On the Temporal Relations between Logging and Code

Zishuo Ding*, Yiming Tang*, Yang Li†, Heng Li‡, Weiyi Shang*
*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
†Beijing University of Posts and Telecommunications, Beijing, China
‡Polytechnique Montréal, Montreal, Canada
Email: *{zi_ding, t_yiming, shang}@encs.concordia.ca, †liyang1999@bupt.edu.cn, ‡heng.li@polymtl.ca

***Abstract*—Prior work shows that misleading logging texts (i.e., the textual descriptions in logging statements) can be counterproductive for developers during their use of logs. One of the most important types of information provided by logs is the *temporal information* of the recorded system behavior. For example, a logging text may use a *perfective aspect* to describe a fact that an important system event has *finished*. Although prior work has performed extensive studies on automated logging suggestions, few of these studies investigate the temporal relations between logging and code. In this work, we make the first attempt to comprehensively study the temporal relations between logging and its corresponding source code. In particular, we focus on two types of temporal relations: (1) logical temporal relations, which can be inferred from the execution order between the logging statement and the corresponding source code; and (2) semantic temporal relations, which can be inferred based on the semantic meaning of the logging text. We first perform qualitative analyses to study these two types of logging-code temporal relations and the inconsistency between them. As a result, we derive rules to detect these two types of temporal relations and their inconsistencies. Based on these rules, we propose a tool named TempoLo to automatically detect the issues of temporal inconsistencies between logging and code. Through an evaluation of four projects, we find that TempoLo can effectively detect temporal inconsistencies with a small number of false positives. To gather developers' feedback on whether such inconsistencies are worth fixing, we report 15 detected instances from these projects to developers. 13 instances from three projects are confirmed and fixed, while two instances of the remaining project are pending at the time of this writing. Our work lays the foundation for describing temporal relations between logging and code and demonstrates the potential for a deeper understanding of the relationship between logging and code.**

***Index Terms*—software logging, logging text, temporal relations**

I. Introduction

Logging statements are inserted by developers in the source code to record important runtime behaviors of software systems. Logging statements execute and produce useful information (i.e., logs) while systems are running. These generated logs are used in a variety of software engineering activities, such as failure diagnosis and system monitoring [1], [2], [3]. Listing 1 shows an example code snippet, which contains a logging statement (line 3). The logging statement has four components: (1) a logging object "`LOG`", (2) a verbosity level "`info`", (3) a dynamic variable "`this.rmAddress`", and (4) a logging text, "`Connected to ResourceManager at `". The content of a logging statement is typically written by developers.

Listing 1 A code snippet from `Hadoop` with a logging statement (line 3).

```
1 private void registerWithRM( ... {
2   this.resourceTracker = getRMClient();
3   LOG.info("Connected to ResourceManager at " +
        this.rmAddress);  ...
4   RegistrationResponse regResponse =
      this.resourceTracker.registerNodeManager(request)
          .getRegistrationResponse();
5   ...}
```

Logging texts provide high-level human-readable information and are usually written to describe the behaviors of the corresponding source code. Well-written logging texts can provide developers and other software practitioners with valuable information for system comprehension or failure diagnosis. Thus, it is important for developers to write proper logging texts for their logging statements. Recent work shows that incorrect logging texts often make the use of logs counterproductive [2], [4], [5], [6]. For example, according to a Hadoop issue report[1], the logging text of the logging statement in Listing 1 is misleading: the logging text indicates a perfective action ("connected"), while the source code corresponding to the action (line 4) is placed after the logging statement, causing an inconsistency between the textual description of the logging statement and its logical relationship with the corresponding source code. When the logging statement is executed and a log message is produced, the log message will provide the misleading information that the "connection" has been established while it is not. To avoid such confusion, the word "connected" in the logging text was changed to "connecting" (indicating a progressive action) in the patch that fixed the issue.

Prior work has performed extensive studies on software logging, including the studies that perform empirical investigations of logging practices [7], [8], [9], [10], that characterize and detect logging-related issues, as well as the studies that propose automated tools to support where to log [11], [12], [13], [14], what to log [7], [15], [16], and how to choose log levels [17], [18]. While a few studies indicate and discuss the importance of the relationship between logging and code [7], [2], none of them offer a satisfactory solution to address this

[1]https://issues.apache.org/jira/browse/MAPREDUCE-4262

issue. For example, a recent study by He et al. [7] finds that developers insert logging statements to describe three types of program operations (i.e., completed, current, and next operations). In their study, only the relative position between the logging statement and its corresponding code is considered, while the underlying intention in the developer-written logging text is ignored. However, both aspects are critical for logging quality and various log analysis tasks. As explained previously in the Hadoop logging example (Listing 1), the inconsistency between these two aspects can mislead practitioners who rely on analyzing logs for various tasks (e.g., debugging). Specifically, four of the logging benefits (e.g., knowing the status of an ongoing event) observed in a recent survey [2] would be impaired with such inconsistency. Moreover, by carefully examining real-life log anti-patterns [8] and log bugs data [4], we observe many real-world cases (over 20) fixing such inconsistency.

Therefore, in this work, we make the first attempt to comprehensively study the relationship between logging and its corresponding code. In particular, we focus on the temporal relations. Specifically, we study the logging-code temporal relations from two perspectives: (1) **logical temporal relations**, which can be inferred from the execution order between the logging statement and its corresponding source code; and (2) **semantic temporal relations**, which can be inferred based on the semantic meaning of the logging text. While the logical temporal relations describe *what the actual order is in the code*, the semantic temporal relations describe *what the order is inferred from the generated logs*. When there is an inconsistency between these two relations (i.e., **temporal inconsistency**), it is misleading to practitioners who rely on the clues provided by logs to understand system runtime behaviors [19], [20]. In this work, we first perform qualitative analyses to study these two types of logging-code temporal relations and the issues of temporal inconsistency. Based on the observations from our qualitative analyses, we also propose a tool (named **TempoLo**) to automatically detect the issues of temporal inconsistency in the source code.

We evaluate our tool on four open-source projects. Our evaluation contains three parts: (1) applying our tool to the logging statements used in our qualitative study, which shows that the tool can cover a majority (78.8%) of the manually identified temporal inconsistencies; (2) applying our tool to the remaining logging statements that are not manually analyzed, which shows that the tool can successfully detect another 326 inconsistencies with a relatively small number of false positive cases (48 cases and almost half are caused by the dependent NLP library); and (3) applying the tool to another dataset of temporal inconsistencies which we collected from the commit history of the studied projects, which shows that our tool can detect 83.3% of the inconsistencies. To gather developers' feedback on whether such inconsistencies are worth fixing, we report 15 detected instances from these projects to developers. 13 instances from three projects are confirmed and fixed, while two instances of the remaining project are still pending at the time of this writing.

The contributions of this paper include:

- We provide empirical observations on the temporal relations between logging and code.
- We derive rules to detect the logical and semantic temporal relations between logging and code, as well as rules to detect logging-code temporal inconsistencies.
- We implement a tool that can automatically detect logging-code temporal inconsistencies in the source code.

Our work is an important step toward analyzing the relationship between logging and code. Our empirical observations and tool can raise developers' and researchers' awareness of the importance of the temporal relations between logging and code, to avoid and identify temporal relation-related bugs. Our research also sheds light on promising research opportunities that exploit other types of relations between logging and code (e.g., semantic inconsistencies) to improve software logging or detect logging anti-patterns in the source code, which will potentially improve the overall quality of software logging.

Paper Organization. Section II presents the background of temporal relations. Section III describes our subject projects, and data collection, and gives an overview of our study. Section IV describes the approaches and results of our qualitative study of the temporal relations between logging and code. Section V presents the implementation and evaluation of our tool that automatically detects logging-code temporal inconsistencies which is based on the observations from our qualitative study. Section VI and Section VII discuss the threats to the validity of our results and the related work of our study, respectively. Finally, Section VIII concludes the paper.

II. Background

In this section, we present the concepts of temporal relations that are widely studied in the natural language community.

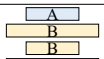
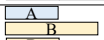
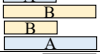
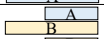
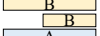
Before describing temporal relations, we first focus on the concept of "event" which is a fundamental term in natural language. **Event** is defined as a situation that happens or occurs [21], and it is often expressed in verbs (as shown in the example below) to describe an action or a transition. Another important concept is the semantic relation, also called **temporal relation**, that holds between relevant events. Given the following example,

> *The server **stopped** unexpectedly, we are **starting** it again.*

According to the definition of events, in this sentence, there are two events, **stopped** (E1) and **starting** (E2). The occurring order of these two events is the temporal relation. In this example, event E2 should happen after event E1.

Identifying the events and the temporal relations among them plays a vital role in many natural language processing (NLP) tasks, such as temporal information extraction (IE) [22], question answering [23] and knowledge base (KB) construction. To better annotate the events and temporal relations, researchers have proposed several representation schemas (e.g., Allen's interval algebra [24], STAG [25], and TimeML) [21]. In our work, we mainly focus on Allen's interval algebra [24], as it has become the standard representation [22], [26].

TABLE I: Allen's 13 temporal relations.

Relation (A to B)	Visualization	Explanation
Before		A ends before B starts
After		A starts after B ends
During		A starts and ends while B is ongoing
Contains		B starts and ends while A is ongoing
Overlaps		A starts before B and ends during B
Overlapped-by		B starts before A and ends during A
Meets		A ends at the point B begins
Met-by		B ends at the point A begins
Starts		Share the start point, but A ends before B ends
Started-by		Share the start point, but B ends before A ends
Finishes		A and B share end point, but A begins later
Finished-by		A and B share end point, but B begins later
Equals		A and B start and end at the same time

Allen [24] first proposed the interval-based algebra for representing temporal relations that may exist between any event pair in natural language. The representation consists of a set of 13 distinct and exhaustive interval relations (i.e., the relations between two time intervals) and are listed in Table I [27], [26], where A and B are two relevant events. The previous example describes two events, E1 and E2, and the corresponding temporal relation should be E2 **after** E1 (or E2 **met-by** E1, as it is possible that E2 starts when E1 ends.).

Since the appearance of Allen's algebra, researchers also propose modified temporal relations for capturing temporal relations in different domains [25], [28], [29], [27]. For example, IV et al. [28] combine Allen's algebra and TimeML schema and identify five temporal relations for clinical narratives. Besides, Mostafazadeh et al. [27] find that using a number of four relations is sufficient to handle the inter-event temporal relation in ROCStories corpus [30]. Considering the wide research of temporal relations for NLP, we would like to examine whether we can formally define our own set of temporal relations to model the relations between the logging statements and source code.

III. Study Setup

In this section, we describe the setup of our study[2]. Figure 1 shows an overview of our study. Overall, our study contains two parts: (1) a qualitative study of the temporal relations between the logging statement and source code, and the logging-code temporal inconsistencies, and (2) the implementation and evaluation of a tool for automatically identifying the temporal inconsistencies. Below, we first present our subject projects and the collection of data for the qualitative study, then we provide an overview of our qualitative study and the implementation and evaluation of our tool.

A. Subject projects

We base our case study on four open-source Java projects: `Hadoop`, `Tomcat`, `JMeter` and `ActiveMQ`. The four selected projects are from different application categories: `Hadoop` is a distributed server-side data processing system; `Tomcat` is a server-side application used for powering web applications; `JMeter` is a client-side software for conducting load testing; `ActiveMQ` is a middleware project that provides useful messaging service for both the server and client-side projects. We choose the subject projects since they are widely used and actively maintained, and have been studied in prior research [8], [5]. The details of the studied versions of these projects are listed in Table II. The source lines of code of the studied projects range from 145K to 1.8M. These projects have about ∼2K to ∼13K logging statements, among which 94.3% to 97.6% have logging texts.

TABLE II: Details of the studied projects.

Project	Version	SLOC	# of logging statements	# of logging statements with text
ActiveMQ	5.6.0	412K	2,139	2,087 (97.6%)
Hadoop	3.4.0	1.8M	13,204	12,463 (94.3%)
JMeter	5.5.0	145K	1,932	1,842 (95.3%)
Tomcat	10.1.0	349K	2,590	2,477 (95.6%)
Total		2.7M	19,865	18,868 (95.0%)

B. Data collection: logging statements and logging statement changes

As shown in Figure 1, we collect logging statements data from the source code of the subject projects to perform our qualitative study of the temporal relations between logging and code and to evaluate our tool that detects temporal inconsistencies. In addition, we collect logging statement changes data from the version control repositories of the subject projects to evaluate our tool. Below, we describe our data collection processes.

1) Collecting logging statements: We collect logging statements from the latest versions (as indicated in Table II) of the four subject projects at the time of writing this paper[3]. We use static analysis and regular expressions to identify the logging statement and the method that contains the logging statement. More specifically, we use JavaParser [31] to find out methods that are invoked in each Java file as logging is typically a method call (e.g., log.info()), then use regular expressions to filter out the logging statements using keywords (e.g., "log", "logger", "logging"). The statistics of the collected logging statements are shown in Table II.

To conduct our manual analysis, we then do random sampling with 95% confidence level and a 5% confidence interval [32] on the collected logging statements. We finally select 326, 373, 321, and 335 logging statements (1,355 samples in total) for `ActiveMQ`, `Hadoop`, `JMeter` and `Tomcat`, respectively. Note that we have sampled from each subject individually, rather than combining all the logging statements into one dataset for sampling, though it would dramatically reduce our manual labeling efforts (from 1,355 samples to 377 samples). The main reason is that the distribution of the logging statements across different projects is highly imbalanced (i.e., the differences in the number of logging

[2]The replication package is available at https://github.com/senseconcordia/TempoLo-replication-package

[3]The time of the data collection is July 2021.

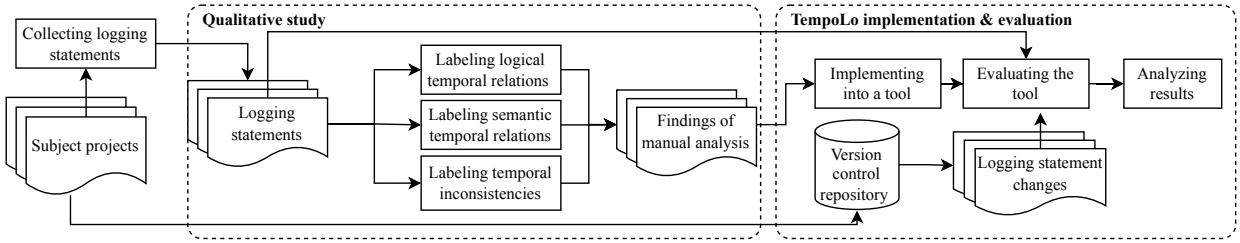


Fig. 1: An overview of our study.

statements in different projects), thus to better understand the characteristics of the temporal relations between the logging statements and the target source code from a wider perspective, we do the sampling for each project individually.

2) Collecting logging statement changes: We also construct an oracle dataset from the commit history of our subject projects to further evaluate the performance of the tool. Our idea is that: if developers change the position of a logging statement relative to its corresponding code, it indicates a potential temporal inconsistency in the first place. We collect all the commits of the four studied projects and keep the commits that have logging statement changes. Then, we manually examine the logging statement changes that fix temporal inconsistencies. We detail the steps in Section V-D3.

C. Overview of the qualitative study

With the sampled logging statements (see III-B1), we manually label the temporal relations between the logging statement and the target source code. Four authors of the paper jointly perform the labeling process.

We analyze the logging-code temporal relations from two perspectives: (1) **logical temporal relations**, where our assumption is that the logging statement and its corresponding source code can be considered as a different but relevant event pair and the temporal relation (in other words, execution order) of the event pair can be inferred from their relative position in the code snippet; and (2) **semantic temporal relations**, where the action described in the logging text and the execution of the logging statement is regarded as an event pair, and the temporal relations can be inferred from the semantic meaning of the logging text, which is similar to existing NLP tasks [29], [22].

We then manually identify the temporal inconsistencies. We consider that there exists a temporal inconsistency if the two types of relations violate each other. We detail our qualitative study steps in Section IV.

D. Overview of our tool implementation and evaluation

In this step, based on the observations from our qualitative study, we implement a tool named **TempoLo** that automatically detects the logging-code temporal relations and the temporal inconsistencies for a given logging statement and its containing method. The tool implements three functionalities: (1) logical temporal relation detection, (2) semantic temporal relation detection, and (3) temporal inconsistency detection.

To evaluate the effectiveness of **TempoLo**, we apply it to the unsampled logging statements (the remaining logging statement after sampling for manual labeling) to examine its ability to detect new temporal inconsistencies. As there is no readily available oracle dataset for temporal inconsistencies, we also construct an oracle dataset from the commit history of the subject projects for further evaluation. Finally, we reported the detected inconsistencies as bugs through issue reports and pull requests to the developers of the four studied projects. We detail the steps and results in Section V.

IV. A Qualitative Study

In this section, we describe the steps of our qualitative study that aims to understand the temporal relations between logging and code, as well as to identify the inconsistencies between the temporal relations inferred from the code (i.e., logical temporal relations) and that inferred from the semantic meaning of the logging text (i.e., semantic temporal relations).

A. Studying logical temporal relations

In this section, we manually label and analyze the logical temporal relations.

1) Labeling logical temporal relations: We follow a three-step manual labeling process:

Step 1. We start by manually labeling a random sample of 406 logging statements (i.e., 30% of the total 1,355 sampled logging statements). In this step, we employ four authors of the paper to do the labeling. The logical temporal relation of each logging statement is labeled by two annotators separately. Therefore, each annotator is assigned 203 logging statements to label. Each logging statement together with its corresponding code snippet is provided to the annotators using a URL that locates the logging statement in the corresponding GitHub repository. The annotators decide the most appropriate relation for each logging statement from Allen's 13 temporal relations.

Step 2. Once the 406 logging statements are labeled, the four annotators compare their labeling results. The results in this step have a substantial agreement for the labeling of the logical temporal relations (Cohen's Kappa of 0.81). As each logging statement is labeled by two annotators, and thus, when there is any disagreement of the labeling, the other two annotators would join and discuss until reaching a consensus.

Step 3. Based on the common understanding of the labels obtained from the last step, the remaining 949 from the total sample of 1,355 logging statements are equally distributed to the four annotators, then, each author labels around 237 logging statements individually.

Note that there exist some logging statements that do not have explicit temporal relations with the source code, and the

annotators simply label the temporal relation as "N/A". For example, considering the code snippet in Table III(a), there is no clear target source code for the logging statement, and thus, it is impossible to infer a temporal relation.

TABLE III: An overview of the logical temporal relations.

Relation	Code snippet

N/A	`private void handleBrowse(SampleResult ...`
	`  LOGGER.debug("isBrowseOnly");`
	`  StringBuilder sb = new ...`
	`  }`
	(a) JMSSampler.java (JMeter)

During	`public void setRunning(boolean running,`
	`  String host) {`
	`  log.info("setRunning({}, {})", running,`
	`  host); ...`
	`  }`
	(b) JMeterMenuBar.java (JMeter)

Before	`public void run() { ...`
	`  log.debug("Sampler start");`
	`  ...`
	`  sample();`
	`  ...}`
	(c) AbstractPerformanceSampler.java (ActiveMQ)

After	`public void onAMQPData(Object command) ...`
	`  frame = header.getBuffer();`
	`  ...`
	`  LOG.trace("Server: Received from`
	`  client: {} bytes", ...`
	`  }`
	(d) AmqpConnection.java (ActiveMQ)

Meets	`void waitForAuthentication() ...`
	`  LOG.debug("Waiting for authentication`
	`  response");`
	`  handler.waitForAuthentication();`
	`  }`
	(e) Application.java (ActiveMQ)

Met-by	`protected void unregisterProducer( ...`
	`  managementContext.unregisterMBean(key);`
	`  } catch (Throwable e) {`
	`  LOG.warn("Failed to unregister MBean`
	`  {}", key);`
	`  ...`
	`  }`
	(f) ManagedRegionBroker.java (ActiveMQ)

2) Identified logical temporal relations: In total, we have identified five logical temporal relations between the logging statement and its corresponding source code, as shown in Table III. As compared with Allen's 13 relations (cf., Section II), we do not include the relations **contains, overlaps, overlapped-by, starts, started-by, finishes, finished-by** and **equals**. The reason is that we consider the execution of a logging statement as a time point and the corresponding code as a time interval. Below, we discuss each of our identified logical temporal relations.

During: The logging statement executes while the target source code is ongoing. As shown in Table III(b), the corresponding code of the logging statement is the whole method, and the logging statement executes while the method is ongoing, thus, we label the relation as **during**.

Before: The logging statement executes before the corresponding source code. As shown in Table III(c), the corresponding code of the logging statement is the method "`sample`", and there is other source code between the logging statement and the corresponding code, thus, we label the relation as **before**.

After: The logging statement executes after the corresponding source code. As shown in Table III(d), the target code of the logging statement is the assignment operation "`frame = header.getBuffer();`", and there is another source code between the logging statement and the corresponding code, thus, we label the relation as **after**.

Meets: The logging statement executes right before the corresponding source code. As shown in Table III(e), the corresponding code of the logging statement is the method "`waitForAuthentication()`", and there is no other source code between the logging statement and the corresponding code, thus, we label the relation as **meets**.

Met-by: the logging statement executes right after the corresponding source code. As shown in Table III(f), the corresponding code of the logging statement is the method "`unregisterMBean(key)`", and there is no other operations between the logging statement and the corresponding code, thus, we label the relation as **met-by**.

3) Findings from labeling logical temporal relations: During the labeling process, the most challenging part is locating the target source code of the logging statement. However, we find that the main verb in the logging text, which typically shows the action that occurs in a sentence, can be effectively used for locating the corresponding source code. Therefore, we list observations for locating the corresponding source code based on the logging text. The number in the parentheses is the proportion of these matches respectively.

Finding L.1: The main verb in the logging text (partially) matches the corresponding source code (40.4%). For instance, in Table III(e), the main verb of the logging text "Waiting for authentication response" is "waiting", which partially matches "waitForAuthentication" in the statement "`handler.waitForAuthentication();`". As the logging statement is logically executed right before the corresponding code statement, we can label the relation as **meets**.

Finding L.2: The direct object or subject of the main verb in the logging text (partially) matches the corresponding source code (18.3%). The direct subject (object) is usually a noun or noun phrase that performs (receives) the action of the main verb. For instance, in Table III(c), the logging text is "Sampler start" and the main verb is "start", whose subject "Sampler" partially matches the statement "`sample();`". There are other statements between the logging statement and the corresponding code, we, therefore, label the relation as **before**.

Finding L.3: The synonyms of the main verb in the logging text match the corresponding source code (11.1%). Developers also use synonyms of the main verb to describe the source code. As shown below, the corresponding code for the logging statement invokes the "append" method; while developers choose to use its synonym, "add" to describe the action.

```
private void initUserTags(BackendListenerContext ...
   userTagBuilder.append(',')
      .append(...tagToStringValue(tagName)) ...
   log.debug("Adding '{}' tag with '{}' value ",
       tagName, tagValue); ...}
```

Finding L.4: The main verb in the logging text implicitly matches an operator of the corresponding source code (17.3%). For example, in Java, the phrase "instantiating a class" means calling the constructor of a class and creating an object of that class, which is done by using the "new" keyword. Below is an example, where "Instantiating" is used to describe the "new" operation.

```
private void startWebApp() { ...
   AHSWebApp ahsWebApp = new AHSWebApp(...
   LOG.info("Instantiating AHSWebApp at " + ...}
```

Finding L.5: The open clausal complement of the main verb in the logging text matches the target source code (10.0%). The open clausal complement of the main verb is usually a predicative or clausal complement, which can also convey actions of the source code. For instance, in Table III(f), the logging text is "Failed to unregister MBean" and the main verb is "Failed", whose open clausal complement, "unregister", matches the method "`unregisterMBean()`".

B. Studying semantic temporal relations

In this section, we study semantic temporal relations that can be inferred from the semantic meaning of logging text.

1) Labeling semantic temporal relations: To get the labels of the logging texts, we follow the same three-step manual labeling process as in Section IV-A1. We also achieve a substantial agreement on the semantic temporal relations (Cohen's Kappa of 0.83). Besides, there also exist logging statements of which the textual descriptions cannot be used to infer temporal relations. For instance, in Table IV(a), there is no explicit event that can be extracted, and the authors simply label the temporal relation as "**N/A**".

TABLE IV: An overview of the semantic temporal relations.

Relation	Logging statement
N/A	(a) `log.debug("Arg: {}", arg)`
During	(b) `log.info("setRunning({}, {})", running, host);`
Before	(c) `LOG.debug("Active scan starting")`
After	(d) `log.info("Thread started:{}", ...);`

2) Identified semantic temporal relations: In total, we have identified three semantic temporal relations inferred from the logging text, as shown in Table IV.

Considering the intrinsic similarity between this labeling process and the 2007 TempEval challenge in NLP [29], [22] as well the fact that it is hard to be certain about whether two events occur exactly during the same time or starting/ending right after/before each other [27] (e.g., we are unable to ensure that the "starting" would exactly occur right after the logging statement.), we thus follow the previous work [29] and restrict the original Allen's 13 interval relations to a set of three (**before**, **after** and **during**) temporal relations, which based on observation are enough to capture the semantic temporal relations. Below, we discuss our identified set of three semantic temporal relations.

During: The logging statement is called while the event described in the logging statement is ongoing. As shown in Table IV(b), the event is the "`setRunning`", and we can infer that the logging statement should execute while the event is ongoing, thus, we label the relation as **during**.

Before: The logging statement is called before the event. As shown in Table IV(c), the event is "`starting`", which should occur after the execution of the logging statement, thus, we label the relation as **before**.

After: The logging statement is called after the event. As shown in Table IV(d), the logging statement should be executed after the event "`started`", thus, we label the relation as **after**.

3) Findings from labeling semantic temporal relations: We discuss the findings that can be used for inferring the semantic temporal relations. We find that the tense and aspect of the main verb are two major pieces of evidence that can be used to infer semantic temporal relations. Tense and aspect are two important concepts in natural language embodying the linguistic encoding of time [33], [34]. There are two tenses in English, past and present, and two aspects, perfective and progressive, indicating that the action is complete or ongoing, respectively.

Finding S.1: If the main verb has a past tense or perfective aspect, the relation is often after. As shown in Table IV(d), the main verb "started" has a tense of past, and thus is labeled a relation, **after**. However, we find that in some cases, just tense solely can not determine the semantic relation, for instance, the expression "have stopped" has an **after** relation, but the tense is present. In this situation, we consider the aspect, as the aspect is perfective.

Finding S.2: If the main verb has a present tense or progressive aspect, the relation is often before. In Table IV(c), the main verb "starting" has a present tense and progressive aspect, and the temporal relation is labeled as **before**. Note that the main verb with a past tense can still have a progressive aspect, for example, given the verb phrase "was stopping", it has a progressive aspect but past tense, for such cases, we label them as **after**.

Finding S.3: If the logging text is in CamelCase and does not contain any explicit tense or aspect, the relation may be during. In Table IV(b), the logging text "setRunning" is in CamelCase and does not contain any verbs and thus the temporal relation is labeled as **during**.

C. Studying temporal inconsistencies

We have labeled the temporal relations for the collected logging statements in previous steps. As one of our goals is to uncover the patterns of temporal inconsistencies (i.e., the inconsistencies between the logical and semantic temporal relations), in this section, we first describe how to identify the temporal inconsistencies and then describe the findings.

Our way to identify the inconstancy is straightforward: if the temporal relation inferred from the source code (i.e., logical

temporal relation) violates the temporal relation inferred from the textual description of the logging statement (i.e., semantic temporal relation), we label the logging statement as a logging statement with a temporal inconsistency. Below, we describe three intuitive rules for such inconsistencies (see Table V).

Rule 1: The temporal relation inferred from the source code is After or Met-by, but the temporal relation inferred from the logging text is Before. For instance, in Table V(a), the logging statement "`LOG.info("Adding a new node: ");`" starts right after its target source code "`clusterMap.add(node)`", thus has a **met-by** relation inferred from the code. However, the semantic temporal relation inferred from the text of the logging statement is **before**, as the logging statement should execute earlier than the "Adding" event. The correct event expression should be "Added", or the logging statement should be moved to the line above the code "`if (clusterMap.add(node)){`".

Rule 2: The temporal relation inferred from the source code is Before or Meets, but the temporal relation inferred from the logging text is After. For instance, in Table V(b), the logging statement "`log.warn("Existing AuthManager {} superseded by {}")`" executes before its target source code "`setProperty()`", and thus has a **before** relation inferred from the code. However, the temporal relation inferred from the text of the logging statement is **after**, as the execution of the logging statement should occur later than the "superseded" event.

Rule 3: The temporal relation inferred from the source code is During, but the temporal relation inferred from the logging text is After or Before. For instance, in Table V(c), both the logging statements execute while the target method is ongoing, and thus they have a **during** relation. However, the temporal relations inferred from the texts of the two logging statements are **after** and **before**, respectively, as the execution of the logging statements should occur later and earlier than the "stopped" and "starting" events, respectively. Ideally, the first logging statement should execute after the "`registerHost()`" method, and the second one should execute before the "`registerHost()`" method. However, as we stated in Section IV-A2, the execution of a logging statement is considered as a time point and thus, both should be moved to the end and start point of the method, respectively.

D. Summary of our qualitative study

In previous sections, we have described our findings from the qualitative study. Here, we discuss the distribution of the logical and semantic temporal relations, as well as the temporal inconsistencies that we identified in the sampled dataset. Table VI shows the statistics of the temporal relations and the identified inconsistencies in the four studied projects.

Developers prefer to insert the logging statement right after/before the corresponding source code. Table VI shows that about 73.4% of the sampled logging statements are located next to (either right before or after) the corresponding source code (i.e., with **meets** or **met-by** logical temporal relations).

Compared to inserting the logging statements before the target source code, developers prefer to see logs after the target source code being executed. Table VI shows that about 61.8% of the sampled logging statements are inserted (right) after the target source code (i.e., with **met-by** or **after** logical temporal relations). This observation is consistent with the findings of previous work [8], [15] that logging statements are more relevant to their pre-log code.

There exist a non-negligible amount of inconsistencies (i.e., 2.4%) between the logical and semantic temporal relations, which can potentially confuse the end users and make the use of logs counterproductive [2]. Our Rule 1 can detect more than twice the temporal inconsistencies as Rule 2. This gap may be caused by the fact that some developers just insert the base form (i.e., with present tense) of a verb in the logging text and put the logging statement after the target source code, paying little attention to the tense or aspect of the action itself. Note that Rule 3 is deduced from the correct cases. In particular, we find that developers may put the method name together with a verb indicating the start of the execution at the first line of the method body. Therefore, we did not detect any inconsistency by Rule 3 during our labeling.

Inspired by our qualitative study and the non-negligible amount of inconsistencies, we decide to implement our findings into a tool to assist developers in automatically detecting the logging-code temporal inconsistencies in the source code.

V. TempoLo: Automatically Detecting Temporal Inconsistencies between Logging and Code

In this section, we propose a tool, **TempoLo**, which automatically detects inconsistencies between the logical and semantic logging-code temporal relations, based on our findings from Section IV. Formally, given a logging statement and the method that contains the logging statement, the proposed tool can automatically analyze both their logical and semantic temporal relations and detect whether there is a temporal inconsistency. **Tempolo** is built as a static code analyzer that could be integrated into an IDE in practice, of which the storage needed is in a matter of KBs and time cost is almost negligible. We detail our implementation and evaluation in the rest of this section.

A. Detecting semantic temporal relations

As we discussed in Section IV-B3, the aspect and tense of the main verb can be effectively used to detect the semantic temporal relations. In this section, we describe how to extract the logging text, and its main verb with tense and aspect.

1) Extracting the logging text: Following the approach used in prior work [15], we first extract the logging texts and variables from the logging statements which are collected in Section III-B. Since our focus is on the main verb of the logging text, we replace the variables with a wildcard (`VID`). For instance, the extracted logging text of the logging statement in Table IV(d) is "`Thread started: VID`". Note that some projects (e.g., Tomcat) use an internationalization/localization helper class to fill the text in the logging statement instead of inserting the logging text directly. For example, Tomcat provides multiple local strings files with different languages and uses a helper class and a key (e.g.,

TABLE V: An overview of the rules for labeling the inconsistencies of temporal relations.

Rule 1	Rule 2	Rule 3
The temporal relation inferred from the source code is **After** or **Met-by**, but the temporal relation inferred from the logging text is **Before**.	The temporal relation inferred from the source code is **Before** or **Meets**, but the temporal relation inferred from the logging text is **After**.	The temporal relation inferred from the source code is **During**, but the temporal relation inferred from the logging text is **After** or **Before**.
(a) NetworkTopologyWithNodeGroup.java (`Hadoop`)	(b) HTTPSamplerBase.java (`JMeter`)	(c) Modified MapperListener.java (`Tomcat`)

TABLE VI: The statistics of the manually labeled temporal relations.

	Types	ActiveMQ	Hadoop	JMeter	Tomcat	Total
Logical temporal relations	During	4	5	7	0	16
	Before	11	17	8	12	48
	After	24	44	24	34	126
	Meets	82	93	64	44	283
	Met-by	178	171	171	192	712
	N/A	27	43	47	53	170
Semantic temporal relations	During	5	5	5	0	15
	Before	93	119	75	61	348
	After	194	199	179	213	785
	N/A	34	50	62	61	207
Temporal inconsistencies	Rule 1	4	13	3	3	23
	Rule 2	4	1	4	1	10
	Rule 3	-	-	-	-	0
	Total	8	14	7	4	33

Note: We did not find any inconsistency that matches Rule 3, as Rule 3 is deduced from the correct cases during labeling.

in "`sm.getString("requestFacade.nullRequest")`", "sm" is the helper class, and "requestFacade.nullRequest" is the key) to retire the specific string. In such cases, we first convert the keys into the corresponding logging text using the underlying resource bundle (e.g., locale-specific resource).

2) Identifying the main verb and its tense and aspect: In this step, we use spaCy [35], an open-source NLP library, to perform dependency parsing and part-of-speech (POS) tagging on the extracted logging text. Dependency parsing is the task of defining the dependency relations (e.g., subject, object) between the tokens of a sentence. Part-of-speech (POS) tagging is the task of categorizing each token in the sentence with different types (e.g., verbs). Figure 2 shows the result (a dependency parsing tree) of performing dependency parsing and POS tagging on an example logging text. The arrows and labels at the top are the syntactic dependency relations and the tags at the bottom are the identified part-of-speech tags. Based on the dependency parsing tree, we consider the following two types of tokens as the main verb : (1) a token with a verb tag and its head (i.e., parent node) in the dependency tree is the token itself, or, (2) the first token with a verb tag. For example, in Figure 2, the head of the first token "Starting" is itself, and it has a tag "verb", thus "starting" is considered as the main verb of the logging text. Once we get the main verb, we can get the tense and aspect of the main verb using spaCy.

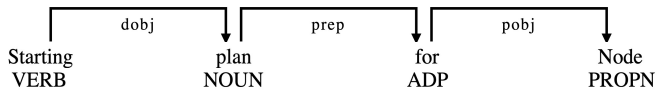


Fig. 2: A dependency parsing tree of an example logging text.

3) Identifying the semantic temporal relation: To identify the semantic temporal relation, we apply the findings (i.e., Finding S.1 - Finding S.3) in Section IV-B3 to the results from the previous step: if the main verb has a past tense or perfective aspect, we label the relation as **after**, and if the main verb has a present tense or progressive aspect, we label the relation as **before**. If the logging text is in CamelCase and does not contain any explicit tense or aspect, we label the relation as **during**.

B. Detecting logical temporal relations

As we discussed in Section IV-A3, locating the target source code of the logging statement plays a vital role in detecting the logical temporal relations. In this section, we use the same approach (cf., Section V-A) to extract the logging text and identify the main verb of the logging statement, after which, we perform lemmatization on the main verb to get its base form using spaCy. Below, we describe how we locate the source code using the lemmatized main verb.

1) Extracting all the potential statements: During our manual analysis, we also observe that the target source code is mainly method calls, assignment, return, if, else, or break statements. Therefore, in this step, we try to extract such statements as the potential statements. We first apply `srcML` to the method that contains the logging statement. `srcML` converts the source code into an XML tree, in which the tags provide the information of the potential statements. For example, all method calls are wrapped with a "call" tag. We then adopt XPath and Beautiful Soup[4] to extract the statements.

2) Locating the corresponding statements: Once having extracted statements, we need to locate the corresponding statement. We implement the five rules observed (i.e., Finding L.1 - Finding L.5) in Section IV-A3, to identify the corresponding statement[5]. To collect the synonyms of a given main verb, we adopt WordNet, which is a large lexical database of English [36].

3) Identifying the logical temporal relation: After we located the target source code, we extract its line number, which is an attribute of the code component in the XML tree. Then we compare it with the line number of the logging

[4]https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[5]We also implemented an ML-based approach to locate the target source code, but the top-1 accuracy is not satisfactory (only around 50%). Therefore, we opt to use the rule-based approach instead of the ML-based approach. Please refer to the replication package for more details.

statement and identify the logical temporal relations defined in Section IV-A2.

C. Detecting temporal inconsistencies

Based on the result from the previous two steps, we implement the rules observed in Section IV-C to detect temporal inconsistencies. For each logging statement, **TempoLo** would scan the pair of the detected logical and temporal relations to check whether they violate each other. Note that on one hand, there may exist some cases that do not have a verb in the logging text or we cannot match any target source code based on the rules, for such cases, we do not label their temporal relations. On the other hand, there is a possibility that one main verb may match more than one target source code in Section V-B2, and thus we collect multiple logical temporal relations for the logging statement. For these cases, we keep all the logical temporal relations, and if all the relations violate the semantic temporal relation, we then label it as a temporal inconsistency. By doing this, we can have a relatively acceptable accuracy and low false positive rate.

D. Evaluation

1) Evaluation on the manually identified logging statements: We first apply **TempoLo** to our manually identified logging statements. As shown in Table VI, we manually identify a total of 33 temporal inconsistencies, we run our tool on this dataset and can successfully detect 26 logging statements out of the 33 (i.e., 78.8%, with a false negative rate of 21.2%) having an inconsistent temporal relation. The results verify the usefulness of the findings and the correctness of the implementation. We then manually check the undetected cases, in order to further understand the reasons that may cause our tool to fail. We find two main reasons contributing to such detection errors: (1) **Mismatch of the target code.** In the step of determining the logical temporal relation of the logging statement, our tool returns the incorrect target source code. An example is shown below.

```
public NodePlan plan(DiskBalancerDataNode node) ...
      LOG.info("Starting plan for Node : {}:{}",
          node.getDataNodeName(), ...}
```

The main verb of the logging text is identified as "starting". The term "get" in the method call "getDataNodeName()" is one of the synonyms of "start". Thus, our tool made a mistake by matching the logging text to the "getDataNodeName()" method. (2) **No main verb found in the logging text.** Our tool relies on spaCy to determine the main verb as well as the tense and aspect of the verb. There exist some cases in which spaCy may not return an incorrect result. For example, given the following logging statement, "`log.trace("Registering key for read:"+ key)`", we can infer that the main verb is "registering" with a progressive state. However, spaCy recognizes "registering" as a noun.

2) Evaluation on the remaining unsampled logging statements: In this part of the evaluation, we would like to check whether **TempoLo** can detect unseen temporal inconsistencies while having a low false positive rate. Therefore, we evaluate **TempoLo** on the remaining unsampled logging statements (the remaining logging statements after sampling for manual labeling). We manually check the detected cases, in order to understand the false positive rate and the reasons that may cause our tool falsely report a correct case as an inconsistency.

TempoLo reports a total of 326 logging statements with temporal inconsistencies. Two authors manually checked each of them to determine whether it is a true positive or false positive. The two authors achieve an agreement ratio of 78.3% and finally identify 48 false positives after reaching a consensus. In general, the false positives are caused either by (1) an incorrect main verb or (2) an incorrect target source code. Below, we detail the reasons that may lead to false positives:

(1) Incorrect identification of the main verb. If the logging text contains more than one verb, **TempoLo** may not detect the correct main verb. For example, **TempoLo** wrongly identifies "ask" as the main verb for the logging statement "`LOG.info("Requested container ask: "+ ...);`" (the correct one should be "requested"). This error is essentially caused by spaCy, as our way to detect the main verb relies on the dependency parsing tree generated by spaCy. Besides, **TempoLo** may wrongly return a non-verb token as the main verb. For example, in the logging statement, "`LOG.info("included nodes = "+ ...);`", **TempoLo** identifies "included" as the main verb, while it is an adjective. This error is caused by the POS tagging of the tokens, which is provided by spaCy. It is reasonable that spaCy has low accuracy in analyzing logging texts (which is different from sentences in NLP) even though we use the state-of-the-art transformer-based model. Future work can train a model on the annotated software engineering data to improve the performance of spaCy for software engineering.

(2) Incorrect match of the target source code. **TempoLo** may not be able to locate the target source code. For example, in the logging statement "`LOG.info("..., moving files from ... "`", developers use "moving files" to describe the actions of the source code, however, the target source code is "`mergePaths()`". Moreover, some descriptions in the logging statements mean no action in the source code. For example, some developers use "stopping" in the catch block to indicate the end of the method, and **TempoLo** cannot detect such cases. To address this problem, developers can manually construct a project-orientated dictionary that maps the token in the logging text to the statements in the source code.

3) Evaluation on the logging statements collected from commit history: Finally, we evaluate whether **TempoLo** can detect historical issues of temporal inconsistencies between logging and code. Since such issues may not be all reported in the issue tracking system, we construct an oracle dataset from the commit history to further evaluate **TempoLo**. Below, we describe how we construct the dataset.

We first clone the git version control repositories of our subject projects and use "`git log`" to extract all the code commits. We then use "`git show`" to analyze the changes. In order to make sure the commit changes are relevant to logging statements, we focus on two types of commits: (1) only the

locations of the source code are changed and the changed code contains logging statements; and (2) logging statements are only modified by changing their temporal information (e.g., verb tense in the sentence) in the logging text. After this step, we gathered a total of 1,273 code changes from our studied projects. Intuitively, not all commits are indeed related to logging statements. Therefore, two authors manually examine each of the commits to confirm that the commit is related to a logging statement. The two authors achieve an agreement ratio of 86.0% and collect 59 commits that are logging statement-related code changes. Furthermore, as the focus of this study is the temporal inconsistency, we find that a majority of the 59 commits are caused either by (1) regular code changes (e.g., new code is added or deleted, causing the change of the logging statement) or (2) logging efficiency (e.g., moving the logging statement out of loops). Therefore, we continue filtering the commits and finally we extract a total of six logging statement changes that are related to temporal inconsistencies.

We then evaluate the tool on the six collected logging statements on their versions before the commit and **TempoLo** can successfully detect five of them (i.e., 83.3%, which is similar to the accuracy when being evaluated on the manually sampled dataset, with a false negative rate of 16.7%.). The one inconsistency case that our tool failed to detect is as follows:

```
+  log.info("Ending thread " + ...);
   allThreads.remove(thread);
-  log.info("Ending thread " + ...);
```

Our tool fails to detect this case since we cannot successfully match the action "remove" with the event "ending".

4) Reporting issues to developers: To gather developers' feedback on whether such inconsistencies are worth fixing, we report our detected instances to developers. To avoid spamming developers, we iteratively and gradually report issues to developers (e.g., by issue reports or PRs). We first only select and report two instances for each project to developers to know whether developers care about this kind of issue. Then if developers confirm the instances (e.g., by fixing the issue or accepting the PR), we further report more instances for that project. In the instance selection process, we prioritize the instances without ambiguity (e.g., we avoid instances that contain verbs that are the same in the present and past tense (e.g., "read")). We have reported 15 instances covering the four projects. So far, all 13 instances from three projects are confirmed and fixed, and two instances of the other project are still under discussion. There are two main strategies for fixing the reported inconsistencies: (1) moving the logging statement to the proper position, and (2) correcting the tense or aspect of the main verb of the logging statement[6].

> **TempoLo** can successfully detect the temporal inconsistencies in the source code with a low false positive rate of 14.7%. 13 out of 15 reported inconsistency instances have been fixed by developers and received positive feedback.

[6]The details of the pull requests can be found in our replication package.

VI. Threats to Validity

External Validity. As the study involves four Java open-source projects, the number of studied subjects and programming languages may pose a threat to the study's validity. To mitigate this, careful consideration goes into the selection of subjects. These four analyzed projects are well-known and have gained considerable attention from developers and researchers, based on the stars on GitHub and existing research papers [8], [5]. Besides, we did obtain consistent empirical results across the four studied projects. Furthermore, Java is a mature programming language that provides built-in logging and third-party logging frameworks. Many studies [37], [15], [5], [17] focus only on Java logging because of its abundance of logging usage. Considering the universality of logging, the findings from Java logging studies may be applicable to other programming languages, which will require further research.

Internal Validity. Since the study involves a manual study, its validity can be influenced by the knowledge and experiences of the participants. As a way to mitigate human bias, we use peer review to reach a consensus as a baseline for further review. Participants are all professional researchers in the field of software engineering.

Construct Validity. This study leverages several third-party tools to preprocess the source code, such as JavaParser and spaCy. These tools could have their limitations. For example, almost half of the detected false positives are caused by spaCy. However, all of the tools used in this study were used in previous studies [38], [39] and are well recognized in the Computer Science community. For example, JavaParser has ∼4K Github stars and spaCy has ∼23K stars at the time of writing this paper. We observe a 15% (48/326) false positive rate, while we admit that after all, our tool is a static analysis-based technique. 85% is rather on-par or above most static analysis-based tools. Besides, as our tool can pinpoint the location of inconsistencies, the cost of manually verifying a false positive is relatively low.

VII. Related Work

Studies on logs in classical SE research area. Following the first empirical study [40] on logging practice in 2012, a slew of further studies on the logging have emerged in SE research area. Shang et al. [41] perform a manual study of email threads from three open source systems' mailing lists, as well as sampled logging statements to better comprehend log lines. Kabinna et al. [42] manually review logging library migrations for Apache Software Foundation (ASF) projects. Hassani et al. [4] investigate log-related issues using a combination of empirical study, manual analysis, and an automated approach. Chen and Jian [8] analyze the characteristics of logging practices in ASF projects through a replication study of [40]. The studies mentioned above all involve manual inspections, which are also used in this work. In addition, there are also many studies providing automated software solutions for log analysis. For example, Tan et al. [43] use their automated approach to track control-flow and data-flow execution in distributed systems by looking at logs. Yuan et

al. [44] propose an automated approach of enhancing logging content using data flow analysis and control flow analysis. Tang et al. [45] present an automated approach to rejuvenate feature log levels. Our work also includes some automated work to reduce the amount of manual effort required for manual analysis. Furthermore, a prior work [5] groups log studies into three categories based on the problems each study is attempting to solve: how-to-log, what-to-log and where-to-log. In our work, we seek to alleviate the problem of how-to-log and where-to-log, by defining and leveraging temporal relations between logging and code.

Prior NLP research related to sequences. Cutting-edge NLP techniques have piqued the interest of many researchers in past few years. Many NLP studies involve sequence analysis or may facilitate it in the future. For example, Mostafazadeh et al. [27] present a semantic annotation framework for investigating the relations between events, including event sequence analysis. Derczynski [26] pinpoints the importance of event sequences and conducts several studies on the topic. Myers and Palmer [46] propose a Neural Network based approach to identify tense and verb aspects, and these results can be used to help order events. Despite the recent surge in NLP research, the majority of related work focuses on pure natural languages rather than logs that use natural languages to interpret programming languages but retain a lot of programming language morphology and syntax.

Prior work on log study, which is at the crossroads of NLP and software engineering. Research on NLP and logs typically focuses on log mining and analysis. For example, Locke et al. [47] apply n-gram models to identify event sequences for log analysis. Kobayashi et al. [48] propose using NLP to generate log templates from logs to assist with log analysis. Aussel et al. [49] leverage NLP for log parsing to enhance the performance of log mining. To the best of our knowledge, there has never been an NLP study that examines the sequence between logging statements and their context.

VIII. Conclusion

In this paper, we have formally defined two sets of temporal relations between the logging statement and the corresponding source code: logical and semantic temporal relations. Based on the defined temporal relations, we have concluded three rules for detecting the temporal inconsistencies that can jeopardize the quality of logging. We then implement the rules as a tool to automatically detect such inconsistencies. By analyzing the results, we find that our tool can successfully detect the temporal inconsistencies in the source code with a relatively low false positive rate. We have reported some detected inconsistencies to the developers of each of our subject projects and received positive feedback. Moreover, our research sheds light on the promising research opportunity of formalizing other logging-code relations to assist in various downstream software engineering tasks (e.g., improving the quality of the automatically generated logging texts [15] and more accurately representing the actual temporal status of the events described in the logs [50]). Future research may build more and improve existing log analysis approaches by incorporating the defined temporal relationship.

References

[1] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '10, 2010, pp. 143–154.

[2] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[3] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The bones of the system: A case study of logging and telemetry at microsoft," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, ser. ICSE Companion '16, 2016, pp. 92–101.

[4] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3248–3280, 2018.

[5] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 71–81.

[6] Z. Li, T. P. Chen, J. Yang, and W. Shang, "Studying duplicate logging statements and their relationships with code clones," *IEEE Transactions on Software Engineering*, pp. 1–19, 2022.

[7] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 178–189.

[8] B. Chen and Z. M. J. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[9] J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 341–352.

[10] Z. Li, T. P. Chen, J. Yang, and W. Shang, "DLFinder: characterizing and detecting duplicate logging code smells," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 152–163.

[11] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 415–425.

[12] Z. Li, T. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, 2020, pp. 361–372.

[13] Z. Ding, H. Li, W. Shang, and T.-H. P. Chen, "Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks," *Empirical Software Engineering*, vol. 27, no. 3, p. 63, 2022.

[14] ——, "Towards learning generalizable code embeddings using task-agnostic graph convolutional networks," *ACM Trans. Softw. Eng. Methodol.*, jun 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3542944

[15] Z. Ding, H. Li, and W. Shang, "Logentext: Automatically generating logging texts using neural machine translation," in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '22, 2022.

[16] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," *arXiv preprint arXiv:2201.04837*, 2022.

[17] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[18] Z. Li, H. Li, T. Chen, and W. Shang, "DeepLV: Suggesting log levels using ordinal based neural networks," in *Proceedings of the 43rd International Conference on Software Engineering, ICSE 2021*, 2021, pp. 1–12.

[19] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 397–400.

[20] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 402–411.

[21] J. Pustejovsky, R. Ingria, R. Saurí, J. M. Castaño, J. Littman, R. J. Gaizauskas, A. Setzer, G. Katz, and I. Mani, "The specification language timeml," in *The Language of Time - A Reader*, I. Mani, J. Pustejovsky, and R. J. Gaizauskas, Eds. Oxford University Press, 2005, pp. 545–558.

[22] X. Ling and D. S. Weld, "Temporal information extraction," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, M. Fox and D. Poole, Eds. AAAI Press, 2010. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1805

[23] F. Schilder and C. Habel, "Temporal information extraction for temporal question answering," in *New Directions in Question Answering, Papers from 2003 AAAI Spring Symposium, Stanford University, Stanford, CA, USA*, M. T. Maybury, Ed. AAAI Press, 2003, pp. 35–44.

[24] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[25] A. Setzer and R. Gaizauskas, "A pilot study on annotating temporal relations in text," in *Proceedings of the Workshop on Temporal and Spatial Information Processing - Volume 13*, ser. TASIP '01. USA: Association for Computational Linguistics, 2001. [Online]. Available: https://doi.org/10.3115/1118238.1118248

[26] L. Derczynski, "Determining the types of temporal relations in discourse," Ph.D. dissertation, University of Sheffield, 2013.

[27] N. Mostafazadeh, A. Grealish, N. Chambers, J. Allen, and L. Vanderwende, "CaTeRS: Causal and temporal relation scheme for semantic annotation of event structures," in *Proceedings of the Fourth Workshop on Events*. San Diego, California: Association for Computational Linguistics, jun 2016, pp. 51–61. [Online]. Available: https://aclanthology.org/W16-1007

[28] W. F. S. IV, S. Bethard, S. Finan, M. Palmer, S. Pradhan, P. C. de Groen, B. J. Erickson, T. A. Miller, C. Lin, G. K. Savova, and J. Pustejovsky, "Temporal annotation in the clinical domain," *Trans. Assoc. Comput. Linguistics*, vol. 2, pp. 143–154, 2014.

[29] M. Verhagen, R. J. Gaizauskas, F. Schilder, M. Hepple, G. Katz, and J. Pustejovsky, "Semeval-2007 task 15: Tempeval temporal relation identification," in *Proceedings of the 4th International Workshop on Semantic Evaluations, SemEval@ACL 2007, Prague, Czech Republic, June 23-24, 2007*, E. Agirre, L. M. i Villodre, and R. Wicentowski, Eds. The Association for Computer Linguistics, 2007, pp. 75–80. [Online]. Available: https://aclanthology.org/S07-1014/

[30] N. Mostafazadeh, N. Chambers, X. He, D. Parikh, D. Batra, L. Vanderwende, P. Kohli, and J. F. Allen, "A corpus and cloze evaluation for deeper understanding of commonsense stories," in *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, K. Knight, A. Nenkova, and O. Rambow, Eds. The Association for Computational Linguistics, 2016, pp. 839–849.

[31] S. Nicholas, B. Danny van, and T. Federico, "javaparser," 2017. [Online]. Available: https://leanpub.com/javaparservisited

[32] Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests: Are we there yet?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1435–1446. [Online]. Available: https://doi.org/10.1145/3377811.3380351

[33] F. Hamm and O. Bott, "Tense and Aspect," in *The Stanford Encyclopedia of Philosophy*, Fall 2021 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2021.

[34] "Mood & modality and dialogue sentiment | towards data science," https://towardsdatascience.com/mood-modality-and-dialogue-sentiment-b06cd36eca88, Jul. 2020, (Accessed on 03/13/2022).

[35] M. Honnibal and I. Montani, "spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing," 2017, to appear.

[36] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[37] B. Chen and Z. M. J. Jiang, "Studying the use of java logging utilities in the wild," in *Proceedings of the 42th International Conference on Software Engineering*, ser. ICSE '20, 2020.

[38] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[39] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, and M. Lanza, "Pattern-based mining of opinions in q&a websites," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 548–559. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00066

[40] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE, 2012, pp. 102–112.

[41] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14, 2014, pp. 21–30.

[42] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 154–164.

[43] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines," in *Proceedings of the First USENIX Conference on Analysis of System Logs*, ser. WASL'08. USA: USENIX Association, 2008, p. 6.

[44] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, p. 3–14, mar 2011. [Online]. Available: https://doi.org/10.1145/1961295.1950369

[45] Y. Tang, A. Spektor, R. Khatchadourian, and M. Bagherzadeh, "Automated evolution of feature logging statement levels using git histories and degree of interest," *Science of Computer Programming*, vol. 214, p. 102724, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321001179

[46] S. Myers and M. Palmer, "ClearTAC: Verb tense, aspect, and form classification using neural nets," in *Proceedings of the First International Workshop on Designing Meaning Representations*. Florence, Italy: Association for Computational Linguistics, aug 2019, pp. 136–140. [Online]. Available: https://aclanthology.org/W19-3315

[47] S. Locke, H. Li, T.-H. P. Chen, W. Shang, and W. Liu, "LogAssist: Assisting log analysis through log summarization," *IEEE Transactions on Software Engineering*, may 2021.

[48] S. Kobayashi, K. Fukuda, and H. Esaki, "Towards an nlp-based log template generation algorithm for system log analysis," in *Proceedings of The Ninth International Conference on Future Internet Technologies*, ser. CFI '14. New York, NY, USA: Association for Computing Machinery, 2014.

[49] N. Aussel, Y. Petetin, and S. Chabridon, "Improving performances of log mining for anomaly prediction through nlp-based log parsing," in *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*. IEEE Computer Society, 2018, pp. 237–243.

[50] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 267–277. [Online]. Available: https://doi.org/10.1145/2025113.2025151