Querying Sequential Software Engineering Data

Chengnian Sun[†] Haidong Zhang[§] Jian-Guang Lou[§] Hongyu Zhang[§]
Qiang Wang[§] Dongmei Zhang[§] Siau-Cheng Khoo[‡]

[†]University of California, Davis, USA
[§]Microsoft Research, Beijing, China
[§]National University of Singapore, Singapore

cnsun@ucdavis.edu, khoosc@nus.edu.sg
{haidong.zhang, jlou, honzhang, qiang.wang, dongmeiz}@microsoft.com

ABSTRACT

We propose a pattern-based approach to effectively and efficiently analyzing sequential software engineering (SE) data. Different from other types of SE data, sequential SE data preserves unique *temporal* properties, which cannot be easily analyzed without much programming effort. In order to facilitate the analysis of sequential SE data, we design a sequential pattern query language (*SPQL*), which specifies the temporal properties based on regular expressions, and is enhanced with variables and statements to store and manipulate matching states. We also propose a query engine to effectively process the *SPQL* queries.

We have applied our approach to analyze two types of SE data, namely bug report history and source code change history. We experiment with 181,213 Eclipse bug reports and 323,989 code revisions of Android. *SPQL* enables us to explore interesting temporal properties underneath these sequential data with a few lines of query code and low matching overhead. The analysis results can help better understand a software process and identify process violations.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*

General Terms

Languages, Management

Keywords

sequential data, pattern matching, query, mining software repository

FSE '14, November 16–22, 2014, Hong Kong, China

1. INTRODUCTION

Over the years of software practice, there is a plethora of software engineering (SE) data produced by different processes and tools. These data include source code, bug reports, change logs, metric data, etc. Many organizations are now maintaining large software repositories and are willing to mine or even share their SE data to facilitate the exchange of results and to improve their current practices by learning from others. The construction of software repositories is further facilitated by the open-source software movement, which also offers a vast amount of SE data to the public. The increase in number and size of software repositories brings both opportunities and challenges for researchers and practitioners.

There is a great variability in SE data. Many existing approaches (e.g., [19, 22, 32, 34]) focus on the relational aspect of SE data and treat SE data as structured data in a database. Other approaches (e.g., [5, 7, 23, 28, 27, 30, 33, 31]) focus on the textual aspect of SE data and treat SE data as a set of documents. We find that many kinds of SE data also have the *sequential nature*, which is often displayed as a sequence of activity records. For example, during the lifetime of a bug report from its initially creation to final close, there is usually a series of status changes. During the lifetime of a source code file, there is often a series of revisions. The sequential nature of data provides invaluable insight into the software maintenance process, and opens up the possibility of conducting more holistic study of software projects in ways previously unexplored.

In this paper, we propose a highly-adaptable pattern-matching approach for effective and efficient query of sequential SE data. With our query language *SPQL* (sequential pattern query language), a user can specify a query describing the characteristics of sequential SE data. The matching results can be returned by the *SPQL* query engine.

Query Language The language is a hybrid of regular expressions and several constructs from imperative programming languages (i.e., variables, if-statement, assignment statements and function calls). This design choice makes the language capable of expressing complex queries, and also easy to learn, especially for developers with experience on regular expressions.

Query Engine The query engine is powered by a proposed extension of non-deterministic automata (NFA) simulation algorithm that supports variables and actions. The classical

NFA simulation algorithm for regular expression matching over strings is stateless, that is, no matching state is maintained. In contrast, in our engine a user can store necessary information in variables that can be used later in subsequence matching. The query engine is designed independent of the sequential data under analysis, and therefore it is highly adaptable to new data sources.

Our technique can be applied to analyze sequential SE data such as bug report history and source code change history. These analysis have real-world motivations. Considering the following example that was given by a test team leader in Microsoft during an interview with the authors:

> *The bug report histories are recorded in bug tracking systems, but currently no tool is available to analyze them. This type of information can be important. For example, a bug can be reopened multiple times after it is fixed before it is eventually closed, and such a bug often indicates a problematic bug, or the ignorance of the assigned developers.*

There is no easy way for current bug tracking systems to retrieve such bug reports. Often a user can query bug reports of which a bug status was changed to the current value, but cannot query if the bug status was ever changed to a temporary value in a sequence of revisions.

In our approach, we can write the query in Figure 1, with one auxiliary variable, three revision events of interest, a pattern body described in the syntax of regular expressions (specifying the pattern of "initially resolved, then reopened one or more times, and finally closed"), and a where clause specifying that we are only interested in bug reports that are reopened more than twice.

```
int counter = 0;
/* an event defines an element of interest
 * in a sequence, indicated by the expression
 * surrounded by the brackets.*/
event resolved (status == "resolved") {}
event reopened (status == "reopened") {
// an action to update the variable
  counter = counter + 1;
}
event closed (status == "closed") {}
// the following is the pattern body
resolved reopened+ closed $ where counter > 2
```

Figure 1: *SPQL* query to retrieve bug reports which are resolved first, then reopened more than twice and finally closed

Current bug tracking systems do not support this query. Even if we have direct access to the back-end database, it is hard for us to express this query in SQL, as the query involves temporal properties among rows of the table, exceeding the expressiveness of SQL. In order to fulfill the same task, we need to write a program to directly interact with the back-end database. Assume that the revision history of each report is already read into a list *seq*, we need to manually write the following Java code, which implements exactly the behavior of the *SPQL* query in Figure 1. Note that the code is simplified for illustration purpose. Besides the function *match*, the whole program should also implement other functions to retrieve the bug reports into memory by executing SQL statements, and construct a list of revision history for each bug report.

```
boolean match(List<Revision> seq) {
  int index = -1;
  boolean fixed = false;
  while (++index < seq.size()) {
    Revision r = seq.get(index);
    if (r.field.equals("status")
            && r.new_value.equals("resolved")) {
      fixed = true;
      break;
    }
  }
// whether the report is fixed before?
  if (!fixed) return false;
  int counter = 0;
  while (++index < seq.size()) {
    Revision r = seq.get(index);
    if (r.field.equals("status")
            && r.new_value.equals("reopened")) {
      ++counter;
    }
  }
// whether it is reopened more than twice?
  if (counter <= 2) return false;
  Revision last = seq.get(seq.size() - 1);
// whether the last status is closed?
  return last.field.equals("status")
            && last.new_value.equals("closed");
}
```

Figure 2: A sample Java program implementing the query in Figure 1 (partial)

We have implemented the proposed language and its runtime engine, and applied it to analyze the evolution of 181,213 Eclipse bug reports between 2001 and 2007, and 323,989 code revisions of Android between 2010 and 2012. The case studies reveal that the proposed approach can enable stakeholders to investigate complex sequential SE data in a repository with a few lines of code, and the matching overhead is low. The results show that the proposed language and tool can complement the searching functionality of the existing software repository management systems, and can help improve software engineering practice.

We summarize our contributions as follows:

1. We propose a new pattern-matching based approach for analyzing sequential software engineering data. We design a query language *SPQL* and a matching algorithm for retrieving the data records that satisfy specified sequential patterns.

2. We implement an efficient runtime engine to support *SPQL*, and the retrieval overhead is negligible.

3. We explore the application scenario of the proposed approach in software development and maintenance process.

This paper is organized as follows. Section 2 describes necessary background. Section 3 introduces the syntax and the semantics of the SPQL query language. Section 4 describes the design of query engine that supports SPQL. Section 5 describes the overview of the tool implementation. Section 6

describes the case studies that illustrate the application scenarios of the proposed query tool. Section 7 surveys related work, followed by Section 8 that concludes the paper with future work.

2. BACKGROUND

Many kinds of SE data exhibit the sequential nature. A series of revisions can be made to these data. Each revision may update the values of certain fields in the data records. We refer to the current value of the data records as a *snapshot* of the evolution history. Specifically, the sequential SE data can be expressed as a series of snapshots,

$$seq = \langle e_1, e_2, \cdots, e_n \rangle$$

where e_i $(1 \leq i \leq n)$ is a snapshot and is sorted in chronological order. Notation-wise, we use $|seq|$ to denote the number of snapshots in seq, $seq[i]$ to denote the i-th snapshot e_i, and $seq[i, j]$ (where $1 \leq i \leq j \leq |seq|$) to denote the subsequence of seq $\langle e_i, e_{i+1}, \cdots, e_j \rangle$.

In this paper, we use two types of SE data, bug report history and source code change history, to illustrate the proposed query approach.

2.1 Bug Report History

Nowadays, bug tracking systems such as Bugzilla [2] and JIRA [3] have been widely adopted in software development and maintenance processes. They enable developers, testers and users to track the bug status. Initially, when a bug is found by a tester or user, he/she can file a new report in the system. If reproducible, the bug is accepted and assigned to a developer. After the bug is fixed, the tester verifies and closes it. All these status transitions are recorded in the repository. Moreover, stakeholders can update the fields of a report at any time. During the lifetime of a bug report from its being initially opened to finally closed, there is usually a series of such revisions, referred to as evolution of bug reports. Such history information is an essential indicator of software process maturity in an organization [14, 13].

Table 1 shows the evolution of the Eclipse Bug Report #1749. This bug report was originally filed on October 11 2001 and went through 7 following-up revisions. Therefore the history of bug report has 8 snapshots. Each row in the table represents one snapshot, and the changed fields are highlighted in bold. For example, in the second snapshot, the severity of the bug was promoted from *normal* to *critical* and the priority was promoted from $P3$ to $P1$.

Table 1: The evolution history of Eclipse bug report #1749, containing 8 snapshots

	Date	Product	Comp.	Severity	Pri.	Status
1	01-10-11	jdt	debug	normal	P3	new
2	01-11-28	jdt	debug	**critical**	**P1**	new
3	01-11-30	jdt	debug	critical	P1	**assigned**
4	01-11-30	**platform**	**ui**	critical	P1	assigned
5	01-11-30	platform	ui	critical	P1	**new**
6	01-11-30	platform	**debug**	critical	P1	new
7	01-11-30	platform	**ui**	critical	P1	new
8	01-12-03	platform	ui	critical	P1	**resolved**

2.2 Source Code Change History

Software is constantly evolving due to changing user requirements, new product features, bug fixes, or refactoring. Changes to software are inevitable. The developers are assigned a new feature or a defect to work on upon a change request. After coding is complete they commit the changes for integration via a version control system (such as SVN or Git). During a long period of software evolution, many developers could be involved in modifying a same file [10].

Table 2 shows a segment of the revision history of the file core/java/android/webkit/WebView.java in Android Git repository. This file went through 5 revisions. Each row represents a revision to the file. Note that the table only shows three attributes of each revision and the other attributes are hidden due to space constraint.

Table 2: A segment of the revision history of WebView.java in Android

	Date	Committer	Log
1	10-01-03	Leon Scroggins	If the DOM changes textfield ...
2	10-01-03	Nicolas Roard	layers support
3	10-01-04	Cary Clark	extract selected text from ...
4	10-01-05	Leon Scroggins	Show label information as ...
5	10-01-06	Leon Scroggins	info only when accessing a ...

3. QUERY LANGUAGE

We design a sequential pattern query language (*SPQL*), to query sequential software engineering data. In this section we introduce the syntax of *SPQL* and its dynamic semantic model.

3.1 Language Syntax

The *SPQL* language is designed to be similar to regular expression, and the underlying matching is performed based on non-deterministic finite automata (NFA). However, the expressiveness of regular expressions is limited as the matching is simple character-equality testing and stateless. In order to meet the requirements of SE data analysis, our pattern language incorporates complex matching predicates, and also introduce global variables, which can save necessary information during matching and such information can be used in the later matching tests.

Generally, a pattern definition consists of the following four components,

1. **Variable Declaration**. Variables are globally accessible in all matching predicates. They can be used to store necessary information or to evaluate matching predicates.

2. **Event Definition**. Each event definition specifies the condition under which a snapshot is matched, and the actions to update global variables if the matching is successful.

3. **Pattern Body**. The body defines the sequential pattern. The syntax is similar to regular expressions.

4. **Where Clause**. This clause specifies the last criterion whether the SE data evolves in the given pattern. After matching the pattern body against a sequence of snapshots, the global variables are updated with new values. This clause tests whether the valuation of the global variables satisfies a certain condition.

Figure 3 displays the Backus-Naur Form (BNF) grammar of our query language. As shown in the rule $\langle type \rangle$, the

⟨*type*⟩ ::= 'int' | 'string' | 'bool' | 'float'
| 'list' | 'set'

⟨*var_decl*⟩ ::= ⟨*type*⟩ ⟨*var_name*⟩ '=' ⟨*exp*⟩ ';'

⟨*field*⟩ ::= the attributes defined in the schema

⟨*arg_list*⟩ ::= (⟨*exp*⟩ (',' ⟨*exp*⟩)*)?

⟨*list_exp*⟩ ::= '[' ⟨*arg_list*⟩ ']'

⟨*set_exp*⟩ ::= '{' ⟨*arg_list*⟩ '}'

⟨*exp*⟩ ::= ⟨*int*⟩ | ⟨*string*⟩ | ⟨*bool*⟩ | ⟨*float*⟩ | ⟨*char*⟩
| ⟨*list_exp*⟩ | ⟨*set_exp*⟩
| ⟨*field*⟩ | ⟨*var_name*⟩
| ⟨*uop*⟩ ⟨*exp*⟩ | ⟨*exp*⟩ ⟨*bop*⟩ ⟨*exp*⟩
| ⟨*func_name*⟩ '(' ⟨*arg_list*⟩ ')'

⟨*stmt*⟩ ::= ⟨*var_name*⟩ '=' ⟨*exp*⟩ ';'

⟨*event*⟩ ::= 'event' ⟨*event_name*⟩ '(' ⟨*exp*⟩ ')'
'{' ⟨*stmt*⟩* '}'

⟨*regexp*⟩ ::= ⟨*event_name*⟩ | '.' | ⟨*regexp*⟩ ⟨*kleene*⟩
| ⟨*regexp*⟩ ⟨*regexp*⟩ | '(' ⟨*regexp*⟩ ')'
| ⟨*regexp*⟩ '|' ⟨*regexp*⟩

⟨*kleene*⟩ ::= '*' | '+' | '?'

⟨*where*⟩ ::= 'where' ⟨*exp*⟩

⟨*pattern*⟩ ::= ⟨*var_decl*⟩* ⟨*event*⟩+
'^'? ⟨*regexp*⟩ '$'?
⟨*where*⟩?

Figure 3: BNF Grammar of *SPQL* Language

language supports four primitive types and two aggregate types of data, types frequently used in software engineering data, matching predicates and statistics computation. A variable declaration ⟨*var_decl*⟩ specifies the type and the name of a variable with its initial value. An expression ⟨*exp*⟩ evaluates to a value from a string/boolean/number literal, a read from a variable, a field of a snapshot, compositional computation over expressions with binary operators ⟨*bop*⟩ (e.g., '+', '−', '×', '>') or unary operators ⟨*uop*⟩ (e.g., '!' and '−'), or a function call.

The language is independent of domain-specific snapshots. The production rule ⟨*field*⟩ is a list of attribute names defined in the provided schema file.

An event ⟨*event*⟩ represents a specific snapshot of interest in sequential SE data, and specifies the following two components,

- **Matching Predicate** is the expression ⟨*exp*⟩ enclosed within the brackets, the condition under which a snapshot is matched.
- **Post Action**. If ⟨*exp*⟩ is evaluated against a snapshot to *true* then the statements within the curly braces are executed subsequently. We refer to these statements as *post action*, and they update the variables according to the current values of global variables and the fields of the snapshot under matching.

The pattern body follows the syntax of regular expressions to specify temporal properties. A pattern body can be:

- a **single** event
- ⟨*regexp*⟩ ⟨*regexp*⟩, **concatenation** of pattern bodies
- ⟨*regexp*⟩ | ⟨*regexp*⟩, **alternative** between two pattern bodies
- ⟨*regexp*⟩ ⟨*kleene*⟩, **Kleene** structures

Moreover, it supports the operator '^' to force the matching to start from the very beginning of the data, '$' to force the matching to end at the last snapshot of the data, operator '.' to match any single snapshot. It also supports '?', '+' and '∗' to postfix a pattern p: p? to match zero or one p, p+ to match one or more p, and $p*$ to match zero or more p.

```
int start_day = 0;
int end_day = 0;
// event definitions
event non_p1 (priority != "P1") {}
event first (priority == "P1") {
  start_day = date;
}
event other (priority == "P1") {}
event last (priority == "P1"
                 && status == "closed") {
  end_day = date;
}

^ non_p1* first other* last $  //pattern body
where (end_day - start_day) <= 10  //where clause
```

Figure 4: A query to retrieve high-priority bug reports fixed within 10 days

Figure 4 shows an example query. It retrieves bug reports, which are fixed within 10 days after their priorities become the highest level $P1$. Given an evolution history of a bug report $h = \langle e_1, e_2, \cdots, e_n \rangle$, this pattern requires the following:

1. the matching should start from the first snapshot e_1 of h (indicated by "^"),
2. zero or more snapshots from the start of h (i.e., none or $h[1, i]$ where $0 < i < |h|$) can be with non-$P1$ priority (indicated by "non_p1*"),
3. the rest of the snapshots must be with $P1$ priority and the last snapshot must be *closed* (indicated by "first other* last$")
4. the global variable *start_day* stores the date when the bug report is assigned with $P1$, and the variable *end_day* stores the date when the bug report is closed.
5. the *where* clause specifies that the time span between the date when the report becomes $P1$ priority and the date when it is closed should not exceed 10 days.

3.2 Dynamic Semantic Model

In order to facilitate describing the execution model, we first formulate necessary concepts for matching predicates, post actions and where clause defined in pattern queries.

Given a pattern query q, let *Pred* denote the set of matching predicates, *Act* denote the set of post actions defined in the event definitions, and *where* denote its *where* clause.

DEFINITION 1 (VALUATION). *Let $\langle v_1, \cdots, v_n \rangle$ be the list of n variables defined in q, and the cartesian product of the variable domains is denoted as $\mathcal{D}(q)$,*

$$\mathcal{D}(q) = dom(v_1) \times \cdots \times dom(v_n)$$

then a valuation of q is an element of $\mathcal{D}(q)$

$$\langle val_1, \cdots, val_n \rangle \in \mathcal{D}(q)$$

where $\forall i \in [1, n]$, $dom(v_i)$ denotes the domain of variable v_i and $val_i \in dom(v_i)$.

Note that a valuation is not a value of a single variable. It is a vector defined over the whole set of global variables, and an element in the vector is a value corresponding to a distinct variable.

Let Σ denote the set of snapshots, the matching predicates $pred \in Pred$ can be formally defined as

$$pred : \Sigma \times \mathcal{D}(q) \rightarrow \{true, false\} \quad \text{(Matching Predicate)}$$

where *true* represents that a snapshot $s \in \Sigma$ matches the predicate *pred* under the variable valuation $v \in \mathcal{D}(q)$, *false* otherwise. Similarly, a post action $act \in Act$ can be formally defined as

$$act : \Sigma \times \mathcal{D}(q) \rightarrow \mathcal{D}(q) \quad \text{(Post Action)}$$

which takes as input a snapshot and a variable valuation and returns a new valuation after computation. The *where* clause can be formalized as

$$where : \mathcal{D}(q) \rightarrow \{true, false\} \quad \text{(Where Clause)}$$

testing whether a variable valuation satisfies a criterion.

The dynamic semantics of our query language is based on the interpretation of NFA. Each pattern query is compiled into an execution model. The pattern body is translated into an NFA by Thompson's method [29], and the matching predicates and post actions in the event definitions are compiled as the labels of the transitions.

DEFINITION 2 (NFA). *The NFA corresponding to the pattern body is defined as a tuple $F = \langle Q, E, \tau, \alpha, \theta, \varphi \rangle$, where*

- *Q: a set of vertices, referred to as* states
- *$E : Q \times Q$: a set of directed transitions between states*
- *$\tau : E \rightarrow Pred \cup \{\epsilon\}$, a function assigning a matching predicate in Pred to a transition. An ϵ transition can be taken without consuming a bug report snapshot from an evolution history. It is introduced when we compile pattern bodies containing* alternatives *or* kleene *structures by Thompson's method [29].*
- *$\alpha : E \rightarrow Act \cup \{\varepsilon\}$, a function associating each transition with an action. An ε action simply returns the input valuation without modification.*
- *θ: the start state of this NFA.*
- *φ: the accepting/final state of this NFA.*

The special symbols '^' and '$' are not encoded into the model, but handled in Algorithm 1.

Figure 5 shows the execution model compiled from the query in Figure 4. Graphically, we represent a transition as ($s \xrightarrow[act]{pred} t$), where s is the source state, t is the sink state, *pred* above the arrow is the matching predicate, and *act* below the arrow is the post action. If *pred* is evaluated to true, then the action *act* is executed; otherwise not. If *pred* or *act* is empty, we just leave it blank.

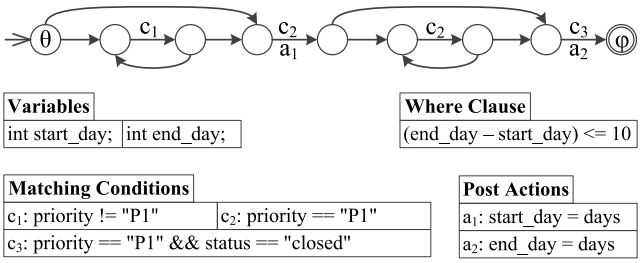

Figure 5: The NFA compiled from the query in Figure 4. This NFA is associated with two variables and one where clause. The labels for ϵ predicates or ε post actions are omitted.

Similar to the standard interpretation of NFA with strings, given a sequence of bug report snapshots $\langle e_1, e_2, \cdots, e_n \rangle$, we interpret the NFA from the start state θ with the sequence, and at each non-ϵ transition ($s \xrightarrow[act]{pred} t$) we consume a snapshot e from the sequence and test whether the current valuation v of global variables and e can satisfy $pred(e, v)$. If yes, we set the global variable valuation with $act(e, v)$. During matching, if the final state φ is reached and the valuation of global variables satisfies the *where* clause, we report that the given snapshot sequence matches the pattern query.

4. QUERY ENGINE

This section describes the algorithm underneath the runtime engine for implementing query matching testing. The proposed matching algorithm is based on the concept of NFA simulation. Nonetheless different from the classic NFA simulation for regular expressions over simple strings, *SPQL* queries support global variables which can not only participate in the matching predicates but also be updated during matching. Section 4.2 describes the details of this pattern matching algorithm, and Section 4.3 analyzes its time and space complexity.

4.1 Main Algorithm

Algorithm 1 lists the main algorithm for the pattern query matching. Given a set of sequential SE data *seq* and a pattern query q, this algorithm returns *true* if *seq* matches q. It first compiles the query into an NFA *nfa*, and then delegates the matching test to the function *Match* defined in Algorithm 2. Generally, *Match* tests whether a subsequence $seq[i, j]$ (where $1 \leq i \leq j \leq |seq|$) can be accepted by *nfa*.

If q has been specified that the matching should start from the head of *seq* with the symbol '^', then *Match* is called at line 4 with the whole *seq*. Otherwise, the algorithm iterates through *seq* at line 5, and invokes *Match* for each subsequence $seq[i, |seq|]$ until it finds a match.

The query q is likely specified with the symbol '$' that the matching must end at the tail of *seq*, i.e., *matchingEnd*, and this part is handled in the function *Match*.

Algorithm 1: Main Algorithm

Input: *seq*, the sequential SE data
Input: q, the pattern query
Output: *true* if *seq* matches q, *false* otherwise
1 let *nfa* denote the NFA compiled from the query q
2 let *matchingEnd* denote whether q has '$'
3 **if** q *has* '^' **then**
    /* matching starts from the head of *seq* */
4   **return** *Match*(*nfa*, *seq*, *matchingEnd*)
5 **for** $i := 1$ *to* $|seq|$ **do**
6   **if** *Match*(*nfa*, $seq[i, |seq|]$, *matchingEnd*) **then**
7   **return true**

4.2 Matching Algorithm

In classic regular expressions, each pattern is first translated into an NFA before matching. Due to the nondeterminism of NFA, there are three ways to test whether a string can be accepted by the NFA. The first way is to convert an NFA to a deterministic finite automata (DFA), as for a character at each state in DFA there is at most one available transition to take as the next step. The second way is backtracking, such as the implementation of regular expression in Perl programming language. It searches for a valid path from the start state of NFA to the final state. If a path becomes not feasible, it backtracks to a previous state and branches to another path. Since the number of paths in an NFA can be exponential, this implementation may suffer from severe performance overhead. The last approach is multiple-state simulation of NFA proposed in [29]. It maintains a set S of reached states. Initially S contains the start state θ. Given S and a character, the algorithm searches the NFA for a set S' of next states reachable from the states in S. In the next iteration, the algorithm uses S' as the starting point to search for the reachable states. It repeats this process until the accepting state is reached. Compared to the backtracking approach, as the multiple-state simulation does not need to maintain a path, it is much efficient. Suppose the length of the string is m and the number of states in NFA is n, then the time complexity is $O(mn)$.

In our runtime engine, we opt for the simulation approach mainly for its efficiency. The reason that we do not choose the first approach is the difficulty in converting the NFA of *SPQL* to DFA. Different from regular expressions, our query language involves complex matching predicates, and it is possible that two matching predicates may partially overlap. For example, say there is one NFA state with two outgoing matching predicates , $x > 0$ and $x < 2$. To convert this NFA, we need to analyze these two predicates and separate them into three disjoint predicates $x \leq 0$, $0 < x < 2$ and $x \geq 2$. This separation process is complex and expensive as a matching predicate can be a compound of multiple predicates. Moreover, a matching predicate can be associated with a post action, and this further complicates the DFA approach.

Algorithm 2 implements the core matching algorithm. It takes as input an NFA *nfa* compiled from the query, a *list* of report snapshots and a boolean value *matchingEnd* indicating whether the matching should end at the tail of the *list*. If *matchingEnd* is true, this algorithm returns *true* if the whole *list* can be accepted by *nfa*. If *matchingEnd* is false, it returns *true* if there exists a subsequence $list[1, i]$ ($i \geq 1$) which can be accepted by *nfa*.

Algorithm 2: *Match*(*nfa*, *list*, *matchingEnd*)

Input: *nfa*, NFA compiled from the query
Input: *list*, a list of snapshots
Input: *matchingEnd*, whether to match the end of *list*
Output: whether *list* matches q
1 $prev := \{(\theta, init)\}$
  /* where θ is the start state of *nfa* and *init* is the initial valuation of global variables */
2 **for** $i := 1$ *to* $|list|$ **do**
3   $snapshot := list[i]$
4   $current := \varnothing$
5   **foreach** $(state, valuation)$ *in prev* **do**
6   *trans* := the set of non-ϵ transitions reachable from *state* via 0 or more ϵ transitions
7   **foreach** $s \xrightarrow[act]{pred} t$ *in trans* **do**
8   **if** $pred(snapshot, valuation) =$ **false then**
          /* *snapshot* and *valuation* dissatisfy the matching predicate *pred* */
9   **continue**
10      $valuation' := act(snapshot, valuation)$
11      $current := current \cup \{(t, valuation')\}$
12   **if** $t \neq \varphi \vee \neg where(valuation')$ **then**
          /* t is not final state or *where* clause is not satisfied */
13   **continue**
        /* t is the final state and *where* clause is satisfied */
14   **if** *matchingEnd* **then**
15   **if** $i = |list|$ **then return true**
16   **else**
17   **return true**
18    $prev := current$
19 **return false**

As aforementioned, the multiple-state simulation in regular expression matching maintains a set of reached NFA states. Differently this algorithm maintains a set of pairs, as our approach supports global variables. In particular in each pair $(state, valuation)$, the first element is an NFA *state*, and the second element is a *valuation* of global variables. A pair means that there exists at least one path from the start state θ to the *state*, such that the list of post actions associated to this path updates the global variables to the *valuation*.

The set *prev* declared at line 1 is stores the reached state-valuation pairs. Initially, it only contains the pair $(\theta, init)$ where θ is the start state of *nfa* and *init* is the initial valuation of global variables. The loop at line 2 iterates through the *list* from the head to the tail and advances the matching in *nfa*. The set *current* declared at line 4 is the set storing the next state-valuation pairs by consuming the snapshot $list[i]$, initially $\varnothing$. The loop at line 5 implements the logic to propagate *current* from *prev*. After the propagation is done, the set *prev* is *cleared* and assigned with the content of *current* at line 18 for the next iteration of propagation.

The following generally describes the process to propagate *current*. For each pair $(state, valuation)$ in *prev* (i.e. line 5),

the algorithm searches *nfa* to collect all the non-ϵ transitions, which are reachable from the *state* via zero or more ϵ transitions. For each $s \xrightarrow[act]{pred} t$ of such transitions, if the current *snapshot* and the variable *valuation* can satisfy the matching predicate *pred*, then we compute a new variable valuation *valuation'* by executing the associated post action *act* over *snapshot* and the old *valuation*. (i.e. line 10).

4.3 Complexity Analysis

This section analyzes the complexity of Algorithm 2. Given a sequence *list* of bug report snapshots, a *SPQL* query q and the NFA *nfa* compiled from q, we use the symbols in Table 3 to facilitate the analysis of time and space complexity.

Table 3: Symbols for Complexity Analysis

L	the length of *list*, $L = \|list\|$
m	the number of non-ϵ transitions in *nfa*
$\mathcal{D}(q)$	as defined in Definition 1, it denotes the set of all possible valuations for variables in the query q
c	We assume that for each NFA state, it has a constant number $c > 0$ of reachable non-ϵ transitions via 0 or more ϵ transitions, i.e., $\|trans\| = c$ at line 6. In the worse case, $c = m$.

Based on the loop at line 5, for each pair in *prev*, it is possible that the c non-ϵ transitions in *trans* at line 7 generate c distinct state-valuation pairs. Assuming *prev* has x pairs, then after the loop at line 5, *prev* will have $(c \times x)$ pairs. Initially, the set *prev* contains only 1 element at line 1. For the first snapshot in *list*, *prev* grows to $(c \times 1)$ elements. For the second snapshot, *prev* grows to c^2 elements. Then c^3, c^4, $\cdots$ and for the last snapshot, *prev* has c^L elements in the worst case. The total number of pairs ever in *prev* is $O(c + c^2 + \cdots + c^L)$. Hence, the time complexity is $O(L)$ when $c = 1$ and is $O(c^L)$ when $c > 1$.

The space complexity is $O(c^L)$, which is the maximum size of the set *prev*. As each element in *prev* is a pair of an NFA state and a valuation, the size is also bounded by $m \times \mathcal{D}(q)$. Again, similar to time complexity, the space complexity is also manageable.

5. TOOL IMPLEMENTATION

We have implemented the proposed language *SPQL* and the runtime engine in Java. Figure 6 displays the overall framework of our tool. Its core is the query engine, which accepts as input *SPQL* queries from users, searches the underlying databases, and returns the matching results to the users.

The tool is designed to be independent of specific SE data sources, and is adaptable to new data sources and application domains (e.g., behavior logs). When applying *SPQL* to a domain, users only need to prepare the following two components:

- Schema, which describes the attributes of data in a domain. Each attribute has an associated type and name. For example, the schema of bug report includes creation date, product, severity, priority, etc.
- Adaptor: The query engine abstracts and models data as sequences of snapshots. An adaptor is an intermediate layer between this abstraction and the domain-specific database. It allows the engine to access the database without knowing the details and design of the database. The adaptor must be implemented consistently with the corresponding schema. That is, when the adaptor converts a data record in the database into a snapshot in a sequence, all the attributes of the snapshot must conform to the schema.

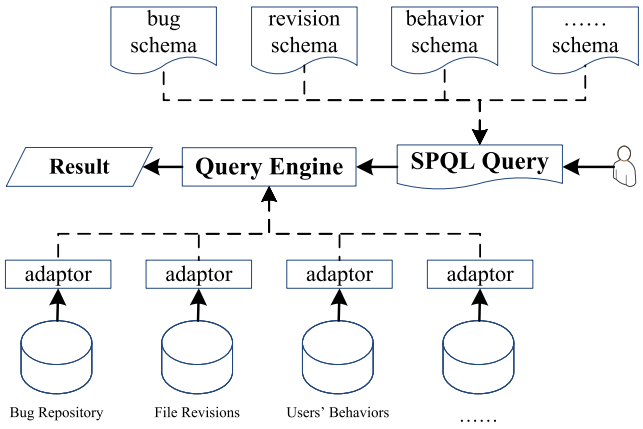

Figure 6: The overall framework of our tool

6. APPLICATION SCENARIOS

In order to evaluate the effectiveness and efficiency of the proposed approach in querying different types of SE data, we describe two studies in this section and show how the proposed approach can be used to accomplish them.

The case studies are conducted on a laptop with Intel Core i7 @ 2.70GHz and 8 Gb memory.

6.1 Detecting Violations in Bug Management Process

In this task, we use *SPQL* to express several violation patterns in the bug management process. These violation patterns are learnt from historical process execution data using a contrasting pattern mining algorithm [26]. They are meaningful to the bug management process of a specific commercial bank introduced in [14, 13]. However, we believe that even though they may not be deemed as anomalous to bug management processes of other organizations, they can still provide some hints to further improve the process quality. Moreover, our query language is general, and stakeholders can always deploy their own organization-specific violation patterns to monitor their bug-handling process.

In this study, we collect the Eclipse bug reports from 2001 to 2007. For each bug report, its evolution history is recovered by crawling its modification history page. After removing the reports with no revision history or required fields (*product*, *component*, *severity* etc.), we get 181,213 bug reports in total.

Table 4: Distribution of Evolution History Length

Range	$[0, 5)$	$[5, 10)$	$[10, 20)$	$[20, 30)$	$[30, 40]$
Number	138903	39652	2619	36	3
Percent.	76.65%	21.88%	1.45%	0.02%	0%

Table 4 shows the statistics on the number of bug report snapshots. The first row is the range of number, the second

row is the total number of bug reports within that range, and the last row is the percentage. The results show that around 23.35 % bug reports' status have been updated more than 5 times (5 snapshots). The maximum number of bug report snapshots is 40.

6.1.1 Closed Right After Assigned

When a bug report is assigned to a developer, the developer needs to set the report status to *resolved* before closes it. A violation pattern is that a bug report is closed right after it is assigned [14, 13] without any other status in between, which is not allowed by the Eclipse Bugzilla Usage Guide [1] either. We use the following query to fulfill this task.

```
event closed(status == "closed") {}
event assigned(status == "assigned") {}
assigned closed
```

This query takes 0.78 second to finish, and returns 4 matching reports. We manually read the returned bug reports and confirmed their validity. For example, Bug Report #173785 was initially *new*, then *assigned* to a developer and later directly *closed* without being resolved. According to the Eclipse Bugzilla Usage Guide [1], there should exist at least one state of *resolved* before the bug is closed. Therefore, the handling process of these 4 bug reports violates the guidelines.

We could integrate our tool to a bug tracking system to automatically detect the violation patterns in a bug repository. Such an integration can also automatically alert the users when they try to directly close the bug without resolving it, thus avoiding a process violation. This could be an interesting application of our approach in future.

6.1.2 Product Change Analysis

In this task, we analyze the following problem: how many bug reports' *Product* field is changed? The Product field is important for bug triage [5, 17]. It is set by the bug reporter and supposed not to change frequenly over time. However, the reporter may not thoroughly understand the bug at the beginning, and may assign the bug to a wrong product. When developers get to know the bug better, they can change this field accordingly. We use the following query to fulfill this task.

```
string prev_product = "";
event first (true) {prev_product = product;}
event change (prev_product != product) {
    prev_product = product;
}
event no_change(prev_product == product){}
^first (no_change* change* no_change*) *$
```

Table 5: Distribution of Product Changes

Changes	No.	Pct.	Changes	No.	Pct.
0	171489	94.63%	5	13	0.01%
1	8503	4.69%	6	1	0
2	977	0.54%	7	0	0
3	182	0.10%	8	1	0
4	47	0.03%	>8	0	0

The distribution of product changes returned by this query is displayed in Table 5. The column *Changes* shows the number of changes made to the *product* field, the column *No.* shows the absolute number of reports with the corresponding changes, and the column *Pct.* shows the percentage. The results show that around 5.5 % of bug reports (9,724) have their Product status changed at least once.

6.2 Analyzing File Revision History

In this section, we show how *SPQL* can be applied to analyze file revision history. Previous empirical studies found that when more people work on a file, it has more failures [9, 10]. Thus it is interesting to know the number of people who have ever touched a particular file. This could help project teams identify potential problematic code and design better source code change policies.

Table 6: Distribution of Revision History Length

Range	$[0, 10)$	$[10, 20)$	$[20, 30)$	$[30, 40)$	$[40, \infty)$
Number	266448	7138	3654	1570	4177
Percent.	94.16%	2.52%	1.29%	0.55%	1.48%

We collected the revision history of the Android source code files between 2010 and 2012. There are 282,987 files, and each file is associated with a sequence of revisions. In total, there are 323,989 code revisions. Table 6 shows the statistics of the length of each file revision history. There are 5.84 % files that have gone through more than 10 revisions.

We apply *SPQL* to find out which files are revised by multiple developers within a period of time. The following query searches for the Android files that are modified by more than 10 developers within a month. The variable *authors* stores the committer names and the *where* clause tests whether the retrieved segment of the file revision sequence satisfies the criterion.

```
int first_day = 0;
int last_day = 0;
set authors = {};

event first_event(true) {
  first_day = committer_date;
  authors = add(authors, committer);
}

event other_event(true) {
  last_day = committer_date;
  authors = add(authors, committer);
}

first_event other_event+
where (last_day - first_day) < 31
                          && size(authors) > 10
```

It takes 43 seconds to execute this query, and 14 files are returned, including the file WebView.java in Table 2. Compared to the experiments on Eclipse bug report evolution, querying Android file revision history takes longer time. This is because of the larger amount of the file revision data.

Figure 7 shows the details of the runtime performance. The x-axis is the length of the revision sequences, and the y-axis is the average time taken to match the query against the sequences. From the plot, we can draw the conclusion that in

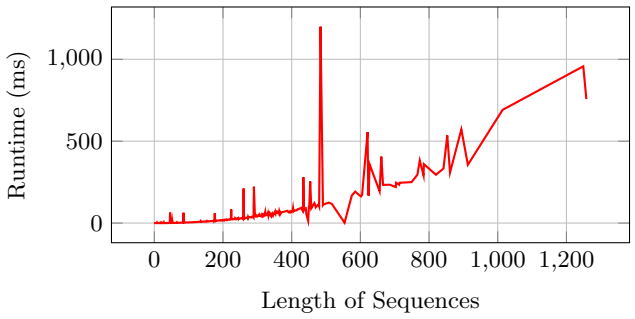

Figure 7: Runtime (millisecond) of matching w.r.t length of revision sequences

general, the runtime performance increases with the length of sequences, and the maximum time is around 1 second even for sequences containing more than 1,000 snapshots.

6.3 Discussions

Our language is a general language that can be applied to many application scenarios. We have shown how *SPQL* can be used to analyze temporal properties beneath the bug report and file revision data. All the queries are concise, and most of them are within 10 lines of code. The following discusses how *SPQL* is related to the SQL-based and programming-based approaches.

SQL. The backend of a bug repository is a relational database. Assume we have direct access to the backend database, we could perform queries in SQL. However, we do not believe this is a good solution. First, as illustrated by Figure 1, not all queries, especially those involving temporal properties, can be expressed in SQL. Second, the bug tracking system needs to expose the implementation details to the user and the users are required to master advanced SQL features such as table joins and group clauses.

Programming-based approach. All of the queries described in this paper can be answered via programming. However, it could cost much more time than writing a short *SPQL* query. For example, the program in Figure 2 is much longer than the *SPQL* query in Figure 1. Our approach provides users a simple and lightweight method for answering complex queries.

6.4 Threats to Validity

Our case studies are subject to two threats to validity.

Internal Validity. The first is threat to internal validity, which is the possible flaws in the implementation of the *SPQL* runtime engine. To minimize this threat, we first wrote and executed considerable unit and system test cases to explore different aspects of the runtime engine. Second, in addition to the queries used in this paper, we also wrote other queries to investigate bug report history. Lastly, we implemented a variant of the matching algorithm Algorithm 2, specially optimized for queries without variables. This variant runs faster, and also serves as a reference to check the correctness of the implementation of Algorithm 2.

External Validity. Our empirical evaluation is subject to threat to external validity, with regard to the concern that the experimental results might not generalize to other queries or datasets. Regarding the runtime, as the evolution history of a bug report in the Eclipse repository is usually not long, our engine can respond quickly within 2 seconds. The experiment on Android file revision history takes longer time, which is mainly due to the longer sequences and the retrieval of all qualified segments within a sequence. We believe this could be alleviated by confining the length of a segment to search, as people are mostly interested in revisions within a finite interval of time. The runtime performance also depends on the complexity of queries.

7. RELATED WORK

This section surveys three lines of research related to our work: querying software repository, software process evaluation, and program matching and monitoring.

7.1 Querying Software Repository

Many software repository management tools provide users with searching functionalities. For example, in Bugzilla [2] and JIRA [3], a user can specify predicates on fields in bug reports and the two systems return the bug reports satisfying the conditions to the user. A predicate could be like "the field *summary* should contain a specific word" or "the field *product* should be a specific value". Both systems also provide support for retrieving bug reports with evolution history predicates. Especially JIRA has an advanced and expressive query language JQL [4] to retrieve bug reports with predicates on revision histories. For example, JQL has a keyword "*was*" to retrieve bug reports which currently have or previously had the specified value for the specified field. The other keyword "*changed*" is used to retrieve bug reports having a value which had been changed for the specified field.

From an evolutionary perspective, the expressiveness of JQL is still limited. It only supports querying the modification in a single bug report revision, but cannot capture temporal properties involving multiple revisions, such as the analysis task in Figure 1. Moreover, it does not support variables in queries, and therefore it is impossible to collect information during matching for complex analysis tasks.

In contrast, our query language *SPQL* supports global variables and is able to express complex temporal patterns involving multiple revisions. The variables and the *where* clause make *SPQL* capable of fulfilling considerable types of analysis tasks, as shown in our case studies. Regarding the implementation, JQL translates queries into SQL statements and uses its database system as the runtime engine, whereas *SPQL* bases its runtime engine on NFA to capture temporal properties in a sequence of revisions. However, *SPQL* is not a replacement but a good complement of JQL. We believe that incorporating *SPQL* into the current bug tracking systems will further improve their searching functionalities.

Kenyon is a system that extracts source code change histories from SCM systems such as CVS and Subversion [8]. Kenyon automatically checks out the source code for each revision and extracts change information such as the change log, author, change date, etc. Kim et al. proposed TA-RE as an exchange language for mining software repositories [19]. It is envisioned that TA-RE can support the representation and integration of software change, transaction and project data mined from SCM (software configuration management) systems. Researchers have also proposed semantic web based approaches to integrate and query software engineering data collected from different software

repositories [20, 18]. However, the above related work does not provide features for users to perform temporal properties related queries. Hindle and German [16] proposed SCQL, a first-order temporal logic based query language for source control repositories. SCQL is able to represent the temporal relationships such as "before" and "after". SCQL requires a formal model of source control repositories, while our SPQL specifies the temporal properties using regular expressions. SPQL is also enhanced with variables and statements to store and manipulate matching states.

In particular, our work is different from DebugAdvisor [6] and BOA [15]. In DebugAdvisor, when a developer is diagnosing a bug, he/she can issue a fat query containing natural language description of the bug, core dumps, execution traces etc. to the system, and the system returns all the associated artifacts including patches of similar bugs which have been fixed before. In this way, the developer may find a clue or hint quickly to solve the bug. BOA is a language and infrastructure to ease the analysis of software repositories. Users can write small BOA programs to implement complex analysis tasks in research on software repository mining. Similar to these work, our work also proposes a language and tool for querying software repository. However, we focus more on the temporal properties and enable users to analyze sequential SE data.

7.2 Software Process Evaluation

Chen et al. [14, 13] propose a machine learning approach to evaluating a software defect management process. In detail, the lifetime of a bug report corresponds to an execution of defect management process, and such an execution is represented as a sequence of status transitions of the bug report. Based on the process specification and process evaluators' expertise, each execution is classified as *normal* or *anomalous*. Their approach first learns a binary classification model from the historical evaluation results, and then applies the model to classify future process executions. Sun et al. [26] further improves this approach by mining explicit evaluation rules from evaluation history. Each rule is capable of explaining why a process execution is evaluated as *normal* or *anomalous*.

Our query language can be used to express the mined evaluation rules as violation patterns, which can be deployed to monitor the running process execution. If an execution matches a violation pattern, the defect process monitor can inform stakeholders of the anomalous state of the execution.

7.3 Program Matching and Monitoring

A line of remotely related research is program matching and monitoring. Brunel et al. [11] propose an extension to computation tree logic with variables and use this logic extension to match code snippets in system code of Linux in order to perform collateral code evolutions [25]. Martin et al. [21] propose Program Query Language to check whether runtime behavior of an application conforms to specified design rules. Meredith et al. [24] propose to use parametric context free grammars to express desired program behaviors, and base the runtime monitor on LR(1) parsing algorithm to check property conformance.

Our work is different from these studies. First the application domains are different. Second, the declaration of variables in *SPQL* is more flexible, and the variables can be used to aggregate information. But those variables used in these studies are mainly for selecting runtime events of interest.

8. CONCLUSIONS AND FUTURE WORK

Many types of software engineering data have the sequential nature and preserve temporal properties. In this paper, we propose a pattern-based approach to effectively and efficiently analyzing sequential software engineering data. We have presented a query language *SPQL* as well as the associated query engine. We have shown that the proposed approach can be used to analyze different types of sequential SE data, including bug evolution, source code revisions, and user logs. The analysis results can help users better understand a software process and identify violations.

There are many interesting directions we would like to explore in future work. For example:

Parametric Queries. In a parametric query, a user can use parameters in initial values of variable declarations, matching predicates, post actions and *where* clauses. Before executing the query, the user instantiates the parameters, and the parametric query becomes non-parametric. With this feature, a category of analysis tasks can be further simplified by varying parameters.

More Query Languages. It would be interesting to support more types of query languages. Similar to [12], we can allow users to write queries in different logics. For example, a query can be written in regular expressions, linear temporal logic, computation tree logic or even more expressive context free grammars. Due to different degrees of expressiveness and different design purposes of these logics, this feature will provide users with more freedom for expressing their intentions.

Intgeration. Our approach can be integrated into existing bug tracking systems and version control systems to help users identify problems in software development and maintenance process.

User Studies. Although *SPQL* is designed to be a lightweight query language and the tool is designed to be adaptable to new data sources, the usability of the language and tool should be evaluated. In future, we will perform user studies to further evaluate the effectiveness of the proposed approach in practice.

9. ACKNOWLEDGMENTS

We are grateful to Dr. Shao Jie Zhang at Singapore University of Technology and Design for her insightful comments on the initial draft. We also thank the anonymous reviewers for their valuable comments and suggestions.

10. REFERENCES

[1] Eclipse Bugzilla Usage Guide. `http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use`, Aug. 2013.

[2] Bugzilla. `http://www.bugzilla.org/`, Aug. 2013.

[3] Jira. `https://www.atlassian.com/software/jira`, Aug. 2013.

[4] JIRA Query Language. `https://confluence.atlassian.com/display/JIRA/Advanced+Searching`, Aug. 2013.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Who Should Fix This Bug? In *ICSE*, pages 361–370, 2006.

[6] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A Recommender System for Debugging. In *ESEC/FSE*, pages 373–382, 2009.

[7] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What Makes a Good Bug Report? In *ESEC/FSE*, pages 308–318, 2008.

[8] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating Software Evolution Research with Kenyon. In *ESEC/FSE*, pages 177–186, 2005.

[9] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality?: An empirical case study of windows vista. *Commun. ACM*, 52(8):85–93, Aug. 2009.

[10] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don'T Touch My Code!: Examining the Effects of Ownership on Software Quality. In *ESEC/FSE*, pages 4–14, 2011.

[11] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking. In *POPL*, pages 114–126, 2009.

[12] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *TACAS*, pages 546–550, 2005.

[13] N. Chen, S. Hoi, and X. Xiao. Software Process Evaluation: a Machine Learning Framework with Application to Defect Management Process. *Empirical Software Engineering*, pages 1–34, 2013.

[14] N. Chen, S. C.-H. Hoi, and X. Xiao. Software Process Evaluation: A Machine Learning Approach. In *ASE*, pages 333–342, 2011.

[15] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE*, pages 422–431, 2013.

[16] 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*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[17] G. Jeong, S. Kim, and T. Zimmermann. Improving Bug Triage with Bug Tossing Graphs. In *ESEC/FSE*, pages 111–120, 2009.

[18] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparol and a software evolution ontology. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 10–, 2007.

[19] S. Kim, T. Zimmermann, M. Kim, A. Hassan, A. Mockus, T. Girba, M. Pinzger, E. J. Whitehead, Jr., and A. Zeller. Ta-re: An exchange language for mining software repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 22–25, 2006.

[20] Y.-F. Li and H. Zhang. Integrating software engineering data using semantic web technologies. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 211–214, 2011.

[21] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *OOPSLA*, pages 365–383, 2005.

[22] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, Jan. 2007.

[23] T. Menzies and A. Marcus. Automated Severity Assessment of Software Defect Reports. In *ICSM*, pages 346–355, 2008.

[24] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient Monitoring of Parametric Context-Free Patterns. In *ASE*, pages 148–157, 2008.

[25] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Eurosys*, pages 247–260, 2008.

[26] C. Sun, J. Du, N. Chen, S.-C. Khoo, and Y. Yang. Mining Explicit Rules for Software Process Evaluation. In *ICSSP*, pages 118–125, 2013.

[27] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards More Accurate Retrieval of Duplicate Bug Reports. In *ASE*, pages 253–262, Nov 2011.

[28] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In *ICSE*, pages 45–54, 2010.

[29] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[30] Y. Tian, D. Lo, and C. Sun. Information Retrieval based Nearest Neighbor Classification for Fine-grained Bug Severity Prediction. In *WCRE*, pages 215–224, 2012.

[31] Y. Tian, D. Lo, and C. Sun. DRONE: Predicting Priority of Reported Bugs by Multi-Factor Analysis. In *ICSM*, pages 200–209. IEEE, 2013.

[32] J. Wang and H. Zhang. Predicting Defect Numbers based on Defect State Transition Models. In *ESEM*, pages 191–200, 2012.

[33] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. ReLink: Recovering Links Between Bugs and Changes. In *ESEC/FSE*, pages 15–25, 2011.

[34] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, 2007.