Assessing Semantic Frames to Support Program Comprehension Activities

Arthur Marques, Giovanni Viviani, Gail C. Murphy
Department of Computer Science
University of British Columbia, Canada
{msarthur, vivianig, murphy}@cs.ubc.ca

Abstract—Software developers often rely on natural language text that appears in software engineering artifacts to access critical information as they build and work on software systems. For example, developers access requirements documents to understand what to build, comments in source code to understand design decisions, answers to questions on Q&A sites to understand APIs, and so on. To aid software developers in accessing and using this natural language information, software engineering researchers often use techniques from natural language processing. In this paper, we explore whether frame semantics, a general linguistic approach, which has been used on requirements text, can also help address problems that occur when applying lexicon analysis-based techniques to text associated with program comprehension activities. We assess the applicability of generic semantic frame parsing for this purpose, and based on the results, we propose SEFrame to tailor semantic frame parsing for program comprehension uses. We evaluate the correctness and robustness of the approach finding that SEFrame is correct in between 73% and 74% of the cases and that it can parse text from a variety of software artifacts used to support program comprehension. We describe how this approach could be used to enhance existing approaches to identify meaning on intention from software engineering texts.

Index Terms—Empirical Software Engineering, Software Discussions, Natural Language

I. INTRODUCTION

Much critical information about software is captured in natural language. For instance, software requirements are often expressed in natural language text, question-and-answer web sites that enable developers to help each other rely on natural language explanations, and comments written in natural language within source code help a developer understand what a piece of code is intended to do.

Researchers have long recognized the value of this natural language text, utilizing various techniques to extract information from this text that can be embedded in tools for software developers. A number of the techniques commonly employed by researchers are based on the frequency of co-occurrence of words (or phrases) in documents. An early example is Maarek and Smadja’s use of lexical relations to index software libraries [1]. Since this early use, software engineering researchers have continued to leverage advances in these approaches, such as when Maletic and Marcus applied Latent Semantic Analysis (LSA) [2] to help cluster software components to aid program comprehension of a software system [3], or when Nguyen and colleagues applied Word2Vec [4] to support the retrieval of API examples [5].

At times, software engineering researchers have argued that general lexicon analysis techniques from natural language processing are insufficient to address text appearing in software engineering artifacts. For example, Di Sorbo and colleagues argued that lexicon analysis, like LDA, was insufficient to classify emails based on developers’ intentions [6]. Based on a manual analysis of existing text in developer emails, they defined rules to distinguish sentences discussing feature requests, asking for an opinion, or proposing solutions, amongst others. As another example, Gu and Kim argued lexicon analysis was insufficient to assess the intent of user reviews and, like Di Sorbo, proceeded with a manual analysis [7].

These arguments that lexicon-based natural language techniques are not applicable is often based on a need for access to the meaning of sentences in the natural language text [6]–[8]. Whenever similarity arises in the reasons for why existing approaches are insufficient, it raises the question of whether there exists an approach that might fill the gap.

In this paper, we consider whether frame semantics [9], [10]—a general linguistic approach—might be such a gap filler. The frame semantics theory is based on how readers comprehend the roles that words take in a sentence with respect to events of interest [10], [11]. We provide a detailed explanation of frame semantics in the next section (Section II).

To the best of our knowledge, there have been only a few uses of frame semantics in software engineering research [12]–[15] (Section III). These approaches have largely focused on text associated with software requirements, leaving open the question of applicability of the approach to text in documents supporting program comprehension activities. One paper [15] applies semantic frames to such documents, but does so from the viewpoint of finding patterns in text rather than considering the applicability of the technique to the text. It is thus an open question whether semantic frames can help identify the meaning of software engineering text associated with helping developers build and comprehend programs, such as text in API documents and bug reports.

To determine whether frame semantics are applicable to a broad range of software artifacts used for program comprehension, we consider three questions:

1) Do generic frame parsing approaches apply to text
used for program comprehension activities? We assessed whether generic frame parsing as provided by the SEMAFOR tool [16] applies to such text by having two annotators label 1,866 sentences sampled from 1,802 documents drawn from existing datasets (Section IV). We found that 5 frames required modification to properly capture the meaning of the text from documents associated with program comprehension activities and that 10 other frames were not suitable. We introduce SEFrame that supports this tailoring of frame parsing to program comprehension related software engineering artifacts (Section V).

2) Does SEFrame produce correct frames? We assessed whether our SEFrame approach provides results acceptable to a broader range of evaluators by having 10 evaluators evaluate 360 sentences and one of their associated frames (Section VI). We found that the frames parsed by SEFrame were identified as correct in 73% of the cases, a number comparable to SEMAFOR’s 75% correctness score, indicating that our approach produces correct frames.

3) Is SEFrame robust? We applied SEFrame to an additional 5 types of software engineering artifacts and had an additional 20 evaluators assess the results as a means of further testing the generalizability of SEFrame (Section VII). We found that the evaluators found 74% of the frames to be correct, indicating that our approach is generalizable to a broader range of artifacts associated with program comprehension.

As a means of addressing the question of whether semantic frames might fill the gap for at least some techniques, we discuss the application of semantic frames to two existing software engineering techniques that rely on manual classification (Section IX). While our work provides a foundation from which to use semantic frames for software engineering, future work should continue to extend the investigation of the applicability of less commonly occurring frames.

This paper makes four contributions:

- It introduces a refined version of semantic frame parsing for program comprehension text called SEFrame.
- It demonstrates that SEFrame produce correct and robust frames for text from software engineering artifacts associated with program comprehension.
- It presents labelled data of sentences extracted from a variety of software artifacts and their associated frames.
- It provides SEFrame as a Python API.

We begin by providing an example of semantic frame parsing (Section II) and reviewing related work (Section III). We then describe our investigation of the applicability of frame semantic to the program comprehension related text (Section IV), leading to the development of SEFrame (Section V). We describe our evaluation of SEFrame (Sections VI and VII) and place those results in the context of threats to their validity (Section VIII). We discuss implications and final remarks in Sections IX and X, respectively.

---

Fig. 1: Example of frames and frame elements

Our data is available at this link\(^1\) and will be made publicly available upon acceptance of this paper.

II. BACKGROUND

Program comprehension activities often require inquiry of information available in text documents. Within these documents, a reader extracts information based on the roles that words take in a sentence with respect to events of interest [10], [11]. Frame semantics [9] theory provides lens to explain such a process, where a ‘frame’ is the key unit that assists understanding readers’ comprehension. The FrameNet project realizes frame semantics. It is a large collection of more than 1,200 frames recognizable in the English language [17] and semantic role parsers [11] have leveraged annotated data available in the FrameNet project for automatically identifying frames and their associated data [18]–[20]. SEMAFOR is an example of such a semantic frame parser [16] and it uses a statistical model for determining which word tokens evoke frames from FrameNet. SEMAFOR is also able to identify predicates not seen in the FrameNet lexicon, what makes the tool particularly useful for identifying frames in domain-specific tasks (e.g., [21] or [22]).

To illustrate frame semantics in action, we utilize an example from Di Sorbo et al. [6], whose approach aimed to classify the content of development emails in six categories, such as ‘solution proposal’ and ‘problem discovery’. Let’s consider the following two sentences:

1. we could use a leaky bucket algorithm to limit the bandwidth; and
2. the leaky bucket algorithm fails in limiting the bandwidth.

These two sentences are lexically similar, making it difficult to use lexicon analysis techniques to categorize them. Di Sorbo’s work categorized both sentences as ‘solution proposal’, even though a more accurate label for the second one would have been ‘problem discovery’.

If we apply frame semantics to these sentences, using SEMAFOR, the differences become apparent. Figure 1 presents the results of a frame analysis for the sentences. The frames of each sentence (in grey) represent a triggering event and the frame elements (fe) (in red) are arguments needed to understand the event. The enclosing square brackets mark all lexical units, or words, associated with either a frame or a frame element.

\(^1\)https://figshare.com/s/323a80bd801d45ed7b89
In the first sentence, the *Using* frame captures that an instrument, the leaky bucket algorithm, is manipulated to achieve a purpose, namely to limit band-width. In contrast, in the second sentence, the *Success or Failure* frame identifies the entity, or agent, that fails to achieve the goal of limiting the band-width. With the frame semantics it becomes possible to distinguish meanings of these two lexically similar sentences in such a way that the categorization desired by Di Sorbo et al. might occur. For instance, a categorization technique might associate a *Using* frame with a category such as ‘solution proposal’.

### III. RELATED WORK

Within software engineering, frame semantics have largely been applied to problems related to requirements engineering. Recognizing the limitations of approaches based on lexicon analysis, Jah and Mahmoud used frame semantics to represent text in user reviews of applications to improve the determination via a classifier of which reviews are feature requests, which are bug requests and which are something else [12]. They concluded that a frame semantics classifier based on SEMAFOR reduced the chance of overfitting and led to better performance over unseen data. Jah and Mahmoud applied frame semantics without any assessment of how well the general frame parsing approaches perform on software engineering text.

Kundi and Chitchyan used frame semantics to assist with requirements elicitation [13]. Their work seeks to identify actors, roles, and relationships according to the frames and frames elements extracted from requirements specification text. As part of this research, they observed that certain frame elements do not reflect the text from which they were extracted [13].

Alhosan et al. also applied frame semantics to problems associated with software requirements. Their investigation on the use of frame semantics in 18 requirements documents [23], [24] has identified that only some of the frames from FrameNet (123 frames) relate to text in requirements engineering. The identified frames were then used in a follow-up study [14] for the design of a technique able to identify textual related requirements using both Word2Vec [4] and FrameNet. Although their study suggests that frames improve measuring semantic relatedness for text in requirements engineering, their results are bound to the small set of artifacts from which frames were extracted and they discuss the need to extend their corpus and evaluation [24].

Marques et al. used SEMAFOR to investigate the relevance of text in software artifacts relevant to completing software engineering tasks [15]. These artifacts include ones that support program comprehension activities, such as referring to bug report discussion to confirm a system’s behaviour. In this study, semantic frame parsing was used as one technique to look for patterns in text identified as relevant by participants. A total of 346 distinct frames over 20 software development natural language artifacts were identified and results suggest consistency in the frames of sentences considered relevant. This work did not investigate whether the identified frames were meaningful for the sentences parsed.

This paper extends these investigations by considering text from documents associated with program comprehension activities, by tailoring frequently occurring frames in these artifacts to appropriate meanings for program comprehension, and by investigating the correctness of frames parsed with a larger set of evaluators.

### IV. DO GENERIC FRAME PARSING APPROACHES APPLY TO PROGRAM COMPREHENSION?

Ideally, frames identified by semantic frame parsers would find appropriate meaning in text related to software engineering artifacts ‘out-of-the-box’ without change. As already seen in the investigation by Alhosan et al. on a small number of software requirement documents [24], this assumption does not hold.

To explore how many frames might need alteration for software engineering text, we ask: “do generic frame parsing approaches apply to text that appears in software engineering artifacts?” Specifically, we investigate the frames that result when a standard parser, SEMAFOR, is applied to a broader range of artifacts associated with program comprehension activities. We are particularly interested in how the presence of jargon or text specific to software such as method signatures, stack traces, or command-line arguments, might affect the meaning of the frames identified.

To answer this question, we had two annotators perform an open coding of frames parsed from three different datasets drawn from corpora published in previous software engineering studies. Our open coding analysis comprises the inspection of the 50 most frequently occurring distinct frames that appear in 1,866 sentences.

#### A. Method

1) **Datasets**: For this investigation, we chose artifact types that relate to program comprehension tasks [28], [29]. These artifact types have been the target of many researchers given the prominent role they play in the activities of software developers:

- developers seek Q&A websites, such as Stack Overflow, to analyze and adapt solutions available online for their current development tasks [30]–[34]; and

![Fig. 2: Distribution of frames over sampled sentences](image)
• bug reports serve as a central place for several program comprehension activities, such as documentation, coding, or testing [35]–[37].

To minimize any bias in our selection of artifacts, we chose to rely on existing datasets published at major software engineering conferences. For this investigation, we use the first three datasets listed in Table I.

2) Sampling: As selected datasets comprise more than 53,800 sentences, and given that sentences can contain more than one frame, manually inspecting the frames in each sentence is infeasible. We thus sample a statistically significant number of sentence-frame pairs for inspection.

We follow the procedures described by Bacchelli and colleagues [38], [39], statistically sampling sentences from each dataset with a 90% confidence level and a 5% error level with respect to the original number of sentences of that dataset. As Table I shows, sample sizes range from 552 to 653 sentences from each source. We then use SEMAFOR [16] to semantically parse each sentence and obtain its set of frames, i.e., when SEMAFOR is applied to a sentence, it may identify multiple frames for the sentence. We chose to use SEMAFOR because previous studies in the field have used the tool and adopting the same parser facilitates comparison (Section III).

After applying SEMAFOR to the dataset, we identify 444 distinct frames. Figure 2 plots the distribution of all sentences per frame. The distribution follows Zipf’s law with certain frames appearing in a high number of sentences and a long tail where some frames are specific to a few sentences. The most occurring frames appear in 30% (topmost frame) to 10% (top 10th frame) of the sampled sentences. In comparison, the 100th frame appears in no more than 25 sentences, which accounts for 1% of the sentences sampled. We are most interested in those frames that occur frequently and so, we produce the final set of sentence-frame pairs for inspection by sampling sentences for each of the top 50 frames. That is, given all sentences associated to a frame, select a statistically significant number of sentences for inspection.

Our sampling procedures resulted in a set of 1,866 sentence-frame pairs for inspection that comprise all occurrences of the top 50 distinct occurring frames.

3) Annotation Process: Similar to Alhoshan and colleagues [23], we use an annotation approach which consists of inspecting the frame, its definition according to the FrameNet database, and its frame elements and associated lexical units. For each frame, annotators indicated if the frame was correct and, if not, suggested either a replacement from FrameNet or an entirely new frame.

As a first step to get familiarized with the definition of each frame encountered, the annotators read a total of 180 pairs of sentences with their associated parsed frame. Then, the two annotators (both authors of the paper) proceeded iteratively. In the first iteration, 300 sentence-frame pairs were individually annotated by the two authors. At the end of this iteration, the two annotators discussed potential strategies for the resolution of disagreed frames, resulting in an agreed output schema for the final iteration: a frame was annotated as valid, invalid, or requires modification (with suggestions for a replacement). At this point, the annotators individually annotated the final set of 1,866 sentence-frames. We did not exclude the previous 300 pairs from this set, as returning to them ensures consistency in how conflicts were resolved.

B. Results

Table II summarizes results according to annotation outcomes: valid, invalid, or modify. To assist discussion, we refer to the examples from Table III by their unique identifiers (UID).

We found that the annotators agreed that SEMAFOR’s parsing of the semantic frames was valid for 35 (64%) of the frames inspected. An annotation of valid means that the meaning of frames is similar in both every-day English text and software engineering text. In valid frames, SEMAFOR is also able to correctly identify the predicates, namely the frame elements (in red) and lexical units (enclosing brackets) of software engineering specific text. For example, it successfully identifies ‘turning off XBitHack’ as the cause for the observed behaviour in UID436 (Table III). For the Using frame (UID122), it also identifies the ‘JNI function’ as the instrument needed for achieving the purpose of creating a certain class.

It is worth noting that even if a frame is considered valid overall, some instances of where that frame is parsed may be invalid. UID120 in Table III is an example of an invalid instance for the Using frame, where ‘application’ does not

| Dataset         | Source | Description                                                                 | Size          | Sentences | Sample |
|-----------------|--------|----------------------------------------------------------------------------|---------------|-----------|--------|
| InfoTypes       | [8]    | Annotated data of 15 complex issue threads across three projects hosted on GitHub | 15 GitHub issues | 5981      | 597    |
| AnswerBot       | [25]   | Technical questions and answers from Stack Overflow covering a diversity of aspects of Java programming | 100 SO questions | 3306      | 552    |
| MissingInf      | [26]   | Bug reports from nine software projects of different types and domains. Bug reports mined both from Bugzilla and GitHub | 2,912 bug reports | 44554     | 653    |
| APIPatterns†    | [27]   | Annotated data of knowledge patterns found in the JDK 6 and .NET 4.0 reference API | 5,574 API elements | 4451      | 577    |

† Comprises only Java API documentation
TABLE II: Summary of frames and annotation outcome with percentage of sentences containing the frames in the dataset

| 35 frames (64%) | Valid  |
|-----------------|--------|
| Intentionally act | Quantity | Using |
| Temporal collocation | Being obligated | Capability |
| Cardinal numbers | Causation | Likelihood |
| Locative relation | Increment | Relational quantity |
| Point of dispute | Existence | Intentionally create |
| Attempt | Instance | Frequency |
| Desiring | Predicament | Desirability |
| Required event | Event | Identicality |
| Age | Gizmo | Similarity |
| Time vector | Relative time | Awareness |
| Measure duration | Scrutiny | Sole Instance |
| Possession | Grasp | |
| 5 frames (16%) | Modified to Execution |
| Arriving | Means | Aggregate |
| Request | Leadership | |
| 10 frames (20%) | Invalid |
| Statement | Type | Placing |
| Being named | Purpose | Roadways |
| Contingency | Connectors | Text |

3SEMAFOR has an average precision of 75.54% for the FrameNet lexicon.

evoke usage. Occurrences of invalid instances represent a small fraction of the data annotated frames and we attribute invalid instances to misclassification errors inherent to SEMAFOR.

A total of 5 (16%) frames required modification. Annotators observed that the meaning of verbs such as ‘get’ or ‘run’ diverge from FrameNet’s meaning as identified by SEMAFOR. UID_{1181} illustrates a frame that requires modification. The verb ‘run’ evokes the Leadership frame which is defined as control by a leader over a particular entity or group. However, the sentence related to this frame in Table III conveys executing an evaluation benchmark. Both annotators suggested renaming the frame to Execution so that it reflects a meaning appropriate to the software engineering domain. The same logic applies to the Arriving frame in UID_{611}.

The 10 frames that were annotated as invalid account for 20% of the inspected data. Annotators found that the presence of source-code related nouns is a common element to the invalid frames. Roadways and Connectors are examples of invalid frames. The sentences for these frames, as shown in Table III, indicate that the nouns that evoke these frames are code-related, such as ‘command line’ and ‘string’, and the frames parsed by SEMAFOR are not suitable for such cases.

Disagreements occurred in scenarios where one annotator suggested removal of a frame while the other said otherwise. As an example, the annotators disagreed on the resolution for the Being obligated frame (Table III). One annotator indicated that Possession would be a more appropriate frame, while the other indicated that given that only a few sentences with the frame were invalid, the situation is similar to a misclassification error inherent to using the SEMAFOR tool. For other cases, such as in Type, annotators observed that a single lexical unit was not enough to justify a new frame and thus, annotators agreed that the frame was invalid.

Out of the top 50 occurring frames detailed in Table II, 35 (64%) of them apply to text from software engineering without modifications, 10 (20%) are invalid and 16% require modification.

V. SEFrame: Tailoring FrameNet for Software Engineering

Our initial investigation found that modifications are needed to SEMAFOR, and some of the frames defined in FrameNet, to support appropriate parsing for software engineering text. We describe SEFrame, a tool we have built that tailors SEMAFOR for software design and other text related to program comprehension activities.

We started the design of SEFrame by gathering all frames that required modification or that are invalid and their respective lexical units. Then, we sought to identify patterns in the lexical units from these frames that would produce a simple set of heuristics to be used by SEFrame to refine the frames extracted by SEMAFOR.

The first pattern observed by the annotators is that if the lexical unit of a frame is a verb with a particular meaning in the software engineering domain (e.g., run, call, etc.), the frame requires modification. A second pattern arises from frames evoked by source-code related nouns, which are deemed as invalid. Table III provides examples for such patterns where the verb ‘get’ in UID_{611} and the word ‘string’ in UID_{1493} are not correct for the Arriving and the Connectors frames, respectively.

The new proposed frame—Execution—addresses verb related modifications. SEFrame makes verbs such as get, return, request, run and process evoke the new frame instead. Interestingly, this change applies to all the frames that require modification in Table II since all of their verbs evoke executing system calls or running some procedure.

To address the second identified pattern, SEFrame removes any frames that are marked as invalid in Table II. Annotators identified that invalid frames are often related to code-related nouns what could lead to the creation of a reference list for ignored lexical units. Creating such a list poses the question of where code elements come from and whether the list is complete. Therefore, SEFrame opts to simple discard the frame rather than referring to which specific nouns and code-related terms would trigger removal. We discuss this and other design decisions threats in Section VIII.

Intuitively, SEFrame can be seen as a Decorator [40] that extends the functionalities of the SEMAFOR tool. Its current version is written in Python and all its dependency libraries are provided via Docker containers [41].

VI. DOES SEFrame PRODUCE CORRECT FRAMES?

For SEFrame to be usable by software engineering researchers, it must be seen as parsing frames that are...
TABLE III: Excerpt of results from annotation procedures

| UID | Frame   | Sentence                                                                 | Annotation |
|-----|---------|--------------------------------------------------------------------------|------------|
| 436 | Causation| [Turning off XBitHack in my config] fe:Cause: [made] Causation: [this behavior go away] fe:Effect | Valid      |
| 122 | Using   | I'm trying to [use] Using a JNI function fe:Instrument [to create a Java class] fe:Purpose and set some properties of that class using the DeviceIdJava constructor method | Valid      |
| 1181| Leadership| Does anyone want to [run] Leadership [a benchmark] fe:Governed           | Modify     |
| 611 | Arriving| I can't even run this simple tensorflow script, as its result, I [get] Arriving ImportError: No module named tensorflowpythonclient fe:Goal | Modify     |
| 120 | Using   | Its a desktop standalone Java [application] Using                         | Invalid    |
| 1176| Roadways| The [command line] Roadways is what almost every other application will use to build your JAR file | Invalid    |
| 1493| Connectors| This means that at least the string-to-int mapping will stay consistent, even if [strings] Connectors are passing out of memory | Invalid    |
| 263 | Being   | obliged [!] fe:Responsible_party [have] being_obliged two classes in a parent-child relationship | Disagreement |
| 692 | Type    | [50] fe:Category [version] Type 71 on the server                         | Disagreement |

meaningful—correct—by more than those who defined SEFrame (i.e., the two annotators). To explore if the results produced by SEFrame are meaningful, we asked 10 evaluators, distinct from the annotators in the initial investigation (Section IV), to assess the correctness of semantic frames parsed from sentences selected for all datasets shown in Table I.

A. Method

We applied SEFrame to the datasets listed in Table I. To provide a more generalizable result, we extended the first three datasets used to create SEFrame with a fourth dataset, APIPatterns, which has text extracted from Java documentation (Table I). We include a new dataset as a first step towards investigating if and how frames apply to other types of sentences and also because API documentation is a resource that developers commonly refer to as part of a program comprehension task [42], [43], [43]. For this study, we considered the 36 frames that are included in SEFrame.

The 10 evaluators were all currently graduate students in an English-language based Computer Science program and most of them had at least some previous industry experience, averaging around 1.5 years. All evaluators were remunerated for their time spent on this work.

For each frame, we sampled 10 sentences that contained the specific frame, for a total of 360 sentences. These sentences were divided in 10 batches, each one containing one sentence for each frame. We then assigned three evaluators to each batch. We assigned evaluators with an approach based on similar studies [44]: any group of three evaluators is evaluating maximum one batch together and any group of two evaluators is evaluating a maximum of two batches together.

We used an online tool to present each sentence. Each sentence and the associated frame was presented to the evaluator through an online tool, an example of which can be seen in Figure 3.

For each sentence, the evaluator was asked to indicate whether the associated frame correctly represented the intention behind the words used in the sentence or not. Whenever the evaluator indicated a frame to not be correct, a follow-up question displayed the original frame parsed by SEMAFOR for that sentence and asked whether the frame was more suitable. The evaluators shared a common chatroom with the authors where some clarificatory questions were exchanged before the start of the study. This common chatroom meant every evaluator had access to the same information.

B. Results

Table IV presents a summary of the results with the percentage of frames that were evaluated as correct. We define a frame to be correct if at least 2 out of the 3 evaluators agreed that it represents the meaning of its associated sentence. Overall, for 73% of the 360 sentences were identified to be useful. The frame Execution, that we introduced in SEFrame, was identified as correct in 8 of the 10 sentences it appeared in. In the two sentences where it was considered incorrect, evaluators agreed that the original SEMAFOR frame was not a better candidate. We consider this result evidence that SEFrame is able to detect correct frames.

VII. Is SEFrame Robust?

Having shown in the previous section that SEFrame provides correct results in the view of evaluators independent from the creators of the tool, we turn to the question of whether SEFrame can produce correct results over a broader range of kinds of artifacts. To explore if SEFrame generalizes across artifacts, we extend our dataset to cover an additional five types of software artifacts—pull requests, vulnerability reports, a broader set of questions and answers, mailing lists and app reviews—and evaluate the correctness of frames parsed by individuals unknown to the SEFrame creators.
Returns 0 if the view is not visible, otherwise it calls the superclass method \texttt{obtain the maximum span}

| Frame          | Ratio | Frame          | Ratio |
|----------------|-------|----------------|-------|
| Predicament    | 100%  | Being obligated| 70%   |
| Required event | 100%  | Ordinal numbers| 70%   |
| Attempt        | 100%  | Temporal collocation | 70%  |
| Identicality   | 90%   | Measure duration | 70%  |
| Awareness      | 90%   | Age             | 70%   |
| Aggregate      | 90%   | Quantity        | 70%   |
| Likelihood     | 90%   | Sole instance   | 70%   |
| Existence      | 90%   | Point of Dispute| 70%  |
| Desiring       | 90%   | Relative time   | 60%   |
| Instance       | 80%   | Similarity      | 60%   |
| Scrutiny       | 80%   | Frequency       | 60%   |
| Using          | 80%   | Relational quantity | 60%  |
| Intentionally create | 80% | Giving | 50% |
| Capability     | 80%   | Possession      | 50%   |
| Grasp          | 80%   | Time vector     | 50%   |
| Inclusion      | 80%   | Causation       | 50%   |
| Desirability   | 80%   | Cardinal numbers| 50%  |
| Execution      | 80%   | Locative relation| 30% |

TABLE IV: Results of Correctness Study

A. Method

We extend the dataset with sentences gathered from five different data sources, which are associated with software design and programming:

1) Pull requests from the top 5 most starred projects on Github. From each project, we selected the top 10 most commented pull requests, and randomly sampled 50 comments for analysis. We filtered comments with less than 50 characters to avoid common comments found in pull requests, e.g., “Approved” or “Looks good to me”.

2) Security threats related to vulnerability management data that are described using the Security Content Automation Protocol (SCAP) [45]. For this dataset, we randomly select the description field of threat reports from 2019.

3) A new set of questions from Stack Overflow due to the fact that the AnswerBot dataset used in our first study (Section VI) concerns only certain topics on the Java programming language [25]. For this new set of questions, we selected the 10 most commented questions from each one of the 5 programming languages mostly discussed on the platform.

4) Developers’ Mailing lists as represented by randomly selecting 6 archive threads from established projects of the Apache foundation. From each thread, we parsed individual emails while ignoring any text in quote blocks such that we ignore duplicated text.

5) App reviews from the 10 most popular apps from the Google Play Store. From each app, we selected the top 50 reviews and randomly sampled 25 reviews for analysis.

Table V details the sources used. Overall, our selection criteria led to a total of 2530 sentences containing 302 distinct frames. We chose to focus on the top 20 most occurring \textit{SEFrame} frames as, similar to our initial investigation in Section IV, these frames appear in 33% of the sentences in this new dataset. We therefore selected the top 20 frames and randomly sampled one sentence for each.

We then recruited 20 evaluators unknown to us through Amazon Mechanical Turk. These evaluators had to have obtained the Master Qualification after repeated participation in the platform. With this approach, we did not have direct control over the selection of evaluators. We introduced two additional sentences in those seen by an evaluator to validate the evaluator’s responses and to avoid exploitation of the platform with the goal of remuneration. The two validation sentences were hand-picked by the authors. The first one had an obviously correct frame associated with (Figure 4a), while the other had an obviously wrong frame(Figure 4b). The correct one was placed as the first sentence in the study, while the incorrect one was placed at the end. Neither of these frames were considered in the results. This validation process lead to two evaluators being replaced, as they had indicated...
TABLE V: Datasets - Robustness Study

| Dataset                  | Description                                                                 | Size     | Sentences | Sample |
|--------------------------|------------------------------------------------------------------------------|----------|-----------|--------|
| Pull Requests            | Pull requests from some of the most starred projects available on Github, i.e., Twitter's Bootstrap, Google's Flutter, Facebook's React, FreeCode-Camp, and OhMyZsh | 25 pull requests | 648       | 328    |
| Security Vulnerability   | Security vulnerabilities available on the U.S. National Vulnerability Database | 1866 reports | 4553      | 579    |
| Stack Overflow           | Stack Overflow most upvoted questions for 5 of the most popular programming languages available, i.e., javascript, java, python, c#, and php. | 50 SO questions | 14196     | 633    |
| Mailing lists            | Mail archives from popular projects hosted by the Apache foundation, namely, Apache Commons, Couch DB, HTTPD, Maven, Lucene, and Spark. | 6 mail archives | 3650      | 561    |
| Android App Reviews      | User reviews for the ten most popular Android applications available on Google play store. | 250 user reviews | 1212      | 429    |

(a) Correct frame

Sentence 1
I hope this gives a useful clue as to what causes this freeze

Proposed Frame
Causation

[|Cause: what, Effect: this freeze]

(b) Incorrect frame

Sentence 22
This is a general bug / feature not having anything in common with NLS or BiDi

Proposed Frame
Leadership

[|Leader: bug]

Fig. 4: Validation frames. The frame in Figure 4a is a correct representation of the sentence, while the frame in Figure 4b is clearly wrong

B. Results

Table VI summarizes the results of our study. Overall, in 296 cases (74%) the participants indicated that the frame was correct for the sentence. In the majority of cases, 13 out of 20 frames, the majority of participants (at least 3/4 of them) indicated that the frame extracted was correct. In only one case, Sufficiency, the majority of participants disagreed with the proposed frame, but 9 out of 20 still marked it as correct.

Consistent results from a different set of evaluators on different datasets increase our confidence that our approach generalizes to a broad range of text that appears in software engineering artifacts.

VIII. Threats

We performed three investigations as part of exploring the applicability of semantic frames to text associated with program comprehension activities. Although the details of each investigation vary, there are similar threats to validity across the studies.

1) Internal and Construct Validity: We developed SEFrame that tailors semantic frames for software engineering based on data sampled from three datasets described in Table I. As this dataset is not representative of all kinds of text appearing in software engineering artifacts associated with program comprehension activities, it is possible that SEFrame may not adequately parse text from artifacts not considered by our investigation. To address this threat, we included an additional dataset when we investigated the validity of SEFrame (Section VI) and then used a broader set of artifact types and new data—Table V—in our investigation of the generalizability of SEFrame (Section VII).

Another threat lies in the sentences sampled from the datasets, which may not be representative. To strike a balance between enough examples drawn from a diverse number of frames, we considered the distribution of frames over sampled sentences such that the focus was on the frames most likely to appear if SEFrame was applied. This led to the selection of the top 50 (Section IV) and top 20 (Section VII) most occurring
TABLE VI: Results of Robustness Study

| Frame            | Correct | Incorrect | Ratio |
|------------------|---------|-----------|-------|
| Likelihood       | 20      | 0         | 100%  |
| Required Event   | 18      | 2         | 90%   |
| Reasoning        | 17      | 3         | 85%   |
| Existence        | 17      | 3         | 85%   |
| Intentionally Act| 17      | 3         | 85%   |
| Relative Time    | 17      | 3         | 85%   |
| Time Vector      | 17      | 3         | 85%   |
| Events           | 16      | 4         | 80%   |
| Sole Instance    | 16      | 4         | 80%   |
| Capability       | 16      | 4         | 80%   |
| Quantity         | 15      | 5         | 75%   |
| Using            | 15      | 5         | 75%   |
| Execution        | 15      | 5         | 75%   |
| Inclusion        | 13      | 7         | 65%   |
| Similarity       | 13      | 7         | 65%   |
| Increment        | 12      | 8         | 60%   |
| Aggregate        | 12      | 8         | 60%   |
| Causation        | 11      | 9         | 55%   |
| Temporal Collocation | 10   | 10        | 50%   |
| Sufficiency      | 9       | 11        | 45%   |
| **Overall**      | **296** | **104**   | **74%** |

frames in the datasets. To ensure we were not biased in the artifacts from which we selected sentences, we employed a sampling approach where we computed the frames for each sentence in every dataset, and randomly sampled out of these, regardless of which artifact they belong. This methodology allowed us to focus on the topmost occurring frames regardless of how often they appear in the data.

A key aim in our investigations is to determine the validity of frames parsed for a sentence. We relied on a variety of evaluators for the determination of validity. Across the investigation of validity and robustness, 30 individuals, none of whom are authors of this paper, assessed frames and sentences. While 1/3 of these evaluators were known to the authors (Section VII), 2/3 were not. The use of a broad range of individuals at arms-length from the authors decreases potential bias and increases our confidence in the results. A threat does arise from our choice of not requiring our evaluators recruited through the MTurk platform to have some certification of English fluency. Recent investigations in the demographic of the MTurk population has shown that the large majority of MTurkers is from the USA (75%) [46]. Additionally, our description of the study was provided in English, and we argue that it is unlikely that someone could have understood the task and passed the validation questions without a good understanding of the language.

There are also threats related to the design of SEFrame itself. In Section V, a common pattern that causes a frame to be incorrect for software engineering text is when source-code related nouns in a sentence evoke a frame. Instead of creating a list with source-code related terms that SEFrame could ignore, we opt to discard the frame entirely. This design decision errs on the side of caution because creating and ensuring that a list of source-code related terms is complete and always up-to-date is challenging. However, this means losing frames that are potentially correct in certain cases.

2) **External Validity:** In Section VII, we investigate the generalizability of our study [47]. To achieve this goal, we selected 5 new sources of text to use in place of the datasets used in the previous study. Based on our methodology and the diverse set of data we used, we believe our results are generalizable to different kinds of text in the domain.

IX. DISCUSSION

Software engineering researchers have recognized limitations of natural language techniques based on lexicon analysis for processing software engineering text. To overcome these limitations, some software engineering researchers have started to make use of semantic frame parsing. We have shown that many semantic frames apply to text appearing in a large range of software artifacts associated with program comprehension activities and that minor modifications can help address differences in use of language in software engineering text. However, questions remain about whether semantic frames can help disambiguate software engineering text where other approaches struggle and whether our modifications address problems seen with text in software engineering applications. In this section, we explore these questions and how they might be addressed in future work.

A. Using Semantic Frames to Disambiguate SE Text

There are limitations associated with applying lexicon analysis to text associated with program comprehension and software engineering activities as outlined in Section II. To overcome these limitations, researchers have used manual labelling combined with classification approaches. In defining these classification approaches, researchers rely on discourse patterns in the form of key phrases and generic placeholders to infer the *meaning* of software engineering text. For example, Di Sorbo et al. define a pattern—“*someone* should add *something*”—to recognize sentences that propose a solution to a problem [6]. While these approaches produce acceptable results, they lag in performance because a pattern-based approach is prone to misclassification [48].

A common misclassification error occurs due to the presence of modal verbs—an auxiliary verb that expresses a possibility, likelihood, suggestions, etc.—in a sentence. In Di Sorbo et al.’s pattern-based approach for mining intentions from development emails, the presence of the modal verb ‘would’ in the sentence—“*One way would be to add them in an #ifdef Q QDOC block and document them*”—causes a misclassification to a feature request instead of a solution proposal [48].

While a sentence related to feature request often contain users’ desires or aspirations, a sentence describing a solution proposal should contain directives or instructions; this meaning would be captured by the *Execution* frame in semantic frame parsing. Unfortunately, generic frame parsing does not suggest such a frame. However, SEFrame, when applied to this sentence, identifies such a frame what could have assisted correctly classifying Di Sorbo et al.’s example.
In another piece of software engineering research, Chapparo et al. attempt to classify if a sentence of a bug report contains ‘observed behaviour’, ‘expected behaviour’, or ‘steps to reproduce the bug’ [26]. Similar to Di Sorbo, Chapparo et al. also observe misclassification errors. For example, the presence of a modal verb causes their approach to misclassify a sentence discussing an observed behaviour “this problem could also be related to some sites not copying URLs” as an expected behaviour. For this sentence, SEFrame parses the Predicament frame, which is a frame that evaluators consistently agreed is related to undesired behaviours. The presence of this frame, as identified by SEFrame, could have assisted determining that the sentence is about a bug’s observed behaviour.

We suggest that future work consider how classifiers based on SEFrame compares to pattern matching-based classifiers (e.g., [6], [26], [49]).

B. Use and Limitations of SEFrame

Software engineering researchers have noted the need to interpret text in a structured way. For example, Robillard et al. show that the structure of information in API documentation can help facilitate discussions about the content of APIs [27], [49]. The structure extracted from a sentence by frame semantics could help in these situations and in particular, SEFrame may help with extracting structure from software engineering text in applications such as those described by Robillard et al. Consider an example from Robillard and Chhetri [49] where a sentence contains a ‘directive’ for cloning objects in the Java API “By convention, the returned object should be obtained by calling super.clone”. Figure 5 shows the same sentence in a structured format what suggests obligations, or requirements, for proper usage of the clone method.

![Fig. 5: Example of structured information for an API directive from [49]](image)

We have assessed and tailored frame semantics across a broad range of software engineering text associated with program comprehension. Upon application of SEFrame to program comprehension tasks, it may be found that further improvements are needed by analyzing more types of frames or by adopting more complex semantic parsers (e.g., [50], [51] or [52]). It is likely this analysis will require a similar approach to what we have taken in this paper that includes manual inspection and annotation. Future studies can benefit from the sampling methodology and annotation framework we outline to aid in these extensions. The frames on which to focus may be guided by a larger analysis of the distribution of frame occurrence over a wider set of software engineering artifacts.

X. Conclusion

Software engineering researchers have noted that general natural language processing techniques based on lexicon analysis are not sufficient for all software engineering problems. In this paper, we have investigated whether frame semantics might help fill this gap for a range of software artifacts associated with program comprehension activities. Frame semantics is a general linguistic approach that extracts semantic frames from natural language sentences, aiming to represent the intended meaning of the words in a sentence.

We developed a tool, called SEFrame, that tailors an everyday English semantic frame parser (SEMAFOR) for artifacts associated with program comprehension activities. We evaluated SEFrame in two ways: first to evaluate its correctness (Section VI) and then to evaluate its robustness (Section VII). In both studies, SEFrame performed well, extracting the correct frame in 73% of the cases in the first study and 74% in the second study. Based on these results, we argue that SEFrame can reliably extract the correct frame for sentences across a broad range of software engineering text appearing in artifacts used to support program comprehension. These frames could be used to successfully categorize semantic information across a variety of artifacts commonly sought during software development.

We plan to continue expanding SEFrame to include an evaluation of all the frames presents in the FrameNet project to more accurately classify the semantic information in software engineering text and to explore its application to software engineering problems.

Acknowledgements

The authors would like to thank the participants of the correctness and robustness studies, and the anonymous reviewers for their valuable feedback. We also thank Shaunak Tulsibagwale for his help implementing some of the datasets samplers.

References

[1] Y. S. Maarek and F. Z. Smadjia, “Full text indexing based on lexical relations an application: software libraries,” in ACM SIGIR Forum, vol. 23, no. SI, 1989, pp. 198–206.
[2] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” Journal of the American society for information science, vol. 41, no. 6, pp. 391–407, 1990.
[3] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, 2003, pp. 125–135.
[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” arXiv preprint arXiv:1301.3781, 2013.
[5] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, “Exploring api embedding for api usages and applications,” in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), 2017, pp. 438–449.
Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, “API method recommendation without worrying about the task-API knowledge gap,” in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM Press, 2018, pp. 293–304.

M. P. Robillard and Y. B. Chhetri, “Recommending reference API documentation,” Empirical SE, vol. 20, no. 6, pp. 1558–1586, 2015.

M. Roth and M. Lapata, “Context-aware frame-semantic role labeling,” Transactions of the Association for Computational Linguistics, vol. 3, pp. 449–460, 2015.

N. FitzGerald, O. Täckström, K. Ganchev, and D. Das, “Semantic role labeling with neural network factors,” in Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, 2015, pp. 960–970.

S. Swayamdipta, S. Thomson, C. Dyer, and N. A. Smith, “Frame-semantic parsing with softmax-margin segmental rnns and a syntactic scaffold,” arXiv preprint arXiv:1706.09528, 2017.