Reasoning over the Evolution of Source Code using Quantified Regular Path Expressions

Andy Kellens, Coen De Roover, Carlos Noguera, Reinout Stevens, Viviane Jonckers
Software Languages Lab, Vrije Universiteit Brussel
Email: {akellens | cderoove | cnoguera | resteven | vejoncke}@vub.ac.be

***Abstract*—Version control systems (VCS) have become indispensable to develop software. Next to their immediate advantages, they also offer information about the evolution of software and its development process. Despite this wealth of information, it has only been leveraged by tools that are dedicated to a specific software engineering task such as predicting bugs or identifying hotspots. General-purpose tool support for reasoning about the information contained in a version control system is limited. In this paper, we introduce the logic-based program query language ABSINTHE. It supports querying versioned software systems using logic queries in which quantified regular path expressions are embedded. These expressions lend themselves to specifying the properties that each individual version in a sequence of successive software versions ought to exhibit.**

I. INTRODUCTION

Version control systems have become indispensable software engineering tools. They enable a developer to work on a private copy of a jointly developed project and provide support for merging her changes with the contributions of other team members later on. As these systems track the history of a project, they contain a wealth of information that can be leveraged by other software engineering tools.

This information can be of use to various stakeholders. Developers spend a significant amount of time understanding source code before altering it. They need to find answers to questions such as "*why were these changes introduced?*" or "*who modified this piece of code most?*" [1] that require information about the history of the source code. Managers might need to verify whether a development team obeyed all imposed process constraints. The questions they need answered include "*has the team correctly applied test-driven development*" and "*were all changes to the system after its beta release corrective only?*". Tool builders are therefore increasingly leveraging version control systems to provide answers to a particular stakeholder. However, existing tools are dedicated to specific software engineering tasks and as such, are ill-suited to answer custom queries. There is therefore a need for a general-purpose tool that answers questions about a system's history that are formulated by the stakeholders themselves.

In this paper, we present ABSINTHE as such a general-purpose tool for querying versioned software. It represents the history of a software system as a directed acyclic graph. The graph's nodes represent individual versions of the software, while its edges correspond to the successor relation between versions. ABSINTHE offers a logic-based language to query this history representation. Next to logic predicates for reasoning about a system's *state* in a particular version, this language offers quantified regular path expressions for reasoning about the *evolution* of the system throughout successive versions.

II. MOTIVATING EXAMPLE

To motivate the need for a general-purpose tool for querying a system's history, we discuss an application scenario that stems from the domain of agile software development. One of the practices that the agile process actively advocates is that of test-driven development. The idea is to write the unit tests for a method before implementing it. As a result, every method is covered by a unit test, and faults in the implementation can be spotted early on.

Consider a manager that needs to assess whether test-driven development was applied throughout a system's history by finding violations to the practice. To identify methods for which the unit tests were introduced later on, the manager needs the history query "*find me all methods in the system for which there was never a unit test added, or for which the unit test was added after the method was introduced*" to be answered. While the versions in a repository do not allow determining whether a unit test was implemented before its corresponding method, they allow identifying tests that were added to the system in a later version.

This particular history query is difficult to answer. Without adequate tool support, the manager has to retrieve all versions of the system from the repository. To link unit tests to their corresponding method, she has to analyze the source code of individual and possibly even across versions. Finally, she has to verify the temporal relations between the versions in which the unit test and the method were introduced respectively. Clearly, tool support to answer such questions is desirable.

III. OVERVIEW OF ABSINTHE

ABSINTHE is a tool for answering questions about the evolution of versioned software.

Our tool offers a logic-based language for querying the repository representation. Its syntax and semantics extend those of SOUL [2] with regular path expressions. Regular path expressions consist of logic conditions to which regular expression operators are applied. They enable specifying which characteristics the history of the system should exhibit along successive versions. Each condition within a regular path

expression is therefore evaluated implicitly against a different version of the software.

Conditions on a regular path expression can use any predicate from the existing SOUL libraries for reasoning about source code. For instance, CAVA [3] and LICOR [4] provide predicates for reasoning about Java and Smalltalk code respectively. ABSINTHE provides additional predicates of its own that reify the information within its repository representation. The following sections discuss each component of ABSINTHE in detail.

A. Repository Representation

Although ABSINTHE features a logic language for specifying history queries, it relies on an object-oriented representation of the history of a versioned software system. This history is modeled as a directed acyclic graph. The graph's nodes represent individual software versions, while edges represent the successor relation between two versions. Note that a node can have multiple outgoing edges. This is the case for versions that initiate a new branch in the software's history. Nodes can also have multiple incoming edges. This is the case for versions that resulted from the merger of different branches.

History-specific information (i.e., time stamp, log message, author, revision number) can be accessed for each version in the repository representation. In addition, each version contains a snapshot of the source code in that particular version. This snapshot contains:

- A coarse-grained representation of the software's structure. We store information about the structural entities (i.e., packages, classes, interfaces, fields and methods) declared in the source code and information about how they are related (i.e., inheritance, containment). Each structural entity has a reference to the version of the software in which it was defined. In addition, frequently accessed sub-method information is available for each method in this representation (e.g., the methods it calls, the classes it references).
- A reference to the complete source code of the version as stored in the version control system. This code can be quantified over using one of the existing predicate libraries for SOUL.

To minimize the memory footprint of the repository representation, our implementation uses the same object for an entity that remains unchanged across successive versions. To this end, we introduced an additional layer of indirection between version snapshots and the objects that implement their structural source code entities. Each structural entity is assigned a unique identifier. All entity-related information (e.g., binary relations such as containment and inheritance) within a version snapshot is stored in terms of these identifiers. They remain constant throughout the history of the entity. In turn, each version snapshot maintains a mapping from entity identifiers to implementation objects. This mapping is only updated for entities that changed since the previous version. As a result, version snapshots share implementation objects for entities that remain unchanged. This implementation strategy is inspired by the work of Laval et al. [5].

A number of importers can be used to populate ABSINTHE's repository representation from a version control system. Currently, we support importing Java programs from Subversion repositories and importing Smalltalk programs from either VisualWorks Store or Monticello repositories.

B. Reification of the Repository Representation

ABSINTHE provides a library of logic predicates for quantifying over its repository representation. These can be used within history queries. Table I depicts an excerpt from this predicate library.

Note that the syntax for a predicate in SOUL closely resembles the one of Smalltalk for a message that is sent to the first argument of the predicate. Logic variables start with a question mark. The Prolog equivalent of a condition `?a isAuthorOfVersion : ?v` is therefore `isAuthorOfVersion(A,V)`. Throughout the remainder of this paper, we refer to predicates using their name and arity (e.g., `isAuthorOfVersion/3`).

In Table I three categories of predicates can be discerned: (1) predicates that reify history-specific information about a version (e.g., `isVersion/1`), (2) predicates that reify the structural entities within a version snapshot (e.g., `isClass/1`) and their relations (e.g., `isMethodInClass:/2`), and (3) predicates that reify frequently used sub-method information (e.g., `methodReads:/2`).

Unary predicate `isVersion/1` belongs to the first category. It succeeds if its argument `?v` unifies with a version that is stored in the repository representation. As a result, conditions can use this predicate to verify whether `?v` is bound to a version as well as to bind `?v` to one of the versions in the representation. This kind of multi-directionality is supported by all predicates in the library.

The second category of predicates reifies information about the structural entities in a particular version snapshot. For example, binary predicate `isMethodInClass:/2` reifies the relation between methods and the classes in which they are declared. These predicates can only be evaluated relative to a version snapshot. To make this explicit in the predicate library, these predicates have been annotated (indicated after the colon) with an additional logic variable `?version` that specifies the version snapshot of which the structural entities are queried. Predicate `isParsetreeOf:/2` is of special interest. It serves as an interface between ABSINTHE and the existing predicate libraries of SOUL. The predicate succeeds if its first argument unifies with a structural entity (e.g., a method or class) and its second argument unifies with the corresponding AST of that entity. The latter is retrieved on-the-fly through the source code link that is associated with each version snapshot.

The third category of predicates reify frequently used sub-method information. For instance, predicate `methodReads:/2` succeeds if its first argument unifies with a structural entity that represents a method and its second argument unifies with one of the fields read from by this method.

Predicate	Description
Versions	
?v isVersion	Entity is a version
?v isOrigin	Is the version an origin
?v isTerminal	Is the version a terminal
?v isVersionAtDate: ?d	Find the version at a particular date
?d isCommitMessageOfVersion : ?v	Retrieve the time stamp of a version
?a isAuthorOfVersion : ?v	Retrieve the author of a version
Structural entities within version snapshots	
?c isClass : ?version	Entity is a class in a particular version
?c isClassWithName: ?n : ?version	Class in version has name
?i isInterface : ?version	Entity is an interface in a particular version
?i isInterfaceWithName: ?n : ?version	Interface in version has name
?m isMethod : ?version	Entity is a method
?m isMethodInClass: ?c : ?version	Method belongs to class
?m isMethodWithName: ?n inClass: ?c : ?version	Method with particular name in class
?p isPackageWithName: ?n : ?version	Package with name
?v isInstanceVariableWithName: ?n inClass: ?c : ?version	Entity is field with name in class
?c isSuperclassOf: ?sub : ?version	Class is a direct superclass of a subclass
?e isParseTreeOf: ?x : ?version	Retrieve the original AST node from the repository for an entity
?e wasChanged : ?version	Entity was altered in a particular version
Frequently used sub-method information	
?m methodReferencesClass: ?c : ?version	Method refers to a particular class
?m methodSendsMessage: ?msg : ?version	Method sends a particular message
?m methodReads: ?var : ?version	Method reads from a field
?m methodWrites: ?var : ?version	Method writes to a field

TABLE I
EXCERPT FROM OUR LIBRARY OF LOGIC PREDICATES.

C. Quantified Regular Path Expressions

ABSINTHE extends the syntax and semantics of SOUL, a Prolog-like language with specialized features for querying code, with quantified regular path expressions (path expressions for short) [6]. Path expressions are an intuitive formalism to quantify over the paths through a graph. As such, they are a natural fit to quantify over the information in ABSINTHE's repository representation.

Quantified regular path expressions 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 ABSINTHE, they match sequences of successive versions from the repository representation. Using the predicates from ABSINTHE's predicate library, each condition within a regular path expression quantifies over the structural entities in a different version snapshot.

Predicate `matches:start:end:/4` can be used to embed a quantified regular path expression in a history query:

```
?quantifier(?exp)
   matches: ?path
   start:?start end: ?end
```

The predicate succeeds if `?path` unifies with a list of successive versions (i.e., a path), between a version that unifies with `?start` and a version that unifies with `?end` (both inclusive), that matches the regular path expression bound to `?exp`. The `?start` and `?end` variables do not necessarily need to be bound: when they are unbound, our tool will backtrack over all possible combinations of `?start` and `?end` for which the regular path expression holds.

The actual regular path expression `?exp` consists of a number of comma-separated logic conditions to which the following operators can be applied:

- $(C)^{*<}$: Consume zero or more versions for which C holds. The operator is matched *greedily*: initially, it consumes as many versions on the path as possible; shorter parts of the path are considered upon backtracking.
- $(C)^{*>}$: Consume zero or more versions for which C holds. The operator is matched *lazily*: initially, it consumes zero versions on the path; longer parts of the path are considered upon backtracking.
- $(C)^{+<}$: Consume one or more versions for which C holds. This operator is matched *greedily*.
- $(C)^{+>}$: Consume one or more versions for which C holds. This operator is matched *lazily*.
- $(C)^{times:n}$: Consume n versions for which C holds.
- $not(C)$: Consume the current version if C does not hold in it.

In addition, meta-variable `?THIS_VERSION` always unifies with the current version against which a condition is evaluated. The resulting binding is local to this condition. The following meta-symbols can be used alongside the logic conditions in a path expression as well:

- $origin$: Only consumes the current version if is not the successor of any other version.
- $terminal$: Only consumes the current version if it does not have any successors.

Finally, the `?quantifier` variable determines whether a path expression is universally or existentially quantified. To illustrate these concepts, consider the graph of versions depicted in Figure 1. This graph consists of 9 versions with V_1 being the single origin, and versions V_5 and V_9 being terminals. When `?start` is bound to version V_1 and `?end`

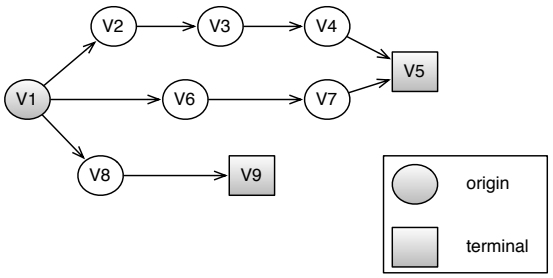

Fig. 1. Illustration of a graph of versions.

```
1 if
2 ?start isOrigin,
3 ?end isTerminal,
4 a(
5   not(? isClassWithName: Evaluator),
6   (?class isClassWithName: Evaluator)+<)
7     matches: ?path from: ?start to: ?end
```

Fig. 2. Example of a universal path expression.

is bound to V_5, an existentially quantified expression `e(?exp) matches: ?path start: ?start end: ?end` verifies whether there exists a path between V_1 and V_5 that matches `?exp`. In other words, either the path $(V_1, V_2, V_3, V_4, V_5)$ or the path (V_1, V_6, V_7, V_5) should match `?exp`. The universally quantified expression `a(?exp) matches: ?path start: ?start end: ?end`, in contrast, verifies whether `?exp` holds for all paths between `?start` and `?end`. Note that not all paths through the graph need to match `?exp`, but rather only the paths between `?start` and `?end`. Upon backtracking, variable `?path` receives a binding for each path that matches `?exp`.

Lines 4–7 of Figure 2 depict a universally quantified regular path expression that is embedded in a logic query. This query identifies pairs of origins `?start` and terminals `?end` for which there always exists a class `Evaluator` on all paths between them —except in the first version of the path. The first two conditions of the query (lines 2 and 3) bind variables `?start` and `?end` to an origin and a terminal respectively. The universally quantified regular path expression on lines 4–7 consists of two conditions. The first condition (line 5) verifies that no class named `Evaluator` exists in the first version on the path. The second condition (line 6), to which the `+<` operator has been applied, verifies that a class named `Evaluator` exists from the second version on the path until the end of the path.

Both conditions within the path expression use predicate `isClassWithName:/2`. As mentioned above, its annotated variable `?version` parameterizes the predicate with the version snapshot over which it is to quantify. Ordinary conditions that use this predicate therefore have to provide a binding for this variable. This is not necessary for conditions within a path expression. They are implicitly evaluated against a version by the path expression evaluator.

IV. Motivating Example Revisited

Above we introduced as a motivating example the situation

```
1 if
2 ?start isOrigin,
3 e(
4  (true)*>,
5  (and(?m isMethod,
6       not(? isTestFor:?m)))+<,
7  or(not(?m isMethod),
8     terminal,
9     ?test isTestFor:?m))
10  matches: ?path
11  start: ?start
12  end: ?end
```

Fig. 3. Verifying test-driven development.

in which a manager needs to find violations of her development team against the principle of test-driven development. In natural language, we formulated the history-related question that the manager needs answered as "all methods in the system for which there was never a unit test added, or for which this unit test was added after the method was introduced".

Figure 3 depicts the corresponding ABSINTHE query. It assumes that a predicate `isTestFor:/2` exists which verifies whether a unit-test tests a particular method.

The query consists of a single existential regular path expression that quantifies over the history of the system to identify a `?path` starting from an origin `?start` (line 2) that contains methods for which there either never was a unit test provided, or for which the unit test was introduced in a later version than the method. The path expression is structured as follows:

- Starting from the origin `?start`, the path expression consumes zero or more versions on the path (line 4) until it encounters a sequence of one or more versions (using the `+<` operator – lines 5 and 6) that contain a method `?m` for which there exists no corresponding unit test.
- Violations are indicated by verifying in the next version (lines 7–9) that either method `?m` is no longer present (line 7), we reach the end of the path (`terminal` in line 8) meaning that there never existed a unit test for the method, or the unit test for the method was added after the introduction of the method itself (line 9).

In each solution to this query, variable `?m` is bound to a method that violates the test-driven development practice. A user of ABSINTHE can then further investigate these methods, or use them in subsequent queries. For instance, to find out which developer made the violation. Note that we did not bind version `?end` beforehand. Depending on whether the method got removed, there never existed a unit test for the method, or the unit test was introduced afterwards, `?end` will respectively be bound to the version in which the method was removed, the last version on the path, or the version in which the unit test was added. Also note that the same logic variable `?m` is used in multiple parts of the path expression. As the binding for this variable can be used throughout the path expression, this allows us to actually express properties about the source-code entity that cross version boundaries.

V. Related work

Mining Software Repositories: In the mining software repositories community, a number of approaches that analyze the information contained in version control systems have been proposed. While a complete overview of these approaches lies outside the scope of this paper, we provide a number of illustrative examples. Hassan [7] proposes a set of complex metrics over the changes in a repository to predict possible faults. Hindle et al. [8] use the commit messages of each revision to track what each developer is working on. They present a way to visualize the topics developers work on, allowing a team leader to track the performance of their team members. Giger et al. [9] track the semantic evolution of a software repository, in combination with a bug tracker, for bug prediction. Bradley and Murphy present Rationalizer [10], a tool that integrates historical information into the source code editor, providing developers information regarding what was changed by whom, and why.

The work presented in this paper complements these approaches: while the above aim at supporting one particular task or solve one particular problem, the goal of ABSINTHE is to offer stakeholders a tool to create *custom* queries over the history of the source code, to retrieve information that is necessary to solve the task at hand.

Logic Program Querying: One of the cornerstones of ABSINTHE is the use of a logic programming language to query software. In particular, we have extended the SOUL program query language with quantified regular path expressions for reasoning about the evolution of versioned software. A number of similar logic-based program query languages have been proposed. Examples include JQuery [11], JQL [12] and PQL [13]. However, none of these languages can be used to reason about the history of a system. They are limited to reasoning about a single version.

Querying Source Code History: There exist a number of query languages that are closely related to ABSINTHE. SCQL [14] is a query language to reason over the evolution of a version repository. Internally it represents a version control system as a graph. Each author, file and revision is a vertex in this graph. Each revision is assigned a timestamp and is connected with the corresponding files and author for that revision. It provides a temporal specification language that allows a user to express relationships as "previous", "after", "always", "never" etc. SCQL does not link version snapshots to source code and therefore does not support queries that are as fine-grained as the ones supported by ABSINTHE.

In earlier work we introduced Time warp [15]. Time warp is a prototype extension of SOUL that has a similar goal as ABSINTHE. It offers an ad-hoc specification language for quantifying over instances of HISMO [16] models. While this work has inspired the development of the approach detailed in this paper, ABSINTHE uses a custom repository representation for scalability reasons and offers a more structured approach to quantify over its information. The main contribution of this paper over previous work is the use of quantified regular path expressions to analyze the history of the source code of a system.

VI. Summary

This paper introduced the ABSINTHE tool for reasoning over version repositories. The contributions of this paper are:

- A logic-based approach for querying the history of versioned software systems.
- The identification of quantified regular path expressions as a suitable means for specifying which characteristics the history of the system should exhibit along successive versions.

Acknowledgments

This research is supported by the IAP Program of the Belgian State. Coen De Roover is funded by the *Stadium* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen).

References

[1] T. Fritz and G. Murphy, "Using information fragments to answer the questions developers ask," in *International Conference on Software Engineering*, 2010, pp. 175–184.

[2] R. Wuyts, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Vrije Universiteit Brussel, Belgium, 2001.

[3] J. Brichau, C. De Roover, and K. Mens, "Open unification for program query languages," in *International Conference of the Chilean Computer Science Society*, 2007.

[4] K. Mens, I. Michiels, and R. Wuyts, "Supporting software development through declaratively codified programming patterns," in *Software Engineering and Knowledge Engineering Conference*, 2001.

[5] J. Laval, S. Denier, S. Ducasse, and J.-R. Fallery, "Supporting simultaneuous versions for software evolution assessment." *Science of Computer Programming*, 2010.

[6] O. de Moor, D. Lacey, and E. Van Wyk, "Universal regular path queries," *Higher-Order and Symbolic Computation*, vol. 16, pp. 15–35, 2003.

[7] A. Hassan, "Predicting faults using the complexity of code changes," in *International Conference on Software Engineering*, 2009, pp. 78–88.

[8] A. Hindle, M. Godfrey, and R. Holt, "What's hot and what's not: Windowed developer topic analysis," in *International Conference on Software Maintenance*, 2009, pp. 339–348.

[9] E. Giger, M. Pinzger, and H. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Working Conference on Mining Software Repositories*, 2011, pp. 83–92.

[10] A. Bradley and G. Murphy, "Supporting software history exploration," in *Mining Software Repositories*, 2011, pp. 193–202.

[11] K. De Volder, "JQuery: A generic code browser with a declarative configuration language." in *International Symposium on Practical Aspects of Declarative Languages*, 2006.

[12] T. Cohen, J. Y. Gil, and I. Maman, "JTL: the Java Tools Language," in *Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2006.

[13] M. Martin, B. Livshits, and M. 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.

[14] A. Hindle and D. German, "SCQL: A formal model and a query language for source control repositories," in *Mining Software Repositories*, 2005, pp. 100–105.

[15] V. Uquillas, A. Kellens, J. Brichau, and T. D'Hondt, "Time warp, an approach for reasoning over system histories," in *Joint international and annual ERCIM workshops on Principles of Software Evolution and software evolution*, 2009, pp. 79–88.

[16] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance: Research and Practice (JSME)*, vol. 18, pp. 207–236, 2006.