Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1579

An exploratory study of the evolution of communicated information about the execution of large software systems

Weiyi Shang[1,*,†], Zhen Ming Jiang[1], Bram Adams[2], Ahmed E. Hassan[1], Michael W. Godfrey[3], Mohamed Nasser[4] and Parminder Flora[4]

[1]*School of Computing, Queen's University, Kingston, Ontario, Canada*
[2]*Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, Montréal, Québec, Canada*
[3]*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada*
[4]*Performance Engineering, Research In Motion (RIM), Waterloo, Ontario, Canada*

SUMMARY

Substantial research in software engineering focuses on understanding the dynamic nature of software systems in order to improve software maintenance and program comprehension. This research typically makes use of automated instrumentation and profiling techniques after the fact, that is, without considering domain knowledge. In this paper, we examine another source of dynamic information that is generated from statements that have been inserted into the code base during development to draw the system administrators' attention to important run-time phenomena. We call this source communicated information (CI). Examples of CI include execution logs and system events. The availability of CI has sparked the development of an ecosystem of *Log Processing Apps* (*LPA*s) that surround the software system under analysis to monitor and document various run-time constraints. The dependence of LPAs on the timeliness, accuracy and granularity of the CI means that it is important to understand the nature of CI and how it evolves over time, both qualitatively and quantitatively. Yet, to our knowledge, little empirical analysis has been performed on CI and its evolution. In a case study on two large open source and one industrial software systems, we explore the evolution of CI by mining the execution logs of these systems and the logging statements in the source code. Our study illustrates the need for better traceability between CI and the LPAs that analyze the CI. In particular, we find that the CI changes at a high rate across versions, which could lead to fragile LPAs. We found that up to 70% of these changes could have been avoided and the impact of 15% to 80% of the changes can be controlled through the use of robust analysis techniques by LPAs. We also found that LPAs that track implementation-level CI (e.g. performance analysis) and the LPAs that monitor error messages (system health monitoring) are more fragile than LPAs that track domain-level CI (e.g. workload modelling), because the latter CI tends to be long-lived. Copyright © 2013 John Wiley & Sons, Ltd.

Received 31 March 2012; Revised 17 September 2012; Accepted 16 November 2012

KEY WORDS: reverse engineering; software evolution; communicated information; execution log analysis

1. INTRODUCTION

Software profiling and automated instrumentation techniques are commonly used to study the run-time behaviour of software systems [1]. However, such techniques often impose high overhead and slow down the execution, especially for real-life workloads. Worse, software profiling and instrumentation are done after the system has been built, on the basis of limited domain knowledge. Therefore, extensive instrumentation often leads to an enormous volume of results that are impractical to interpret meaningfully.

*Correspondence to: Weiyi Shang, School of Computing, Queen's University, Kingston, Ontario, Canada.
†E-mail: swy@cs.queensu.ca

Copyright © 2013 John Wiley & Sons, Ltd.

In practice, system administrators and developers typically rely on the software system's communicated information (CI), consisting of the major system activities (e.g. events) and their associated contexts (e.g. a time stamp) to understand the high-level field behaviour of large systems and to diagnose and repair bugs. Rather than generating tracing information in a blind way, developers choose to explicitly communicate some information that is considered to be particularly important for system operation. Therefore, we call such information as CI because it is the information communicated automatically by the system during its execution, whereas traces are generated by the people who analyze the system after the fact. One common medium for CI are execution logs. The purpose and importance of the information in such logs varies on the basis of their purpose. For example, detailed debugging logs are relevant to developers (i.e. implementation CI), whereas operation logs summarizing the key execution steps are more relevant to operators (i.e. domain-level CI).

The rich nature of CI has introduced a whole new market of applications that complement large software systems. We collectively call these applications the *log processing apps* (LPAs). The apps are used, for example, to generate workload information for capacity planning of large-scale systems [2, 3], to monitor system health [4], to detect abnormal system behaviours [5] or to flag performance degradations [6]. As such, these LPAs play an important role in the development and management of large software systems, to the extent that major decisions such as adding server capacity or changing the company's business strategy can depend on the information derived by the LPAs.

Communicated information changes often break the functionality of the LPAs. Often, LPAs are in-house applications that are highly dependent on the CI. Although they are typically built on commercial platforms by IBM [7] and Splunk [8], the actual link between the LPAs and the monitored system depends heavily on by the specific kind and format of CI in use. Hence, the apps require continuous maintenance as the format or type of CI changes and as the needs change. However, because little is known about the evolution of CI, it is unclear how much maintenance effort LPAs require in the long run.

In our previous research [9], we explored the evolution of CI by examining the logs of 10 releases of an open source software system (*Hadoop*) and 9 releases of a closed source large enterprise application, which we call *EA*. Because the CI during typical execution of a system may not cover all the CI that the system is capable of, this paper extends our research by performing a lower-level study than the previous research. We study the CI on the basis of the logging statements in the source code. We call such logging statements CI potential because their output potentially will show up in the CI during execution if the code that they are in is executed. We perform CI potential case studies on the 10 studied releases of Hadoop and five releases of another open source software system (*PostgreSQL*). Our study is the first step in understanding the maintenance effort for LPAs by studying the evolution of the CI (i.e. their input domain).

Our study tracks the CI for the execution of a fixed set of major features across the lifetime of the three studied systems and analyzes the source code of the two studied open source systems for CI potential. This allows us to address the following research questions:

RQ1:
How much does CI change over time?We find that over time, the amount of CI communicated about our covered features at execution level increases 1.5–2.8 times compared with the first-studied release. The CI potential (logging statements) increases 0.17–2.45 times. We note that down to only 40% of the CI at execution level stays the same across releases and up to 21.5% of the CI at execution level is modified with context modifications across releases. The modifications to the CI may be troublesome as they cause the LPAs to be more error prone.

RQ2:
What types of modifications happen to CI?Examining the CI and CI potential modifications across releases, we identify eight types of modifications. Of these changes, 10–70% can be avoided, and the impact of 15–80% of them can be minimized through the use of robust analysis techniques. The remaining modifications are risky and should be tracked carefully to avoid breaking LPAs.

RQ3:
What information is conveyed by short-lived CI?We find that short-lived CI at code level focuses more on system errors and warnings. Besides the high-level conceptual information, the content

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

of short-lived CI at execution level also contains implementation-level details and system error messages. With these findings, more resources should be allocated to maintain LPAs that heavily depend on implementation-level information or monitor system errors.

Our study highlights the need for tools and approaches (e.g. traceability techniques) to ease the maintenance of LPAs.

The rest of this paper is organized as follows: Section 2 presents an example to motivate our work. Section 3 presents the data preparation steps for our case study. Section 4 presents our case studies and the answers to our research questions. Section 5 discusses the limitations of our study. Section 6 discusses prior work. Section 7 concludes the paper.

2. A MOTIVATING EXAMPLE

We use a hypothetical, but realistic, motivating example to illustrate the impact of CI changes on the development and maintenance of LPAs.

The example considers an online file storage system that enables customers to upload, share and modify files. Initially, there were execution logs used by operators to monitor the performance of the system. The information recorded in the execution logs contained system events, such as 'user requests file', 'start to transfer file' and 'file delivered'.

2.1. *Release n*

Let us suppose that operators identified a performance problem in the release $n-1$. In order to diagnose the problem, developer Andy added more information to the context of the execution events in CI, such as the ID of the thread that handles file uploads. Using the added context in the execution logs, the operators identified the root cause of the performance problem and developer Andy resolved it. A simple LPA was written to continuously monitor for the re-occurrence of the problem by scanning the CI for the corresponding system events.

2.2. *Release* $n+1$

The file upload feature was overhauled, during which the developers changed the communicated events and their associated log entries. These context changes to the log files led to failures of the LPAs because they could no longer parse the log lines correctly to find the start and end time stamps of each transaction.

The application also started giving false alarms. After several hours of analysis, the root cause of the false alarm was identified. The CI had been changed by another developer, Bob, who was not aware that others made use of this information because there is no traceability between the information and the LPA. To avoid these problems reoccurring in the future, developer Andy marked the dependence of the LPA on this CI event in an ad hoc manner through basic code comments.

From the motivating example, we can observe the following:

- CI is crucial for understanding and resolving field problems and bugs.
- CI is continuously changing because of development and field requirements.
- LPAs are highly dependent on the CI.

Unfortunately, today, there are no techniques to ensure traceability between source code and LPAs, leading LPAs to be very fragile, as they have to adapt to continuously changing CI.

3. CASE STUDY SETUP

To understand how CI evolves, we mine a commonly available source of CI: the execution logs. We mine both the actual CI communicated dynamically at run time as well as the static CI potential, that is, the logging statements in the source code. Depending on the user scenario, some of the CI

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table I. Overview of the studied releases of Hadoop (minor releases in italic).

Release	Release date	K SLOC
0.14.0	20 August 2007	122
0.15.0	29 October 2007	137
0.16.0	7 February 2008	181
0.17.0	20 May 2008	158
0.18.0	22 August 2008	174
0.19.0	21 November 2008	293
0.20.0	22 April 2009	250
0.20.1	14 September 2009	258
0.20.2	26 February 2010	259
0.21.0	23 August 2010	201

potential will result in actual CI, whereas other statements will not be executed and hence not communicated. In this section, we present the studied systems and our approach to recover CI from the execution logs.

3.1. Studied systems

We chose two open source systems and one closed source software system with different sizes and application domains as the subjects for our case study. We chose 10 releases of Hadoop,* five releases of PostgreSQL[†] and nine releases of a closed source large enterprise application, which we will refer to as EA.

Hadoop is a large distributed data processing platform that implements the MapReduce [10] data processing paradigm. We use releases 0.14.0 to 0.21.0 for our study as shown in Table I. We chose these releases because 0.14.0 is the earliest one that is able to run in our experimental environment and 0.21.0 is the most recent release at the time of our previous study [9]. Among the studied releases, 0.20.1 and 0.20.2 are minor releases of the 0.20.0 series. Because Hadoop is widely used in both academia and industry, various LPAs (e.g. *Chukwa* [11] and Salsa [12]) are designed to diagnose system problems as well as monitor the performance of Hadoop.

PostgreSQL is an open source database management system written in C. We chose releases 8.2 to 9.1 for our study because they are the releases that are able to run on our experimental environment (Windows Server). The overview of the releases is shown in Table II. All the releases used in our study are major releases. Various LPAs have been developed for PostgreSQL; for example, *pgFouine* [13] analyzes PostgreSQL logs to determine whether certain queries need optimization.

The EA in our study is a large-scale communication application that is deployed in thousands of enterprises worldwide and used by millions of users. Because of a non-disclosure agreement, we cannot reveal additional details about the application. We do note that it is considerably larger than Hadoop and has a much larger user base and longer history. We studied nine releases of EA. The first seven minor releases are from one major release series, and the later two releases are from another major release. We name the release numbers 1.0 to 1.6 for the first major release and 2.0 to 2.1 for the second major release. There are currently several LPAs for the EA. These LPAs are used for functional verification, performance analysis, capacity planning, system monitoring and field diagnosis by customers worldwide.

3.2. Uncovering CI from logs

We perform both execution-level and code-level analyses on logs to uncover CI. Studying CI only at execution level or code level is important because execution-level CI contains the information that LPAs actually depend on and the code-level CI has the potential to be communicated during

*http://hadoop.apache.org/, last checked March 2011.
[†]http://www.postgresql.org/, last checked February 2012.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table II. Overview of the studied releases of PostgreSQL.

Release	Release date	K SLOC
8.2	5 December 2006	471
8.3	4 February 2008	533
8.4	1 July 2009	576
9.0	20 September 2010	613
9.1	12 September 2011	654

execution. In our previous research [9], we only studied CI at execution level, but in this paper, we extend our study of CI by also performing source code analysis of logging statements.

3.2.1. Execution level. Our execution-level approach to recover the CI of software systems consists of the following three steps: (1) system deployment; (2) data collection; and (3) log abstraction.

3.2.1.1. System deployment. For our study, we seek to understand the CI of each system on the basis of exercising a same set of features across the releases of these systems. To achieve our goal, we run every version of each application with the same workload in an experimental environment. The experimental environment for Hadoop consists of three machines. The experimental environment for EA mimics the setup of a very large field deployment. The experimental environment of PostgreSQL is a Windows Server.

3.2.1.2. Data collection. In this step, we collect execution logs from the three subject systems.

The Hadoop workload consists of two example programs, *wordcount* and *grep*. The wordcount program generates the frequency of all the words in the input data, and the grep program searches the input data for a string pattern. In our case study, the input data for both wordcount and grep is a set of data with a total size of 5 GB. The search pattern of the grep program in our study is one particular word (fail).

To collect consistent and comparable data for the EA, we choose a subset of features that are available in all versions of the EA. We simulate the real-life usage of the EA system through a specialized workload generator, which exercises the configured set of features for a given number of times. We perform the same 8-h standard load test [14] on each of the EA releases. A standard load test mimics the real-life usage of the system and ensures that all of the common features are covered during the test.

We deploy the database of the *Dell DVD Store* [15] on a PostgreSQL database server for each of the releases. We extract database queries from the source code of the Dell DVD Store as the typical workload of the Dell DVD Store database. We leverage *Apache JMeter* [16] to execute the queries repetitively for 2 h.

Note that at the execution level, we use realistic run-time scenarios and workloads. Hence, our experiments cannot guarantee full coverage of all features and hence run-time events. This is the reason why we also need to perform the code-level analysis presented in the next sub-section.

3.2.1.3. Log abstraction. We recover the CI by analyzing the generated execution logs. Execution logs (e.g. Table III) typically do not follow a strict format. Instead, they often use inconsistent formats [17]. For example, one developer may choose to use ‘,’ as separators in logs, whereas another developer may choose to use ‘\t’. The free-form nature of these logs makes it hard to extract information from them. Moreover, log lines typically contain a mixture of static and dynamic information. The static values contain the descriptions of the execution events, whereas the dynamic values indicate the corresponding *context* of these events.

We need to identify the different kinds of system events on the basis of the different event instances in the execution logs. We use a technique proposed by Jiang *et al*. [18] to automatically extract the execution events and their associated context from the logs. Figure 1 shows the overall process of log abstraction. As shown in Table IV, the descriptions of task types, such as ‘Trying to launch’, are static values, that is, system events. The time stamps and task IDs are dynamic values (i.e. the context for these events). The log abstraction technique normalizes the dynamic values and uses the

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table III. Example of execution log lines.

#	Log lines
1	time = 1, Trying to launch, TaskID = 01A
2	time = 2, Trying to launch, TaskID = 077
3	time = 3, JVM, TaskID = 01A
4	time = 4, Reduce, TaskID = 01A
5	time = 5, JVM, TaskID = 077
6	time = 6, Reduce, TaskID = 01A
7	time = 7, Reduce, TaskID = 01A
8	time = 8, Progress, TaskID = 077
9	time = 9, Done, TaskID = 077
10	time = 10, Commit Pending, TaskID = 01A
11	time = 11, Done, TaskID = 01A

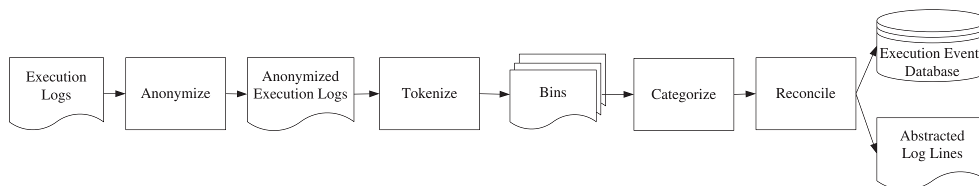

Figure 1. Overall framework for log abstraction.

Table IV. Abstracted execution events.

Event	Event	#
E_1	time = $t, Trying to launch, TaskID = $id	1,2
E_2	time = $t, JVM, TaskID = $id	3,5
E_3	time = $t, Reduce, TaskID = $id	4,6,7
E_4	time = $t, Progress, TaskID = $id	8
E_5	time = $t, Commit Pending, TaskID = $id	10
E_6	time = $t, Done, TaskID = $id	9,11

static values to create abstracted execution events. We consider the abstracted execution events as representations of communicated system events.

First, the anonymize step uses heuristics to recognize dynamic values in log lines. For example, ‘TaskID = 01A’ will be anonymized to ‘TaskID = $id’. The tokenize step separates the anonymized log lines into different groups (i.e. bins) according to the number of words and estimated parameters in each log line. Afterwards, the categorize step compares log lines within each bin and abstracts them into the corresponding execution events (e.g. ‘Reduce’). Similar execution events with different anonymized parameters are categorized together. Because the anonymize step uses heuristics to identify dynamic information in a log line, there is a chance that the heuristic might fail to anonymize some dynamic information. The reconcile step identifies such dynamic values by analyzing the difference between the execution events to each other within the same bin. Case studies in previous research [18] show that the precision and recall of this technique both are high, for example, over 80% precision and recall on the EA logs.

3.2.2. Code level. Our code-level approach consists of two steps: (1) code abstraction and (2) identification of logging statements.

3.2.2.1. Code abstraction. We download the source code of each release of the subject systems. For Hadoop, a Java-based system, we leverage the *Eclipse JDT parser* [19] to create abstract syntax trees for each source code file. For PostgreSQL, we use *TXL* [20] to create abstract syntax trees.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

3.2.2.1. Identification of logging statements. Software projects typically leverage logging libraries to generate logs. One of the most widely used logging libraries in Java systems is *Log4j* [21]. After manually browsing the source code of each project, we identify that Hadoop uses Log4j as logging library, whereas PostgreSQL uses its own logging library.

Using this knowledge, we analyze the generated abstract syntax trees to identify the logging source code fragments and changes. Typically, logging source code contains method invocations that call the logging library. For example, in Hadoop, a method invocation such as 'LOG' is considered to be a piece of logging source code. Changes that include such code are considered as log churn in Hadoop. Although some of the logging statements can be identified by performing text analysis, such as using 'grep', tracking the source code changes in our code-level analysis is difficult using the text analysis.

We only study the CI at the code level for Hadoop and PostgreSQL because of the lack of access to the source code of EA.

4. CASE STUDY RESULTS

In this section, we present the findings on our research questions. For each research question, we present the motivation, our approach and the corresponding results.

4.1. RQ1: How much does CI change over time?

4.1.1. Motivation. The evolution of CI impacts the maintenance of LPAs. The frequent change of the CI makes LPAs fragile because of the lack of established traceability techniques between LPAs and CI. Hence, change to execution events gives an indication of the complexity of designing and maintaining LPAs.

4.1.2. Approach. At execution level, we use the number of different abstracted execution events (we referred to all the abstracted execution events as 'events' for short in the later part of this paper) as a measurement of the size of CI. We also study the CI changes by measuring the percentages of unchanged, added and deleted execution events. Given the current release n and the previous release $n-1$, the percentages of unchanged and added execution events are defined by the ratio of the number of unchanged and added events in release n over the number of total execution events in the current release (n), respectively. The percentage of deleted execution events is defined the same as unchanged and added events, except over the number of total execution events in the previous release ($n-1$). For example, the percentage of unchanged execution events in release n $\left(P_{\text{unchanged}_n}\right)$ is calculated as follows:

$$P_{\text{unchanged}_n} = \frac{\#\text{unchanged events}_n}{\#\text{total events}_n} \quad (1)$$

We identify modified events by manually examining added and deleted events. We use the frequency of execution events in two releases to assist in mapping between modified events with similar wording across releases. Given the large number of events of EA (over 1900 across all releases), we only examine the top 40 most occurring events because they represent more than 90% of the total size of logs. This means that for EA, we use a similar equation as Equation (1), except that the number of *total execution events* for EA is always 40 as we only examine 40 execution events.

At code level, we use the number of logging statements to measure the size of CI potential. The unchanged, added and deleted logging statements at code level are studied in a similar way as at the execution level. To assist in understanding the CI change at code level, we also calculate the code churn (total number of added and deleted lines of code) in each studied release. We leverage *J-REX*, a high-level evolutionary extractor of source code history similar to C-REX [22], to identify the modification of logging statements during the development of the systems.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

4.1.3. Results: The total amount of CI.

4.1.3.1. Execution level. We first study the amount of CI in the history of both systems. Figure 2 shows the growth trend of CI in Hadoop. The growth of CI is faster than the growth of source code (shown in Table I). At the execution level, we note that the CI in the last studied release (0.21.0) is 2.8 times the amount of the first studied release (0.14.0), whereas the size of source code is only increased by less than 30% (201 KLOC to 259 KLOC). In particular, the amount of CI increases significantly in release 0.21.0 even though the size of the corresponding source code decreases by 20%. We also note that the CI increases more between major releases than between the studied two minor releases. The large increase of CI size at execution level in major releases indicates that additional maintenance effort might be needed for LPAs to continue operating correctly even if the existing LPAs do not use CI about new features or the additional CI about the currently analyzed features.

The CI of PostgreSQL, shown in Figure 3, also shows a growth trend at the execution level. Unlike Hadoop, there is not one release in PostgreSQL that has significantly larger amounts of CI growth than other releases. We believe that the reason is that PostgreSQL is a mature and stable system with more than 20 years history, whereas the development of Hadoop started in 2005 and did not officially release version 1.0 yet. Therefore, developers of PostgreSQL would be unlikely to add or delete substantial features in the system or significantly change the architecture of the system in one release. The stable nature of the CI in PostgreSQL may suggest that its LPAs are easier to maintain. The developers of LPAs can focus more on adding features of the LPAs by analyzing the added CI and focus less on maintaining features on the basis of old CI.

For the EA system, we only study execution level CI because of lack of access to the source code data. Because we study only two major releases, we study the amount of CI in each major release (nine releases in total) instead of generalizing a trend over the two studied releases. We note that the amount of CI does not change significantly in the first major release, whereas the CI increases significantly in the second major release. The CI of the last studied release (2.1) is 1.5 times the amount of the first-studied release (1.0).

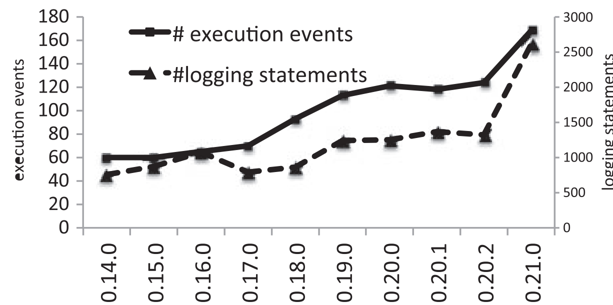

Figure 2. Growth trend of CI (execution level and code level) in Hadoop.

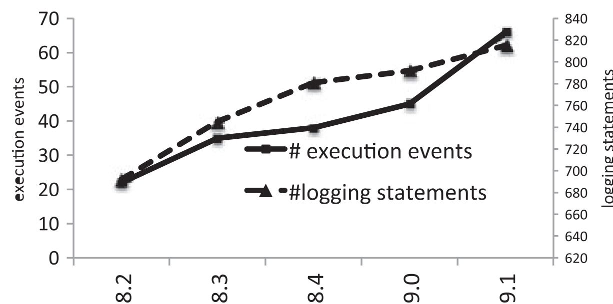

Figure 3. Growth trend of CI (execution level and code level) in PostgreSQL.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

4.1.3.2. Code level. The trend of CI potential is similar to that at execution level. As an exception, the CI in release 0.17.0 of Hadoop increases at the execution level but decreases at the code level. Table VI shows that over 48% of the CI in release 0.16.0 is removed in release 0.17.0. The release notes [23] showed that the *HBase* component of Hadoop of release 0.16.0 became a separate project right before the release of 0.17.0 of Hadoop.

In both Hadoop and PostgreSQL, less than 10% of the CI potential at code level is observed at execution level. Operators typically examine the logs that appear in the field when the software system is upgraded to a new release and adapt their LPAs accordingly. However, our results indicate that most of the CI potential is not observed during the typical execution of the system. The changes to such CI may cause problems in the LPAs. Developers of the software system may consider documenting all the CI potential in the system and transferring such knowledge to the operators in the field. On the other hand, it may be good news for LPA developers because most of the changed CI potential*s* may not show up during execution, therefore not causing LPAs to break in practice.

4.1.4. Results: The amount of changed CI.

4.1.4.1. Execution level. A closer analysis of the CI across releases shows that for all the studied systems at execution level, most (over 60%) of the old CI remains the same in new major releases (Tables V–VII). The CI events are more stable across minor releases (on average, over 80% remains the same). This is good news for developers of LPAs. However, from Tables V–VII, we observe that on average, around 1% (for EA), 7.5% (for Hadoop) and 6.3% (for PostgreSQL) of the CI are CI modifications. Such CI may be troublesome for maintainers of LPAs because they might need to modify their code to account for such changes.

It is important to note that all these CI changes are for a fixed set of executed features; that is, although the executed features remain the same, the CI clearly does not. We studied the release

Table V. Percentage of unchanged, added, deleted and modified CI in the history of EA (bold font indicates large changes).

	Unchanged (%)	Added (%)	Modified (%)	Deleted (%)
1.1	92.1	7.9	0.0	2.8
1.2	**64.8**	**32.9**	**2.3**	**34.7**
1.3	77.8	19.7	2.5	10.3
1.4	86.0	13.4	0.7	19.5
1.5	85.7	13.5	0.8	15.4
1.6	96.5	3.2	0.3	3.2
2.0	**61.2**	**38.0**	**0.7**	**20.8**
2.1	74.2	25.6	0.2	14.2

Table VI. Percentage of unchanged, added, modified and deleted CI (in execution level and code level) in the history of Hadoop (bold font indicates large changes).

	Execution level					Code level				
	Total	Unchanged (%)	Added (%)	Modified (%)	Deleted (%)	Total	Unchanged (%)	Added (%)	Modified (%)	Deleted (%)
0.15.0	60	81.7	10.0	8.3	10.0	871	69.7	**30.3**	5.1	15.5
0.16.0	65	80.0	16.9	3.1	10.0	1074	72.3	**27.7**	3.5	11.2
0.17.0	70	81.4	10.0	8.6	3.1	794	87.7	12.3	0.4	**48.3**
0.18.0	93	38.7	**39.8**	**21.5**	**20.0**	862	78.8	21.2	3.8	12.8
0.19.0	113	66.4	29.2	4.4	14.0	1239	65.7	**34.3**	0.7	6.2
0.20.0	121	71.9	24.0	4.1	18.6	1250	80.7	19.3	2.1	25.7
0.20.1	118	91.5	5.9	2.5	8.3	1370	89.7	10.3	0.6	1.4
0.20.2	124	95.2	4.8	0.0	0.0	1321	99.0	1.0	0.0	5.0
0.21.0	168	38.7	**46.4**	**14.9**	**27.4**	2605	43.6	**56.4**	0.7	13.6

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table VII. Percentage of unchanged, added, modified and deleted CI (in execution level and code level) in the history of PostgreSQL.

	Execution level					Code level				
	Total	Unchanged (%)	Added (%)	Modified (%)	Deleted (%)	Total	Unchanged (%)	Added (%)	Modified (%)	Deleted (%)
8.3	35	45.7	**48.6**	5.7	18.2	745	73.7	**22.6**	3.8	4.0
8.4	38	73.7	10.5	15.8	2.9	781	85.7	12.4	1.9	2.0
9.0	45	82.2	15.6	2.2	0.0	792	93.3	5.1	1.6	1.7
9.1	66	66.7	**31.8**	1.5	0.0	815	86.6	11.7	1.7	1.8

information for both releases and read through the change logs to better understand the rationale for large CI changes. We find that internal implementation changes often have a big impact on the CI. For example, according to Table VI, release 0.18.0 (in bold font) is one of the releases with the highest percentage of CI changes. Release 0.18.0 introduced new Java classes for Hadoop jobs (a core functionality of Hadoop) to replace the old classes. Release 0.21.0 officially replaced the old MapReduce implementation named 'mapred', with a new implementation named 'MapReduce'. Table VI shows that releases 0.18.0 and 0.21.0 have the largest amounts of code churns, which shows evidence that both releases have significant changes in the source code. Similarly, releases 1.3 and 2.0 (bold in Table V) of EA have significant behavioural and architectural changes compared with their previous releases. Similarly, although PostgreSQL with a much longer history than Hadoop and mature architecture and design, we still observe CI added, deleted and modified across releases. For example, release 8.3 has the largest (48.6%) added CI, which corresponds to the new 'autovacuum' feature. It appears that the subject systems communicate a significant amount of implementation-level information, leading their CI to vary considerably because of internal changes.

4.1.4.2. Code level. We observed different percentages of unchanged, added, deleted and modified CI at code level compared with the percentages at execution level. For example, in Hadoop, releases 0.15.0, 0.16.0 and 0.19.0 have a large number of added CI potential at code level, but the added CI at execution level is low. From the release notes and development history, we found out that new components with extensive logging had been added in the source code of these releases, but these components are not deployed in the release. For example, HBase is added into Hadoop in release 0.15.0, but it is not, by default, deployed with Hadoop. In release 0.19.0 of Hadoop, a collection of sub-projects is added into the 'contrib' folder of Hadoop, including Chukwa [11] and Hadoop *Streaming* [24]. Neither of these sub-projects is deployed with Hadoop by default. However, we cannot observe the same in PostgreSQL. The release 8.3 of PostgreSQL has the largest added CI at both execution level and code level. According to the release notes [25], the large addition of CI is according to a new feature to support 'multiple concurrent autovacuum processes'.

> The CI for the studied systems tends to contain implementation-level information that provides LPAs with detailed knowledge of the internals of the systems. However, this makes such LPAs fragile, in particular for major releases (e.g., in 0.21.0 of Hadoop, only 38% CI keeps unchanged)). For minor releases, we still see a large percentage of CI added or modified (e.g., only 64.8% CI is unchanged in EA 1.2). The CI at execution level only covers less than 10% of the CI potential at code level, and the CI potential at code level may change differently compared to the CI at execution level. Developers may consider explicitly transferring knowledge about CI potential to users of the CI.

4.2. *RQ2: What types of modifications happen to CI?*

4.2.1. Motivation. RQ1 shows that up to 21.5% (Table VI) of communicated events is modified. These modified CI have a crucial impact on LPAs because LPAs expect certain context information and are likely to fail when operating on events with modified context. In contrast, newly added CI is not likely to impact already developed LPAs because those applications are unaware of the new CI

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

and will simply ignore it. In short, changes to the context of previously communicated events are more likely to introduce bugs and failures in LPAs. For example, during the history of Hadoop, ‘task’ (an important concept of the platform) was renamed to ‘attempt’, leading to failures of monitoring tools and to confusion within the user community about the communicated context [23]. Therefore, we wish to understand how communicated contexts change.

4.2.2. Approach. We follow a grounded theory [26] approach to identify modification patterns to the context. We manually study all events at execution level with a modified context and all the modified logging statements at code level. For each of them, we analyze what information is modified and how the information is modified. We repeat this process several times until a number of modification types emerge. We then calculate the distribution of different types of modifications. The percentage of each type of modifications is calculated as the ratio of the number of occurrences of a type across all the releases over the total number of modifications across all the releases. For example, the percentage of modified CI of type p $\left(P_{\text{modified}_p}\right)$ is calculated as follows:

$$P_{\text{modified}_p} = \frac{\#\text{modified events}_p}{\#\text{total modified events}} \tag{2}$$

4.2.3. Results: CI modification types. Table VIII tabulates the six types of CI modifications identified from our manual examination on the CI at the execution level. The table defines each type and gives a real-life example of it from the studied data. Among all the types, *Rephrasing* and *Redundant contexts* are avoidable modifications because neither of them brings any additional information to the CI and only cause changes in LPAs. The *Adding context* modification is typically unavoidable, but a robust log parsing parser should still be able to parse the log correctly. For example, *Island Grammars* [27] can be leveraged in this case to ignore the added information in CI during the parsing of the log lines. Therefore, Adding context is a recoverable modification and has a less negative impact than the avoidable modifications. The other three types of modification (i.e. *Merging*, *Splitting* and *Deleting context*) are unavoidable, but the LPAs still need to adapt for these modifications. Developers can use a *null* value for the deleted context to make the CI consistent. However, such deleted context may correspond to the removed features, and such preferentially deleted context

Table VIII. CI modification types and examples of the execution level analysis.

Pattern	Definitions	Examples	
		Before	After
Adding context (recoverable)	Additional context is added into the communicated information.	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit m	ShuffleRamManager memory limit *n* MaxSingleShuffleLimit *m* mergeThreshold *Q*
Deleting context (unavoidable)	Context is removed from the communicated information.	Got *n* map output known output *m*	Got *n* output
Redundant context (avoidable)	Some redundant information is added or the added information can be inferred without being included in the context.	Task is in COMMIT_PENDING	Task is in commit_pending, status: COMMIT_PENDING
Rephrasing (avoidable)	The CI is replaced (partially) by new CI.	Hadoop mapred Reduce task fetch *n* bytes	Hadoop MapReduce task Reduce fetch *n* bytes
Merging (unavoidable)	Several old CI are merged into one.	MapTask record buffer MapTask data buffer	MapTask buffer
Splitting (unavoidable)	The old CI is split into multiple new ones.	Adding task to tasktracker	Adding Map Task to tasktracker; Adding Reduce Task to tasktracker

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

with *null* values may cause the CI hard to maintain in the long term. Developers of the system should document the unavoidable modifications well and inform people who make use of the modified CI. We note that although some Splitting modifications look similar to Adding context, the two types of modifications are essentially different. Splitting is dividing one CI event into multiple ones (e.g. splitting the recording of buffer size to different types of buffers, such as data buffer and task buffer), whereas Adding context is providing extra information to the CI; for example, in addition to recording the buffer size, the free space of the buffer is also recorded.

At the code level, we identify two additional types of CI modifications shown in Table IX. Both *Changing logging level* and *Changing arguments* are recoverable modifications because a robust log parser that analyzes the generated logs from these logging statements would likely not be impacted by such CI modifications.

4.2.4. Results: CI modifications distribution.

4.2.4.1. Overall. Figure 4 shows the classification distribution of the CI modification types at both execution and code levels across all the studied releases, and Table X shows the percentage of avoidable, recoverable and unavoidable CI modifications. We find that the majority of the CI modifications are either avoidable or recoverable. Only a small portion of the CI modifications is unavoidable. Simply put that developers can improve the maintenance of the LPAs by avoiding the avoidable modifications and documenting the unavoidable CI modifications.

4.2.4.2. Execution level. Tables XI, XIV, XII, XV, XIII show the percentages of context modifications for Hadoop, PostgreSQL and EA at execution level, broken down per release and pattern. Table XI shows that the two largest numbers (in bold) of context modifications are both instances of Rephrasing context. They were introduced in releases 0.18.0 and 0.21.0 of Hadoop. Table XIII shows that many Rephrasing context instances are introduced in version 0.1.2 of EA. As noted in RQ1, all these three releases (0.18.0 and 0.21.0 of Hadoop and 1.2 of EA) have significant changes to the systems. These results indicate that most of the Rephrasing context modifications may have a high correlation to the major changes introduced into the software systems. For example, in release 0.21.0, the old MapReduce library, which is the most essential part of Hadoop, was replaced by a whole new implementation. Therefore, the word 'mapred' was replaced by the word 'MapReduce'. As both implementations have the same features, the operator should not need to worry about such implementation changes of the library. However, such rephrasing modifications require updating any LPAs to ensure their proper operation.

In contrast, even though release 1.3 of the EA has many Adding context modifications, it does not have a large number of added or deleted CI. This result indicates that even though some releases do not introduce major changes into the system, CI may still be modified significantly. Developers of LPAs may not only spend maintenance effort when the system is upgraded to a release with major system changes. A release without major system changes may also impact the LPAs significantly.

In PostgreSQL, only Adding context, Deleting context and Rephrasing context are observed in the studied releases. Eighty per cent of the context modification to the CI of PostgreSQL at execution level is recoverable (Adding context). Developers of the LPAs of PostgreSQL may spend more effort on creating robust log parsers.

Table IX. CI modification types discovered from code-level analysis (in addition to the types in Table VIII).

Pattern	Definitions	Examples	
		Before	After
Changing logging level (recoverable)	The logging level is changed.	*Log.info*('property'' + key + ''is' + val)	*Log.debug*('property'' + key + ''is' + val)
Changing arguments (recoverable)	Arguments in the logging statements are changed.	*Log.info*('created trash checkpoint:' + checkpoint)	*Log.info*('created trash checkpoint:' + checkpoint.touri ().getpath())

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

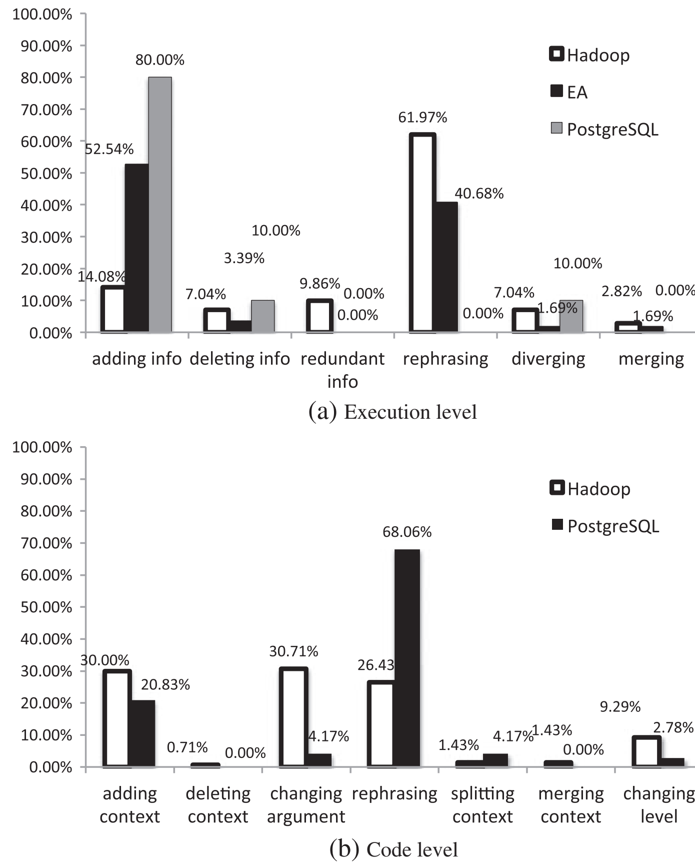

Figure 4. Distributions of the different types of CI modifications across all studied releases.

Table X. Percentage of avoidable, recoverable and unavoidable CI modification in Hadoop and EA.

	Execution level			Code level	
	Hadoop (%)	EA (%)	PostgreSQL (%)	Hadoop (%)	PostgreSQL (%)
Avoidable	71.83	40.68	10.00	70.00	68.06
Recoverable	14.08	52.54	80.00	26.43	28.78
Unavoidable	14.09	6.78	10.00	3.57	4.17

Table XI. Detailed percentages of different types of context modifications in Hadoop (execution level).

Release	Adding context	Deleting context	Redundant context	Rephrasing	Merging	Splitting
0.15.0	1.41	0.00	2.82	2.82	0.00	0.00
0.16.0	0.00	0.00	2.82	0.00	0.00	0.00
0.17.0	0.00	0.00	0.00	8.45	0.00	0.00
0.18.0	0.00	0.00	0.00	**28.17**	0.00	0.00
0.19.0	0.00	2.82	0.00	4.23	0.00	0.00
0.20.0	2.82	1.41	1.41	0.00	1.41	0.00
0.20.1	0.00	1.41	0.00	1.41	1.41	0.00
0.20.2	0.00	0.00	0.00	0.00	0.00	0.00
0.21.0	9.86	1.41	1.41	**16.9**	2.82	2.82

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table XII. Detailed percentages of different types of context modifications in PostgreSQL (execution level).

Release	Adding context	Deleting context	Redundant context	Rephrasing	Merging	Splitting
8.3	10.00	10.00	0.00	0.00	0.00	0.00
8.4	60.00	0.00	0.00	0.00	0.00	0.00
9.0	10.00	0.00	0.00	0.00	0.00	0.00
9.1	0.00	0.00	0.00	10.00	0.00	0.00

Table XIII. Detailed percentages of different types of context modifications in EA.

Release	Adding context	Deleting context	Redundant context	Rephrasing	Merging	Splitting
1.1	0.00	0.00	0.00	0.00	0.00	0.00
1.2	6.78	1.69	0.00	**20.34**	0.00	0.00
1.3	**22.03**	0.00	0.00	10.17	0.00	1.69
1.4	6.78	0.00	0.00	1.69	0.00	0.00
1.5	8.47	1.69	0.00	0.00	0.00	0.00
1.6	1.69	0.00	0.00	1.69	0.00	0.00
2.0	6.78	0.00	0.00	3.39	1.69	0.00
2.1	0.00	0.00	0.00	3.39	0.00	0.00

Table XIV. Detailed percentages of different types of CI modifications in Hadoop (code level).

Release	Adding context	Deleting context	Changing argument	Rephrasing	Splitting	Merging	Changing logging level
0.15.0	8.57	0.71	5.71	2.86	0.71	0.00	1.43
0.16.0	9.29	0.00	10.00	3.57	0.00	0.00	5.71
0.17.0	0.00	0.00	1.43	0.00	0.00	0.00	0.71
0.18.0	1.43	0.00	2.86	14.29	0.00	0.00	0.00
0.19.0	2.14	0.00	0.00	2.14	0.00	0.00	0.00
0.20.0	5.00	0.00	7.14	2.14	0.00	0.71	0.00
0.20.1	1.43	0.00	0.71	0.71	0.71	0.71	0.71
0.20.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.21.0	2.14	0.00	2.86	0.71	0.00	0.00	0.71

Table XV. Detailed percentages of different types of CI modifications in PostgreSQL (code level).

Release	Adding context	Deleting context	Changing argument	Rephrasing	Splitting	Merging	Changing logging level
8.3	8.33	0.00	0.00	**0.56**	0.00	0.00	0.00
8.4	5.56	0.00	0.00	11.11	1.39	0.00	0.00
9.0	5.56	0.00	4.17	9.72	1.39	0.00	2.78
9.1	1.39	0.00	0.00	16.67	1.39	0.00	0.00

We observe that in Hadoop, most of the CI modifications belong to the type of Rephrasing, whereas most of the CI modifications in PostgreSQL are in the type of Adding context. We believe the reason is that Hadoop as a new project has more structural changes, which may lead to the CI rephrasing. On the other hand, as a project with long history, PostgreSQL mainly has added features to it, which may contribute to most of the Adding context modifications.

4.2.4.3. Code level. Tables XIV and XV show the percentage of context modifications for Hadoop and PostgreSQL at code level, broken down per release and pattern. For Hadoop, the results of some releases are similar at code level and execution level, whereas the results of some other releases are different at code level and execution level. For example, release 0.18.0 has a large

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

number of Rephrasing contexts at both code level and execution level, whereas release 0.21.0 of Hadoop has large number of Rephrasing context at execution level but the number is low at code level. The high number of Rephrasing context at execution level but low at code level is because the re-implementation of the MapReduce library of Hadoop without removal of the old implementation from the source code. Therefore, at code level, the new implementation is considered as 'added' CI potential, whereas at execution level, the new implementation is considered as Rephrasing because it replaces the old implementation during the execution.

In PostgreSQL, more Rephrasing context is observed at code level. For example, the large percentage of Rephrasing at release 8.3 corresponds to rephrasing 'can't' to 'cannot' in all logging statements. These modifications to the CI are not observed in typical execution, thus likely to be neglected by developers of LPAs. If these CI appear during the execution of the system, the LPAs may not be able to process them correctly. Developers of the software system may consider spending more effort on tracking these Rephrasing type CI modifications using code analysis and informing the users of such CI about any modifications.

> We have identified eight types of communicated context modifications. Two of the types (Rephrasing and Redundant context) are avoidable, three types (Adding context, Changing logging level and Changing arguments) are unavoidable but their impact can be controlled through the use of robust parsing techniques. The other three types (Merging, Splitting and Deleting context) are unavoidable and have a high chance introducing errors. Around 90% of the modifications can be controlled through careful attention by system developers (avoidable context modifications) or careful robust programming of LPAs (Adding Context).

4.3. *RQ3: What information is conveyed in short-lived CI?*

4.3.1. Motivation. RQ1 shows that CI is added and deleted in every release. Some CI is added by developers and removed in a short period. The LPAs depending on such short-lived CI may be extremely fragile. We study the information conveyed in the short-lived CI to understand why such CI exists only within a short period. By studying the conveyed information, we can understand the CI at a high level of abstraction instead of considering simple counts of added, removed and modified CI like in the previous two questions.

4.3.2. Approach. We consider the CI that exists in only a single release to be short-lived CI. To have an initial idea of the purpose of the short-lived CI, we extract the logging level using the code analysis of the CI to measure the logging levels in short-lived CI. For each logging level, we calculate the percentage of short-lived logging statements.

To further understand the information conveyed by the short-lived CI over time, we generate a latent Dirichlet allocation (LDA) [28] model of the topics in the short-lived CI. Each topic in the model is a list of words that has high probability to appear together in short-lived CI. We put the short-lived CI of each release in a separate file as input documents of LDA. We use *MALLET* [29] to generate the LDA models with five topics. Each word in the topics has a probability indicating its significance in the corresponding topic. We generated the five words with the highest probability in each topic to determine the information conveyed by CI in the topic. Finally, we examine the words in the five topics and generate a one-sentence summary based on our knowledge about the systems to summarize the information conveyed in short-lived CI.

To compare the different characteristic between short-lived and long-lived CI, we perform the same experiments on long-lived CI. We consider the CI that exists in all studied releases as long-lived.

4.3.3. Results.
4.3.3.1. Logging level. The results of the code level analysis in Table XVI shows that most of the short-lived logging statements in Hadoop are at the *info* and *debug* level. Almost 70% of the logging statements at *trace* level and 22% to 25% of the logging statements in debug and *error* levels only appear in one release, while none of the logging statements at trace level exist across all the studied releases.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table XVI. Logging levels of short-lived and long-lived CI.

Short-lived CI				
	Hadoop		PostgreSQL	
	% over total short-lived CI	% short-lived CI in the logging level	% over total short-lived CI	% short-lived CI in the logging level
Trace	1.01	69.70	—	—
Debug	27.36	25.20	3.93	5.21
Info	47.62	14.70	1.43	100.00
Warn	13.96	14.31	0.00	0.00
Error	8.99	22.17	76.07	7.55
Fatal	1.06	10.86	16.43	17.10
Notice	—	—	0.36	9.09
Log	—	—	1.79	4.07
Long-lived CI				
	Hadoop		PostgreSQL	
	% over total short-lived CI	% short-lived CI in the logging level	% over total short-lived CI	% short-lived CI in the logging level
Trace	0.00	0.00	—	—
Debug	18.43	2.48	5.56	5.21
Info	54.98	2.47	0.00	0.00
Warn	16.01	2.39	0.00	—
Error	8.16	2.93	82.83	5.81
Fatal	2.42	3.62	3.54	2.60
Notice	—	—	1.01	9.09
Log	—	—	7.07	4.07

In PostgreSQL, over 75% of the short-lived logging statements are at the error level. The logging statements at error and *fatal* levels account for more than 90% of all the short-lived logging statements. The percentage of short-lived fatal level logging statement is much higher than long-lived logging statements. In contrast to Hadoop, PostgreSQL has more error and fatal level logging than info level logging. Therefore, all info logging statements in PostgreSQL are short-lived and they only account for 1.43% of the total logging statements.

4.3.3.2. Topics. A manual analysis of short-lived CI at execution level reveals that a small part of CI corresponds to exceptions and stack traces. We removed such data because it does not represent short-lived CI but primarily rare errors.

Tables XVII and XVIII show the topics generated by LDA for Hadoop. The words in each topic are sorted by their degree of memberships. From the results in Table XVII, as expected, we find

Table XVII. Topics of CI in Hadoop at execution level generated by LDA.

Short-lived CI		
#	Topic	Summary
1	job output node jobhistory saved	Hadoop saves output to a machine.
2	reducetask jobinprogress choosing server hadoop	Hadoop assigns a reduce task to a machine.
3	mapred map tracker taskinprogress jobtracker	Map task updates its progress.
4	id org local file ipc	Hadoop reads from a local file.
5	task tasktracker attempt outputs tip	Hadoop Attempt saves its output and reports to the task tracker.
Long-lived CI		
#	Topic	Summary
1	attempt starting successfully received	Hadoop starts an attempt successfully.
2	sessionid jvm fetcher processname	The jvm metric starts on a process with session id.
3	job initializing jvmmetrics jobtracker	Hadoop initializes a job with jvm metrics recorded.
4	id node tracker tasks	A task with id is on a node.
5	task reduce map completed	Hadoop map or reduce task completed.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Table XVIII. Topics of CI in Hadoop at code level generated by LDA.

#	Topic	Summary
	Short-lived CI	
#	Topic	Summary
1	block dir stringifyexception start	Hadoop throws exception when it reads a directory from a data block.
2	job integer finish maps	Hadoop job finishes with a number of Map tasks.
3	tostring getmessage path created	A new path is created in the distributed file system.
4	region tostring regionname regioninfo	HBase region server information is printed.
5	error file closing size	Hadoop fails to close a file.
Long-lived CI		
#	Topic	Summary
1	filter tracker defined trackername	Filter a tracker.
2	dst frequency countrecords testpipes	Count records from a server.
3	records addr rename	Rename a record on a node.
4	src patter unknown server	Progress from unknown server.
5	count skipping stringifyexception	Skip records.

that both short-lived and long-lived topics contain high-level conceptional information such as 'job'. However, we also find that the topics in short-lived CI may contain lower-level information, such as the implementation of the system. For example, the word 'ipc' in topic 4 means inter-procedural communication between machines. Because the topic is about reading a remote file, the word 'ipc' corresponds to an implementation detail of how to read the file. In addition, the information about outputting results and choosing a server in topics 1, 2 and 5 also contain implementation-level information.

At code level, two of the topics (topics 1 and 5) are about system exceptions or errors. We browsed the short-lived CI of Hadoop and found over 15% of them contains low-level details. For the topics in long-lived CI, we find that most of the topics correspond to the system events that do not often happen, such as filtering a node and skipping a record. The fact that these features are not in the hot spot of the system might be the reason that these CI are not changed during the development of the system. We performed the same study on EA at execution level, with the results similar to the results of Hadoop. For example, both long-lived and short-lived CI contains high-level domain knowledge but three topics in short-lived CI contain error messages or implementation details.

We observe that the topics in short-lived CI at execution level differ to the topics at code level. The reason is that only a small part of the CI at code level is executed at run time. In addition, Table XVI shows that almost 70% of the short-lived CI of Hadoop at code level is at trace level. These CI would not be generated during execution with default configuration.

We do not find short-lived CI of PostgreSQL at execution level. Table XIX shows the generated topics by LDA for PostgreSQL. Because we do not have experience in the development of PostgreSQL, we only show the generated topics without a summary. Different to the results of Hadoop, we can observe error messages, such as 'fail' and 'invalid', in both short-lived and long-lived CI of PostgreSQL at code level. Such observation confirms the results of logging level (Table XVI) that most of the logging statements (both short-lived and long-lived) in PostgreSQL are at error level.

Table XIX. Topics of the short-lived CI in PostgreSQL at code level generated by LDA.

#	Short-lived topic	#	Long-lived topic
1	spi type process tuple	1	type block invalid list
2	failed arguments pipe gin	2	relationgetrelationname rel fired page
3	cache lookup invalid extract	3	failed spi number trigger
4	file number create check	4	index relname owner add
5	failed relation join ttdummy	5	lookup relation returned manager

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Some LPAs are designed for recovering high-level information about the systems, for example, system workload rather than implementation details. Such LPAs would not need the implementation-level information and hence would not be impacted by changes to this kind of CI. However, there are a few LPAs that are designed for debugging purposes. Such applications require the implementation-level information and error messages in the short-lived CI, and would be fragile as their corresponding CI is continuously changing.

> Short-lived CI contains implementation-level details and error messages to facilitate system development and testing. LPAs depending on implementation-level information and error messages are likely to be more fragile. More maintenance effort is needed for LPAs that depend on implementation-level information and error messages.

5. THREATS TO VALIDITY

This section presents the threats to validity of our study.

5.1. External validity

Our study is an exploratory study performed on Hadoop, PostgreSQL and an EA. Even though all the subject systems have years of history and large user bases, more case studies on other software systems in the same domain are needed to see whether our findings can generalize. Similarly, the studied logs are collected from specific workloads, which may not generalize. We plan to study in-field execution logs in our future work.

5.2. Internal validity

Our study includes several manual steps, such as the analysis of log modifications and the classification of log reformatting. Our findings may contain subjective bias in such manual steps.

Our study is performed on both major and minor releases of Hadoop and EA. However, the major and minor releases in the two systems may not contain similar amounts of source code changes. We study Hadoop with mostly major releases, and we study EA with mostly minor releases. The major releases of Hadoop may not contain as many significant changes, and the minor releases of EA may contain large numbers of changes. Therefore, our findings about major and minor releases may be biased. We plan to study more releases in the same systems and more systems to counter this bias.

5.3. Construct validity

We use execution logs to study the CI. Other types of CI, such as code comments, may not evolve in the same manner or contain the same information as execution logs. Studying other types of CI is part of our future work.

Our execution-level study is mainly based on the abstraction of execution events proposed by Jiang *et al.* [18]. This approach, customized to better fit the two subject systems, is shown to have high precision and recall. However, falsely abstracted log events may still exist, which may potentially bias our results. We plan to adopt other log abstraction techniques to improve the precision and to reduce the incorrectly abstracted execution events in our study.

Our code-level study leverages J-REX. The correctness of our code-level study depends on the correctness of J-REX. J-REX has been used in previous research showing good performance and accuracy [30, 31]. Because of the lack of mature techniques to track the genealogy of CI, our approach cannot identify CI modifications automatically. We examined the added and deleted CI and identified the modified CI on the basis of our experiences of using CI. Such results can be treated by the accuracy of our subjective decision on the modified CI. We will re-perform our experiment when a mature tracking technique of CI genealogy is available.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

6. DISCUSSION AND RELATED WORK

In this section, we discuss the prior work related to our study.

6.1. Non-code-based evolution studies

Whereas many prior studies examined the evolution of source code (e.g. [32–34]), this paper studies the evolution of software systems from the perspective of non-code artefacts associated with these systems. The non-code artefacts are extensively used in software engineering practice, yet the dependency between such artefacts and their surrounding ecosystem lacks explicit study. Therefore, understanding the evolution of non-code based software artefacts is important. For example, the evolution of the following non-code artefacts has been studied before:

- **System Documentation**: Software systems evolve throughout the history, as new features are added and existing features are modified because of bug fixes, performance and usability enhancements. Antón *et al*. [35] studied the evolution of telephony software systems by studying the user documentation of telephony features in the phone books of Atlanta.
- **User Interface**: His *et al*. [36] studied the evolution of Microsoft Word by looking at changes to its menu structure. Hou *et al*. [37] studied the evolution of UI features in the Eclipse Integrated Development Environment.
- **Features**: Instead of studying the code directly, some studies have picked specific features and followed their implementation throughout the lifetime of the software system. For example, Kothari *et al*. [38] proposed a technique to evaluate the efficiency of software feature development by studying the evolution of call graphs generated during the execution of these features. Our study is similar to this work, except for using CI instead of call graphs. Greevy *et al*. [39] used program slicing to study the evolution of features.
- **Communicated Information**: We divide CI into CI about code (static CI) and CI about the execution (dynamic CI).
  -**Evolution of Static CI**: A major example of this CI is code comments, which are a valuable instrument to preserve design decisions and to communicate the intent of the code to programmers and maintainers. Jiang *et al*. [40] studied the evolution of source code comments and discover that the percentage of functions with header and non-header comments remains consistent throughout the evolution. Fluri *et al*. [41, 42] studied the evolution of code comments in eight software projects.
  -**Evolution of Dynamic CI**: To the best of our knowledge, this paper is the first work that seeks to study the evolution of dynamic CI.

6.2. Logs as a source of CI

Our case studies use execution logs as a primary source of CI. This is based on our team's extensive experience working with several large enterprises. Many systems today support monitoring APIs to communicate to a system. Users can call such APIs to retrieve the information about the execution of the systems. Typically, users of such systems still make extensive use of execution logs as a valuable source of the information about the execution of these systems, because logging is much simpler and a lighter-weight approach to communicate to users than providing a monitoring API.

In many ways, the logs provide a non-typed, flexible communication interface to the outside world. The flexible nature of the logs makes them easy to evolve by developers (hence faster to respond to changes in the systems). However, this also makes the applications that depend on them very fragile. As additional applications depend more and more on specific CI, it is often the case that such information is then formalized and communicated through more formalized and typed interfaces, such as a monitoring API. For example, application response measurement [43] provides monitoring APIs to assist in system response time monitoring and performance problem diagnosis. Further studies are needed to better understand the evolution of CI from very flexible to well-defined APIs.

We coined the term CI to clearly differentiate it from tracing information. Tracing information is mainly low level and generated in a blind way, whereas CI is intentionally generated by software

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

developers who consider the value of using such information in practice. We firmly believe that CI can (and should) remain consistent even as the lower-level implementation details of an application change. Recent work by Yuan *et al.* [44] has explored how one can improve the CI to easily detect and repair bugs by enhancing the communicated contexts. In future studies, we wish to define metrics to measure the quality of CI and the need for CI changes relative to code changes.

6.3. *Traceability between logs and log processing apps*

Many software developers consider logs as a final output of their systems. However, for many such systems logs are just the input for a whole range of applications that live in the log processing ecosystem surrounding these systems.

Our study is the first study to explore how changes in parts of an ecosystem (CI; i.e. logs), once released in the field, might impact other parts of the system (LPAs). The need for such types of studies was noted by Godfrey and German [45] as they recognized that most software systems today are linked with other systems within their ecosystems. For example, in regression tests, the test suites need to be maintained as the functionality changes. Test suites tend to accrue beyond their usefulness because developers are reluctant to remove any tests that some other developers might be depending on it.

Lehman's earlier work [34] recognizes the need for applications to adapt to the changes in their surrounding environment. In this study, we primarily focused on the environmental changes of LPAs (i.e. changes to CI). To prove the concept that LPAs evolve because of the evolution of CI, we manually examined the items of the issue tracking system JIRA of Chukwa, a log collector for Hadoop. We found six out of 637 items caused by the updating of Hadoop logs. For example, one of the issues (CHUKWA-132[‡]) corresponds to the failure of log parser when Hadoop starts to output logs across multiple lines. Another example is issue CHUKWA-375,[§] which is to update log parsers because of the log changes in Hadoop. The issues spread out across releases during the development history of Chukwa. In future work, we wish to extend this study by studying the changes in all aspects of the ecosystem, namely the system, the CI and the LPAs that process the CI.

Our study and our industrial experience support us in advocating the need for research on tools and techniques to establish and maintain traceability between the CI (logs in our case) and the LPAs. Such a line of work might increase the cost of building large systems. However, it is essential for reducing the maintenance overhead and costs for all apps within the ecosystem of large software systems.

7. CONCLUSION

Communicated information, such as execution logs and system events, is generated by snippets of code inserted explicitly by domain experts to record valuable information. An ecosystem of LPAs analyzes such valuable information to assist in software testing, system monitoring and program comprehension. Yet, these LPAs highly depend on CI and are hence impacted by changes to the CI. In this paper, we have performed an exploratory study on the CI of 10 releases of open source software named Hadoop, 5 releases of another open source software system named PostgreSQL and 9 releases of a legacy EA.

Our study shows that systems communicate more about their execution as they evolve. During the evolution of software systems, the CI also evolves. Especially when there are major source code changes (e.g. a new major release), the CI is changed significantly, although the changes of implementation ideally should not have an impact on CI. In addition, we observed eight types of CI modifications. Among the CI modifications in the studied systems, only less than 15% of the modifications are unavoidable and are likely to introduce errors into LPAs. We also find that short-lived CI typically contains system implementation-level details and system error messages.

[‡]https://issues.apache.org/jira/browse/CHUKWA-132 last checked September 2012.
[§]https://issues.apache.org/jira/browse/CHUKWA-375 last checked September 2012.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Our results indicate that additional maintenance resources should be allocated to maintain LPAs, especially when major changes are introduced into the software systems. Because of the evolution of CI, traceability techniques are needed to establish and track the dependencies between CI and the LPAs.

However, even today, without traceability techniques between CI and LPAs, the negative impact can still be minimized by both the system developers (who generate CI) and the developers of LPAs (who consume CI). System developers should avoid modifying CI as much as possible. The avoidable CI modifications include rephrasing and adding redundant information in the CI. On the other hand, LPA developers should write robust log parsers to avoid the impact of CI changes. In addition, more resources should be allocated to maintain LPAs designed for debugging problems (from short-lived CI).

We also found that the observed CI during typical system execution only covers less than 10% of the logging statements. The CI potential may evolve differently to the CI at execution level. The developers of LPAs are likely not aware of such difference, leading to 'suprise' LPAs bugs and failures.

Because of the difference between CI at the execution level and the CI potential in the source code, developers may consider providing documentation of the CI potential to the users of the CI (i.e. system administrators and developers of the LPAs) to support better usage of CI and to avoid potential problems in the LPAs.

ACKNOWLEDGMENTS

We would like to thank the WCRE 2011 reviewers for their valuable feedback. We are also grateful to Research In Motion (RIM) for providing access to the EA used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's products.

REFERENCES

1. Cornelissen B, Zaidman A, Deursen AV, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 2009; **35**:684–702, doi:10.1109/TSE.2009.28.
2. Hassan AE, Martin DJ, Flora P, Mansfield P, Dietz D. An industrial case study of customizing operational profiles using log compression. *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, ACM: Leipzig, Germany, 2008; 713–723.
3. Nagappan M, Wu K, Vouk MA. Efficiently extracting operational profiles from execution logs using suffix arrays. *ISSRE'09: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, IEEE Press: Bengaluru-Mysuru, India, 2009; 41–50.
4. Bitincka L, Ganapathi A, Sorkin S, Zhang S. Optimizing data analysis with a semi-structured time series database. *SLAML'10: Proceedings of the 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, USENIX Association: Vancouver, BC, Canada, 2010; 7–7.
5. Jiang ZM, Hassan AE, Hamann G, Flora P. Automatic identification of load testing problems. *ICSM '08: Proceedings of 24th IEEE International Conference on Software Maintenance*, IEEE: Beijing, China, 2008; 307–316.
6. Jiang ZM, Hassan AE, Hamann G, Flora P. Automated performance analysis of load tests. *ICSM '09: 25th IEEE International Conference on Software Maintenance*, IEEE: Edmonton, Alberta, Canada, 2009; 125–134.
7. Infosphere streams. Http://goo.gl/nI1a4
8. Splunk. Http://www.splunk.com/
9. Shang W, Jiang ZM, Adams B, Hassan AE, Godfrey M, Nasser M, Flora P. An exploratory study of the evolution of CI about the execution of large software systems. *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, Lero, Limerick, Ireland, 2011.
10. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51** (1):107–113.
11. Boulon J, Konwinski A, Qi R, Rabkin A, Yang E, YM Chukwa, a large-scale monitoring system. CCA '08: Proceedings of the First Workshop on Cloud Computing and Its Applications, Chicago, IL, 2008; 1–5.
12. Tan J, Pan X, Kavulya S, Gandhi R, Narasimhan P. Salsa: analyzing logs as state machines. *WASL'08: Proceedings of the First USENIX Conference on Analysis of System Logs*, USENIX Association: San Diego, California, 2008; 6–6.
13. phfouine. Http://pgfouine.projects.postgresql.org/
14. Beizer B. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Co.: New York, NY, USA, 1984.
15. Dell dvd store. Http://linux.dell.com/dvdstore/
16. Apache jmeter. Http://jmeter.apache.org/

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

17. Bruntink M, van Deursen A, D'Hondt M, Tourwé T. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. *AOSD '07: Proceedings of the 6th International Conference on Aspect-oriented Software Development*, ACM: Vancouver, British Columbia, Canada, 2007; 199–211.
18. Jiang ZM, Hassan AE, Hamann G, Flora P. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution* 2008; **20**(4):249–267, doi:http://dx.doi.org/10.1002/smr.v20:4
19. Eclipse jdt. Http://www.eclipse.org/jdt
20. Cordy JR. Excerpts from the txl cookbook. *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, Springer-Verlag: Berlin, Heidelberg, 2011; 27–91.
21. Log4j. Http://logging.apache.org/log4j/1.2/
22. Hassan AE. Mining software repositories to assist developers and support managers. PhD Thesis, University of Waterloo 2005.
23. Hadoop release notes. Http://hadoop.apache.org/common/releases.html
24. Hadoop stream. Http://hadoop.apache.org/common/ docs/r0.20.2/streaming.html
25. Postgresql release notes of 8.3. Http://www.postgresql.org/docs/8.3/static/release-8-3.html
26. Brower R, Jeong H. Beyond description to derive theory from qualitative data. *Handbook of Research Methods in Public Administration*, Raton B (ed.). Taylor and Francis: Boca Raton 2008; 823–839.
27. Moonen L. Generating robust parsers using island grammars. *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, IEEE Computer Society: Washington, DC, USA, 2001; 13.
28. Blei DM, Ng AY, Jordan MI. Latent dirichlet allocation. *Journal of Machine Learning Research* 2003; **3**:993–1022.
29. Mallet: a machine learning for language toolkit. Http://mallet.cs.umass.edu/
30. Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, ACM: New York, NY, USA, 2010; 4:1–4:10.
31. Selim GMK, Barbour L, Shang W, Adams B, Hassan AE, Zou Y. Studying the impact of clones on software defects. *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, IEEE Computer Society: Washington, DC, USA, 2010; 13–21.
32. Gall H, Jazayeri M, Klösch R, Trausmuth G. Software evolution observations based on product release history. *ICSM '97: Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society: Bari, Italy, 1997; 160–166.
33. Godfrey MW, Tu Q. Evolution in open source software: a case study. *ICSM '00: Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society: San Jose, California, USA, 2000; 131–142.
34. Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM. Metrics and laws of software evolution—the nineties view. *Proceedings of the 4th International Symposium on Software Metrics*, IEEE Computer Society: Albuquerque, NM, USA, 1997; 20–32.
35. Antón AI, Potts C. Functional paleontology: system evolution as the user sees it. *Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society: Toronto, Ontario, Canada, 2001; 421–430.
36. His I, Potts C. Studying the evolution and enhancement of software features. *ICSM '00: Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society: San Jose, California, USA, 2000; 143–151.
37. Hou D, Wang Y. An empirical analysis of the evolution of user-visible features in an integrated development environment. *CASCON '09: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, ACM: Toronto, Ontario, Canada, 2009; 122–135.
38. Kothari J, Bespalov D, Mancoridis S, Shokoufandeh A. On evaluating the efficiency of software feature development using algebraic manifolds. *ICSM '08: International Conference on Software Maintenance*, Beijing, China, 2008; 7–16.
39. Greevy O, Ducasse S, Gîrba T. Analyzing software evolution through feature views: research articles. *Journal of Software Maintenance and Evolution* 2006; **18**:425–456, doi:10.1002/smr.v18:6.
40. Jiang ZM, Hassan AE. Examining the evolution of code comments in postgresql. *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, ACM: Shanghai, China, 2006; 179–180.
41. Fluri B, Wursch M, Gall HC. Do code and comments co-evolve? On the relation between source code and comment changes. *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, IEEE Computer Society: Vancouver, BC, Canada, 2007; 70–79, doi:10.1109/WCRE.2007.21.
42. Fluri B, Würsch M, Giger E, Gall HC. Analyzing the co-evolution of comments and source code. *Software Quality Control* 2009; **17**:367–394, doi:10.1007/s11219-009-9075-x.
43. Johnson MW. Monitoring and diagnosing application response time with ARM. *Proceedings of the IEEE Third International Workshop on Systems Management*, IEEE Computer Society: Newport, RI, USA, 1998; 4–15.
44. Yuan D, Zheng J, Park S, Zhou Y, Savage S. Improving software diagnosability via log enhancement. *ASPLOS '11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM: Newport Beach, California, USA, 2011; 3–14.
45. Godfrey MW, Germán DM. The past, present, and future of software evolution. *FoSM: Frontiers of Software Maintenance*, Beijing, China, 2008; 129–138.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES:

Weiyi Shang is a PhD student in the Software Analysis and Intelligence Lab at Queen's University (Canada). He obtained his MSc degree from Queen's University (Canada). His research interests include mining software repositories, large-scale data analysis, software performance engineering, source code duplication, reverse engineering and dynamic analysis.

Zhen Ming Jiang is a PhD student at Queen's University in Canada. He received both his BMath and MMath degrees from the University of Waterloo in Canada. His research interests include dynamic analysis, software performance engineering, source code duplication, mining software repositories and reverse engineering.

Bram Adams is an assistant professor at the École Polytechnique de Montréal, where he heads the MCIS lab on Maintenance, Construction and Intelligence of Software. He obtained his PhD at Ghent University (Belgium) and was a postdoctoral fellow at Queen's University (Canada) from October 2008 to December 2011. His research interests include software release engineering in general, and software integration, software build systems, software modularity and software maintenance in particular. His work has been published at premier venues such as ICSE, FSE, ASE, MSR and ICSM, as well as in major journals such as TSE, EMSE, JSS and JSME. He has co-organized five international workshops, has been tool demo and workshop chair at ICSM and WCRE and was a program chair for the 3rd International Workshop on Empirical Software Engineering in Practice (2011) in Nara, Japan. He currently is a program co-chair of the ERA-track at the 2013 IEEE International Conference on Software Maintenance, as well as of the 2013 International Working Conference on Source Code Analysis and Manipulation, both taking place in Eindhoven, The Netherlands.

Ahmed E. Hassan is the NSERC/RIM Software Engineering chair at the School of Computing in Queen's University. He spearheaded the organization and creation of the mining software repositories conference and its research community. He co-edited special issues of the IEEE Transaction on Software Engineering and the Journal of Empirical Software Engineering on the mining software repositories topic. Early tools and techniques developed by his team are already integrated into products used by millions of users worldwide. His industrial experience includes helping architect the Blackberry wireless platform at RIM and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. He is the named inventor of patents at several jurisdictions around the world including the USA, Europe, India, Canada, and Japan.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr

Michael W. Godfrey is an associate professor in the David R. Cheriton School of Computer Science at the University of Waterloo. His research interests span many areas of software engineering including software evolution, mining software repositories, reverse engineering, program comprehension and software clone detection and analysis.

Mohamed Nasser is currently the manager of Performance Engineering team for the BlackBerry Enterprise Server at Research In Motion. He has a special interest in designing new technologies to improve performance and scalability of large communication systems. This interest has been applied in understanding and improving communication systems that are globally distributed. He holds a BS degree in Electrical and Computer Engineering from The State University of New Jersey-New Brunswick.

Parminder Flora is currently the director of Performance and Test Automation for the BlackBerry Enterprise Server at Research In Motion. He founded the team in 2001 and has overseen its growth to over 40 people. He holds a BS degree in Computer Engineering from McMaster University and has been involved for over 10 years in performance engineering within the telecommunication field. His passion is to ensure that Research In Motion provides enterprise class software that exceeds customer expectations for performance and scalability.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/smr