A Declarative Foundation for Comprehensive History Querying

Reinout Stevens
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
Email: resteven@vub.ac.be

***Abstract*—Researchers in the field of Mining Software Repositories perform studies about the evolution of software projects. To this end, they use the version control system storing the changes made to a single software project. Such studies are concerned with the source code characteristics in one particular revision, the commit data for that revision, how the code evolves over time and what concrete, fine-grained changes were applied to the source code between two revisions. Although tools exist to analyse an individual concern, scripts and manual work is required to combine these tools to perform a single experiment. We present a general-purpose history querying tool named QWALKEKO that enables expressing these concerns in a single uniform language, and having them detected in a git repository. We have validated our work by means of replication studies as well as through MSR studies of our own.**

I. INTRODUCTION

The use of a Version Control System (VCS) is an industry best practice to develop and maintain software projects. A VCS allows developers to commit changes, share those changes with other developers and undo changes. As a side-effect, a VCS contains the history and evolution of the stored software project up until the granularity of a single commit. VCS are used by the Mining Software Repositories (MSR) community to perform studies about the evolution of the source code of software projects.

For such a study a researcher needs to be able to specify the following characteristics:

Source Code Characteristics: Source code characteristics are concerned with the detection of source code elements that exhibit certain syntactical properties, as well as data- and control flow properties.

Revision Characteristics: Revision characteristics concern meta-information about a revision, such as the author, timestamp and commit message of a revision.

Temporal Characteristics: Temporal characteristics concern in which revision source code elements need to be retrieved, and how these elements evolve over time. Examples are ensuring that a class remains present throughout the evolution of the software project, skipping revisions in which a method is not present or finding consecutive revisions that have the same author.

Change Characteristics: Change characteristics concern fine-grained changes that were applied to the source code between two revisions. Examples are deleting a source element or moving a source element to a different location.

To the best of our knowledge, no tool exists that enables specifying these characteristics in a uniform language.

II. STATE OF AFFAIRS

Currently, a researcher writes a script to help him retrieve the required data from the correct revisions. To this end, he is helped by a Program Query Language (PQL) to specify the sought after source code characteristics.

This approach allows for the following:

Source Code Characteristics: Source code characteristics are expressed using a PQL. A PQL enables a researcher to express the characteristics a snapshot of a program needs to exhibit.

Revision Characteristics: A VCS stores the meta-information of every commit. Researchers can use the VCS directly to access this information. Alternatively, libraries can be used that enables accessing this information from an ordinary programming language.

Temporal Characteristics: A PQL is suited to describe characteristics of a single revision of a program. The researcher needs to combine both the VCS and the PQL so that the correct source code elements are retrieved from the correct revision. For example by checking out a particular revision in a separate directory and run tool on that directory. Problems occur when source code elements retrieved in one revision have to be used in other revisions as well. To this end, information computed in a previous revision that has to be used in a later revision needs to be stored so that it can be accessed later.

Change Characteristics: Change characteristics can be computed using existing tools such as ChangeDistiller [8]. These tools take as input two Abstract Syntax Trees (AST) and output a sequence of operations that, when applied, transform the first AST into the second AST. The researcher has to ensure the original and updated ASTs are given to these tools. These tools do not enable reasoning over the output operations. Instead, it is up to the researcher to reason over these operations and interpret them.

While solutions exist to specify each characteristic individually, no solution exists that enables specifying combinations of these characteristics. As a result, it is up to the researcher to write scripts that act as the glue between the different

solutions. Such endeavours are error-prone and cumbersome to the researcher.

Besides the aforementioned issues, there is the issue of reproducing existing studies. The ad-hoc manner of these scripts makes reproducing the experiment on different data sets hard. The scripts are written to work with a specific data set on the researcher's hardware, and are not easily ported to a different setup.

To address these issues we propose an applicative logic foundation for querying the history and evolution of software project. We have instantiated this approach in the tool named QWALKEKO, which combines the declarative programming language EKEKO with the temporal specification language QWAL, and which also provides its own set of predicates that enable querying revision information and changes made to two ASTs.

III. RELATED WORK

Our work lies at the intersection of program query languages, graph querying and program changes.

Program querying tools identify source code that exhibits user-specified characteristics of interest. For instance, they can be used to check architectural constraints. Enabling users to specify these characteristics in logic-based languages has proven to result in expressive, yet descriptive specifications. To this end, code is reified as data that is queried using logic predicates. Examples of such logic-based program querying tools include CODEQUEST [9], PQL [10], SOUL [11] and EKEKO [4].

Most VCS provide limited facilities to query files (e.g.,see who touched a file, what lines were modified), they are limited to query on a line level and have no notion of the programming language of the stored project. The EVOLIZER platform supports history analyses of versioned software through dedicated plugins. For instance, CHANGEDISTILLER [12] extracts code changes between successive versions through tree differencing. The general-purpose history querying tools that exist, (i.e., SCQL [13] and V-Praxis [14] do not feature a language dedicated to specifying the temporal characteristics of fine-grained code evolutions across multiple versions.

More advanced tools exist, namely Boa [18] and HARMONY [19]. Boa is a query language for large-scale software repositories. It enables 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.

HARMONY provides an extensible framework to perform MSR studies. It supports importing different VCS, provides several predefined analyses and enables the definition of custom analyses on top of the HARMONY model. It has support for parallelism. However, it is lacking a dedicated declarative query language to specify source characteristics. Thus, everything needs to be written in Java, making advanced studies very complex.

In earlier work we have proposed a history querying tool called ABSINTHE [5] [2]. ABSINTHE is an extension of SOUL [11] allowing the querying Monticello repositories using quantified regular path expressions. ABSINTHE features an incremental model that efficiently represent classes and their methods throughout different revisions. The main differences from QWALKEKO is that QWALKEKO directly reasons over the source code.

IV. APPROACH

We propose an applicative logic foundation for querying the evolution of software projects. Such a foundation applies the approach from a PQL to history querying, which identifies source code that exhibit user-specified throughout the history of a software project. The declarative nature of PQLs have proven themselves successfully in querying software project. Thus, we propose a declarative approach for history querying as well.

This foundation is instantiated in the history querying tool QWALKEKO. It combines the the graph query language QWAL with the logic program query language EKEKO, and extends the latter with several predicates that only make sense in the context of history programming. Foremost, it converts a git repository into a graph of commits. This graph also contains information such as the author, timestamp, commit message and modified files of each commit. It uses EKEKO to query revisions in this graph. It uses QWAL to navigate through this graph, and to specify in what revision predicates need to be evaluated. Finally, it features an implementation of a tree differencer called CHANGENODES used to compute and reason over source code changes.

A. Expressing Source Code Characteristics using EKEKO

EKEKO [4] 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.

EKEKO provides 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. 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.

B. Expressing Revision Characteristics using QWALKEKO

QWALKEKO extends EKEKO with a set of predicates that reify meta-information present in the queried VCS. For example, unary predicate `(author ?author)` unifies `?author` with the author of the current revision. This information is added to the fact base EKEKO uses to query a single program revision. Thus, these predicates are indistinguishable from other EKEKO predicates and are used in the same manner.

C. Expressing Temporal Characteristics using QWAL

QWAL enables querying graphs using regular path expressions [15]. 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. Users are not limited to the goals provided by QWAL, but can easily define their own goals.

D. Expressing Change Characteristics using CHANGENODES

QWALKEKO provides our own implementation of a tree distilling algorithm based upon the work of Chawathe et. al [16] called CHANGENODES. 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 replaced with a different node

Chawathe's algorithm has also been used in CHANGEDISTILLER [8]. 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. QWALKEKO also provides predicates to retrieve the AST of a node retrieved in a different version. For example, the binary predicate `compilationunit|corresponding` retrieves the corresponding compilation unit of a given compilation unit in the current revision.

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)))
```

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`.

A current problem we are still facing is working with different change sequences that result in the same modifications made to source code. For example, a method rename can be performed by updating its name, as well as by deleting the original method and inserting a new method with the same body but a different name. A more complex example is detecting a recurring change pattern, but in which each pattern instance has a different list of operations. We are currently working on more advanced predicates that allow reasoning over multiple changes more easily, in which we combine multiple changes into high-level changes that better express the modifications that occurred. We envision that this library will provide the foundation for a change calculus.

V. VALIDATION

As discussed earlier, QWALKEKO consists out of several components. We have validated each individual component, as well as the combination of components, by means of replication studies as well as performing novel MSR studies.

A. Validation of QWAL

QWAL is used as the graph query language to navigate through the graph of revisions. We have performed a study in which we used Linear Temporal Logic (LTL), Computation Tree Logic (CTL) and regular path expressions to write history queries [1] that answer history questions developers frequently ask [17].

We were able to write queries that answered each question in all of the selected expression formalisms. Of those three formalisms, regular path expressions is the most intuitive one, mainly because developers are already familiar with regular expressions.

B. Validation of QWALKEKO

QWALKEKO has been used in several studies. We have used it to detect refactorings across multiple versions [3]. In this study we reimplemented some refactorings specified by Prete et al. in QWALKEKO. By using our graph query language we could detect these refactorings over multiple revisions.

In a different study we used QWALKEKO and CHANGENODES to identify and classify which parts of SELENIUM tests were most prone to change [6] [7]. This study combined the different components of QWALKEKO and has shown it can be used to perform large-scale MSR studies. For a corpus of 8 large open-source projects we needed to identify the different SELENIUM files in every revision, compute changes made to these files and classify these changes.

We plan on performing a more advanced study in which we investigate how code clone instances are removed using refactorings, and whether clone instances are removed in the same manner, or which additional steps are performed. This study forces us to reason over multiple changes at once, while the SELENIUM classification study only reasoned over individual changes.

VI. CONTRIBUTIONS

Our contributions are two-fold. We have created a unified approach with a declarative specification language, instantiated in a history querying tool that enables its users to write queries that express source code characteristics, revision characteristics, temporal characteristics and change characteristics in a single uniform language. We have proven that our chosen expression formalisms for each characteristic is suitable by means of replication studies and novel studies. We have also proven that the combination of these formalisms can be used successfully to perform MSR studies. Each of these studies also have their own scientific value, regardless of the used tool.

In earlier work we have proven our approach to be scalable for history queries up to the level of methods by introducing an incremental model that represents the classes present throughout the history of the queried project [5].

VII. TIMELINE

We are currently lacking a change calculus enabling us to combine low-level changes into higher-level ones in order to eliminate the differences in which an AST can be transformed into another AST. We plan on applying this calculus in the detection of code clone instances that are removed via refactorings, and what additional steps are performed besides the refactoring on each clone instance. We plan to finish this work at the beginning of February 2015.

We plan on starting to write our PhD after this work, and aim to finish our PhD at the end of the year 2015.

REFERENCES

My Publications

[1] R. Stevens, "Source code archeology using logic program queries across version repositories," Master's thesis, Vrije Universiteit Brussel, 2011.

[2] 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 *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE11)*, 2011, pp. 389–393.

[3] 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.

[4] 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.

[5] R. Stevens, C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "A logic foundation for a general-purpose history querying tool," *Elsevier Journal on Science of Computer Programming*, 2014.

[6] L. Christophe, R. Stevens, and C. De Roover, "Prevalence and maintenance of automated functional tests for web applications," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME14)*, 2014.

[7] R. Stevens and C. De Roover, "Querying the history of software projects using qwalkeko," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSMe14), Tool Demo Track*, 2014.

Other References

[8] 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.

[9] E. Hajiyev, M. Verbaere, and O. D. Moor, "Codequest: Scalable source code queries with datalog," in *Proceedings of the 20th European conference on Object-Oriented Programming (ECOOP06)*, 2006.

[10] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: a program query language," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*, 2005.

[11] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers, "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 (PPPJ11)*, 2011, pp. 71–80.

[12] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[13] A. Hindle and D. M. German, "SCQL: A formal model and a query language for source control repositories," in *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR05)*, 2005.

[14] 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.

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

[16] 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.

[17] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE10)*, 2010, pp. 175–184.

[18] 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 35th International Conference on Software Engineering (ICSE13)*, 2013.

[19] J.-R. Falleri, C. Teyton, M. Foucault, M. Palyart, F. Morandat, and X. Blanc, "The harmony platform," *CoRR*, vol. abs/1309.0456, 2013.