A Logic Foundation for a General-Purpose History Querying Tool.

Reinout Stevens[1,1], Coen De Roover[1,2], Carlos Noguera[1,3], Andy Kellens[1,2], Viviane Jonckers[1]

[a] *Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium*

Abstract

Version control systems (VCS) have become indispensable software development tools. The version snapshots they store to provide support for change coordination and release management, effectively track the evolution of the versioned software and its development process. Despite this wealth of historical information, it has only been leveraged by tools that are dedicated to a specific task such as empirical validation of software engineering practices or fault prediction. General-purpose tool support for reasoning about the historical information stored in a version control system is limited. This paper provides a comprehensive description of a logic-based, general-purpose history query tool called Absinthe. Absinthe supports querying versioned Smalltalk system 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. To demonstrate the general-purpose nature of our history query tool, we use it to verify development process constraints, to identify temporal bad smells and to answer questions that developers commonly ask. Finally, we compare a query written in Absinthe to an equivalent one written in Smalltalk.

1. Introduction

Version control systems (e.g., CVS, Subversion, Git and Mercurial) have become indispensable software development tools. They enable a developer to work on a private copy of a jointly developed project and provide support for merging 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.

Email addresses: resteven@vub.ac.be (Reinout Stevens), cderoove@vub.ac.be (Coen De Roover), cnoguera@vub.ac.be (Carlos Noguera), akellens@vub.ac.be (Andy Kellens), vejoncke@soft.vub.ac.be (Viviane Jonckers)

[1]Funded by a doctoral scholarship provided by "IWT Vlaanderen".

[2]Funded by the Stadium SBO project funded by "IWT Vlaanderen".

[3]Funded by the AIRCO project of "FWO Vlaanderen".

This information is potentially of use to several 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, 2] 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?*". Finally, *researchers* in the field of "*Mining Software Repositories*" aim to peruse trends in the history of a system to, among others, predict future defects [3], elicit co-changing entities [4], or to empirically validate engineering practices (e.g., [5]).

Version control systems provide limited support for querying or browsing the history information they store. For example, a user can only retrieve who changed a particular line of code. As a result, software engineering tools are increasingly being developed to answer the history queries of a particular stakeholder. However, existing tools are dedicated to answering a predefined set of queries and are not suited for answering others. There is therefore a need for a general-purpose tool that answers questions about a system's history, formulated by the stakeholders themselves.

In earlier work [6], we introduced a novel logic-based approach to querying the history of versioned software and instantiated it in a general-purpose tool called Absinthe. We primarily focused on the identification of quantified regular path expressions [7] as a suitable mean for specifying which characteristics the software should exhibit along successive versions. In this paper, we provide a more comprehensive overview of Absinthe as a whole. In addition:

- We demonstrate the expressiveness and applicability of Absinthe using several representative history queries that answer questions of various stakeholders. The examples stem from three application domains: verifying development process constraints, answering developer questions and identifying temporal bad smells.
- We discuss the advantages and disadvantages of using Absinthe over Smalltalk by means of a query written in both languages.

This paper is structured as follows. Section 2 introduces our logic-based approach to history querying and its prototype instantiation Absinthe. Section 3 demonstrates Absinthe through history queries that answer questions of various stakeholders such as team leaders, software developers and tool builders. Section 4 compares Absinthe to Smalltalk. Before concluding this paper, we discuss the related work in Section 5.

2. ABSINTHE: A General-Purpose History Querying Tool

In this section, we introduce the ABSINTHE[4] tool for answering questions about the evolution of versioned software. These questions are specified as a logic query that quantifies over a representation of the software's version repository. As such, ABSINTHE is a tool in the tradition of logic program querying —which has already proven itself suitable for answering questions about a single software version. The expressiveness of the logic paradigm facilitates specifying characteristics of sought after code.

2.1. A Logic Program Querying Foundation

SOUL [8] provides the logic program querying foundation for ABSINTHE. SOUL is a Prolog-like language with dedicated features for querying code. Among others, it features a symbiosis with Smalltalk. SOUL variables can be bound to logic as well as Smalltalk values. To this end, SOUL supports embedding Smalltalk expressions in logic queries. ABSINTHE relies on this feature to quantify over its object-oriented repository representation (cf. Section 2.2).

We illustrate SOUL by means of the following query:

```
1  if ?class isClassWithName: #Person,
2     ?method isMethodInClass: ?class,
3     ?method methodSendsMessage: ?msg,
4     ?implementer isMethodWithName: ?msg inClass: ?implementClass
```

The query detects all the methods that may be called by an instance of the class `Person`. This is done by looking at all the instance methods of the class `Person`, finding which messages these methods send, and finally finding a method implementing this message. As Smalltalk is a dynamic language we cannot be sure which of these methods are actually called at runtime.

The first condition of the query uses binary predicate `isClassWithName:/2` to bind variable `?class` to the `Person` class. 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. We indicate the arity of a predicate by means of a suffix. Conditions are separated by commas. The second condition of the query requires `?method` to be one of the methods of the `Person` class. The third condition binds variable `?msg` to one of the messages sent by `?method`. The final condition quantifies over methods with the name of this message, and the class in which they reside.

2.1.1. From Logic Program Queries to History Queries

The history of a versioned system can be represented 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. Section 2.2 details this history representation used by ABSINTHE.

[4] ABSINTHE can be downloaded from `http://soft.vub.ac.be/SOUL/`

To enable querying a graph-based history representation, Absinthe extends the syntax and semantics of Soul with an intuitive formalism to quantify over the paths through a graph: quantified regular path expressions [7]. Consisting of logic conditions to which regular expression operators are applied, quantified regular path expressions enable specifying which characteristics a system should exhibit along the paths throughout its history representation. To this end, each condition within a quantified regular path expression is implicitly evaluated against a different version of the software. These conditions can use any predicate from the existing Soul libraries. Section 2.2 discusses Absinthe's use of quantified regular path expressions in detail.

2.2. History Representation

Absinthe relies on an object-oriented representation of the history of a versioned software system called Ring [9]. Ring models this history 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 merge 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. Ring stores 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). Similar to the HISMO meta-model [10], each structural entity has a reference to the version of the software in which it was defined.

- 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 history representation, Ring uses the same object for an entity that remains unchanged across successive versions. To this end, an additional layer of indirection is introduced 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 (the objects representing classes, methods, etc.). 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. The actual implementation strategy is inspired by the work of Laval et al. [11].

2.2.1. Populating the History Representation

A number of importers can be used to populate ABSINTHE's history representation from a version control system. Currently, we support importing Smalltalk programs from either VisualWorks Store or Monticello repositories. The source code of each version is checked out of the repository and subsequently parsed. In the case of the first version in the repository, the importer creates a full version snapshot. Implementation objects are instantiated for all of its structural entities. For all subsequent versions, the AST of their code is compared with the AST of its predecessor's code. For versions that resulted from the merger of multiple versions, the AST of the merged version is compared with those of all its predecessors. New implementation objects are only instantiated for structural entities that had changed or were added. The mapping from entity identifiers to implementation objects is updated accordingly within the version snapshot. Note that the importers compare structural entities based on their name. As such, they do not take renaming into account. Within a version snapshot, only the presence of a new entity and the absence of the renamed entity is reflected. It is up to the user to detect and define entities that are conceptually the same, but are not identified as such. For example a renamed class can be detected by finding a newly introduced class implements the same methods as a class that was removed in the previous version. ABSINTHE does not do this automatically for the user as there is no clear definition when a refactoring preserves the identity of an entity.

2.2.2. Predicates for Quantifying over the History Representation

ABSINTHE provides a library of SOUL predicates for quantifying over its history representation, which can be used within history queries. Although the history representation is object-oriented, it does not need to be converted to a logic fact base before it can be queried. The symbiosis of SOUL with Smalltalk enables the ABSINTHE predicates to query the objects of the representation directly

Table 1 depicts an excerpt from this predicate library. Three categories of predicates can be discerned: (1) predicates that reify history-specific information about a version (e.g., unary `isVersion/1`), (2) predicates that reify the structural entities within a version snapshot (e.g., unary `isClass/1`) and their relations (e.g., binary `isClassInPackage:/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 instance, binary predicate `isClassInPackage:/2`

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
`?v isVersionBetweenDates: ?start and: ?end`	Find all versions during a particular time interval
`?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 isClassInPackage: ?p : ?version`	Class belongs to package 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 isInterfaceInPackage: ?p : ?version`	Interface belongs to package in a particular version
`?i isInterfaceWithName: ?n : ?version`	Interface in version has name
`?m isMethod : ?version`	Entity is a method
`?m isClassMethod : ?version`	Entity is a class (static) method
`?m isMethodInClass: ?c : ?version`	Method belongs to class
`?m isMethodWithName: ?n inClass: ?c : ?version`	Method with particular name in class
`?p isPackage : ?version`	Entity is a package in a particular version
`?p isPackageWithName: ?n : ?version`	Package with name
`?v isInstanceVariableWithName: ?n inClass: ?c : ?version`	Entity is field with name in class
`?v isClassVariableWithName: ?n inClass: ?c : ?version`	Entity is static field with name in class
`?c isSubclassOf: ?super: ?version`	Class is a direct subclass of superclass
`?c isSuperclassOf: ?sub : ?version`	Class is a direct superclass of a subclass
`?c classImplementsInterface: ?i : ?version`	Class implements a particular interface
`?i interfaceIsImplementedBy: ?c : ?version`	Interface is implemented by a particular class in a version
`?i isSubinterfaceOf: ?super : ?version`	Interface is a subinterface of a particular interface
`?tree isParseTreeOf: ?e : ?version`	Tree is the Abstract Syntax Tree of entity in a version
`?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 1: Excerpt from our library of logic predicates.

reifies the relation between classes and the packages in which they are declared. Note that 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.

The third category of predicates reifies 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 by this method.

2.3. QRPEs for Specifying Paths Through the History Representation

Absinthe extends Soul with Quantified Regular Path Expressions [7] (QRPEs). QRPEs are an intuitive formalism for quantifying over the paths through a graph. They are but one of many graph query formalisms. Other examples include Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [12]. We have opted for QRPEs, as they strike the balance between expressiveness and ease of use [13]. As such, they are a natural fit to quantify over the information in Absinthe's history representation.

Quantified regular path expressions are akin to regular expressions [14], 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, QRPEs match sequences of successive versions from the history 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:

```
if ?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`. Variable `?quantifier` determines whether a path expression is universally or existentially quantified. The existential quantifier, written `e`, denotes that at least one path must exist between `?start` and `?end` for which the expression holds. The universal quantifier, written `a`, denotes that the expression has to hold on all the paths between `?start` and `?end`.

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. Both variables will get bound to their corresponding version for each successful regular path expression.

The actual regular path expression `?exp` is an expression of the form $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$, where each e_i is a logical expression that must hold in version v_i. Each $e_i \rightarrow e_j$ can be annotated by a $*$ or $+$ and single or double arrowhead, each controlling how frequently the condition e_i can be iterated over across

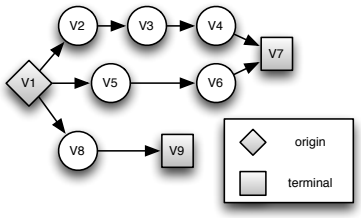

Figure 1: Illustration of a graph of versions.

a number of successive versions. Both $*$ and $+$ have the same meaning as in regular expressions: the conditions can succeed in arbitrary number of successive versions. With a $*$ this number can be zero, while a $+$ means this number must be larger than zero. The head controls whether paths are considered using pre-order ($\rightarrow$) or post-order ($\twoheadrightarrow$) numbering.

The following meta-symbols can be used alongside the logic conditions in a path expression as well:

- *origin*: Only consumes the current version if it is not the successor of any other version (similar to the caret in regular expressions).
- *terminal*: Only consumes the current version if it does not have any successors (similar to the dollar in regular expressions).
- $\$this\_version$: A variable bound to the current version of the path expression.

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_7 and V_9 being terminals. When `?start` is bound to version V_1 and `?end` is bound to V_7, an existentially quantified expression `e(?exp) matches: ?path start: ?start end: ?end` verifies whether there exists a path between V_1 and V_7 that matches `?exp`. In other words, either the path $(V_1, V_2, V_3, V_4, V_7)$ or the path (V_1, V_5, V_6, V_7) 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` —independent of the actual quantifier that was used.

We present a logic query that embeds a universally quantified path expression.

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

This query quantifies over the history representation to identify 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. Lines 2–3 bind two logic variables, `?start` and `?end` to an origin and terminal version respectively. Lines 4–8 contain the actual QRPE. The universally quantified regular path expression on lines 4–5 consists of two conditions. The first condition (line 4) verifies that no class named `Evaluator` exists in the first version on the path. The second condition (line 5), to which the $\twoheadrightarrow +$ operator has been applied, verifies that a class named `Evaluator` exists from the second version on the path until the end of the path.

Note that both conditions within the path expression use predicate `isClassWithName:/2` from Table 1. 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.

2.4. Overview of the Architecture

Figure 2 depicts the architecture of Absinthe. A version control system holds the complete history of the queried software project. From the VCS, the importer builds an instance of the Ring model. The importer supports Monticello repositories storing projects written in Smalltalk. The Ring model provides an object-oriented representation of the history of the software project. This representation works on two levels. First, it provides history information, such as the timestamp of a version, the author, the commit message etc. Second, it stores the entities (classes, methods, etc.) present in a version up until the level of methods. Frequently used sub-method information (for example which messages a method sends) is also stored.

The predicate library reifies this information: it features predicates quantifying over version information and the structural entities within each version snapshot. These predicates are implemented using the Soul programming language. Using Soul, QRPE, and these predicates, the software project under investigation can be easily queried.

This architecture allows for some flexibility. Software projects implemented in a different language or versioned in a different VCS can be supported through another importer, given that the language can be mapped to Ring. A new temporal specification language, such as computation tree logic, can be implemented

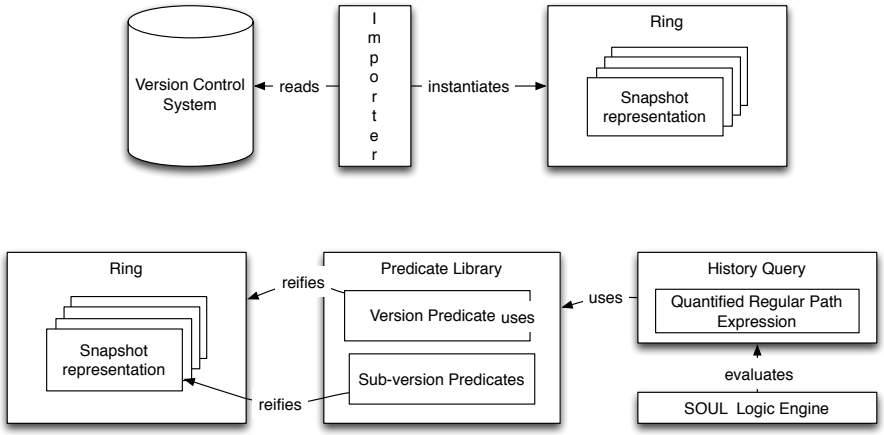

Figure 2: Overview of how the RING model is instantiated (top half) and queried (bottom half).

in SOUL and used to specify the temporal relations. One could even replace SOUL with another program querying language, provided that the predicate library is either changed as well or compatible with the new language. Finally, RING can be replaced with a different meta-model. This requires changing the importer so that it instantiates the new meta-model correctly. However, this may also require changing the predicate library in case the interface to the model is changed, or new information is exposed.

3. Answering History Queries using ABSINTHE

We illustrate the expressiveness and applicability of ABSINTHE by specifying queries that answer several representative history-related questions.

3.1. History Queries Answering Developer Questions

We begin our discussion with examples that illustrate how ABSINTHE can be used by developers to answer some of the questions they commonly ask (e.g., as identified by the surveys [1, 2]).

3.1.1. Identifying Co-Authors of a Class Owned by a Developer

In our first example, a developer wants to find out which people modified some of the classes he owns. A person owns a class when he introduced that class.

First, we create a query to find the version in which a specific class named `Evaluator` was introduced. Second, we create a query to find all the classes introduced by a certain person. Finally, we create a query for the initial history-related question; which people modified classes introduced by our developer.

Find the version in which `Evaluator` was introduced. The following query finds the software version in which a class named `Evaluator` got introduced:

```
1  if
2     ?start isOrigin,
3     e((true)→ *
4       and(?class isClassWithName: Evaluator,
5           ?class wasIntroduced))
6     matches: ?path  from: ?start  end: ?version
```

Line 2 binds `?start` to the root version of the repository. Lines 3–6 represent the actual path expression. We are interested in a path in which its last version has introduced class `Evaluator`. To this end, we use predicates `isClassWithName:/2` and `wasIntroduced/1`. The first predicate succeeds when the current version contains a class with the specified name. The second predicate succeeds when its argument was introduced in the current version. This is done by verifying that the class is present in the current version, but not in one of the predecessors of the current version. Line 3 specifies the quantifier for the QRPE: we use the existential quantifier as we are only interested in one path. The condition on line 3 consumes versions along the path as long as the next line does not succeed. Lines 4–5 specify that the current version must contain a class named `Evaluator`, and that the class was introduced in that version. The last line returns the path, start version and end version for which the QRPE succeeded.

Finding which classes are introduced by a specific author. The previous query identified the version in which the `Evaluator` class was introduced. The following query extends the previous one so that it identifies classes introduced by a specific author.

```
1  if
2     ?start isOrigin,
3     e((true)→ *
4       and(?class isClass,
5       ?class wasIntroduced,
6       bob isAuthorOfVersion))
7     matches: ?path from: ?start end: ?end
```

We use predicate `isAuthorOfVersion/1`, which succeeds when its first argument unifies with the author of the current version. The query only differs from the previous one on lines 4–6. Here, we specify that `?class` needs to be a class introduced in the current version, and that Bob must be the author of that version.

Finding who changed classes introduced by a specific author. Our final query answers the original question: "who changed classes introduced by Bob?".

```
1  if
2     ?start isOrigin,
3      e((true)→ *
4        and(?class isClass,
5            ?class wasIntroduced,
6            bob isAuthorOfVersion)→
7        (true)→ *
8        and(?class wasChanged,
9            ?changer isAuthorOfVersion,
10           not(?changer equals: bob)))
11    matches: ?path from: ?start end: ?end
```

The beginning of the query is the same as the previous one: we are looking for a class that was introduced by Bob. Lines 7–10 specify that this class is modified in a later version by a different author. Line 7 skips an arbitrary number of versions, as the modification may happen in any successor of the version in which the class was introduced. Lines 8–10 identify the successor version in which the class is changed; `?changer` is the author of that version. Line 10 verifies that `?changer` differs from the original author.

3.1.2. Identifying Co-changing Classes

For the second example in this section, consider detecting classes that consistently change together. Finding co-changing classes is interesting, as they might hint at the presence of hidden dependencies [4].

The following ABSINTHE query identifies pairs of classes that are, starting from the first version in which they changed together, always changed together.

```
1  if
2    ?start isOrigin,
3    e((true)→ *
4      and(?classA isClass,
5          ?classA wasChanged,
6          ?classB isClass,
7          not(?classA equals: ?classB)
8          ?classB wasChanged)→
9          (or(and(?classA wasChanged,
10                 ?classB wasChanged)
11             and(not(?classA wasChanged),
12                 not(?classB wasChanged)))↠ +
13       terminal)
14   matches: ?path start: ?start end: ?end
```

Lines 3–8 consume versions on the path until two different classes change together. Predicate `wasChanged/1` (lines 5 and 8) checks if its argument got changed in the current version. Lines 9–12 consume one or more versions in which classes `?classA` and `?classB` are consistently changed together (lines 9–10) or not changed at all (lines 12–13). Finally, the QRPE succeeds if we reach the end of a path (`terminal` on line 14).

3.1.3. Identifying Re-introduced Methods

Our third example identifies methods deleted at one particular point in time, but added again afterwards —possibly under a different name. The following query identifies such methods:

```
1  if
2     ?start isOrigin,
3     e((true)→ *
4        and(?method isMethodInClass:?class,
5            RBMethodNode(?name,?args,?body) isParseTreeOf: ?method)→
6        (true)→ *
7        (not(?method isMethodInClass:?class))↠ +
8        and(?reintroduced isMethodInClass:?class,
9            RBMethodNode(?,?args,?body) isParseTreeOf: ?reintroduced))
10    matches: ?path start: ?start end: ?end
```

Starting from the origin, its QRPE will try to consume one or more versions (line 3) until it reaches a version that contains a method `?method` defined by a class `?class`. Line 5 uses predicate `isParseTreeOf:/2` to retrieve the parse tree of `?method` from the history representation. By unifying this parse tree with an equivalent logic term, the same line extracts the method's `?name`, arguments `?args` and `?body`. The logic term is named after the class representing a method in the AST, namely Refactoring Browser Method Node. Next, the QRPE lazily consumes zero or more versions (line 6) until the method is removed from the class (line 7). Finally, lines 8–9 verify whether there exists a method `?reintroduced` in the same class `?class` for which the arguments `?args` and the method `?body` match that of the original method.

In all solutions to the query, variable `?end` will be bound to the version in which the method got introduced again. Note that the query does not only identify reintroduced methods based on the last method body encountered on the path, but on all such previous method bodies. This is ensured by the use of `(true)→*` on line 6. Upon backtracking, this line will consume a longer path in search for a reintroduced method with the same body.

3.2. History Queries Verifying a Development Process

We now shift our focus to history-related questions managers of a development team might want answered. These questions concern the development process.

3.2.1. Finding Violations against Test-Driven Development

In the first example, a manager needs to find violations against the principle of test-driven development which advocates writing tests prior to implementing the tested functionality. The manager wants to assert that the team actually followed the intended design process. In natural language, the question that the manager needs answered is "what are the methods in the system for which a unit test was never added, or for which the corresponding unit test was added only after the method was introduced?".

The following ABSINTHE query assumes that a predicate `isTestFor:/2` exists which verifies whether a unit-test tests a particular method. Although we do not provide a definition for this predicate, this predicate links a unit test to a method through a naming convention.

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

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 3) until it encounters a sequence of one or more versions (using the $\twoheadrightarrow +$ operator – lines 4 and 5) that contains a method `?m` for which there exists no corresponding unit test. Violations are identified by verifying in the next version (lines 6–8) that either method `?m` is no longer present (line 6), or that we have reach the end of the path (`terminal` on line 7) meaning that there never existed a unit test for the method, or that the unit test for the method was added after method was introduced (line 8).

In each solution to this query, variable `?m` is bound to a method that violates the test-driven development practice. These violations can be investigated in successive Absinthe queries. For instance, to find out which developer committed 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.

The depicted query illustrates an important feature of Absinthe. Logic variables (i.e., `?m`) can be used in multiple parts of a path expression. This enables tracking software entities across version boundaries.

3.2.2. Finding Violations against a Release Schedule

For the second example, consider the situation in which a manager needs to verify whether a prescribed release schedule was followed. The release schedule states that between the alpha and beta release, developers are only allowed to either fix bugs or work on planned features; after the beta release only bugs are allowed to be fixed until the product is released. Suppose also that developers, for traceability purposes, annotate the intent of each revision by stating its purpose (e.g., "worked on feature X", "fixed bug number Y") in the commit message.

The following rules and query depicted verify this release schedule (i.e., all revisions between the alpha and beta release should concern a feature or a bug fix; versions after the beta release should only be bug fixes).

```
1  alpha : ?version if
2    ?version isVersionAtDate:{15 January 2011}
3  beta: ?version if
4    ?version isVersionAtDate:{20 February 2011}
5  release : ?version if
6    ?version isVersionAtDate:{1 May 2011}
7
8  if
9    alpha : ?alphaVersion,
10   a((and(?message isCommitMessageOfVersion,
11          ?message matchesRegex: '.*(feature|bug).*',
12          alpha))↠ *
13     (and(?message isCommitMessageOfVersion,
14          ?message matchesRegex: '.*bug.*',
15          beta))↠ *
16     release)
17   matches: ?path start: ?alphaVersion end: ?releasedVersion
```

First, we define which versions are considered to be an alpha, beta and a product release. To this end, the logic rules on lines 1–6 define three auxiliary predicates. The history query on lines 8–17 verifies the actual release schedule. Its universally quantified QRPE verifies that all paths from `alpha` to `releas` are of the following form. The first version on each path should be the `alpha` release (lines 9–17). After the alpha release, there should be zero or more versions that satisfy the requirement that their commit message contains either strings 'feature' or 'bug' (lines 10–12). We verify this using a regular expression. From the beta release onwards, there should be zero or more versions that contain the string 'bug' in their commit message. The final version on the path should be marked `release`.

Note that the QRPE in this query is universally quantified. This ensures that *all* paths between the different milestones and the release follow the prescribed schedule. Thus, if a developer creates a separate branch in order to work on a feature after the beta version, this will only be marked as a violation if that branch is actually merged back into the main trunk before the release version.

This query demonstrates that Absinthe queries do not necessarily have to reason about the history of the source code. In the above query, we solely reason about the meta-data of versions, such as commit messages.

3.3. History Queries Identifying "Temporal Bad Smells"

As a final example we illustrate the use of Absinthe to detect a temporal bad smell. These are bad smells that only become apparent when analyzing the evolution of the source code of a system. In particular, we introduce the concept of *zombie code*. We define this bad smell as methods in the system that were stopped being used in a particular version but that are not removed (i.e., become dead code), and that are used again in a later version. While the presence of zombie code is not necessarily a problem in the system, instances of this bad smell might point at uses of implicitly deprecated code.

The following Absinthe query retrieves instances of zombie methods:

```
1  if
2    ?start isOrigin,
3    e((true)→ *
4      (and(?m isMethodWithName:?name inClass: ?c,
5           ?invoker methodSendsMessage:?name))→
6      (and(?m isMethodWithName:?name inClass: ?c,
7          not(? methodSendsMessage:?name)))↠ +
8          and(?m isMethodWithName:?name inClass: ?c,
9              ?newInvoker methodSendsMessage:?name))
10   matches: ?path start: ?start end: ?end
```

This query consists of a single QRPE that matches a sub-path that exhibits the following characteristics. Lines 3–5 advance until a version is encountered that contains a method `?m` with name `?name` defined in a class `?c` that is called by a method `?invoker`. Lines 6–7 match one or more versions in which method `?m` is still present, but no callers are found (i.e., the method is implicitly deprecated). Finally, lines 8–9 match a version in which the method `?m` is still present, and it is again called by method `?newInvoker`.

When evaluated, this query will bind the logic variable `?m` to all methods considered as a zombie method. The logic variable `?end` will be bound to the first version in which the zombie method gets called again. Notice that we also retrieve bindings for the methods `?invoker` and `?newInvoker`: the query does not only provide information regarding the zombie methods, but also indicates all callers of such methods.

4. Validation

In the previous section we demonstrated the use of Absinthe for answering several representative history queries. In this section, we discuss the expressiveness of Absinthe's history query specification language and the performance of the mechanism it uses to find solutions to such a query.

We compare the query that detects co-changing entities presented in 3.1.2 to an equivalent Smalltalk implementation. The Smalltalk implementation interfaces directly with the Ring model.

For reference, the following code implements the query in Absinthe:

```
1  if
2    ?start isOrigin,
3    e((true)→ *
4      and(?classA isClass,
5          ?classA wasChanged,
6          ?classB isClass,
7          not(?classA equals: ?classB)
8          ?classB wasChanged)→
9          (or(and(?classA wasChanged,
10                 ?classB wasChanged)
11             and(not(?classA wasChanged),
12                 not(?classB wasChanged)))↠ +
13       terminal)
14   matches: ?path start: ?start end: ?end
```

The following Smalltalk code defines a method `classNamed:cochangesAlongPath:` which, given the name of a class (`aSymbol`) and a path (`aPath`) on which to perform the search, detects which classes are co-changing.

```
1  classNamed: aSymbol cochangesAlongPath: aPath
2    |solutions|
3    solutions := OrderedCollection new.
4    [aPath isEmpty] whileFalse: [
5      |currentVersion targetClass|
6      currentVersion := aPath first.
7      aPath remove: currentVersion.
8      targetClass := self findClassNamed: aSymbol inVersion: currentVersion.
9      (targetClass isNil) ifFalse: [
10       currentVersion classes do: [:aClass |
11         (aClass = targetClass) ifFalse: [
12           (self classCochanges: targetClass with: aClass along: aPath) ifTrue: [
13             solutions add: aClass.
14           ].
15         ].
16       ].
17     ].
18   ].
19   ^solutions.
```

The method starts by looping over all the versions in the path (lines 4–18). For each version, it then retrieves the version-specific entity representing the class with the provided name and whether this entity is still present in the current version (lines 8–9). Next, it loops over all the classes present in the current version and verifies whether a class co-changes for the remaining versions of the path (lines 10–12) by invoking the helper method `classCochanges:with:along:`. If this is the case the class is added to the solutions. Finally, the solutions are returned (line 19).

The method which verifies whether two classes co-change in *all* the versions of a given path is the following:

```
1  classCochanges: candidateClass with: targetClass along: aPath
2    ^(aPath anySatisfy: [ :aVersion |
3      |versionCandidate versionTarget|
4      versionCandidate := aVersion thisHistory: candidateClass.
5      versionTarget := aVersion thisHistory: targetClass.
6      (versionCandidate isNil or: [versionTarget isNil]) ifTrue: [
7        ^true.
8      ].
9      ^(versionTarget wasChanged = versionCandidate wasChanged) not.
10   ]) not.
```

The method detects whether the path contains a version in which the classes do not change together. To this end, it first retrieves the version-specific representation of the two given classes for each version (lines 4–5) and stops when either class no longer exists (lines 6–8) or when they do not co-change (line 9). The result is a boolean indicating whether the two classes co-change along the path or not.

4.1. Discussion

Having introduced the Smalltalk query over the Ring model that finds co-changing classes, we now compare it to the previously introduced equivalent Absinthe expression in terms of expressiveness and performance.

4.1.1. Expressiveness

First, we consider the expressiveness of both solutions by comparing navigation through the version graph, retrieval of relevant entities in each version and reusability.

The Smalltalk code has several issues, which are absent from the Absinthe version of the query:

Version Navigation Absinthe features QRPE to describe paths through the graph of versions. In Smalltalk, this version management needs to be done manually. This is reflected in the code by the looping over all the versions in the path. If the path was not provided, a queue or work list of versions would have to be used, and every time a version is processed its successors would be added to that list. Absinthe also returns the path for which the query succeeded. This path would have to be constructed manually in Smalltalk. This requires that the user correctly adds and removes versions to the path under construction while executing the query.

Entity Retrieval Absinthe looks up the version-specific representation for every variable, even if that variable was bound and used in another version. If the entity is no longer present in the current version, the reasoning engine will backtrack automatically. In Smalltalk, all of the above must be implemented by the user. This results in repetitive code whenever conditions are checked in a version.

Reusability Each variable passed to an Absinthe query can be used both as input and output for that query. For example the "co-changing entities" query can be used to verify whether two given classes co-change, which classes co-change with a given class or to find all the co-changing classes in the system. The Smalltalk code we have provided, in contrast, only finds which classes co-change with a given class.

The Smalltalk code has repetitive patterns that, while necessary, are not related to the main logic of the query. In the `classNamed:cochangesAlongPath:`, only line 12 is actually related to co-changing entities; the other 18 lines deal with navigation and entity retrieval. These additional concerns are handled by the Absinthe runtime, thus allowing the developer to concentrate on the actual logic of the query.

4.1.2. Performance

The superior expressiveness of Absinthe comes at the cost of a lower performance. To illustrate the overhead of Absinthe over Smalltalk we have conducted a benchmark. In this benchmark we compute all the classes that co-change with a given class along a provided path. This benchmark is executed

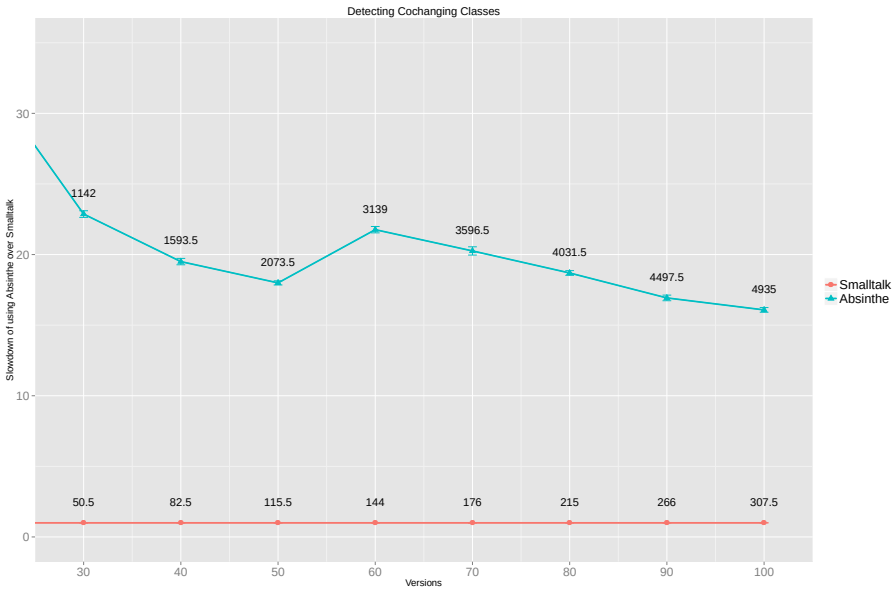

Figure 3: The overhead of using Absinthe over Smalltalk for the query detecting which classes co-change with a given class. The Y-axis depicts the overhead. The X-axis depicts the number of versions. The number with each node denotes the runtime in ms.

against the version repository of the Ring implementation, which comprises 281 versions, of which the latest version contains 249 classes. The benchmark is executed using Pharo 1.3 and the CogVM 4.0.0, running on a single thread of a server with 2.4 Ghz Xeon E5520 processors. Pharo [15] is a popular Smalltalk dialect originally forked from Squeak. The heap of the VM is restricted to 1 GB.

Figure 3 depicts the results for this benchmark. The X-axis shows the number of versions for which the query was executed. The Y-axis depicts the overhead of using Absinthe over Smalltalk. The number of each node denotes the runtime in ms. Note that the graph only shows data when the number of versions is more than 30. The runtime of both queries with a lower number of versions was too low to produce any meaningful result.

For this particular query Absinthe is around 20 times slower than Smalltalk. Note that the overhead of Absinthe in general diminishes as the number of versions increases. At the 60 versions mark, this trend is broken with a sharp increase in runtime for the Absinthe version of the query. Looking at the code of the repository, we see that in version 50 a refactoring that augments the number of classes in the system took place. This implies that the Absinthe query is more sensitive to the size of the search space than the Smalltalk version.

In general, the performance of Absinthe changes depending on several factors. Examples are the number of bound and unbound variables in a query, the number of entities present in each version, the topology of the version graph

etc.

While the Smalltalk version is clearly faster, the point of this comparison is to provide a base line against which to compare the relative advantages of Absinthe in terms of expressiveness versus its runtime performance.

5. Related work

Our work lies at the intersection of mining software repositories and logic program querying.

Mining Software Repositories. In the field of mining software repositories, 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 two illustrative examples. Giger et al. [16] track the semantic evolution of a software repository, in combination with a bug tracker, for bug prediction. Instead of using a line-by-line comparison they use fine-grained source code changes that contain semantic information about the changes. Bradley and Murphy present Rationalizer [17], 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 such approaches: while the above aims at supporting one particular task or solve one particular problem, the goal of Absinthe is to offer stakeholders a tool to create *custom* queries about the history of the source code, to retrieve information that is necessary to solve the task at hand.

Logic Program Querying. One of the foundations of Absinthe is the use of a logic programming language to query software. In particular, we have extended the Soul [8] 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 [18], JQL [19] and SCL [20]. 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 [21] is a query language to reason about 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.

V-Praxis [22] compiles a version repository into a logic fact base. It does this by creating a complete representation of the first version, and storing deltas

between versions. Each fact is associated with a timestamp, allowing a complete reconstruction of the history of a project. Unlike Absinthe, V-Praxis does not feature a dedicated query and temporal specification language, but instead uses regular Prolog. This results in repetitive patterns, such as binding a version to a logic variable before executing a goal in that version. These patterns are absent from Absinthe queries.

6. Conclusions and Future Work

In this paper we have detailed the logic-based history query tool Absinthe. Next to logic predicates for reasoning about a system's *state* in a particular version, Absinthe offers quantified regular path expressions for reasoning about the *evolution* of the system throughout successive versions. While the former have proven themselves for querying source code, the latter have proven themselves for querying the paths through a graph. Together they give rise to an elegant language for querying a system's history. To the best of our knowledge, Absinthe is the first tool that applies QRPEs in this manner.

We have demonstrated the expressiveness and applicability of Absinthe on several representative history queries. The resulting queries are descriptive, but future experiments ought to determine whether Absinthe is sufficiently intuitive for application developers to use.

Finally, we have compared and discussed the "co-changing entities" query written in Absinthe with an equivalent one in Smalltalk in terms of expressiveness and performance. The increase in expressiveness comes at a reasonable performance loss.

In future work we will research whether a specification language closer to the source code would be feasible. We suspect that the granularity of the current specification language is too coarse. The source code is stored in the model, but querying source code entities across multiple versions still remains hard. At the moment it can only be done by specifying the characteristics of the code in each version individually. Better would be to allow some transformational approach, where a user specifies the code in one version, and describe how the code has to evolve in future versions by means of transformations.

Next we would like to replace the in-memory representation of a repository to a database representation. Even though Ring provides a scalable representation of the history of a versioned software project the memory consumption is still too high for large-scaled projects. This can be solved by storing the representation in a database.

References

[1] T. Fritz, 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.

[2] T. D. LaToza, B. A. Myers, Hard-to-answer questions about code, in: Evaluation and Usability of Programming Languages and Tools (PLATEAU10), 2010, pp. 8:1–8:6.

[3] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR10), 2010, pp. 31–41.

[4] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller, Mining version histories to guide software changes, in: Proceedings of the 26th International Conference on Software Engineering (ICSE04), 2004, pp. 563–572.

[5] F. Khomh, T. Dhaliwal, Y. Zou, B. Adams, Do faster releases improve software quality? an empirical case study of Mozilla Firefox, in: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR12), 2012, pp. 179–188.

[6] A. Kellens, C. De Roover, C. Noguera, R. Stevens, 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.

[7] O. de Moor, D. Lacey, E. V. Wyk, Universal regular path queries, Higher-Order and Symbolic Computation (2002) 15–35.

[8] C. De Roover, C. Noguera, A. Kellens, 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.

[9] V. U. Gómez, Supporting integration activities in object-oriented applications, Ph.D. thesis, Vrije Universiteit Brussel - Université des Sciences et Technologies de Lille (October 2012).

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

[11] J. Laval, S. Denier, S. Ducasse, J.-R. Fallery, Supporting simultaneuous versions for software evolution assessment., Science of Computer Programming 76 (12) (2011) 1177–1193.

[12] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, System and Software Verification, Model-Checking Techniques and Tools, Springer, 2001.

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

[14] A. Aho, Algorithms for finding patterns in strings, MIT Press, 1990.

[15] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.

[16] E. Giger, M. Pinzger, H. C. Gall, Comparing fine-grained source code changes and code churn for bug prediction, in: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR11), 2011, pp. 83–92.

[17] A. W. Bradley, G. C. Murphy, Supporting software history exploration, in: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR11), 2011, pp. 193–202.

[18] K. De Volder, JQuery: A generic code browser with a declarative configuration language., in: Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages (PADL06), 2006, pp. 88–102.

[19] T. Cohen, J. Y. Gil, I. Maman, JTL: the Java Tools Language, in: Proceedings of the 21st Annual SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA06), 2006, pp. 89–108.

[20] D. Hou, H. J. Hoover, Using SCL to specify and check design intent in source code, Transactions on Software Engineering 32 (6) (2006) 404–423.

[21] A. Hindle, 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.

[22] A. Mougenot, X. Blanc, 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.