A History Querying Tool and its Application to Detect Multi-version Refactorings

Reinout Stevens*, Coen De Roover†, Carlos Noguera‡ and Viviane Jonckers‡
*Software Languages Lab
Vrije Universiteit Brussel, Brussels, Belgium
Email: resteven—cderoove—cnoguera—vejoncke@vub.ac.be

***Abstract*—Version Control Systems (VCS) have become indispensable in developing software. In order to provide support for change management, they track the history of software projects. Tool builders can exploit this latent historical information to provide insights in the evolution of the project. For example, the information needed to identify when and where a particular refactoring was applied is implicitly present in the VCS. However, tool support for eliciting this information is lacking. So far, no general-purpose history querying tool capable of answering a wide variety of questions about the evolution of software exists. Therefore, we generalize the idea of a program querying tool to a history querying tool. A program querying tool reifies the program's code into a knowledge base, from which it retrieves elements that exhibit characteristics specified through a user-provided program query. Our history querying tool, QWALKEKO, enables specifying the evolution of source code characteristics across multiple versions of Java projects versioned in Git. We apply QWALKEKO to the problem of detecting refactorings, specified as the code changes induced by each refactoring. These specifications stem from the literature, but are limited to changes between two successive versions. We demonstrate the expressiveness of our tool by generalizing the specifications such that refactorings can span multiple versions.**

***Keywords*-program comprehension tools; software repositories; refactoring;**

I. INTRODUCTION

While maintaining software, developers frequently need a wide series of questions answered [1]. Some of these questions (e.g.,what are a method's callers) can be answered using existing integrated development environments. Program querying tools have proven useful for answering more complex questions (e.g., whether all execution paths respect the protocol of an in-house developed API) [2]. Nevertheless, not all questions can be answered using the current state of a software project alone. Even simple questions such as "who introduced this class?" or "was this method renamed?" require the project's history.

One possible source of information about a project's history is the Version Control System (VCS) it is versioned in. The use of a VCS has become a software development best practice. A VCS enables developers to commit, share and undo changes, as well as implementing experimental features in a separate branch. Other sources of information can also be used. For instance, several Eclipse plugins track edit operations on a project.

Currently, tool support for querying these history sources is lacking. A history querying tool ought to offer a dedicated language for specifying how source code evolves. Existing tools do not feature expressive means for specifying the temporal characteristics of fine-grained changes (e.g., how statements change across versions).

In this paper, we present a history querying tool that features a dedicated language for specifying temporal characteristics of fine-grained code evolutions. To this end, QWALKEKO models a project's history as a version graph. History queries navigate the version graph using the QWAL regular path expression language, while code characteristics of individual versions along this path are described using the EKEKO logic program querying language.

QWALKEKO is inspired by an earlier history querying tool of ours called ABSINTHE [3]. Whereas QWALKEKO quantifies directly over versioned AST nodes (i.e., Java versioned in Git), ABSINTHE quantifies over a lightweight model that contains coarse-grained information about a program's structure (i.e., Smalltalk versioned in Monticello). Sub-method information was only available indirectly. As a result, ABSINTHE does not support specifying how individual instructions evolve.

II. RELATED WORK

Program querying tools identify source code that exhibits user-specified characteristics of interest. For instance, to check architectural constraints. Enabling users to specify these characteristics in logic-based languages has proven to result in expressive, yet descriptive specifications. This requires reifying code as data in a logic language. Examples of such logic-based program querying tools include CODEQUEST [4], PQL [5] and SOUL [2]. Our work generalizes this idea to multiple versions of code.

While most VCS provide limited version querying facilities (e.g.,to see who touched a file), they do not support querying the code within a version —let alone across versions. The EVOLIZER platform supports history analyses of versioned software through dedicated plugins. For instance, CHANGEDISTILLER [6] extracts code changes between successive versions through tree differencing. The *general-purpose* history querying tools that exist, (i.e., SCQL [7], V-Praxis [8] and iSPARQL [9]) do not feature a language

dedicated to specifying the *temporal* characteristics of *fine-grained* code evolutions *across multiple* versions.

III. General-Purpose History Querying Tools

We advocate the idea that a general-purpose history querying tool has to identify source code evolutions that exhibit characteristics of interest. From the user's perspective, three dimensions are key in the design of the tool's specification language: the formalism for specifying code characteristics, the formalism for specifying version characteristics, and the formalism for specifying temporal characteristics.

Source code characteristics concern the instructions (i.e., individual AST nodes), overall structure (e.g., subtyping relations), control flow (i.e., the order in which instructions can be executed) and data flow (i.e., the values operated upon by each instruction) within a single version of the program. Once specified in a history query, retrieving source code elements that exhibit these characteristics is left to the querying tool.

Version characteristics primarily concern the meta-data associated with each version of the software. Examples include the author, timestamp and commit message. These are typically provided by the history information source (e.g., a VCS). In addition, version characteristics concern the traceability relation between source code elements across versions. Using source code and version characteristics alone, history queries can be specified that retrieve the source code elements that correspond between two particular versions. Again, tracking entities across versions in order to maintain traceability is left to the querying tool.

Temporal characteristics concern the succession of versions along the project's history. Note that a version can have multiple successor versions. This is the case for versions that initiate a new branch in the project's history. Versions can also have multiple predecessors. This is the case for versions that resulted from the merger of different branches. As a result, there might be multiple sequences of successive versions between any two versions. In combination with the other characteristics, temporal characteristics enable specifying queries that retrieve sequences of versions along which particular source code or version characteristics hold. Such a sequence corresponds to one particular evolution in a program's history.

IV. QwalKeko

We now describe our general-purpose history querying tool QwalKeko. It extends the logic program querying tool Ekeko[1] with an implementation of regular path expressions Qwal[2]. QwalKeko currently supports querying Eclipse JDT projects versioned in Git.

[1]Ekeko is freely available at http://github.com/cderoove/damp.ekeko

[2]Qwal is freely available at http://github.com/ReinoutStevens/damp.qwal

A. Ekeko, a Program Query Language

Ekeko is a tool for answering program queries about JDT projects such as "where does my code implement a double dispatching idiom?". Its specification language is based on the core.logic port to Clojure of Kanren [10]. Source code characteristics are therefore specified as logic conditions. Queries are launched using the `ekeko*` special form which takes a vector of logic variables as its first argument, followed by a sequence of logic conditions:

```
1 (ekeko* [?s ?e]
2   (ast :ReturnStatement ?s) (has :expression ?s ?e))
```

Solutions to a query consist of bindings for its variables such that all conditions succeed. For the above query, the solutions consist of a return statement `?s` and an expression `?e` such that the latter is the former's expression part. Binary predicate `ast/2` quantifies over AST nodes of a particular type, while ternary predicate `has/3` quantifies over the values of their properties. In addition to such AST-related predicates, Ekeko provides predicates that quantify over the structure, control flow and data flow of a Java program.

B. Qwal, a Graph Query Language

Qwal enables querying graphs using regular path expressions [11]. Regular path expressions are an intuitive formalism for quantifying over the paths through a graph. They are akin to regular expressions, except that they consist of logic conditions to which regular expression operators have been applied. Rather than matching a sequence of characters in a string, they match paths through a graph along which their conditions holds.

In the context of QwalKeko, graphs represent a program's history. Nodes correspond to program versions, while edges connect consecutive versions. Applied to such a version graph, Qwal's regular path expressions match sequences of successive versions. The logic conditions within such an expression specify source code characteristics of a single version through Ekeko predicates. Version characteristics can be specified through predicates provided by QwalKeko itself (cf. Section IV-C).

Figure 1 depicts a query that finds classes that are subclassed in a later version. The first line configures the Qwal engine: it specifies the graph, start node and end node of the regular path expression, and introduces two logic variables `?class` and `?subclass`. Note that the end node is an unground logic variable, and will be bound to the end node of the path expression. Line 2 uses the Qwal primitive `q=>*`. This primitive skips an arbitrary number of versions (i.e., including none). Lines 3–4 specify that there must be a class in the current version, which will be bound to `?class`. Line 5 uses the `q=>+` primitive, which is similar to `q=>*`, except that it skips at least one version. Lines 6–8 specify that there must be a subclass `?subclass` of `?class` that is introduced in this version.

```
(qwal graph start ?end [?class ?subclass]
  q=>*
  (qin-current
    (ast :TypeDeclaration ?class))
  q=>+
  (qin-current
    (class-subclass ?class ?subclass)
    (introduced ?subclass)))
```

Figure 1. Finding a class for which a subclass is introduced later.

C. QwalKeko

QWALKEKO converts a Git repository into a version graph that can be queried using QWAL. A separate JDT project is created for each version. As opening all these projects at once will not scale, QWALKEKO opens and closes these projects on-demand as required by the QWAL query.

As stated earlier, history querying tools should maintain traceability links such that users do not have to track elements manually across versions. Our approach is two-pronged. A domain-specific variant of the classical unification procedure unifies code elements that stem from different program versions. It may succeed where the classical unification procedure fails. This is necessary as the same logic variable can substitute for elements from different program versions. For example, two methods from different versions unify if they have the same name, signature and return type. Domain-specific unification does not consider the context of a program element (e.g., the classes declaring the methods). Predicate `same/2`, in contrast, does. It unifies its second argument with a program element from the current version that corresponds to the one given as its first argument. For example, two methods are considered the same in case they unify, have the same defining class and package. Several of the built-in predicates are implemented using `same/2`, such as `introduced/1` and `removed/1`.

V. Detecting Multi-version Refactorings

Prete et al. [12] approach the problem of identifying which refactorings have been applied between two program versions by detecting the changes they induce, specified in the logic-based querying tool REF-FINDER. REF-FINDER populates a fact base from two Eclipse projects, each representing a single version of the software project. The changes induced by a refactoring are specified in a Prolog-like language.

To demonstrate that QWALKEKO is as least as expressive as REF-FINDER, we re-implement the REF-FINDER specification for methods that have been pulled up between two versions. Even though this is a fairly simple refactoring, we foresee no difficulties implementing more complex refactorings. Both REF-FINDER and QWALKEKO use the JDT to construct their representation of a single program version.

Figure 2 depicts the REF-FINDER specification for pulled up methods. A method is pulled up when the method is moved to another class, and this class is a superclass of

```
move_method(fShortName,tChildFullName,tParentFullName)
AND before_subtype(tParentFullName,tChildFullName)  !
pull_up_method(fShortName,tChildFullName,tParentFullName)
```

Figure 2. A method named `tChildFullname` is pulled up to `tParentFullName`. This rule originates from [12].

```
deleted_method(mFullName,mShortName,t1FullName)
AND added_method(newmName,mShortName,t2FullName)
AND similarbody(newmName,newmBody,mFullName,mBody)
AND not(equals(t1FullName,t2FullName))  !
move_method(mShortName,t1Fullname,t2FullName)
```

Figure 3. A method named `t1Fullname` is moved to `t2FullName`. This rule originates from [12].

the method's original defining class. These conditions can be translated into logic rules, depicted on lines 1–2. The first line specifies that the method has to be moved. The second line specifies the superclass requirement. The most interesting condition is `move_method/3`. Its implementation is depicted in figure 3. A method is moved in case the method was removed and a new method with a similar body was introduced in another class.

Figures 4 and 5 depict the corresponding QWALKEKO specifications. The most interesting condition is `removed`. This is implemented using `same/2` and negation.

```
(defn pulled-up [?method ?pulled]
  (fresh [?m-class ?p-class]
    (method-moved ?method ?pulled)
    (declaring-class ?method ?m-class)
    (declaring-class ?pulled ?p-class)
    (superclass ?m-class ?p-class)))
```

Figure 4. Query to detect whether a method `?method`, bound in a previous version, is pulled up to `?pulled` in the current version.

```
(defn method-moved [?moved ?to]
  (all
    (removed ?moved)
    (ast :MethodDeclaration ?to)
    (== ?moved ?to) ;;same signature
    (has-similar-body ?moved ?to)))
```

Figure 5. Query to detect whether a method `?moved`, bound in a previous version, is moved to `?to` in the current version.

The main difference between QWALKEKO and REF-FINDER is that REF-FINDER is limited to reason about two predefined versions. As such, the REF-FINDER specification language does not support temporal characteristics; predicates are already evaluated in the "correct" version. QWALKEKO, in contrast, enables reasoning about multiple versions. Goals are always evaluated with respect to an implicit version, specified by a path expression. In fact, so far the QWALKEKO specification does not yet specify in which versions its conditions have to hold. The specification depicted in Figure 6 does. Lines 2–3 bind `?method` to a method in the start version. Line 4 moves to a successive version. Lines 5–6 use our previously defined rule to find whether the current version contains a method `?pulled` that is the result of pulling up `?method`.

```
(qwal graph root ?end [?method ?pulled]
  (qin-current
    (ast :MethodDeclaration ?method))
  q=> ;;go to next version
  (qin-current
    (pulled-up ?method ?pulled)))
```

Figure 6. A method `?method` is pulled up to `?pulled` in a successive version.

Because of QWALKEKO's dedicated support for temporal characteristics, it is straightforward to generalize the specification such that the refactoring can happen across multiple versions, where intermediate changes do not qualify as a refactoring, but the accumulation of changes does. This only requires changing `q=>` to `q=>+` in the specification. As this operator skips an arbitrary number of versions, intermediate changes are skipped and the refactoring is detected once it has been performed completely. It does not matter in which order the steps of the refactoring occurred, only whether the end result qualifies as a refactoring. Detecting multi-version refactorings using REF-FINDER, in contrast, would require manually decomposing the refactoring into individual changes. This is because REF-FINDER can only assert that such a change occurred between two specific versions. As such, all pairs of successive versions have to be quantified over.

VI. Conclusions and Future Work

We have presented the general-purpose history querying tool QWALKEKO, which identifies source code evolutions that exhibit user-specified characteristics of interest. These evolutions are specified as sequences of successive versions, and the properties that have to hold in each of these versions. Its expressive, dedicated means for specifying the temporal characteristics of fine-grained code evolutions set QWALKEKO apart.

We have applied our tool to the problem of detecting refactorings, originally specified in REF-FINDER [12] by means of the changes induced between two versions. To demonstrate the expressiveness of our specification language, we have generalized these specifications to refactorings that happen across multiple versions, where individual changes are not a complete refactoring, but the accumulated changes are. This only required interchanging temporal operators. In future work, we will conduct an empirical study to assess how frequently such refactorings occur in large-scale software projects.

One of the major challenges is improving the performance of the tool. EKEKO uses computationally intensive static analyses to derive control flow and data flow relations. These are derived for each version. In future work, we intend to incrementalize these derivations —in particular the logic-based ones.

Finally, QWALKEKO can be applied to software engineering problems other than detecting refactorings. One interesting application domain is supporting integrators with the incorporation of a patch for one release of a software project into a variant release. This requires investigating how the entities modified by the patch have diverged.

Acknowledgments

Reinout Stevens is funded by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). Coen De Roover is funded by the Cha-Q SBO project sponsored by the same agency.

References

[1] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering*, 2010.

[2] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proc. of the 9th Int. Conf. on Principles and Practice of Programming in Java*, 2011.

[3] A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers, "Reasoning over the evolution of source code using quantified regular path expressions," in *Proc. of the 18th Working Conference on Reverse Engineering*, 2011.

[4] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable source code queries with Datalog," in *Proceedings of the 20th European Conf. on Object-Oriented Programming*, vol. 4067, 2006.

[5] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proc. of the 20th annual ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications*, 2005.

[6] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, no. 1, 2009.

[7] A. Hindle and D. M. Germán, "SCQL: a formal model and a query language for source control repositories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, 2005.

[8] A. Mougenot, X. Blanc, and M.-P. Gervais, "D-Praxis: A peer-to-peer collaborative model editing framework," in *Proc. of the 9th Int. Conf. on Distributed Applications and Interoperable Systems*, 2009.

[9] C. Kiefer, A. Bernstein, and J. Tappolet, "Mining software repositories with isparol and a software evolution ontology," in *Proc. of the 4th Int. Workshop on Mining Software Repositories*, 2007.

[10] D. P. Friedman, W. E. Byrd, and O. Kiselyov, *The Reasoned Schemer*. The MIT Press, 2005.

[11] O. De Moor, D. Lacey, and E. Van Wyk, "Universal regular path queries," *Higher Order Symbol. Comput.*, vol. 16, no. 1-2, Mar. 2003.

[12] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proc. of the 2010 IEEE Int. Conf. on Software Maintenance*, 2010.