflow of using the FQL toolkit. There are three major steps to use the software: (i) Users input the targeted software's file path





**Fig. 3.** Software components implemented for parsing and executing FQL queries in the FQL toolkit.

(shown as the first rectangle from the left); (ii) Next, the toolkit pre-scans the targeted software (shown as the second rectangle); and (iii) Based on the user’s choice, the FQL toolkit executes particular operations till the user exits the program.

More details of the three major steps are shown as follows:

- (i) *Input the software path by users:* Our toolkit will firstly ask the user to input a file path to search for. This path should be the top-level directory of the targeted software. All files in the specific file path will be scanned by the FQL toolkit recursively.
- (ii) *Select an operation:* In the second step, our software will ask the user to select an operation to operate. There are three available operations.
- (iii) *Execute a selected operation:* Our software will execute an operation based on the user’s selection in the previous step. Three operations are as follows:
  - To list all the predefined questions. This operation is to remind a user of all predefined FQL queries and corresponding questions (in plain English). The user can then execute a specific query by entering the index number of the question.
  - To add a new user-defined query. The FQL toolkit will also make sure the query entered by a user is valid. It will repeatedly ask the user to input the query until a valid query is received.
  - To execute a specific query. Section 3.3 provides details about how to execute an FQL query by the FQL toolkit.

After executing one of the above three additional operations, the toolkit will check whether the user wants to repeat Step iii or not.

### 3.3 Implementation of the FQL Toolkit

To support FQL, we develop a new software toolkit to parse and execute FQL queries. An overview of the process that parses and executes FQL queries is illustrated in Fig. 3.

As shown in Fig. 3, the two yellow round-corner boxes (above the dotted line) represent a user’s input and output. The four green round-corner boxes (at the bottom) represent the data exchanged between several major program components.

Table 2. HPC features of the different software

	QMC- Pack	ParFlow	E3SM	SICM	Truchas	Tusas	ExaMPM	MEUM-APPS
MPI	✓	✓	✓	✓	✓	✗	✗	✓
MPI min. version required	2.0	2.0	2.0	2.0	2.0	-	-	2.0
MPI process topology	Cartesian, Graph	None	Cartesian	Cartesian	None	-	-	None
MPI one-sided communication	✓	✓	✓	✗	✓	-	-	✗
MPI I/O	✗	✗	✓	✗	✗	-	-	✗
OpenMP	✓	✗	✓	✓	✗	✓	✗	✗
Task programming constructs	✓	-	✓	✗	-	✗	-	-
Hybrid MPI/ OpenMP	✓	-	✓	✓	-	✗	-	-
Scheduling method	Static	-	Static	Static, Dynamic	-	Static	-	-
CUDA	✓	✗	✗	✗	✗	✗	✗	✗
Single/double precision	Both	-	-	-	-	-	-	-
Support multiple GPUs	✓	-	-	-	-	-	-	-
OpenACC	✗	✗	✓	✗	✗	✗	✗	✗
Asynchronous operation	-	-	✗	-	-	-	-	-
Atomic operation	-	-	✗	-	-	-	-	-
Min required C compiler	C99	C99	C99	C99	C89	-	C99	-
Fortran standard	Fortran 77	Fortran 77	Fortran 2003	Fortran 2003	Fortran 90	-	Fortran 90	Fortran 77

In total, there are four major program components in the toolkit, which are displayed as four blue rectangles in Fig. 3. They are *lexical analyzer*, *semantic analyzer*, *keyword scanner*, and *result generator*. We will introduce the four major components in details as follows.

- (i) *Lexical Analyzer*: The input of this component is an FQL query which is an array of characters. The *lexical analyzer* will parse the query into a list of tokens. Here, each token is a string with an assigned or predefined meaning.
- (ii) *Semantic Analyzer*: The objective of the *semantic analyzer* component is to find a feature's corresponding keywords from a sequence of tokens. The component translates a list of tokens into keywords. Keywords refer to a set of significant strings that can be used as an indicator of the software feature. For instance, if we find the string `#pragma omp` in the source code, we can say OpenMP is used.
- (iii) *Keywords Scanner*: The objective of the *keywords scanner* component is to find whether the desired keywords exist in the source code or not. This component searches for the keywords derived from the semantic analyzer, then prints out a list of boolean variables (illustrated as the Intermediate Search Result in Fig. 3) to indicate whether each keyword is found or not in the source code.
- (iv) *Result Generator*: The *result generator* component imports the intermediate results from the *keywords scanner*, and presents the results in an easy-to-understand way to users.

In summary, the *lexical analyzer* and *semantic analyzer* components generate a list of keywords from an FQL query. Then, this list is passed to the *keywords scanner* component, which searches the open source code of interest by using the keywords. Finally, the *result generator* component presents the *keywords scanner* results to users.

## 4 Exemplar Applications

For the demonstration purpose, we present the searching results of scientific computing software packages supported by several large-scale DOE programs, such as the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program ([www.doeleadership-computing.org](http://www.doeleadership-computing.org)), Exascale Computing Projects ([www.exascaleproject.org](http://www.exascaleproject.org)), Earth System Modeling ([climatemodeling.science.energy.gov](http://climatemodeling.science.energy.gov)), and Subsurface Biogeochemical Research ([doesbr.org](http://doesbr.org)). For the demonstration purpose, we use five applications in this paper:

1. QMCPACK: A quantum Monte Carlo package designed for the *ab initio* electronic structure calculations [17]. It includes the implementation of a number of numerous Quantum Monte Carlo (QMC) algorithms.
2. ParFlow A parallel watershed flow model used to simulate different kinds of hydrological processes [20].

3. E3SM: A model used to simulate the interaction between human and Earth systems [1].
4. SICM: A tool provides a simple unified interface to simplify the process of managing the complex memory hierarchies [24].
5. ExaAM (includes Truchas, Tusas, ExaMPM and MEUMAPPS) : ExaAM is a software environment to simulate the complex additive manufacturing process (AM) [19]. Since it is an integration of many software, we use the Truchas [27], Tusas [28], ExaMPM [15] and MEUMAPPS [8] as our test cases. As shown in Table 2, there are four columns for each of them.

Exemplar FQL results of these applications are listed in the Table 2. It is obviously that MPI and OpenMP are two of the most widely used HPC features.

## 5 Conclusions

In this paper, we design and develop a software toolkit that automatically collects the software features from scientific codes using a new language, called Feature Query Language (FQL). For specific user-defined questions, we translate and formulate them into FQL queries using the FQL syntax. Then, the toolkit parses and executes the FQL queries over source code to collect information about the software features, such as special hardware, software and architecture requirements. Although we emphasize collecting the HPC features in this study, the capability of the toolkit can be easily extended to other software engineering tasks, such as coding pattern, hardware dependency and portability, as long as these questions can be formulated as valid FQL sentences following the defined FQL syntax that combines command, keyword, and phrase. FQL can also be integrated into other code analysis tools. For instance, FQL is included in an integrated tool called XScan. As described in [34], XScan can be used to analyze the Open Source Community-based Scientific Code.

## References

1. Bader, D., et al.: Accelerated climate modeling for energy (ACME) project strategy and initial implementation plan (2014)
2. Baxter, I.D., Pidgeon, C., Mehlich, M.: DMS@: program transformations for practical scalable software evolution. In: Proceedings of the 26th International Conference on Software Engineering, pp. 625–634. IEEE Computer Society (2004)
3. Beyer, D., Lewerentz, C.: CrocoPat: efficient pattern analysis in object-oriented programs. In: 2003 11th IEEE International Workshop on Program Comprehension, pp. 294–295. IEEE (2003)
4. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Vittek, M.: ELAN: a logical framework based on computational systems. *Electron. Notes Theor. Comput. Sci.* **4**, 35–50 (1996)
5. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. *Softw. Pract. Exp.* **30**(7), 775–802 (2000)

6. Cohen, T., Gil, J.Y., Maman, I.: JTL: the Java tools language. In: ACM SIGPLAN Notices, vol. 41, pp. 89–108. ACM (2006)
7. Collard, M.L., Decker, M.J., Maletic, J.I.: srcML: an infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In: 2013 IEEE International Conference on Software Maintenance, pp. 516–519. IEEE (2013)
8. Cook, J., Finkel, H., Junghans, C., McCorquodale, P., Pavel, R., Richards, D.: Proxy app prospectus for ECP application development projects. Technical report, Lawrence Livermore National Lab (LLNL), Livermore, CA, United States (2017)
9. Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A.: Software engineering by source transformation—experience with TXL. In: Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, pp. 168–178. IEEE (2001)
10. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* **5**(1), 46–55 (1998)
11. Date, C.J., Darwen, H.: A Guide to the SQL Standard: A User’s Guide to the Standard Relational Language SQL. Addison-Wesley, Reading (1989)
12. de Moor, O., et al.: QL: object-oriented queries made easy. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 78–133. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-88643-3\\_3](https://doi.org/10.1007/978-3-540-88643-3_3)
13. De Roover, C., Noguera, C., Kellens, A., Jonckers, V.: The soul tool suite for querying programs in symbiosis with eclipse. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, pp. 71–80. ACM (2011)
14. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buer overows in C. In: ACM Sigplan Notices, vol. 38, pp. 155–167. ACM (2003)
15. ExaMPM (2017). <https://github.com/ECP-copa/ExaMPM>
16. Gobeille, R.: The FOSSology project. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, pp. 47–50. ACM (2008)
17. Kim, J., et al.: QMCPACK simulation suite (2014)
18. Klint, P., Van Der Storm, T., Vinju, J.: RASCAL: a domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, pp. 168–177. IEEE (2009)
19. Exascale Simulation for Additive Manufacturing (2017). <https://github.com/ExascaleAM>
20. Maxwell, R.M., et al.: ParFlow user’s manual. International Ground Water Modeling Center Report GWMI 1(2009), p. 129 (2009)
21. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv. (CSUR)* **37**(4), 316–344 (2005)
22. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in Linux device drivers. In: ACM SIGOPS Operating Systems Review, vol. 42, pp. 247–260. ACM (2008)
23. Rademaker, P.: Binary relational querying for structural source code analysis. University Utrecht, Netherlands (2008)
24. SICM (2018). <https://github.com/lanl/SICM>
25. oss-review-toolkit (2017). <https://github.com/heremaps/oss-review-toolkit>
26. scancode-toolkit (2016). <https://github.com/nexB/scancode-toolkit>
27. Truchas (2017). <https://github.com/truchas/truchas-release>

28. Tusas (2018). <https://github.com/chrisknewman/tusas>
29. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Not.* **35**(6), 26–36 (2000)
30. Verbaere, M., Hajiyev, E., De Moor, O.: Improve software quality with Semmler-Code: an eclipse plugin for semantic code search. In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, pp. 880–881. ACM (2007)
31. Visser, E.: Stratego: a language for program transformation based on rewriting strategies system description of stratego 0.5. In: Middeldorp, A. (ed.) *RTA 2001*. LNCS, vol. 2051, pp. 357–361. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45127-7\\_27](https://doi.org/10.1007/3-540-45127-7_27)
32. Wang, D., Zheng, W., Song, F.: Application software analytics toolkit for facilitating the understanding, componentization, and refactoring of large-scale scientific models. Technical report, Oak Ridge National Lab (ORNL), Oak Ridge, TN, United States (2018)
33. Wilde, N., Huitt, R., Huitt, S.: Dependency analysis tools: reusable components for software maintenance. In: *Proceedings. Conference on Software Maintenance*, pp. 126–131. IEEE (1989)
34. Zheng, W., Wang, D., Song, F.: XScan: an integrated tool for understanding open source community-based scientific code. In: Rodrigues, J.M.F., et al. (eds.) *ICCS 2019*. LNCS, vol. 11536, pp. 226–237. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-22734-0\\_17](https://doi.org/10.1007/978-3-030-22734-0_17)
35. Zitser, M.: *Securing software: an evaluation of static source code analyzers*. Ph.D. thesis, Massachusetts Institute of Technology (2003)