EvoMe: A Software Evolution Management Engine Based on Differential Factbase

Xiuheng Wu, Mengyang Li, Yi Li
Nanyang Technological University, Singapore
{xiuheng001, lime0040, yi_li}@ntu.edu.sg

***Abstract*—Managing large and fast-evolving software systems can be a challenging task. Numerous solutions have been developed to assist in this process, enhancing software quality and reducing development costs. These techniques—e.g., regression test selection and change impact analysis—are often built as standalone tools, unable to share or reuse information among them. In this paper, we introduce a software evolution management engine, EvoMe, to streamline and simplify the development of such tools, allowing them to be easily prototyped using an intuitive query language and quickly deployed for different types of projects. EvoMe is based on differential factbase, a uniform exchangeable representation of evolving software artifacts, and can be accessed directly through a Web interface. We demonstrate the usage and key features of EvoMe on real open-source software projects. The demonstration video can be found at: http://youtu.be/6mMgu6rfnjY.**

***Index Terms*—Software maintenance, software evolution, query languages, program facts**

I. Introduction

Managing a large software system is hard, and managing a fast-evolving one is extremely challenging. Software systems are usually developed and maintained in an incremental manner and various types of development artifacts accumulate in this process. These artifacts, including change histories and code of each version, can be a useful source for program understanding and analysis. Numerous tools and techniques have been developed to make use of those artifacts in order to help improve software quality and reduce development costs, such as regression test selection [1], change impact analysis [2], and semantic history slicing [3]. These are often built as standalone tools, each addressing a different yet related aspects of the evolving software, unable to share or reuse information among them. When trying out a new technique, one often has to start from scratch and re-implement common analysis tasks, such as classifying changes and identifying code dependencies.

To promote information sharing across multiple software evolution management tasks and streamline the development of the associated tools, we encode common knowledge about the evolving software artifacts as *differential facts* [4]. Differential facts are stored in a *differential factbase*, providing a uniform exchangeable representation of the reusable information and supports efficient querying and manipulation. For example, facts about how classes and methods are added or updated between each version can be stored and reused between regression test selection and change impact analysis [4]. In addition, representing software changes as facts makes cross-language analysis possible. The analyses implemented on top of differential facts, which are language-independent, can be applied directly to operate on projects written in different programming languages without modifications. With efficient cross-version analyses, differential facts enable fast prototyping and are especially beneficial in experimenting with new techniques.

Yet, querying in differential factbase involves writing low-level relational algebraic formula and Datalog-like logic inference rules, which is prohibitive for average users. To make the manipulation of differential facts more effortless, we introduce an SQL-like query language, allowing a large set of evolution management tasks to be specified with more intuitive syntax. EvoMe combines the expressive intermediate representation of differential facts with improved usability and efficiency provided by the newly added query engine. In the following example, we demonstrate how EvoMe can be applied in a real-world analysis task.

Example. Figure 1 illustrates a real-world usage scenario, where we implement a lightweight static regression test selection (RTS) algorithm. The goal of RTS is to select a subset of test cases needed to be re-executed after software changes, in order to reduce the time costs of regression testing [1]. The query leverage knowledge about classes (`Class`), methods (`Method`), source code changes (i.e., `Insert` and `Update`), the containment relation (`Containment`) between different program entities and the call graphs (`MethodAccess`). It finds methods whose direct or indirect callees changed between the two software versions, namely, `v1` and `v2`. Then it finds out which classes *contain* those methods and only keep test classes (`isTestClass()`). Since the resulted classes might include inner classes, we select their outer classes as the tests needed to be rerun. Later in section II, we will explain how this query works in more detail.

The key features of EvoMe are summarized as follows. Readers can also get more details about the tool at http://evome.facta.xyz.

- **SQL-like query language with cross-version support.** Compared with logic programming based queries, the language used by EvoMe is influenced by SQL and object oriented programming constructs, which is more user friendly.
- **Language-neutral fact abstraction.** Meanwhile, the query and fact abstraction are both language-independent, thus can be applied to projects written in different programming

```
from Class cl, Insert ins, Update upd,
Method m, Containment ct, MethodAccess ma
range upd @ vRange, ins @ vRange,
ma @ vNew, m @ vNew, cl @ vNew
where exists (cl, ins, upd, m, ct, ma that
  ma.getCaller() = m
  and (upd = ma.getCallee+()
    or ins = ma.getCallee+())
  and (cl = ct.getContainer()
    and ct.getContained+() = m))
  and cl.isTestClass())
select cl.getOuterClass() as selectedTest
define vNew ("v2")
define vRange ("v1".."v2")
```

Fig. 1. Regression test selection implemented using EvoMe query.

languages. Modifications are only necessary when language-specific features are involved in the analysis.

- **Efficient query execution thanks to facts reusing.** The previously extracted facts can be reused between iterative development cycles of software projects, on repeated runs of the same task, such as RTS. A significant portion of facts can also be reused among different tasks, thus making the query execution more efficient.
- **Quick prototyping for new techniques.** With EvoMe, it is much easier for developers to do quick prototyping in order to try out and compare different new techniques. As in the example, since information used by different variants of RTS algorithms have a lot in common, facts can be partly or fully shared. Executing modified queries is also faster than re-running an end-to-end implementation, thus reduces the efforts by developers.

II. Overview of EvoMe

In this section, we describe the architecture and usages of EvoMe. As shown in Fig. 2, users may interact with EvoMe through the front-end web interface (Web UI). Two inputs are taken from the user, including a Git repository URL and a query program. The back-end then sends the query program to the *query translator*, converting it to a Datalog program and also determining the set of facts needed for the query execution. If the facts needed for the task already exist in the *fact storage*, they are fed into the Datalog engine together with the translated Datalog program. Otherwise, the repository will be fetched from the specified URL, if not already cached in the EvoMe server. Then the *fact extractors* take the target repository as input and process the artifacts, including source code and change histories, to produce the differential facts. The newly generated facts are stored in the fact storage.

A Datalog program [5] contains (1) a set of inference rules automatically generated by the query translator, in the form of "$a, b, c \rightarrow d$", and (2) a set of differential facts which are used to infer whether the terms a, b, and c can be evaluated to true. The *Datalog engine* then outputs all instances of d satisfying all the rules. The outputs are then translated to human-readable form and sent back to the user interface for display.

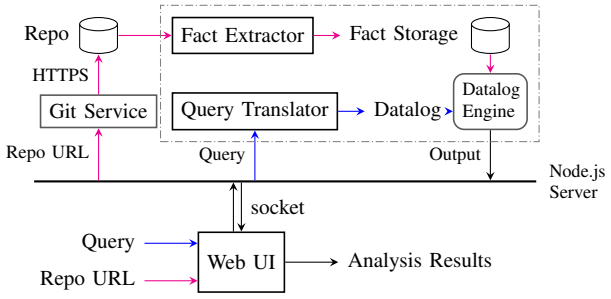

Fig. 2. Architecture of EvoMe.

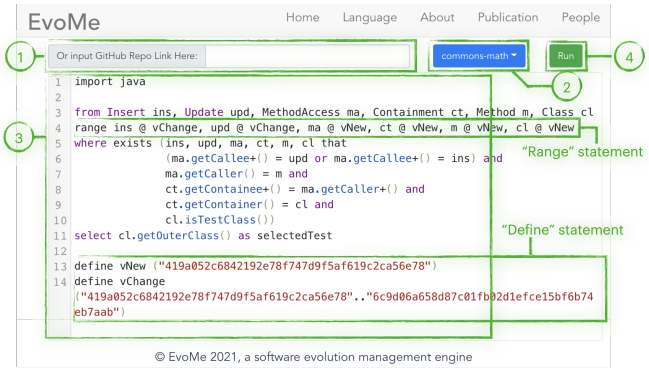

Fig. 3. Overview of the user interface.

A. User Interface

As is shown in Fig. 3, the user interface of EvoMe mainly consists of a ③ *code editor* for inputting query programs and a ① *form* for specifying repository location. Clicking on the ④ *Run* button will trigger the back-end process. Once the process finishes, the outputs are displayed on the web page as shown in Fig. 4. Besides the web interface, power users can also use the command line interface (CLI), where users specify the query file and repository location as arguments to the CLI program.

Selected Test	Version
org.apache.commons.math3.optim.nonlinear.scalar.noderiv.CMAESOptimizerTest	419a052c6842192e78f747d9f5af619c2ca56e78
org.apache.commons.math3.optim.nonlinear.scalar.noderiv.PowellOptimizerTest	419a052c6842192e78f747d9f5af619c2ca56e78

Fig. 4. Output displayed after the query execution.

B. Grammar of EvoMe Query

The SQL-like query language is the key to make good use of EvoMe. The core grammar of it is listed in Fig. 5 in the EBNF format. A query consists of a series of selection statements (*select_stmt*) and optional version selection statements (*ver_select*). Each selection statement can be divided into at most four parts, marked by the corresponding keywords: `from`, `range`, `where`, and `select`.

We now explain each clause using the example introduced in Fig. 1. A *from* clause contains one or more variable declarations, each of which consist of a *type* (*type_id*) and a *name* (*lower_id*)

separated by a space. In the example, six variables are declared with six different types (Lines 1 to 2). A *range* clause is optional, but is usually included for evolution management tasks. With it, a version variable can be associated for each variable declared in the *from* clause, in the form of "*var* @ *version*". In the example, we associate the six variables with either of the two version variables, namely, `vRange` and `vNew`. The *where* clause, like in SQL, determines the selection criteria. It contains a series of predicates connected by logic operators, i.e., "*and*" and "*or*". In Fig. 1, the first three predicates (Lines 6–8) express the relation between the method *m* and the changed program entities in *upd* and *ins*. By applying the "+" suffixed variant of the *getCallee()* method on *ma* (Lines 7–8), we compute a transitive closure on the *MethodAccess* relation, thus require that *m* must have some updated or newly-inserted program nodes on its call chain. The fourth and fifth (Lines 9–10) find classes *cl* which contain the method *m* through the *Containment* relation. The last one (*isTestClass()* on Line 11) restricts *cl* to test classes. The *select* clause determines what to include in the output and the *as* clause can be used to set the corresponding column headers. In our example, the class names are selected to be included in the final output.

Version selections mimic the syntax used by Git for its revision selections [6]. A *ver_select* assigns a version or a version range to a version variable (*lower_id*) so that it can be referenced in the *range* clause in *select_stmt*. In the example, two version variables are defined: `vNew` and `vRange`. While `vNew` is bind to a version, `vRange` is defined as a range `v1..v2`. In accordance with Git, it means `vRange` is a version reachable from `v2` and not reachable from `v1`. Besides the range representation, commonly used ancestry references such as `v2~2` (the grandparent of `v2`) and `v1^3` (the third parent of `v1`) are also supported. All of these representations can be used together, as in "`define v (v1, v2..v3, v4~4)`", which means `v` could be any version of them.

III. Backend Implementation

The backend of EvoMe consists of two components: the fact extractor and the query translator.

Fact Extractors. The fact extractors generate Datalog facts as tab-separated CSV files from versioned software artifacts. Different types of facts are produced by independent extractors, so that only the necessary ones need to invoked for specific projects to improve the efficiency. For analyses in need of currently unavailable facts, such as an unsupported programming language, EvoMe can also be extended easily by adding more extractors. Currently, there are extractors for intra-version facts encoding basic information about classes and methods, and call/reference relations between them. They are implemented by analyzing Java bytecode using the Apache BCEL [7] library. The inter-version fact extractor, which encodes AST node changes between versions, is based on ChangeDistiller [8]. A language-neutral extractor for Git histories is implemented using git2-rs [9] providing libgit2 bindings. All facts contain version information so that they can support versioned queries.

```
stmt = {?import_stmt? | "define" ver_select |
↪  select_stmt};
ver_select = lower_id, ?commit?, {",", ver_select}
  | lower_id, "(", mul_select, ")", {"," ver_select}
  | lower_id, "(", ?commit?, "..", ?commit?, ")",
  ↪  {"," ver_select};
mul_select = ?str_literal?, {",", mul_select};
select_stmt = ["from" from], ["range" range],
↪  ["where" where], "select", select;
select = expr, ["as" lower_id], {",", select};
from = type_id, lower_id, {",", from};
range = lower_id, "@", lower_id, {",", range};
where = quantified, {logic_op, where};
quantified = quantifier, "(", lower_id, {",",
↪  lower_id}, "that", formulas, ")";
quantifier = "exists" | "not exist" | "forall";
logic_op = "and" | "or";
cmp_op = ">=" | ">" | "=" | "!=" | "<" | "<=" | "=";
formula = "(", formula, ")" | "not", formula | call
  | formula, (logic_op|cmp_op), formula;
primary = lower_id | ?str_literal? | ?int_literal? |
↪  call;
call = lower_id, ".", lower_id, "(", {primary}, ")";
lower_id = ?lower?, { ?letter? | ?digit? | "_" };
type_id = ?upper?, { ?letter? | ?digit? | "_" };
```

Fig. 5. Grammar of EvoMe in EBNF.

Query Translator. The translator is implemented using the lexer generator Flex [10] and the parser generator GNU Bison [11]. It translates EvoMe queries into the Datalog dialect used by the Soufflé [5] engine. Classes in EvoMe becomes relations in Datalog, and methods usually correspond to terms in relations. For example, for a variable `cl` of type `Class`, `where cl.isTestClass()` produces `Class(cl, ...,1 ,...)` in Datalog where "1" indicate that `cl` is a testclass in facts. By converting getters and predicate methods into Datalog rules, the whole *where* clause of a selection statement becomes one or more inference rules. Similarly, version selections are translated so that concrete versions assigned to a version variable can be inferred automatically by Soufflé.

IV. Evaluation

To evaluate the applicability and performance of EvoMe on real-world projects, we measured the time and peak memory usage of four different queries on four Maven projects of varying sizes: Commons CLI [12], Commons IO [13], Commons Compress [14], Commons Lang [15].

The experiments were conducted on a desktop computer with Xeon(R) E5-1650 (v4) processor and 16GB RAM. Table I shows the time and memory costs for running the following three different queries in a history range of 90 commits.

- Find methods which are called by other methods in the selected version 2 but are not used in the selected version 1;
- Find methods taking parameter lists of different lengths between the two versions;
- Find methods which return *void* in the selected version 1 but return types other than *void* in the selected version 2.

The second column shows project sizes measured by thousand lines of Java code (kLoC) and the last column is the peak

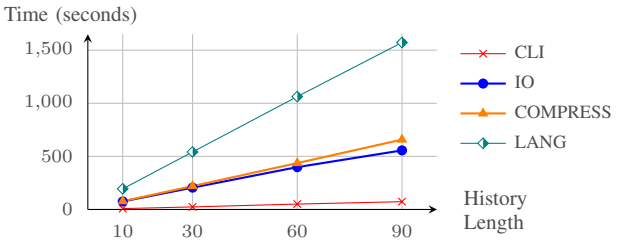

Fig. 6. Total time usage with the history lengths increase.

TABLE I
TIME AND MEMORY USAGE ON FOUR REAL-WORLD PROJECTS

Projects	kLoC	Time (seconds)				Mem (KB)
		Facts	Translate	Query	Total	
CLI	6.4	73	0.02	0.64	74	547
IO	38.7	553.16	0.02	3.18	556	2,256
COMPRESS	52.2	653.70	0.02	4.22	658	2,285
LANG	88.3	1,565.71	0.02	6.44	1,572	2,904

memory usage in kilobytes. Remaining columns list the time usage of the different stages in seconds—column *facts* for time used by the facts extractors, *translate* for the translators, *query* for the query executions (time used by Datalog engine), and *total* for the overall time taken. Note that *translate* and *query* are averaged over three queries. We also measured the time costs when the lengths of the change histories queried over increases from 10 to 30, 60, and 90 commits. The total time usage is plotted in Fig. 6.

It is clear that the time costs of facts extraction and query execution increases when the repository size grows bigger and the history length involved in the query increases. The evaluation also revealed that the majority of the time is spent during the facts extraction stage. Since facts can be stored and reused, time used for extraction is a one-time cost. In contrast, the query stage takes on average 3.6 seconds, providing a reasonable user experience.

Case Studies. We have also tested EVOME on a set of practical evolution management tasks, including regression test selection, change impact analysis, and semantic history slicing. We managed to replicate the existing techniques using EVOME queries and test them on a subset of projects used in [4]. We can obtain matching results at comparable speed. EVOME inherits the fact reusing capability of differential facts, extensively evaluated in the previous work [4], where a 44% time-cost reduction was observed. We also manually inspect the translated program, comparing them with hand-written Datalog rules and verifying the correctness. Finally, the same set of queries were also tested on several C/C++ projects, demonstrating the effectiveness of the language-neutral fact abstraction, although this feature has not been integrated into the Web interface yet.

Expressiveness. While we do not encounter expressiveness issues when implementing queries for evaluation and case studies, there might be expressive limitations, caused by either the implementation of translator, missing information in extracted facts or the expressiveness limitations of Datalog. Issues arisen from the first two causes can be solved by adding features to translators and fact extractors. For queries which can not be expressed in Datalog efficiently, we can also add some pre-processing procedures to handle them before the translation to Datalog. Those can be studied in future work.

V. RELATED WORK

There is a large body of work on analyzing and understanding software histories and using them for evolutionary management tasks. Li et al. defined the problem of *semantic history slicing* [3], [16], [17] and proposed an algorithm CSLICER [3] which computes a sub-history that preserves the desired semantic properties for Java projects hosted in Git repositories. Recently multiple tools for change impact analysis and regression test selection has been developed by both academia and industry [18]–[20]. Zhu et al. [21] proposed a framework for comparing and evaluating different RTS techniques on their correctness (false negatives) and efficiency (false positives). It is also possible to compare different RTS algorithms with the quick prototyping abilities of EVOME.

Converting software artifacts into "facts" or databases and using queries for searching or analyzing them have led to many works. Hajiyev et al. [22] implemented *CodeQuest*, a source code querying tool which uses a Datalog-based language and translates it to SQL for scalable execution. Nowadays, Datalog engines are efficient enough to be used as the query backend for lots of applications, as seen in Doop [23], a framework for points-to analysis using Soufflé. Other works include JQuery [24], which facilitates the navigation of source code using a logic programming based language and PQL [25] which is a query language designed for flexible static program analysis. CodeQL [26] gains popularity in recent years. It provides an SQL-like language and a standard library for security analysis. Some of our syntax were influenced heavily by CodeQL. None of these tools aim at evolution management and they do not provide facilities for handling versions. Wu et al. [4] proposed DiffBase, a fact-based approach for evolution management by considering program changes as first-class objects and implementing fact extractors for them. DiffBase uses both algebraic operations and Datalog as interfaces for manipulating facts. EVOME adopts a more intuitive language over DiffBase, making it more usable.

VI. CONCLUSION

In this paper, we described the architecture, user interface, and usage of EVOME. We also evaluated its applicability and key features on real-world software projects. The SQL-like version-aware query language brings improved usability and efficiency to differential fact-based evolution management, enabling fast and easy prototyping for related tools.

ACKNOWLEDGEMENT

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2019-T2-1-040).

REFERENCES

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification & Reliability*, vol. 22, no. 2, pp. 67–120, March 2012.

[2] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Proceedings of the 19th aAnual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2004, pp. 432–448.

[3] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, Feburary 2017.

[4] X. Wu, C. Zhu, and Y. Li, "Diffbase: A differential factbase for effective software evolution management," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 503–515.

[5] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.

[6] C. Scott and S. Ben, "Git Tools - Revision Selection," https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection.

[7] "Apache Commons Byte Code Engineering Library," https://commons.apache.org/bcel, 2015.

[8] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[9] "git2-rs," https://docs.rs/crate/git2/.

[10] Vern Paxson. Flex. [Online]. Available: https://github.com/westes/flex

[11] Free Software Foundation, "GNU Bison." [Online]. Available: https://www.gnu.org/software/bison/manual

[12] "Apache Commons CLI," https://commons.apache.org/cli.

[13] "Apache Commons IO," https://commons.apache.org/io.

[14] "Apache Commons Compress," https://commons.apache.org/compress.

[15] "Apache Commons Lang," https://commons.apache.org/lang.

[16] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise semantic history slicing through dynamic delta refinement," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2016, pp. 495–506.

[17] C. Zhu, Y. Li, J. Rubin, and M. Chechik, "GenSlice: Generalized semantic history slicing," in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution*, Sep. 2020, pp. 81–91.

[18] M. Parfianowicz, "Open Clover," https://openclover.org, 2017.

[19] M. Gligoric, L. Eloussi, and D. Marinov, "Ekstazi: Lightweight test selection," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 713–716.

[20] L. Zhang, "Hybrid regression test selection," in *2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 2018, pp. 199–209.

[21] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, 2019, pp. 430–441.

[22] E. Hajiyev, M. Verbaere, and O. De Moor, "Codequest: Scalable source code queries with Datalog," in *European Conference on Object-Oriented Programming*. Springer, 2006, pp. 2–27.

[23] Y. Smaragdakis and M. Bravenboer, "Using datalog for fast and easy program analysis," in *International Datalog 2.0 Workshop*. Springer, 2010, pp. 245–251.

[24] D. Janzen and K. De Volder, "Navigating and querying code without getting lost," in *Proceedings of the 2nd international conference on Aspect-oriented software development*, 2003, pp. 178–187.

[25] S. Jarzabek, "Design of flexible static program analyzers with pql," *IEEE Transactions on software engineering*, vol. 24, no. 3, pp. 197–215, 1998.

[26] "Codeql for research," https://securitylab.github.com/tools/codeql.