Querying the History of Software Projects using QwalKeko

Reinout Stevens*, Coen De Roover†*
*Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
†Software Engineering Laboratory, Osaka University, Osaka, Japan
Email: resteven—cderoove@vub.ac.be

***Abstract*—We present the QwalKeko meta-programming library for Clojure that enables querying the history of versioned software projects in a declarative manner. Unique to this library is its support for regular path expressions within history queries. Regular path expressions are akin to regular expressions, except that they match a sequence of successive snapshots of a software project along which user-specified logic conditions must hold. Such logic conditions can concern the source code within a snapshot, versioning information associated with the snapshot, as well as patterns of source code changes with respect to other snapshots. We have successfully used the resulting multi-faceted queries to detect refactorings in project histories. In this paper, we discuss how applicative logic meta-programming enabled combining the heterogenous components of QwalKeko into a uniform whole. We focus on the applicative logic interface to a new implementation of a well-known change distilling algorithm. We use the problem of detecting and categorizing changes made to Selenium-based test scripts for illustration purposes.**

I. Introduction

Software repositories form a large source of information regarding the development of software projects. Researchers can use this information for the analysis of the maintainance and on how software projects are being maintained and evolve. So far, tool support to extract and query this information on a fine-grained level is lacking.

In this paper we present and demonstrate the capabilities of a history querying tool called QwalKeko. QwalKeko is an Eclipse plugin to query the history and evolution of software projects stored in git. A git repository can be viewed as a graph, in which nodes correspond to commits, and successive commits are connected. A branch results in multiple successors, while a merge results in multiple predecessors of a node. QwalKeko combines the graph query language Qwal[1] with the logic program querying language Ekeko [7] to reason over such a graph. Qwal navigates through the graph, while Ekeko is used to specify source code characteristics that have to hold in a specific version. QwalKeko has been applied successfully to detect refactorings [1] and classify changes made to Selenium scripts [2]. The source code, installation instructions[2], code[3] and video used in this paper can be found on Github.

[1]https://github.com/ReinoutStevens/damp.qwal
[2]http://github.com/ReinoutStevens/damp.qwalkeko
[3]http://github.com/ReinoutStevens/damp.qwalkeko/blob/master/src/qwalkeko/demo/icsme_selenium.clj

II. Ekeko

Ekeko [7] is a Clojure library for applicative logic meta-programming against an Eclipse workspace. It provides a library of predicates that can be used to query programs. These predicates reify the basic structural, control flow and data flow relations of the queried Eclipse projects, as well as higher-level relations that are derived from the basic ones.

Throughout this paper we only use predicates that reify structural relations computed from the Eclipse JDT. Binary predicate `(ast ?kind ?node)`, for instance, reifies the relation of all AST nodes of a particular type. Here, `?kind` is a Clojure keyword denoting the capitalized, unqualified name of `?node`'s class. Solutions to the query `(ekeko [?inv] (ast :MethodInvocation ?inv))` therefore comprise all method invocations in the source code. Throughout this paper we prefix logic variables with a `?`.

Ternary predicate `(has ?propertyname ?node ?value)` reifies the relation between an AST node and the value of one of its properties. Here, `?propertyname` is a Clojure keyword denoting the decapitalized name of the property's `org.eclipse.jdt.core.dom.PropertyDescriptor` (e.g., `:modifiers`). In general, `?value` is either another `ASTNode` or a wrapper for primitive values and collections. This wrapper ensures the relationality of the predicate.

III. Qwal

Qwal allows querying graphs using regular path expressions [8]. 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.

A Qwal query is launched using the function `(qwal graph begin ?end [ & vars ] & goals)`. It takes as arguments a graph object, a begin node, a logical variable that is unified with the end node of the expression, a vector of local variables available inside the query and an arbitrary amount of goals. The goals in a query either specify conditions that must hold in the current node of the query, or they modify the node against which conditions are checked by moving through the graph. For example, the goal `q=>` changes the current node to one of its successors. Table I provides an excerpt of the available goals. Users are not limited to

`q=>`	Moves the current version to one of its successors.
`q<=`	Moves the current version to one of its predecessors.
`(qcurrent [curr] & conditions)`	Conditions need to hold in the current version, which is bound to `curr`. Conditions are regular `core.logic` predicates.
`(q* & goals)`	Goals succeed an arbitrary, including zero, amount of times.
`(q=>* & goals)`	Similar to `q*`, except an implicit `q=>` is added after goals. If goals is empty this skips an arbitrary number of nodes.
`(q=>+ & goals)`	Similar to `q=>*`, except goals must succeed at least once. If goals is empty this skips an arbitrary, non-zero number of nodes.
`(q? & goals)`	Goals can either succeed or fail.

TABLE I. EXCERPT OF THE AVAILABLE GOALS IN QWAL

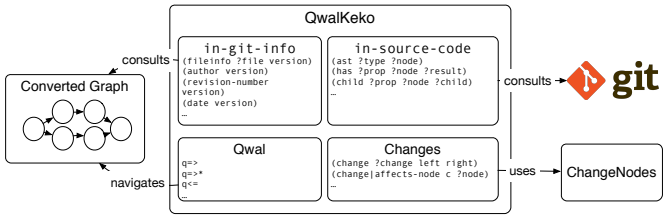

Fig. 1. Overview of QWALKEKO's architecture.

the goals provided by QWAL, but can easily define their own goals.

IV. QWALKEKO

QWALKEKO combines QWAL and EKEKO to reason over the history of a software project. Foremost, it converts a git repository into a graph of commits. This graph also contains information such as the author, timestamp, commitmessage and modified files of each commit. This graph and data is stored in a new Eclipse project, together with a copy of the repository. QWAL is used to navigate through this graph and specifies against which commit predicates are checked. Predicates can either use the metadata stored in the graph, or they can use source code information from a specific commit. For the latter, a new Eclipse project containing that particular revision is created, and predicates written in EKEKO will use that project as their fact base. QWALKEKO also features its own set of QWAL and EKEKO predicates that only make sense in the context of history querying. For example, QWALKEKO provides a set of predicates that reason over changes made to the source code.

A. Overview of the Architecture

Figure 1 depicts how the different components of QWALKEKO interact. QWALKEKO defines the following two QWAL goals: `(in-git-info [c] & conditions)` and `(in-source-code [c] & conditions)`. Both evaluate conditions in the current version `c`. Goal `in-git-info` only allows predicates that reason over the metadata stored in the graph. For example, the predicate `(fileinfo|edit ?info version)` unifies its first argument with a representation of a file that was modified. Goal `in-source-code` allows reasoning over source code information. To this end, it performs a checkout of the code to provide AST information.

The following query finds the compilation units (i.e., the root node of an AST) of every modified file in all the versions of the queried software project. We assume `graph` and `root` are variables bound to respectively the graph and the root version of the queried project.

```
1 (qwalkeko* [?info ?cu ?end]
2   (qwal graph root ?end []
3     (q=>*)
4     (in-source-code [curr]
5       (fileinfo|edit ?info curr)
6       (fileinfo|compilationunit ?info ?cu curr))))
```

On the first line we call `qwalkeko*`, which configures the logic engine and specifies which variables will be the result of the query. The second line configures the QWAL engine. Both `graph` and `root` are already bound, while `?end` will be bound to the end version. On the third line we skip an arbitrary number of versions. This pattern is functionally equivalent to mapping the rest of the query over all the versions in the graph. In the last three lines we specify that we are interested in a modified file, as denoted by `fileinfo|edit`, and its corresponding compilation unit.

B. ChangeNodes

QWALKEKO provides an implementation of a tree distilling algorithm based upon the work of Chawathe et. al [10] called CHANGENODES[4]. It takes as input two AST nodes and outputs a minimal edit script that, when applied, transforms the first AST into the second one. The edit script will contain the following operations:

Insert	A node is inserted in the AST
Delete	A node is removed from the AST
Move	A node is moved to a different location in the AST
Update	A node is updated/replaced with a different node

Chawathe's algorithm has also been used in CHANGEDISTILLER [4]. The main difference between CHANGENODES and CHANGEDISTILLER is that CHANGENODES works directly on top of the JDT nodes. CHANGENODES uses a language-aware representation, while CHANGEDISTILLER uses a language-agnostic representation. The heuristics used in CHANGEDISTILLER are also used in CHANGENODES.

QWALKEKO introduces a new predicate `(change ?change source target)`, which binds `?change` to a single change operation between the source AST and target AST. The predicate `(changes ?changes source target)` binds `?changes` to a collection containing all the changes made to both ASTs.

The following code demonstrates how one can use the `change` predicate to retrieve changes made to two Java classes:

```
1 (qwal graph root ?end [?left-cu ?right-cu ?change]
2   (in-source-code [curr]
3     (ast :CompilationUnit ?left-cu))
4   q=>
5   (in-source-code
6     (compilationunit|corresponding ?left-cu ?right-cu)
7     (change ?change ?left-cu ?right-cu)))
```

[4]https://github.com/ReinoutStevens/ChangeNodes

On line 3 it binds `?left-cu` to a compilation unit in the root version of the graph. It moves to one of the successors of that version on line 4. On lines 5–6 we retrieve the corresponding compilation unit using `compilationunit|corresponding`, which looks for a compilation unit in the same package that defines the same type. Finally we compute the changes between these two compilation units using the predicate `change`.

V. QwalKeko explained by means of an Example

In the following section we will build the queries needed to categorize changes made to Selenium scripts. These queries have been used to perform an earlier study [2]. First, we need to identify which files are Selenium scripts. Next, we need to compute changes made to these files. Finally, we detect whether these changes belong to a predetermined category.

A. Identifying Selenium Scripts

First of all we identify which files of our project are Selenium scripts. To this end, we write a QwalKeko query that loops over all the revisions of the queried software project. For each revision, it inspects the newly added files and identifies whether it is a Selenium script. The latter is done by looking whether the file imports a package which name contains "selenium". Albeit a simple heuristic we have not found any incorrectly identified files.

The following query returns all the Selenium scripts, the compilation unit and the revision of the script of the queried software project. First, it skips an arbitrary (including zero) amount of revisions by using the `q=>*` operator on line 3. Next, we bind `?info` to a newly added file. This predicate is evaluated without checking out the code, and thus if no new files are added no unnecessary operations are performed. Finally, in the last three lines we retrieve the corresponding compilation unit of the added file and verify whether it is a Selenium script.

```
1 (qwalkeko* [?info ?cu ?end]
2   (qwal graph root ?end [ ]
3     (q=>*)
4     (in-git-info [curr]
5       (fileinfo|add ?info curr))
6     (in-source-code [curr]
7       (fileinfo|compilationunit ?info ?cu curr)
8       (compilationunit|selenium ?cu))))
```

Identifying whether a compilation unit is a Selenium script is done purely using Ekeko. On line 2 we define three new logic variables using `fresh`. Line 3 verifies whether `?cu` unifies with a compilation unit. Line 4 unifies `?imp` with one of the import statements on that compilation unit. Lines 5–6 retrieve the name of the imported package. The last line verifies whether the name contains the string ".selenium".

```
1 (defn compilationunit|selenium [?cu]
2   (fresh [?imp ?impname ?str]
3     (ast :CompilationUnit ?cu)
4     (child :imports ?cu ?imp)
5     (has :name ?imp ?impname)
6     (name|qualified-string ?impname ?str)
7     (string-contains ?str ".selenium")))
```

The results of this query are written to a database so that they do not need to be recomputed by other predicates. We introduce a new predicate `(fileinfo|selenium file version)` which verifies whether a file corresponds to a Selenium script. This predicate consults this database and will be used in further examples.

B. Classification of Changes

Having successfully identified the Selenium scripts in the queried software project, we now need to compute and categorize changes made to these files. To this end, whenever a change was made to a Selenium script we will use ChangeNodes to compute the differences between that revision of the file and its predecessor.

The following query computes and classifies changes made to Selenium scripts. It skips a arbitrary, non-zero number of versions using the `q=>+` predicate on line 4. Next, it binds `?info` to a modified file in the current revision. It ensures this file is a Selenium script. If no Selenium scripts are modified in this revision no code is checked out. Next, it binds `?right-cu` to the compilation unit of that Selenium script. On line 10 of the query it moves to one of the predecessors of the current revision using the `q<=` predicate. On line 12 it retrieves the corresponding compilation unit of `?right-cu` and binds it to `?left-cu`. Note that the way the query is written, `?left-cu` is bound to the original script, while `?right-cu` contains the more recent revision of the script. On the last 2 lines it computes the changes made these files. The predicate `classify-change` is responsible for classifying a single change.

```
1 (qwalkeko*
2   [?left-cu ?right-cu ?info ?end ?change ?category]
3   (qwal graph version ?end []
4     (q=>+)
5     (in-git-info [curr]
6       (fileinfo|edit ?info curr)
7       (fileinfo|selenium ?info curr))
8     (in-source-code [curr]
9       (fileinfo|compilationunit ?info ?right-cu curr))
10    q<=
11    (in-source-code [curr]
12      (compilationunit|corresponding ?right-cu ?left-cu)
13      (change ?change ?left-cu ?right-cu)
14      (classify-change ?change ?category))))
```

The following code verifies that a change can be classified as a modification to a locator. To this end it uses `(change|affects-node change ?node)`, which unifies `?node` to any parent node of both the original and the target AST node of the change. This predicate works purely on an AST level, and does not use other sources of information. It then verifies whether one of the affected nodes resides inside either a method invocation with the name "By" or an annotation named "FindBy". These predicates are once again written using Ekeko.

```
1 (defn change|affects-findBy [change ?find-by]
2   (all
3     (change|affects-node change ?find-by)
4     (conde
5       [(methodinvocation|by ?find-by)]
6       [(annotation|findBy ?find-by)])))

7 (defn methodinvocation|by [?x]
8   (fresh [?name]
9     (ast :MethodInvocation ?x)
10    (child :expression ?x ?name)
11    (name|simple-string ?name "By")))

12 (defn annotation|findBy [?x]
13   (fresh [?name]
14     (ast :NormalAnnotation ?x)
15     (has :typeName ?x ?name)
16     (name|simple-string ?name "FindBy")))
```

Predicates for the remaining categories are analogous.

VI. Related Work

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 [4] extracts code changes between successive versions through tree differencing. The general-purpose history querying tools that exist, (i.e., SCQL [5] and V-Praxis [6]) do not feature a language dedicated to specifying the temporal characteristics of fine-grained code evolutions across multiple versions.

Boa [3] is a query language for large-scale software repositories. It allows answering high-level questions over a plethora of software repositories. Examples of questions are "What are the five most used licenses?", "How many Java projects using SVN were active in 2011?" and "What are the projects that support multiple operating systems?". Queries for these questions do not range over source code but rather high-level information of the queried software projects. Boa also features an AST visitor that can be used in queries. This has been used to detect whether and how new language features are adopted in programming projects. Boa users are limited to querying the projects that are made available on their website, and thus no arbitrary projects can be queried. QWALKEKO allows querying any Java project stored in git. It also provides a richer query language to retrieve information from the source code of the queried projects.

VII. Discussion

QWALKEKO provides an easy way to query large-scale software projects in an acceptable time. The REPL provides a nice way to gradually build queries until the expected results are returned. Applying these queries to different data sets requires no modification.

QWALKEKO only provides AST nodes as its source of information. EKEKO provides other predicates that reason over bindings (i.e.,, type and scoping information) provided by Eclipse. In order to generate these bindings a project must compile successfully in Eclipse. This is not always feasible, as libraries may be missing, or some revisions only work with a specific version of a library. We have incorporated the Partial Program Analysis [9] tool in QWALKEKO, but the performance is too slow for large-scale code querying. Thus, queries are either limited to AST information, or the user needs to ensure the project compiles properly.

Some recurring patterns have an unexpected poor performance, mainly due to the logic reasoning engine provided by Clojure. A concrete example is the use of the `q=>*` predicate at the beginning of a query. For large projects this pattern must be rewritten so that it uses Clojure's `map` over all the versions in the graph. An automated rewrite of these patterns would increase the performance, while keeping the readability of the query language.

VIII. Summary

We have presented QWALKEKO as a tool to query over the history of software projects stored in Git. To this end, it combines the programming query language EKEKO with the regular path expressions language QWAL. It provides its own set of predicates that only make sense in the context of history querying. The most important one is `(change ?change left-cu right-cu)`, which reifies change operations made to ASTs. It uses CHANGENODES, which implements a tree differencing algorithmn on top of the JDT AST nodes. We have demonstrated the use of QWALKEKO by writing queries that identify SELENIUM scripts at categorize changes made to these scripts.

Acknowledgments

This work has been supported, in part, by the *Cha-Q* SBO project of the Flemish agency for Innovation by Science and Technology (IWT), by a PhD scholarship of the same agency, by the Japan Society for the Promotion of Science, Kakenhi Kiban (S), No.25220003, and by the Osaka University Program for Promoting International Joint Research.

References

[1] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013.

[2] L. Christophe, R. Stevens, and C. De Roover, "Prevalence and maintenance of automated functional tests for web applications," in *Proceedings of the 30th Internation Conference on Software Maintenance and Evolution*, 2014, to be published.

[3] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[5] A. Hindle and D. M. German, "SCQL: A formal model and a query language for source control repositories," in *Proceedings of the 2005 Working Conference on Mining Software Repositories (MSR05)*, 2005, pp. 100–105.

[6] A. Mougenot, X. Blanc, and M.-P. Gervais, "D-Praxis: A peer-to-peer collaborative model editing framework," in *Proceedings of the 9th International Conference on Distributed Applications and Interoperable Systems (DAIS09)*, 2009, pp. 16–29.

[7] C. De Roover and R. Stevens, "Building development tools interactively using the ekeko meta-programming library," in *Proceedings of the CSMR-WCRE Software Evolution Week (CSMR-WCRE14)*, 2014.

[8] O. de Moor, D. Lacey, and E. V. Wyk, "Universal regular path queries," *Higher-Order and Symbolic Computation*, pp. 15–35, 2002.

[9] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *In Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA08)*, 2008.

[10] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD96)*, 1996, pp. 493–504.