Requirements Engineering
https://doi.org/10.1007/s00766-022-00390-8

ORIGINAL ARTICLE

Testing software's changing features with environment-driven abstraction identification

Zedong Peng[1] · Prachi Rathod[1] · Nan Niu[1] · Tanmay Bhowmik[2] · Hui Liu[3] · Lin Shi[4] · Zhi Jin[5]

Received: 20 January 2022 / Accepted: 6 September 2022

Abstract

Abstractions are significant domain terms that have assisted in requirements elicitation and modeling. To extend the assistance toward requirements validation, we present in this paper an automated approach to identifying the abstractions for supporting requirements-based testing. We select relevant Wikipedia pages to serve as a domain corpus that is independent from any specific software system. We further define five novel patterns based on part-of-speech tagging and dependency parsing, and frame our candidate abstractions in the form of <key, value> pairs for better testability, where the "key" helps locate "what to test", and the "value" helps guide "how to test it" by feeding in concrete data. We evaluate our approach with six software systems in two application domains: Electronic health records and Web conferencing. The results show that our abstractions are more accurate than those generated by a state-of-the-art technique. While the initial findings indicate our abstractions' capabilities of revealing bugs and matching the environmental assumptions created manually, we articulate a new way to perform requirements-based testing by focusing on a software system's changing features. Specifically, we hypothesize that the same feature would behave differently under a pair of opposing environmental conditions and assess our abstractions' applicability to this new form of feature testing.

Keywords Abstractions · Natural language · Environmental assumptions and conditions · Requirements-based testing

✉ Nan Niu
nan.niu@uc.edu

Zedong Peng
pengzd@mail.uc.edu

Prachi Rathod
rathodpt@mail.uc.edu

Tanmay Bhowmik
tbhowmik@cse.msstate.edu

Hui Liu
liuhui08@bit.edu.cn

Lin Shi
shilin@iscas.ac.cn

Zhi Jin
zhijin@pku.edu.cn

[1] University of Cincinnati, Cincinnati, OH, USA

[2] Mississippi State University, Mississippi State, Starkville, MS, USA

[3] Beijing Institute of Technology, Beijing, China

[4] Institute of Software Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing, China

[5] Peking University Beijing, Beijing, China

1 Introduction

In requirements engineering (RE), an *abstraction* refers to a term that has a particular significance in a given domain [1]. For example, "radar" is recognized as an abstraction in the air traffic control problem domain [2], and so is "antenna" in the radio frequency identification (RFID) application domain [3]. In order to reduce the requirements engineer's effort, researchers have developed methods to automatically identify the abstractions from the natural language (NL) documents. While the seminal work of AbstFinder searches for patterns of byte sequences [4], other researchers have located the abstraction candidates by exploiting natural language processing (NLP) techniques (e.g., corpus-based frequency profiling and part-of-speech tagging) [2, 3, 5–7].

Current support is mainly for *early phase RE* where the focus is on understanding the problem domain *before* formulation of the initial requirements [8]. For instance, Sawyer et al. [2] showed in an air traffic control case study that the abstractions extracted from a set of ethnographic fieldnotes by NLP could match the elements of a class diagram at a 75% recall and 12% precision level. Clearly, the relevance

Published online: 20 September 2022

of such NLP results must be vetted by the requirements engineer.

Indeed, Ryan [9] argued that NLP should play only a partial role in requirements validation, i.e., demonstrating convincingly a software system's conformance to stakeholder needs, because validating requirements must remain an informal, social process. Ryan [9] further pointed out that an intrinsic difficulty lies in the identification of *assumptions* that reflect the shared, common sense knowledge of people familiar with the social and technical contexts within which the software system operates.

Significant to RE are the *environmental assumptions* [10], i.e., the conditions over the phenomena of the physical world that one accepts as true irrespective of the software to be built [11]. In this paper, we use *assertion* [11] and *assumption* [12] interchangeably to refer to a *statement* indicating a property over the phenomena in the software's operational context that is accepted as true by the developers [13]. Many software problems originate in missing or flawed environmental assumptions. Notably, the assumption made about the maximum horizontal velocity did not hold for Ariane 5, contributing to the rocket launch failure [14]. Making the assumption statements explicit is therefore key to understanding the informal, social aspects of requirements validation.

A recent empirical study by Bhowmik et al. [15] with 114 developers showed the positive impact of environmental assumptions on requirements-based testing. One concrete result highlighted the assumption: "a doctor's appointment shall be scheduled only for a future timeslot". This statement generally holds independent of any specific software system. As a result, it helped to uncover a defect in a software application where a patient was able to make a doctor's appointment for a past date and time [15]. Despite the positive impact, Bhowmik et al. [15] reported that manually formulating complete and correct environmental assumptions from scratch is challenging.

Using NLP to automatically produce assumption statements, according to Ryan [9] and Sawyer et al. [2], is infeasible; however, narrow domain understanding in the form of abstractions has been shown to be realistic [1–4]. Our objective in this paper is to automatically identify the abstractions that are both indicative of important domain phenomena and amenable to requirements-based testing. To that end, we derive a corpus by selecting pages from Wikipedia, an exceptional repository codifying our shared knowledge about specific and connected topics. We then define a novel set of NLP patterns to extract and rank candidate abstractions. We show the effectiveness of our approach by comparing its results with abstractions identified from the state-of-the-art method [1]. We also demonstrate our approach's usefulness by relating the resulting abstractions to the environmental assumptions created manually [12, 15].

In our previous work [16], we derived four desiderata based on Jackson's conceptualization [11] for abstraction identification in the context of requirements-based testing: machine independent, requirements related, directly testable, and bug revealing. Our interpretation of a *bug* hinged on the failure of requirements fulfillment [16]. However, our analysis of some real-world bugs showed that a non-negligible distance often existed between testing a software feature and fulfilling a stakeholder's need. Thus, in this extension of our previous paper, we address such a distance by refining the notion of bug revealing, which in turn shapes the operationalization of testability. In particular, we depict a new form of acceptance testing when a changing feature is considered to be deployed, where a feature is an increment of functionality usually with a coherent purpose[1]. Rather than testing the software feature in a *single* environment to show the failure of requirements fulfillment [16], the extension of this paper shifts the requirements-based testing's focus toward testing the feature in a *pair* of opposing environmental conditions. The testing expectation (or oracle [17]) is that the feature would behave differently under these contrasting conditions, effectively tackling the distance between observing the execution of a feature and validating the satisfaction of a requirement.

The main contribution of our work is the abstraction identification that goes beyond early phase RE and offers new support for requirements-based testing. Our evaluation with six software applications in two different domains shows the effectiveness of our approach. In what follows, we present background information in Sect. 2. We then clarify the influence of environmental assumptions on requirements-based testing in Sect. 3. Section 4 details our NLP tool chain, Sect. 5 describes the empirical evaluations for the effectiveness and relevance of identified abstractions, Sect. 6 discusses the extended way of testing a software's changing features with support of abstractions, and finally, Sect. 7 concludes the paper.

2 Background and related work

2.1 Abstraction identification in requirements engineering

Abstractions are important domain concepts which the human analyst needs to identify in order to understand the problem domain as well as the constraints on the range of possible solutions [2]. To support the elicitation of the initial requirements, Goldin and Berry [4] developed the

[1] http://www.pamelazave.com/faq.html (last accessed on 2022/09/14 12:07:09).

AbstFinder tool based on the idea that important abstractions would recur frequently as repeated words within the target NL document. AbstFinder thus searches for co-occurring byte sequences within pairs of sentences using a series of circular shifts and returns a ranked list of frequently occurring byte sequences that a human analyst must recognize as parts of words and phrases.

To be successful in supporting early phase RE, the automatic identification of abstractions must achieve a level of completeness at least as good as that achieved by a human analyst. High recall values are often obtained at the cost of low levels of precision [18–20]. As noted by Sawyer et al. [2], when a complex problem is tackled, a precision of 25% or higher represents good abstraction identification performance.

Sawyer et al.'s air traffic control case study confirmed the practically achievable precision level, where a NLP toolset—word frequency and collocation, part-of-speech (PoS) tagging, and shallow semantic analysis—was used to extract abstractions from an aggregated set of ethnographic fieldnotes comprising about 44,000 words [2]. The technique achieved a 21% precision, showing the practical performance when processing a sizable volume of text.

Gacitua et al. [1, 3] used corpus-based frequency profiling to identify single-word abstractions. Given a domain document D and a normative corpus C, frequency profiling computes a term t's log-likelihood value[2] LL_t according to t's observed values in D and its expected values in C. The greater the LL_t value is, the more significant t is in D than in C, and hence the more likely t is an abstraction. Because over 85% domain-specific terms are multiword units [21], Gacitua et al. also recognized multiword abstractions via syntactic patterns based on PoS tagging, namely *adjectives and nouns*, and *adverbs and verbs*. In an experiment with the full text of a book containing 156,028 words, Gacitua et al.'s method achieved a 32% recall and a 32% precision, outperforming AbstFinder's 7% recall and 7% precision [1].

It should be pointed out that Gacitua et al. [1, 3] used AbstFinder as a one-shot tool, which was *not* the intended use as conceived by Goldin and Berry [4]. AbstFinder is aimed at helping a human analyst to identify the abstraction in an interactive and iterative manner. After all, it finds only fragments of words that only an intelligent human can recognize as portions of relevant words and phrases. After each use of AbstFinder on the input that is left, the input is to be strained to remove from it the text related to the abstractions identified in the last use of AbstFinder. Then, AbstFinder is applied to the strained input [4]. We therefore compare our approach introduced in this paper to the abstraction identification mechanism defined by Gacitua et al. [1], as both are intended to be used in a one-shot manner.

In summary, abstraction identification can be thought of as where the expertise of domain expert and requirements engineer meet [3]. Since domain expertise is often available to the requirements engineer as NL documents (e.g., marketing reports, feature-release notes, etc.), abstractions identified from the relevant documents help encapsulate the rich contextual information needed for the framing of the requirements. Framing the requirements, as Jackson [11] pointed out, must consider explicitly the phenomena of the environment in which the software operates.

2.2 Environmental assumptions

An assumption is defined as: "*a thing that is accepted as true or as certain to happen, without proof*"[3]. In software development, an *environmental assumption* is a statement about the software system's operational context that is accepted as true by the developers [13]. For example, the statement: "a train is moving if and only if its speed is non-null" [12] is an assumption made about the physical world, whereas the statement: "the operator will not enter data faster than X words per minute" [22] is an assumption made about the user interactions with the software-intensive system.

Many software problems originate in missing, inadequate, inaccurate, or changing environmental assumptions [12]. Besides the aforementioned Ariane 5 launch failure [14], other examples include the false assumption regarding the stopping distance when heavier New York subway trains were introduced [23], as well as the inadequate assumption about the Therac-25 radiation therapy machine's protective circuits and mechanical interlocks [22]. All these flawed assumptions resulted in accidents and in the Therac-25 case, even fatal accidents.

Diagnosing whether an existing assumption is incorrectly removed or retained in a software product line is addressed by Rahimi and her colleagues [24]. Their diagnostic approach considers only the safety-related assumptions that are linked to the manually conducted Failure Mode, Effects and Criticality Analysis (FMECA) [25]. Rules are then defined to flag potential assumption errors for a new or changed product (e.g., a removed assumption that should have been maintained). Another source of variability is the location specificity of environmental assumptions pointed out by Alrajeh et al. [26]. While in London "polluting vehicles are admitted in a Low Emission Zone (LEZ)", the assumption does not hold everywhere, e.g., polluting vehicles are not admitted in Brussels's LEZ. As a result, the

[2] Mathematical definition is provided later in Eq. (6) of Sect. 5.2.

[3] http://www.oxforddictionaries.com/definition/english/assumption (last accessed on 2022/09/14 12:07:09).

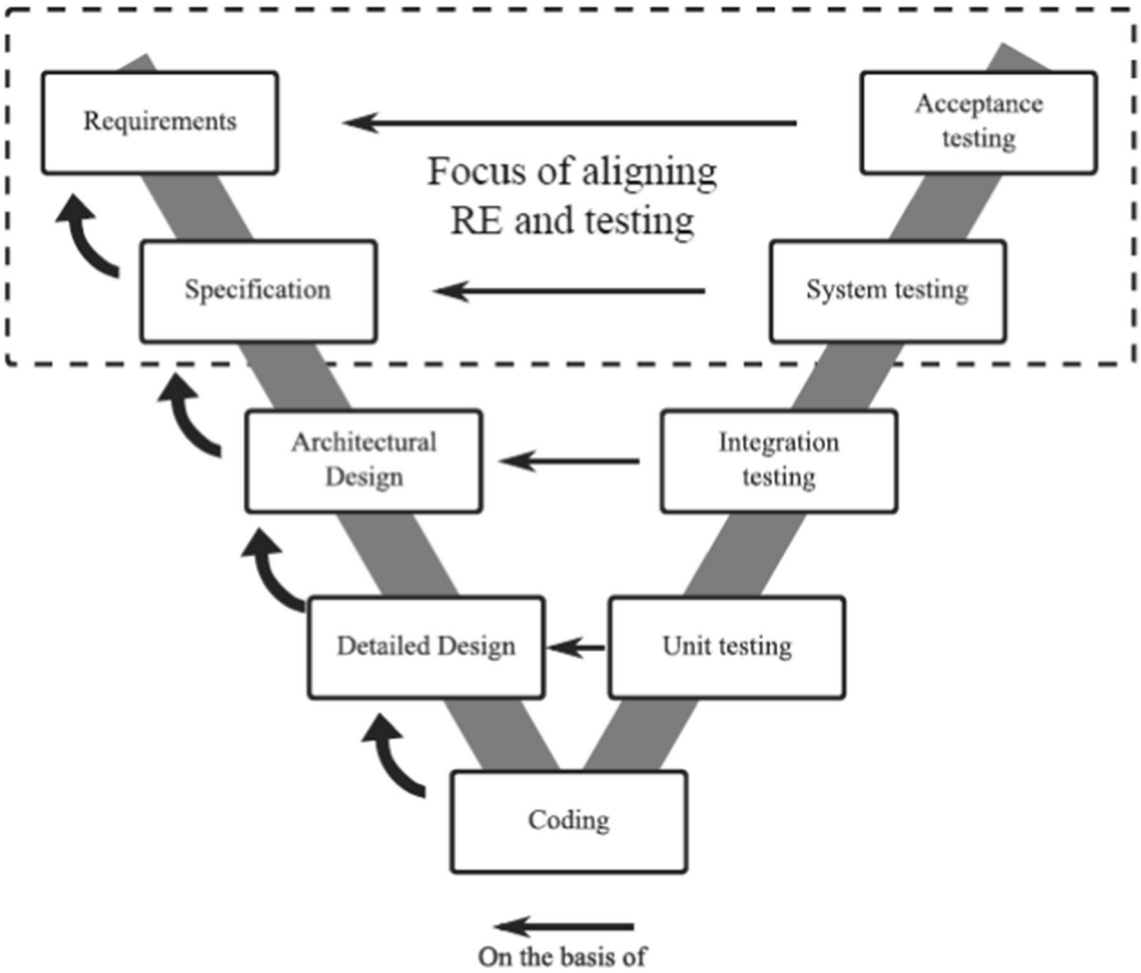


Fig. 1 Understanding the focus of aligning RE and software testing in the context of the V-model (*adapted from Unterkalmsteiner et al.* [35])

requirements goal models need to be adapted toward the varying environments [26].

Although documented assumptions may be flawed, one of the most critical problems is that assumptions are usually kept undocumented in software projects [27], leading to architectural mismatches [28, 29], budget and schedule overruns [30], security vulnerabilities [31, 32], and a multitude of system issues, defects, and failures. Similar to requirements and source code, assumptions are a type of software artifacts being produced, modified, and used by a process [33]. In Sect. 3, we present our proposal for integrating environmental assumptions in the requirements-based testing process.

2.3 Requirements engineering and testing

Software development consists of transitions from system concept, requirements specification, analysis and design, implementation, and test and maintenance [34]. Unterkalmsteiner et al. [35] used the V-Model of Fig. 1 to show the focus of aligning RE and testing in which black-box testing is more applicable than white-box testing. A strengthened RE-testing alignment could lead to benefits like improved product quality [36], cost-effective testing [37, 38], high-quality test cases [39], and early discovery of incomplete requirements [40]. Our use of abstractions is to assist in developing acceptance test cases, as will be shown in Sect. 6.

Since requirements are commonly documented in NL, NLP techniques have been exploited to assist software testing. Garousi et al. [41] reviewed 67 papers from 2011 to 2017 on NLP-assisted software testing and summarized that the top-three exploited techniques were PoS tagging, dependency parsing, and keyword checking. The more recent work by Fischbach et al. [42] employed dependency parsing to help identify causes and effects from a user story's acceptance criteria. Their empirical results show that the NLP-assisted automated approach could generate 56% of the test cases. The domain knowledge missed in the acceptance criteria was a main reason that hindered the scope of NLP's assistance in testing. Our abstraction identification presented

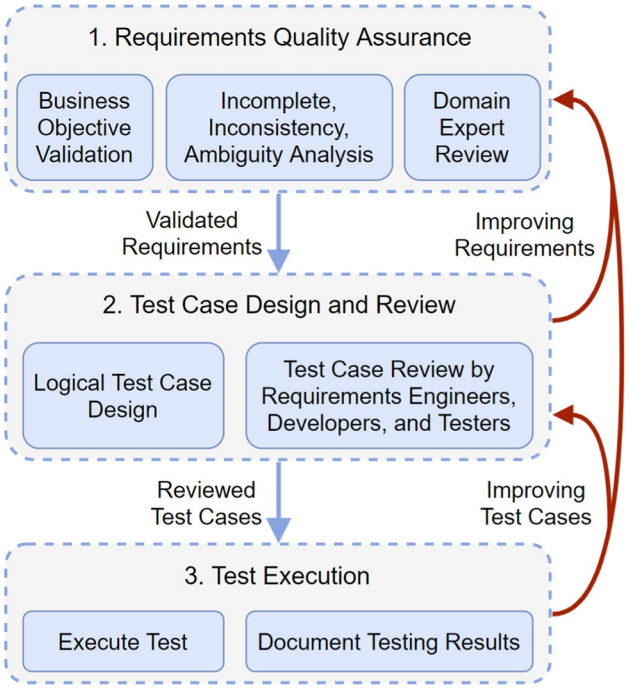


Fig. 2 The RBT process flow (adapted from Skoković and Skoković [43])

in Sect. 4 thus applies NLP to uncover domain knowledge from relevant corpus.

3 Environmental assumptions and requirements-based testing

Skoković and Skoković [43] introduced requirements-based testing (RBT) to address two major issues in software quality assurance: (1) validating the requirements are unambiguous, consistent, and complete, and (2) designing a necessary and sufficient set of test cases from a black-box perspective to cover the validated requirements. Figure 2 shows the three RBT activities, which we use iTrust's[4] "Schedule Appointments" requirement [44] to illustrate. Figure 3 displays a snippet of this use case.

RBT's first activity of requirements quality assurance stands out from other traditional testing techniques. Not only must the requirements be validated against the business objectives, but an initial review shall be conducted to try to find errors in requirements, such as ambiguity, incompleteness, and inconsistency [45].

Given the validated requirements, the next RBT activity is to design such black-box, logical test cases as to achieve high test coverage of the requirements [43]. Designing and reviewing logical test cases can help discover requirements problems, e.g., E1, as currently stated in Fig. 3, is triggered by a new appointment type's name over 30 characters, or by a duration unit not entered in minutes. However, S1 lacks information about whether the name and the duration are mandatory at a new appointment type's creation time. Thus, the requirements can be clarified for better logical test coverage.

The third RBT activity shown in Fig. 2 concerns executing tests by adding data to the logical test cases. Once all

Use Case: Schedule Appointments

Sub-flows:

[S1] The system shall enable the administrator to add a new entry for an appointment type, including its type, name with up to 30 alpha characters and duration in the unit of minutes [E1].

[S2] The LHCP (licensed health care professional) schedules an appointment with a patient, and enters comment (optional) up to 1000 characters such as reason for the appointment [E2].

[S3] The patient selects an LHCP from his or her provider list. The patient selects the type of appointment, enters the appointment date and start time. If the requested appointment time does not conflict with any existing appointment for the LHCP, the request is saved. If the requested appointment time does conflict with an existing appointment, the patient is presented with a list of the three next non-overlapping available appointment times within 7 days of the requested date. The patient selects one of these appointments and the request is saved.

Alternative Flows:

[E1] The user inputs invalid information and is prompted to try again.

[E2] The comment is empty and the text "No Comment" (without link) is displayed instead of the "Read Comment" link.

Fig. 3 Snippet of iTrust's "Schedule Appointments" use case (adapted from [15])

[4] iTrust is a Java application that provides patients with a means to keep up with their medical records and to communicate with their doctors [44].

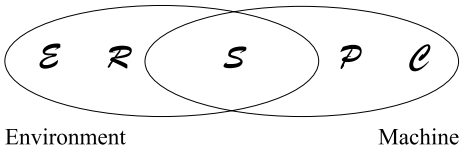


Fig. 4 The environment is the part of the world with which the machine (i.e., the software-intensive system) will interact (adapted from Jackson [11] Zave and Jackson [46], and Gunter et al. [47])

of the tests designed according to the validated requirements execute successfully against the code, Skoković and Skoković [43] argue that 100% of the functionality has been verified and the code is ready to be delivered into production.

Despite RBT's attentions paid to requirements, we believe the process of Fig. 2 can be enhanced by two shifts, both involving environmental assumptions. Note that the shifts that we propose concern the second step of test case design in Fig. 2. This shows a focused use of environmental assumptions in the RBT process. We do not apply the assumptions to the first step of requirements quality assurance in Fig. 2 because we take the requirements descriptions directly from the software vendors. We currently perform the step 3 of test execution in Fig. 2 manually and aim to improve the test execution efficiency in future work. The first shift that we propose over the RBT process of Fig. 2 is revisiting the meaning of requirements, which is defined by Jackson as [11]:

$$\mathcal{E}, \mathcal{S} \vdash \mathcal{R} \tag{1}$$

in which $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{R}$ represent environmental assumptions, specifications, and requirements respectively. Figure 4 depicts the conceptual distinction and overlap between the environment and the machine (software-to-be). A customer requirement $\mathcal{R}$ expresses a condition over the phenomena of the environment that we wish to make true by installing the machine, whereas an environmental assumption $\mathcal{E}$ expresses a condition over the phenomena of the environment that we accept to be true irrespective of the properties and behavior of the machine [11, 46]. The $\vdash$ among $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{R}$ is an entailment, meaning that if a machine doing $\mathcal{S}$ is installed in an environment having $\mathcal{E}$, then we know $\mathcal{R}$ will be fulfilled. Compared with validating $\mathcal{R}$ against business objectives shown in the top activity of Fig. 2, Eq. (1) shifts requirements validation toward the entailment relationship, $\vdash$, to which $\mathcal{E}$ is integral.

In Fig. 4, $\mathcal{P}$ and $\mathcal{C}$ are private to the machine domain: $\mathcal{P}$ denotes the program implementing the specification, and $\mathcal{C}$ denotes the computing platform on which $\mathcal{P}$ runs. The correctness of a software implementation is given by Zave and Jackson [46] and Gunter et al. [47] as:

$$\mathcal{P}, \mathcal{C} \vdash \mathcal{S} \tag{2}$$

and we further denote the software under test (SUT) by $(\mathcal{P}, \mathcal{C})$ to indicate that software testing, including RBT, is to stimulate $\mathcal{P}$ on $\mathcal{C}$ as a whole with concrete input data. From Eqs. (1) and (2), we have:

$$\mathcal{E}, \text{SUT} \vdash \mathcal{R} \tag{3}$$

deducing requirements validation in the absence of $\mathcal{S}$.

The second shift that we propose to the RBT process shown in Fig. 2 is to highlight the practical value of software testing. Dijkstra famously said, "*Testing shows the presence, not the absence of bugs*"[5]. We argue that searching for those environmental assumptions and test inputs such that:

$$\mathcal{E}, \text{SUT} \nvdash \mathcal{R} \tag{4}$$

would be more valuable to RBT than trying to achieve a 100% test coverage against the requirements. Let us revisit iTrust's "Schedule Appointment" requirement in Fig. 3. Subflow [S3] describes two branches after the patient enters the requested appointment time: no time conflict with the LHCP or otherwise. Two logical test cases can then be designed to have both branches covered. However, an unstated assumption about [S3] is that: "If a doctor's appointment cannot be made at the patient's preferred time, the patient will accept an alternative appointment time within 7 days of the preferred time."

Making this assumption explicit allows us to construct a concrete $\mathcal{E}$ (e.g., "a patient wants to know the options, and possibly to schedule the appointment, beyond 7 days of the unfulfilled, preferred time") such that $\mathcal{E}$, SUT $\nvdash \mathcal{R}$, effectively showing the presence of a bug[6] and potentially provoking requirements changes. From this example and the discussions of this section so far, we derive the desiderata of the kinds of $\mathcal{E}$ that best support our revised RBT process:

- $\mathcal{E}$ shall be **machine independent**. Regardless of iTrust or any other machine being the SUT, $\mathcal{E}$ is in the indicative mood [11], expressing what is assumed to be true in the environment.
- $\mathcal{E}$ shall be **requirements related**. Although $\mathcal{E}$ is independent of the machine, it should not be indifferent to $\mathcal{R}$. Even for the same SUT, different requirements, in principle, need different assumptions to support the RBT.

[5] https://en.wikiquote.org/wiki/Edsger_W._Dijkstra (last accessed on 2022/09/14 12:07:09).

[6] By *bug*, we mean Eq. (4) is evaluated to be true (i.e., the entailment relationship fails to hold) under a specific set of $\mathcal{E}$, SUT, and $\mathcal{R}$. We refine the notion of bug in Sect. 6 by formulating the test oracle without directly referring to $\mathcal{R}$ in order to address the distance between observing the software executions and validating the stakeholder goals.

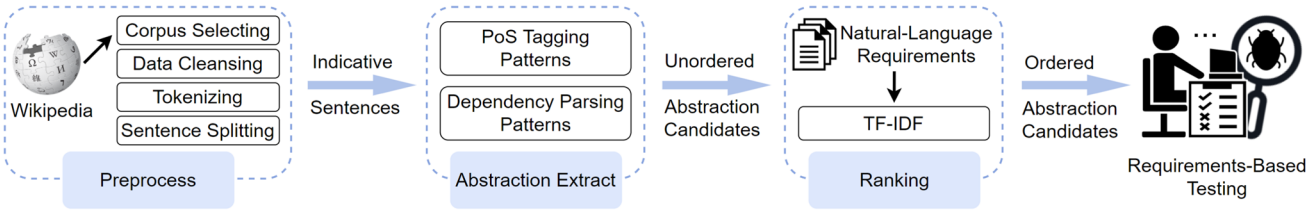


Fig. 5 Components of the NLP-aided abstraction identification approach

- $\mathcal{E}$ shall be **directly testable**. The $\mathcal{E}$ that gives rise to executable tests is preferred. In addition to being closely related to $\mathcal{R}$, $\mathcal{E}$ shall induce as concrete test inputs as possible to trigger the SUT.
- $\mathcal{E}$ shall be **bug revealing**. The practical value of showing the presence of bugs implies that the RBT helps to uncover flawed assumptions, faulty implementations, or invalidated requirements.

Automatically generating $\mathcal{E}$ with all the desiderata from NL documents is unrealistic [9], but the less ambitious goal of assisting the discovery of problem domain properties is feasible, as evidenced by abstraction identification in RE (cf. Sect. 2.1). The next section presents our automated support for identifying abstractions driven by the desiderata of $\mathcal{E}$.

4 Abstraction identification for environmental assumptions

Figure 5 shows an overview of our NLP tool chain for extracting and ranking abstractions. The candidate abstractions are then used to assist the human analyst in performing the RBT. Our approach consists of three major steps: corpus selection from Wikipedia, abstraction identification via NLP patterns, and abstraction ranking according to the textual similarity between the extracted abstractions and the NL requirements of a software-intensive system. This section uses iTrust [44], a software system helping to manage electronic health records, to illustrate our approach.

4.1 Preprocessing

The data source that we use to identify the abstractions is Wikipedia. The main rationale is to achieve the **machine independent** property of the resulting abstractions. Wikipedia is a vast online source of human knowledge that describes the concepts across a wide range of domains. These descriptions are independent from any specific software solutions, and are indicative stating what is believed to be true rather than what is wished to be true by introducing a software solution. Additionally, Wikipedia is a text corpus that collectively contains the current conventional wisdom of the subject matters [48]. This ensures the descriptions are less biased than individuals' subjective opinions, e.g., tweets or reviews about a software product.

To select a corpus for our purpose of abstraction identification, a seed page is required to anchor the domain interests. Our approach relies on the human analyst to provide such a seed page. For iTrust, for example, the Wikipedia page on "Electronic health records"[7] is manually chosen as the seed page, from which a corpus containing related pages is derived. We build on Ezzini et al.'s recent work where they used a domain-specific corpus for detecting requirements ambiguities [49]. While using a corpus that is too small would be ineffective in recognizing significant terms and their relationships, building and using a corpus that is too large would be time-consuming and, more importantly, would defeat the goal of being domain-specific [49]. In Ezzini et al.'s work, 50–250 keywords were tested, and in our work here, we empirically set the corpus's size to be 250 Wikipedia pages.

To grow from the single seed page to the 250-page corpus, we distinguish two kinds of Wikipedia pages: *content page* introducing a topic and *category page* listing a set of content pages that belong to a topic as well as the subcategories of the topic. Figure 6 illustrates the distinction: the sub-figure to the left is a content page whereas the one to the right is a category page, showing that "Electronic health records" contain two sub-categories and 45 pages. We further note the hyperlinks within each content page, e.g., "health care" and "information systems" of Fig. 6a. These links not only provide an efficient navigation mechanism over the Wikipedia contents, but also represent some semantic relationships between pages or categories [50]. Our corpus is constructed with the following procedure: While the total number of pages is less than 250,

1. Add the manually identified seed page, resulting in the "Electronic health records" corpus size to be 1 page;

[7] https://en.wikipedia.org/wiki/Electronic_health_record (last accessed on 2022/09/14 12:07:09).

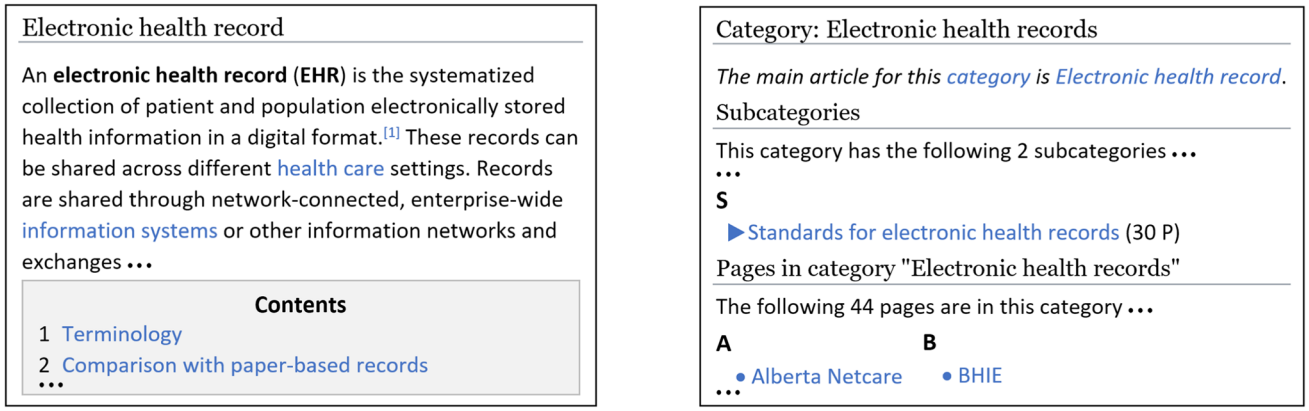


Fig. 6 Sample content page (used in our domain corpus) and sample category page (used to find more content pages to be included in the corpus)

2. Add all the content pages belong to the seed page's topic, making the corpus's size grow to (1+44) = 45 pages;
3. Add all the content pages belonging to the sub-categories of the seed page's topic, leading to a (45 ± 30 ± 30) = 105-page corpus; and
4. From the already added content pages, add the Wikipedia content pages of the hyperlinks *before* each page's structured table of contents (cf. Fig. 6a). The reason that we include into our corpus only the hyperlinked pages before the table of contents is because these topics provide substantial background information for the main topic of interest. We operate 4) based on when a page is added by following the above 1), 2) and 3) ordering, till a total of 250 Wikipedia pages is reached.

We implemented our page selection logic by using the Beautiful Soup Python library [51]. Our Python-based corpus builder also ensured that no duplicate Wikipedia page was selected. Once the corpus's 250 pages were chosen, Fig. 5 shows that data cleansing, tokenizing, and sentence splitting would take place. For each page, our data cleansing removed the figures and the formatting information (e.g., table of contents). Following prior work [52, 53], we then applied spaCy's tokenizer [54] to break each page into tokens: words, numbers, punctuation marks, or symbols. Finally, we used spaCy's sentencizer [54] to split the text into sentences based on conventional delimiters (e.g., period).

4.2 Extracting abstractions

We process each sentence from the selected Wikipedia pages in order to find **directly testable** abstraction candidates. We operationalize *testability* by formatting an abstraction as a <key, value> pair. This hash structure is intended to separate a domain concept (key) from its manifestation (value). We expect the "key" to help locate "what to test", and the "value" to guide "how to test it" by feeding in concrete data.

Building on the NLP-based abstraction identification approaches [2, 3], we define five patterns by exploiting the syntactic and grammatical roles that words play in a sentence. Figure 7 lists these patterns. Note that Gacitua et al. [1, 3] used two PoS patterns—*adjectives and nouns* and *adverbs and verbs*—to identify abstractions; however, the two PoS patterns of our approach, PoS_P1 and PoS_P2 shown in Fig. 7, consider <key, value> explicitly.

Dependency parsing [55] is the task of identifying the grammatical structure of a sentence by determining the linguistic dependencies between the words based on a pre-defined set of *dependency types*. For example, in the sentence: "The system shall refresh the display", "display" is the direct object (*dobj*) of the main verb "refresh" whereas "shall" is the auxiliary verb (*aux*) adding modality to the main verb. Dalpiaz et al. [56] recently exploited dependency parsing to classify functional and non-functional requirements. Although our objective is to identify abstractions, we believe dependency parsing, just like PoS tagging, can constitute an effective NLP toolset that offers deep domain understandings [2]. We detail each of the five patterns as follows:

– **PoS_P1** extracts the NN or NNS that precedes the parentheses as key, and the content inside the parentheses as value. Cohen et al. [57] showed that the parenthesized material in biomedical text often contains data value or list element useful for information extraction, which we also observe in Wikipedia pages. This pattern therefore extracts <"adult", "age 15 +"> from *S1* of Fig. 7.

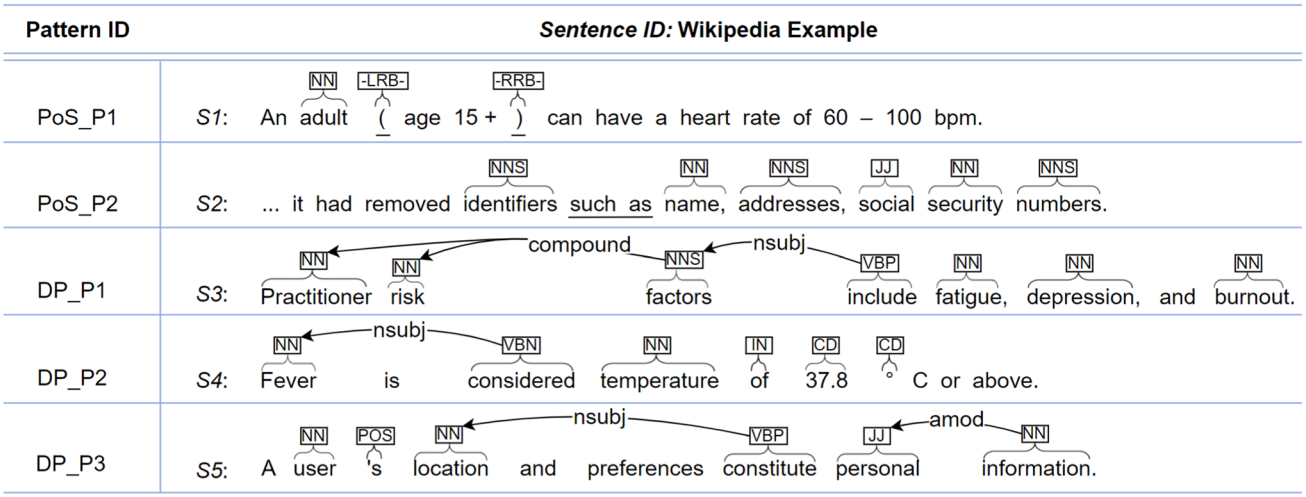


Fig. 7 Illustration of our five patterns for identifying <key, value> abstractions in a sentence. PoS (part-of-speech) tags shown here include: "NN" (noun, singular), "-LRB-" (left round bracket), "-RRB-" (right round bracket), "NNS" (noun, plural), "JJ" (adjective), "VBP" (verb, non-3rd person singular present), "VBN" (verb, past participle), "IN" (preposition), "CD" (cardinal number), and "POS" (possessive ending). Dependency parsing types shown here include: "compound" (noun compound modifier), "nsubj" (nominal subject), and "amod" (adjectival modifier)

- **PoS_P2** leverages a common lexico-syntactic way of providing examples [58] so as to identify the NN or NNS to be a key, and the hyponym(s) [59] following "such as" to be its value(s). The pattern thus outputs <"identifiers", "name" "addresses" "social security numbers"> as an abstraction candidate for Fig. 7's *S2*.
- **DP_P1** recognizes the nominal subject ("nsub") or the subject's compound ("compound") as a key, and the parallel NN's after the main verb "include" (or "includes") as values. Different from the "such as" pattern used in **PoS_P2**, "include" signals a part-whole relationship [60], listing several concrete facets of the concept. In Fig. 7's *S3*, <"practitioner risk factors", "fatigue" "depression" "burnout"> is identified as an abstraction candidate.
- **DP_P2** treats the nominal subject as a key in the same manner as **DP_P1**; however, its value is extracted based on the sequence of "NN, IN, and CD" appearing after the main verb. **DP_P2** is informed by Dalpiaz et al.'s finding that "IN and CD" often distinguished requirements types [56]. For *S4* of Fig. 7, **DP_P2** uncovers <"fever", "temperature of 37.8° "> as a candidate pair.
- **DP_P3** also considers the nominal subject to be a key, with the value identified via the NN that follows the main verb and that is modified by an adjectival ("amod"), nominal ("nmod"), or numeric ("nummod") modifier. Finin [61] noted that modifier takes a head concept and a potential modifying concept and produces a set of possible interpretations, which we adopt to derive **DP_P3**. According to this pattern, <"user's location", "personal information"> is recognized from *S5* of Fig. 7.

The above set is by no means an exhaustive list of grammatical features that must be associated with environmental-assumption statements, but a means of automatically extracting directly testable abstractions. The NLP patterns are informed by the relevant literature [56–61] and further realized by our PoS tagging and dependency parsing implementations built on top of the open-source spaCy library [54] written in Python. We also made the implementations of our preprocessing and NLP steps publicly available on Google Colab [62], facilitating the users to identify abstractions according to our approach inside their web browsers.

4.3 Ordering abstractions

So far, our approach processes information related to a *domain*, e.g., Electronic health records. To ensure that our abstraction identification is **requirements related**, we use the NL requirements of each specific *product* [63] to rank the abstraction candidates. In this way, even though some software-intensive systems are in the same domain, their *ranked* abstractions will be different due to the differences of the requirements at the product level.

Given a product's requirements R={ r_1, r_2, ..., r_m }, e.g., iTrust's 54 use cases, we first compute the cosine similarity measure between a candidate abstraction (*abst*) and R with TF-IDF weighting [64–66]:

$$\text{cosine}(abst, R) = \sum_{i=1}^{m} \text{cosine}(abst, r_i). \quad (5)$$

Table 1 Domain and subject system characteristics

	Electronic health records	Web conferencing
# of sentences	33,988	43,744
Category depths	3	3
# (%) of sentences to which a pattern applies requirements		
PoS_P1	5437 (16.0%)	4553 (10.4%)
PoS_P2	1962 (5.8%)	1182 (2.7%)
DP_P1	2133 (6.3%)	785 (1.8%)
DP_P2	3236 (9.5%)	2945 (6.7%)
DP_P3	17,393 (51.2%)	15,973 (36.5%)
# of product-level requirements	54 (iTrust)	49 (Teams)
	27 (OpenEMR)	26 (Webex)
	39 (OpenMRS)	31 (Zoom)

Zoom_r1:	Presenters can share their whole desktop or individual applications.
Zoom_r2:	Primary camera view will automatically toggle to the active speaker.
Zoom_r3:	Browser, client, and plugin scheduling options, including delegation for co-hosts and schedulers.
Zoom_r4:	Record meetings locally and upload to Box, OneDrive Video, or Youtube.
Zoom_r5:	Zoom sessions can be expanded to allow larger groups, up to 500 interactive participants in Large Rooms or 10,000 viewers via Zoom Webinars.
Zoom_r6:	Hide all participants whose cameras are disabled.
Zoom_r7:	Automatically transcribe the audio of a meeting or a webinar; the host can edit the transcript.

Fig. 8 Sample NL requirements of Zoom

Note that R can be a set of selected requirements, or even a single requirement of interest. We then rank all abstraction candidates by their cosine similarity scores in a descending order.

5 Evaluating the effectiveness and relevance of identified abstractions

This section presents a comprehensive evaluation of our abstraction identification approach, comparing its performance with the effectiveness of a state-of-the-art technique: relevance-based abstraction identification (RAI) [1]. Furthermore, to assess relevance, we examine the resulting abstractions' matching with manually created environmental assumptions, as well as their capabilities of revealing software defects. We answer three research questions (RQs) in this section by investigating the effectiveness (RQ1), partialness (RQ2), and bug revealingness (RQ3) of the produced abstractions. All our experimental materials are publicly available at https://doi.org/10.5281/zenodo.5858384 (last accessed on 2022/09/14 12:07:09).

5.1 Domains and subject systems

We chose to study two domains, Electronic health records and Web conferencing, due to our familiarity with them and the availability of software applications in them. We ran our Python-based preprocessing steps in February 2021. Table 1 shows that, when we manually selected https://en.wikipedia.org/wiki/Electronic_health_record and https://en.wikipedia.org/wiki/Web_conferencing (both links last accessed on 2022/09/14 12:07:09) to be the seed page respectively, a total of 33,988 and 43,744 sentences were collected. In both domains, the selected 250 Wikipedia pages were within 3 category depths of each other, implying these pages' topical closeness [67]. The five NLP patterns' applicability ranged from 1.8% to 51.2% of the sentences. In both domains, **DP_P3** had a wide influence, showing that many Wikipedia sentences have described the nominal subject ("nsub") with adjectival, nominal, or numeric modifiers.

Table 2 Top-ten abstraction results of iTrust and OpenEMR by RAI [1] and by Our <Key, Value> Pair (KVP) approach

Rank	iTrust		OpenEMR	
	RAI	KVP	RAI	KVP
1	cardiac causes	<"factors", "vascular damage">	subsequent recovery	<"identifier type", "uniform">
2	healthcare	<"factors", "pathogens" "dysfunctions">	subsequent amendment	<"cell type", "cytokines">
3	security	<"factors", "mental protection">	subsequent surgical	<"type", "pharmacologic">
4	natural causes	<"safety checks", "potential dose">	subsequent complications	<"procedure", "patient id" "name" "sex" "age">
5	reversible causes	<"x - rays", "absorbed dose">	ambulatory care pharmacists	<"term", "patient type">
6	annual meeting	<"rash", "adverse reaction">	surgical nurses	<"type", "type private">
7	pediatric radiology	<"dose", "equivalent dose">	specific patient types	<"product type", "type stage">
8	biological causes	<"stage", "ideal dose">	dynamic nuclear polarization	<"symptoms", "shortness breath" "chest pain">
9	preventable causes	<"radiation dose", "average dose">	initial tomography acquisition	<"type", "additional type">
10	itchy rash	<"radiation dose units", "dose units">	chest pain	<"type", "type college">

We selected three software applications for each chosen domain. Table 1 lists the number of NL requirements for these applications. In addition to iTrust, we investigated two open-source medical record systems: OpenEMR [68] and OpenMRS [69]. OpenEMR is one of the most popular electronic medical records in use today with over 7,000 downloads per month, and its project repository lists 27 features, such as patient scheduling, prescriptions, and medical billing [70]. OpenMRS allows for customizable electronic medical record systems to support the delivery of health care in developing countries; 39 requirements are introduced in OpenMRS's user guide [71], including viewing and creating patient records, patient dashboard, etc.

Web conferencing has become one of the most prevalent and useful tools due to the COVID-19 pandemic. We studied three popular products: Zoom, Cisco Webex, and Microsoft Teams. While our experimental materials list all the product-level requirements and the online resources from which we collected the requirements, Fig. 8 shows a few Zoom features.

5.2 Effectiveness of abstraction identification

As did Gacitua et al. [1], we measure abstraction identification's effectiveness by precision and recall. In addition to our <key, value> pair (KVP) abstractions, we also experimented with RAI, which uses corpus-based frequency profiling to compute the log-likelihood (*LL*) value for a word *w* [1]:

$$LL_w = 2 \cdot \left(w_d \cdot \ln \frac{w_d}{E_d} + w_r \cdot \ln \frac{w_r}{E_r} \right) \tag{6}$$

in which w_d is the number of times w appears in the domain documents of the 250 Wikipedia pages, and w_r is the number of times w appears in the product-level requirements. While w_d and w_r are observed values of w, E_d and E_r in Eq. (6) are expected values: $E_d = \frac{n_d \cdot (w_d + w_r)}{n_d + n_r}$ and $E_r = \frac{n_r \cdot (w_d + w_r)}{n_d + n_r}$, where n_d is the total number of words in the domain documents, and n_r is the total number of words in the product-level requirements. We ranked the single-word abstraction candidates in the ascending order of LL_w to obtain testable constructs: the smaller is LL_w, the significance of w in domain documents is closer to that of w in product-level requirements, making w's ranking closer to the top. The multiword abstraction candidates were recognized by the *adjectives and nouns* and *adverbs and verbs* PoS patterns, and then ranked by an aggregated *LL* for the multiword unit [1].

Table 2 presents a comparison of RAI and our approach. Due to space constraints, we show only the top-ten abstraction candidates of iTrust and OpenEMR from the healthcare domain. To quantitatively and practically evaluate different abstraction identification techniques, we measured precision of the topmost 20, 100, and 200 results. Sawyer et al. [2] hypothesized that human analysts would be reluctant to

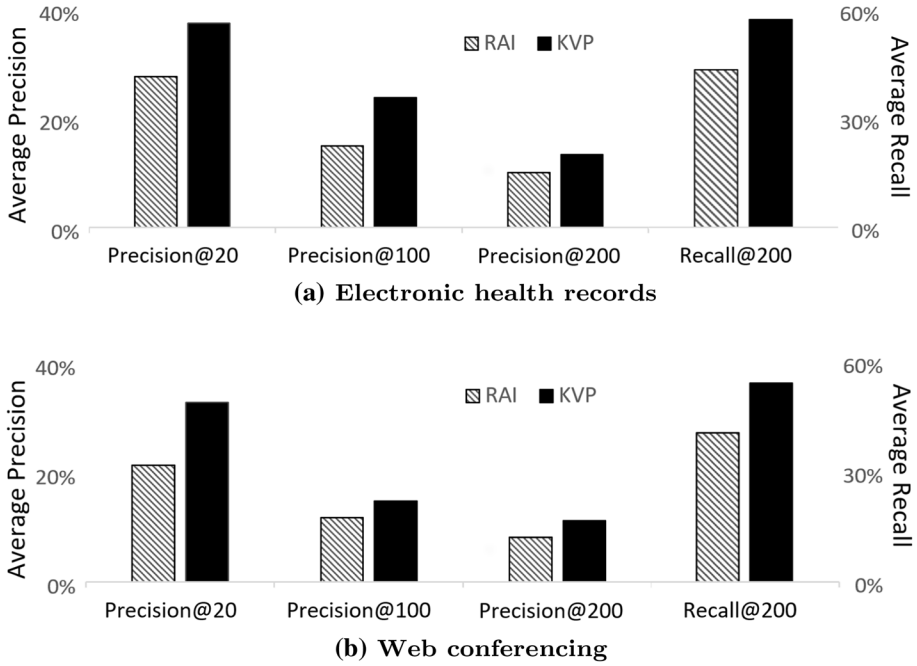


Fig. 9 Accuracy of candidate abstractions

explore a long list of abstraction candidates and used the 20 highest ranked candidates to simulate the hypothesis. Gacitua et al. [1] evaluated the topmost 200 candidates as a practical upper bound. Therefore, we also computed recall@200 without exceeding this limit.

The answer set for each subject system was constructed by two researchers: first individually and then jointly to resolve discrepancies. Cohen's kappa before the joint meeting was 0.57, signifying a moderate level of inter-rater agreement. The main challenge was to handle abstractions with varying lengths yet overlapping partially, e.g., in iTrust, RAI's 153rd output was "medical imaging", and our 31st pair was <"medical imaging", "time">. The research team decided to unify these candidates with shortest-common-supersequence [72], i.e., creating *one* element of "medical imaging time" in the answer set and then using *'contained in'* to establish a match. In this way, RAI's 153rd and KVP's 31st outputs were considered as matching the answer set's element of "medical imaging time" because both candidates were 'contained in' that element. Note that the 'contained in' match was not ordered, so "time medical" would be a match, too. The two researchers collaboratively defined the answer set for each subject system based on their individual judgments and the shortest-common-supersequence unifying step. A third researcher reviewed and agreed on the answer sets, which we share as a part of our experimental materials.

Figure 9 plots the average precision and recall among the three systems in each domain. When considering precision, we note that KVP outperforms RAI. Such effects are observed more prominently in the healthcare domain than web conferencing. One reason may be the relatively homogeneous domain concepts in web conferencing; in contrast, healthcare covers wider domain phenomena, e.g., symptoms, health conditions, and treatments. Given that a precision over 25% represents practically achievable good abstraction identification performance [2], both methods perform well at Precision@20 in the healthcare domain; however, KVP also performs well at Precision@20 in web conferencing and even at Precision@100 in healthcare.

We offer a couple of qualitative insights into the performance differences. First, KVP is contextually richer than RAI. In iTrust, RAI outputs "blood pressure" as its 45th candidate, a rather general concept. A relevant KVP generated by our approach is the 98th candidate: <"blood pressure", "140">, depicting a testable and indicative domain phenomenon, i.e., 140 mm Hg or above implies Stage 2 high blood pressure. Our second observation is that KVP tends to group relevant domain phenomena which are spread out otherwise. In OpenEMR, for instance, <"symptoms", "shortness breath" "chest pain"> is ranked 8th. RAI, on the other hand, recognizes "chest pain" as its 10th candidate and "shortness breath" as its 352nd candidate. The distance is so large that the two candidates are unlikely to be related with one another. As shown in Fig. 9, KVP's Recall@200 reaches about 60% in both domains, covering more relevant domain phenomena than RAI.

5.3 Comparison with manually formulated assumptions

To gain further insights into abstractions' completeness, we compare them with manually created environmental

Table 3 Coverage of the eight gists of the environmental assertions about iTrust's "schedule appointments" use case [15]

Gist	RAI	KVP
Valid user accounts	√	√
Valid appointment time	√	√
Valid appointment type		
Unique patient account	√	√
Valid schedule alternatives		
Designated LHCP		
Responding to appointment requests		√
Displaying patient message	√	√

coverage over the eight gists listed in Table 3. For instance, "valid user account" was covered by RAI's 89th candidate ("valid patient") and KVP's 59th candidate (<"user", "valid patient">). Compared to RAI, KVP had the highest coverage. Notably, the 142nd KVP, <"respondents", "regular patients">, corresponded to the assertion that the respondents (e.g., a licensed health care professional) shall approve or deny the patients' requests for appointment. Such an assertion was not covered by any of the top-200 candidates generated by RAI.

In a study on environmental assumptions, Rahimi and her colleagues [24] collected 150 statements from the literature: textbooks, papers, and websites. From this collection,

Table 4 Top-ranked <Key, Value> pair matching meeting scheduler's environmental assumptions ("−−" means no relevant pair was found within the top-200 abstraction candidates to match the assumption)

Environmental assumption	Teams	Webex	Zoom
A participant cannot attend multiple meetings at the same time	67th: <"offline attacks", "multiple user accounts">	29th: <"shared password, "another person citation">	59th: <"addition users", "passwords bookmarks" "history" "cookies">
Participants will promptly respond to e-mai requestsl	58th: <"e-mail", "Microsoft 365 Business Basic">	−−	−−
A participant is on the invitee list if & only if he or she is invited to that meeting	−−	−−	188th: <"user", "required path">
A meeting is scheduled if & only if its time and location are set	5th: <"meetings", "Microsoft Teams">	133rd: <"meeting", "time meeting">	15th: <"meetings", "up to 100 devices" "40-minute time restriction">
A meeting is scheduled only if it is requested.	−−	−−	−−
Saturdays are excluded dates for meetings	−−	−−	−−
Confidentiality rules can prevent non-privileged participants being aware of constraint	181st: <"messenger rooms", "limit participant of 50">	155th: <"server hosts", "private sharing">	153rd: <"security", "unauthorized person">
Confidentiality rules can prevent non-privileged participants being aware of meetings	173rd: <"access control list", "OneDrive folder">	165th: <"person meeting", "private meeting">	129th: <"ip address", "host identification">

assumptions. Our comparison is twofold: matching the 150 iTrust assertions studied by Bhowmik et al. [15], and matching the 8 meeting scheduler assumptions shared by Rahimi et al. [24]. In both cases, two researchers manually extracted the gists of the iTrust assertions and the meeting scheduler assumptions, and then matched the automatically generated abstractions with those gists in a joint session.

Table 3 shows the coverage results of iTrust. As the environmental assertions in [15] are about iTrust's "Schedule Appointments" use case (cf. Fig. 3), we used only this use case's NL descriptions, instead of the NL requirements of all 54 iTrust's use cases, to rank the automatically generated abstraction candidates. Inspecting the top-200 abstractions identified by RAI and our KVP led to a 50% and 62.5%

we selected all the assumptions of the meeting scheduler domain. These eight statements, shown in the leftmost column of Table 4, were taken from van Lamsweerde [12]. The rest of Table 4 provides the top-ranked, relevant KVP identified by our approach that matched the assumption. Although some statements are idiosyncratic, such as "Saturdays are excluded dates for meetings", no KVP within the top-200 abstractions was found to be relevant to the gist of not scheduling meetings on some special date. Overall, the coverage of the eight meeting scheduler assumptions [12] is 58.3% ($\frac{14}{24}$). A noteworthy finding is that, even for the same assumption, different abstractions are identified for different software products, enabling more specialized testing for each

system, e.g., checking if a participant who is excluded from the "access control list" in the "OneDrive folder" could join a Teams meeting, or "ip address" should authenticate "host identification" in Zoom.

5.4 Bug revealing capability of abstractions

As we discussed in Eq. (4), revealing bugs shows the practical value of the abstractions in the RBT process. Among our six subject systems, we focused on the known bugs of iTrust, Teams, Webex, and Zoom. Bhowmik et al. [15] highlighted two defects of iTrust discovered manually in the RBT process. In our analysis, both bugs could be revealed with support of the KVP results. We manually judged which abstractions, if known to the testers, could help detect the bugs. We found that the 139th pair, <"appointment", "last year">, could help uncover the bug that iTrust allowed an appointment to be made for a past time, and the 79th pair, <"time", "scheduling conflicts"> could help detect the bug that iTrust allowed a patient to schedule appointments with multiple doctors at the same time.

Our manual web search found 7 bugs for Teams, 5 bugs for Webex, and 7 bugs for Zoom, all of which are shared in our experimental materials. Analyzing the top-200 KVPs manually, we were able to use the abstractions to help reveal 3 Teams's bugs (43%), 2 Webex's bugs (40%), and 2 Zoom's bugs (29%). For example, Teams's 162nd pair was <"meeting organizer", "repetition occurrence meeting">, helping to define a path of RBT as follows:

1. Scheduler organizes a daily meeting series;
2. Scheduler invites a guest to join only on day #3;
3. Guest accesses the meeting series's text chats on day #4.

If the assumption at step 2) is *one-time* invitation only, then step 3) is expected to fail. However, step 3) was successful in Teams, because Teams assumed the guest invitation was from day #3 *onward*. Constructing the above testing path also needs the deep understanding of inviting a guest to a "repetition occurrence meeting". This emphasizes that the automatically identified abstractions are supporting the human analysts, rather than replacing them, in performing RBT. Nevertheless, our 65th KVP, <"sharing feature", "video sharing" "audio sharing" "desktop sharing" "file sharing" "whiteboard sharing" "text sharing">, clearly suggests some new testing paths similar to the above, where step 3) can concentrate on guest's access to day #4's uploaded files or day #4's shared whiteboard. In fact, Teams provides separate entry points to chats, files, and whiteboards. Resolving guest's chat access does *not* resolve the file or whiteboard access. Thus, related KVPs can improve the efficiency of uncovering related bugs.

5.5 Threats to validity

A threat to construct validity is that our answer set's building adopted shortest-common-supersequence [72] in order to unify the abstractions identified by different techniques. This caused the matching between abstraction candidates and answer set elements to be judged on a *'contained in'* basis, which must be taken into account when interpreting our reported recall and precision values. Another threat is that our assessment of the bug revealing capability of abstractions is only speculative in this work. In the next section, we offer a refined interpretation of bug revealingness thereby improving the testing's objectiveness.

We believe the internal validity is high in that the factors potentially affecting the abstractions' accuracy, coverage, and bug revealingness measures are under our direct control. This makes the abstraction identification techniques the cause of observed differences. One factor worth noting is that we ran our Python-based preprocessing steps in February 2021 in order to collect the 250 Wikipedia pages for each domain; however, the collected pages affected both techniques. The comparison results between RAI and our KVP approach therefore remain valid. In addition, our Google Colab tool [62] helps mitigate this limitation in that the users can identify up-to-date abstractions in a dynamic fashion.

Our evaluation results may not generalize to other subject systems or other domains, a threat to external validity. Studying more software applications within and beyond Electronic health records and Web conferencing will be valuable. Another threat here is our reliance on Wikipedia for abstraction identification. From a computational linguistic point of view, Wikipedia provides a balance in size, quality, and structure, between the highly-structured, but limited in coverage, linguistic databases like WordNet, and the large-scale, but less-structured, corpora such as the entire Web [48]. Nevertheless, using other corpora, including user forums [73], or combining Wikipedia with additional NLP support like WordNet could be interesting directions to expand our work.

6 Feature testing in opposing environmental conditions

Building on the identified abstractions, we present in this section a new way to perform RBT by targeting a software system's changing features. To that end, we first revisit the notion of *bug* and point out the distance of software testing and requirements validation in Sect. 6.1. We then review the use of finite state machines to design acceptance test cases in Sect. 6.2, articulate our feature testing approach in Sect. 6.3, demonstrate the extended support from the abstractions in Sect. 6.4, and discuss the implications of

environment-driven, abstraction-supported feature testing in Sect. 6.5.

6.1 Revisiting bug revealingness

We previously interpreted bug in RBT according to Eq. (4), i.e., the SUT, once executed, fails to entail $\mathcal{R}$. In practice, this view faces some challenges. Let us revisit the Teams's bug of inviting a guest to join a recurring meeting series. For the stakeholders who want the guest to join at a certain point in time and onward, Teams's current implementation would *not* be buggy, i.e., $\mathcal{E}, \text{SUT} \vdash \mathcal{R}$ for this group of stakeholders. Yet, for those who want the guest invitation to be valid for one time and one time only, a solution could be using Teams to schedule a separate, standalone meeting with the guest and the regular attendees without resorting to the recurring series at all. Then, $\mathcal{E}, \text{SUT} \vdash \mathcal{R}$ also holds for the one time only guest invitation scenario, and hence no Teams's bug would be found.

Clearly, two different features of the SUT are involved: inviting guest to a recurring meeting series and scheduling a standalone meeting. Since different features help satisfy different, and in this example, diverging goals, recognizing which feature to test is important to RBT. In another word, being requirements related means choosing the right feature to test. Besides feature selection, another challenge of RBT's bug revealingness lies in the inherent distance between testing a feature and reasoning about a requirement's fulfillment. For Teams's features, whether meeting with a guest in a recurring series or in a standalone meeting, a direct observable is the outcome of the actual scheduling, i.e., whether the intended meeting with the right parties is successfully scheduled or not. However, the requirements-level concern here is the extent to which the guest is permitted to access the shared information, such as files and chats. In fact, inviting the guest to a meeting series or an individual meeting may result in a pass if the test oracle (i.e., expected outcome) hinges on scheduling confirmation. A gap exists between observing a meeting's scheduling confirmation and guarding the guest's permissible access to the sharables.

To further illustrate this gap, let us also revisit the iTrust's bug which allowed a patient to schedule appointments with multiple doctors at the same time [15]. From the patient's perspective, the requirement $\mathcal{R}$ might be: "to consult with the family doctor and other epidemiology experts about the holiday travel under the wide and quick spread of Omicron". Note that this $\mathcal{R}$ is optative and refers to only the private phenomena of the environment. The $\mathcal{R}$ is fulfilled if the consultation actually happens and the patient receives proper travel advice. Therefore, even though iTrust's buggy implementation allowed multiple doctor's appointments to be made, the patient's $\mathcal{R}$ could still be satisfied, e.g., by consulting with the multiple doctors over a Teams's virtual session. In this example, iTrust's appointment-scheduling feature exhibits a non-negligible distance to the validation of $\mathcal{R}$. The buggy implementation of iTrust, according to [15], turns out to be capable of meeting the patient's needs.

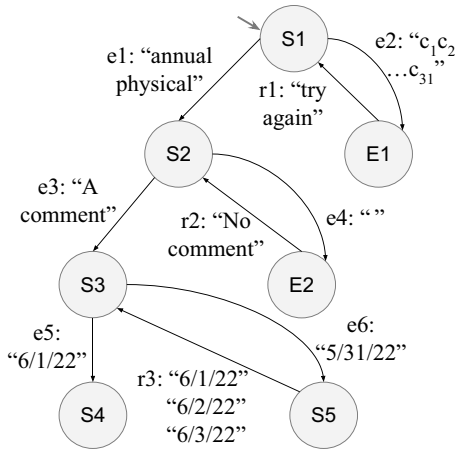


Fig. 10 Modeling iTrust's Schedule Appointments use case (cf. Fig. 3) with a finite state machine in which the additional states "S4" and "S5" represent "request is saved" and "time conflict is detected" respectively. The transition is labeled with "e" (user-entered input) or "r" (system-generated response)

In summary, there exist a couple of challenges of interpreting bug based on the failure of requirements fulfillment: (1) choosing the right feature to test and (2) addressing the gap between software testing and requirements validation. Before presenting our approach to tackling these challenges in Sect. 6.3, we review the existing way of using state machines to design acceptance test cases.

6.2 Acceptance test cases

Hsia et al. [74] were among the first to link acceptance testing and requirements engineering. They introduced scenario analysis in which requirements analysts manually constructed a finite state machine for the user or customer view. For illustrative purposes, we manually built a finite state machine model for iTrust's Schedule Appointments use case (cf. Fig. 3), and our model is shown in Fig. 10.

The finite state machine formalism helps to systematically generate acceptance test cases. From Fig. 10, for instance, we could derive the test cases, such as $[e1]$, $[e2]$, and $[e1 \circ e3]$. The formalism encapsulates the *logical* test case design advocated by Skoković and Skoković [43]. The literature also distinguishes positive test cases from negative ones, e.g., $[e1]$ over Fig. 10 tests the positive outcome of adding a new appointment type, whereas $[e2]$ tests the negative outcome. Having both positive and negative tests is

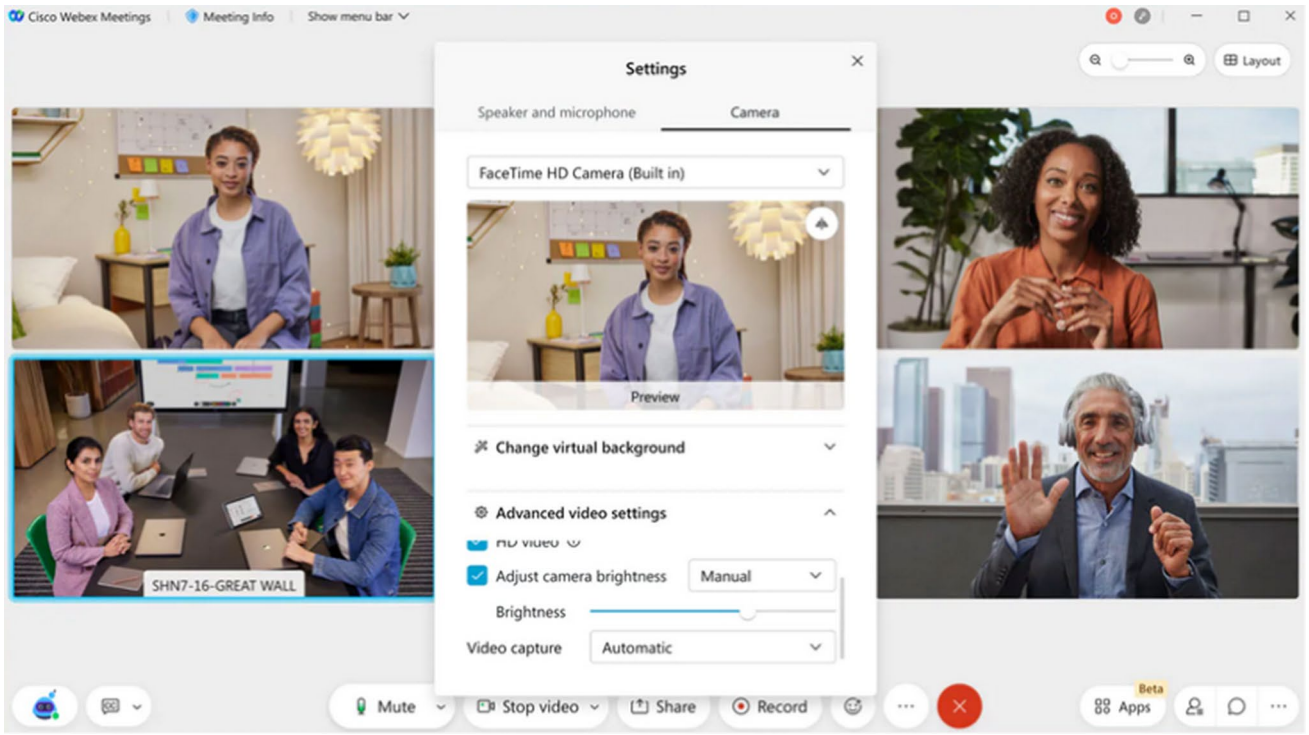


Fig. 11 Adjusting camera brightness feature of Webex [76]

important for acceptance testing [42, 75]. However, we argue that both types are about what is desired from the implemented software: desired to confirm a success or desired to throw an exception. Such a desire is what the optative $\mathcal{S}$ shows in Fig. 4 and is defined only *within* the machine boundary. Next, we explore a new way to inject $\mathcal{E}$ explicitly into a feature's acceptance testing.

6.3 Testing software's changing features

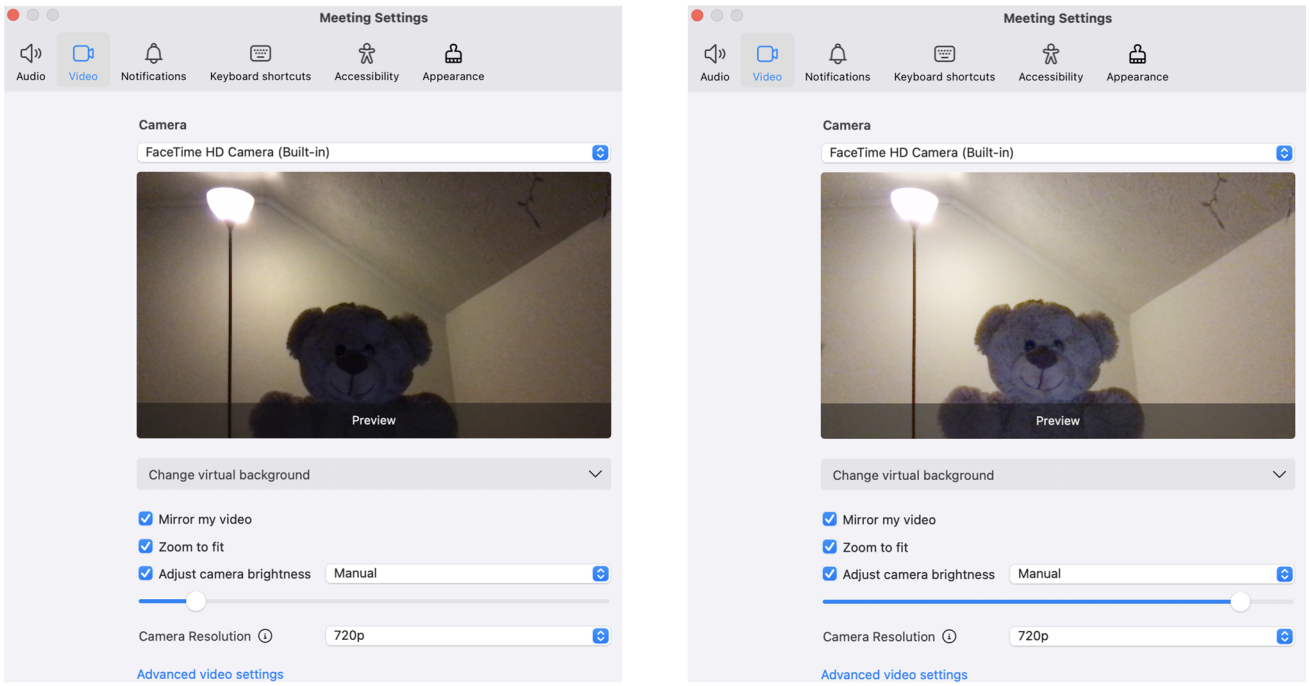
In a rapidly evolving technological landscape, many software vendors seek to test and launch their products and services continuously. For example, Cisco released 27, 16, 22, and 25 Webex features in August, September, October, and November of 2021, respectively [76]. One of August's changing features is to adjust camera brightness. Figure 11 shows Cisco's description of this feature, from which the texts introduce the functionalities (i.e., automatic and manual adjustments), how to invoke them in the software (e.g., checking the box, a slider control, etc.), and the constraints (e.g., the software feature's dependency on Hardware Acceleration for video). The pictorial part of Fig. 11, which is only optional to feature description, helps illustrate the invocation, operation, and expected behavior of the functionalities.

Our current work focuses on testing the changing features like Webex's adjusting camera brightness. This is because not only that a feature represents the increment of functionality and hence a coherent $\mathcal{S}$ of the software, but also that the new code associated with the changing feature tends to have more bugs and be less well tested for edge cases [77]. Performing the RBT on the changing feature, rather than on the software as a whole, addresses the challenge of choosing the right feature to test discussed in Sect. 6.1. To address the other challenge of testing's distance from validating $\mathcal{R}$, we define the oracle as follows:

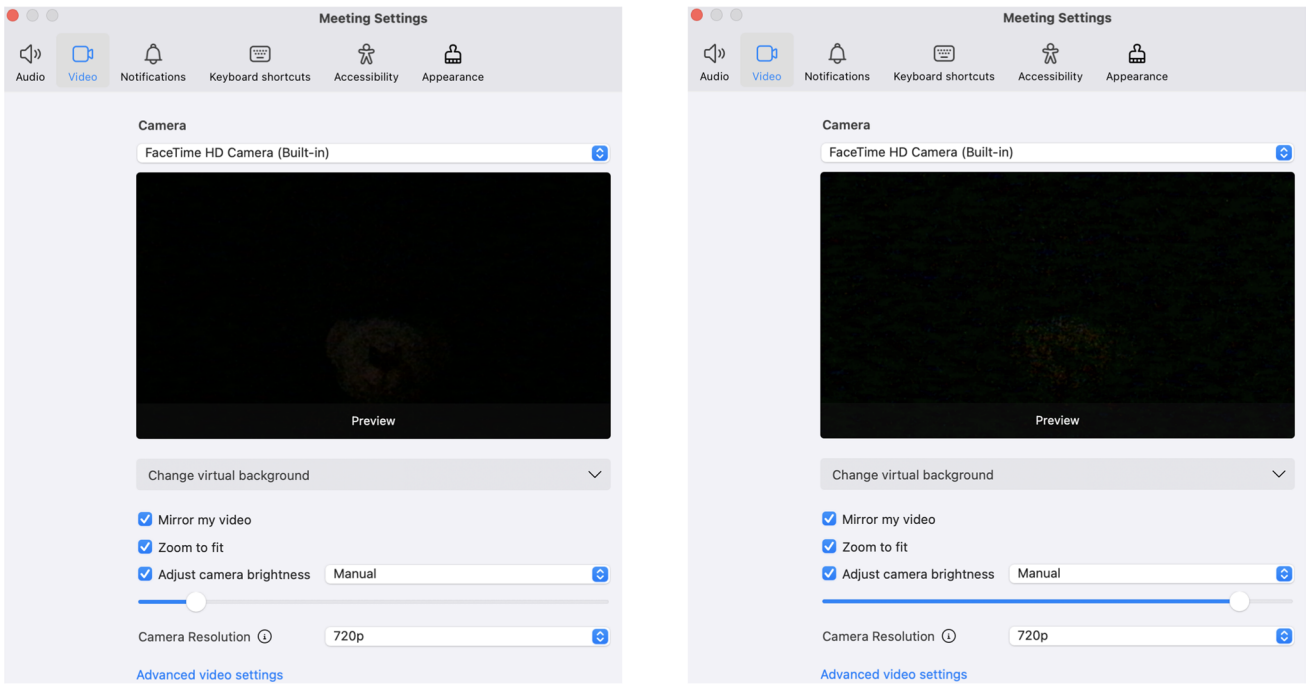
$$\mathcal{E}, \text{feature} \neq \neg\,\mathcal{E}, \text{feature} \tag{7}$$

in which $(\mathcal{E}, \neg\,\mathcal{E})$ consists of a pair of opposing environmental conditions. Compared to Eq. (4), this formulation eliminates the need for $\mathcal{R}$ and also pins down the unit of testing to a specific feature. Intuitively, Eq. (7) asserts that the same feature would behave differently when tested in an opposing pair of conditions, i.e., we expect the behavior of the feature observed in $\mathcal{E}$ to be different from the behavior of the same feature observed in $\neg\,\mathcal{E}$. Note that the test oracle of Eq. (7) always compares *two* executions of a particular feature, instead of expressing the anticipated outcome of that feature in a solo condition.

To better understand the feature testing approach characterized by Eq. (7), we consider Webex's adjusting camera brightness feature introduced in Fig. 11 and identify a relevant $(\mathcal{E}, \neg\,\mathcal{E})$ pair to be ("the user is in a bright room", "the user is in a dark room"). Both $\mathcal{E}$ and $\neg\,\mathcal{E}$ refer to the



(a) Adjusting the camera brightness from 10% (left) to 90% (right) in $\mathcal{E}$="the user is in a bright room"



(b) Adjusting the camera brightness from 10% (left) to 90% (right) in $\neg\mathcal{E}$ ="the user is in a dark room"

Fig. 12 Results of testing Webex's adjusting camera brightness feature

indicative, private phenomena of the environment [11]; however, they represent the opposing ends of a locational facet of the user of Webex, or more specifically, the user of the "adjusting camera brightness" feature. Following Eq. (7), we expect ("the user is in a bright room", "adjusting camera brightness") to behave differently from ("the user is in a dark room", "adjusting camera brightness"), which serves as the oracle of our RBT. Carrying out the actual testing with the guidance of this oracle is straightforward. Figure 12 shows the testing results. From the standpoint of seeing the

Table 5 Antonyms of the top-10 abstractions of Webex's *side-by-side mode for dual camera support*

Rank	Abstraction	Antonyms of key	Antonyms of value
1	<"side", "human side">	"bottom", "top"	"nonhuman", "bottom", "top"
2	<"side", "short side">	"bottom", "top"	"retentive", "long", "tall", "bottom", "top"
3	<"side", "left side">	"bottom", "top"	"center", "right", "bottom", "top"
4	<"side", "right side">	"bottom", "top"	"wrong", "center", "left", "incorrect", "bottom", "top"
5	<"video camera", "professional camera">	−−	"unprofessional", "nonprofessional"
6	<"video camera", "digital camera">	−−	"analog"
7	<"effect", "back side">	−−	"forward", "front", "bottom", "top"
8	<"hand", "hand camera">	−−	−−
9	<"video", "quality video at 2">	−−	−−
10	<"video interpreting", "live video">	−−	"dead", "recorded"

user more clearly, we could conclude that "adjusting camera brightness' in $\mathcal{E}$ (cf. Fig.12a) has a different effect compared to observing the same feature's outcome in $\neg\,\mathcal{E}$ (cf. Fig. 12b). The testing results thus support the oracle of Eq. (7).

What this example shows can be regarded as a confirmed boundary condition, which, if known before the feature's deployment, could better inform the end users about their expectations of the newly released functionalities of the software. For instance, an additional constraint might be inserted into the feature description of Fig. 11 to state that the adjusting camera brightness feature does not work well in a dark environment. Furthermore, making the feature work well in $\neg\,\mathcal{E}$ can be added to the backlog as a new requirement, allowing the developers to improve the capabilities of the software and to make innovations in their product. The software failure here—its inability to display the user more clearly—is uncovered *unexpectedly* with respect to the development team and yet purposefully by our approach. Hence, our environment-driven feature testing is capable of revealing bugs that drive continued innovation.

6.4 Quantitative analysis

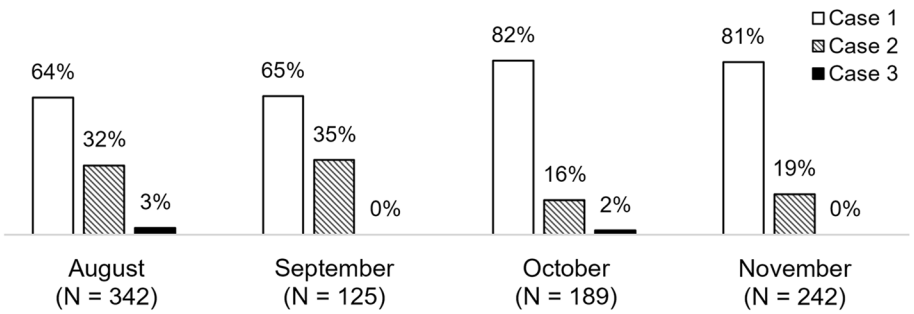
To provide automated support for the feature testing approach of Sect. 6.3, we follow the process of Fig. 5 to generate a ranked list of abstractions in the form of <key, value> pairs. The abstraction ranking is performed by calculating the cosine similarity with TF-IDF weighting (cf. Eq. 5) between each abstraction candidate and a given feature's NL description. Our current NLP support does not consider the pictorial part but only the NL part of a feature's description (cf. Fig. 11). As our feature testing operates on $\mathcal{E}$ and $\neg\,\mathcal{E}$ according to Eq. (7), we treat the identified abstraction as $\mathcal{E}$, and add a new NLP method to automatically search for $\neg\,\mathcal{E}$ of a given $\mathcal{E}$. Our implementation makes use of WordNet's antonyms() function as part of Python's NLTK package [78]. This function returns a set of antonyms for a single English word if they exist in WordNet's lexical database. Previous work leveraged this function for various purposes, e.g., Lilian et al. [79] identified antonyms for the adjective in a word embedding model, and Killawala et al. [80] produced all possible antonyms and synonyms for the given word to develop a framework to automate the process of quiz and exam question generation.

Inspired by the previous work, we compute the antonyms for key and for value separately from the abstractions that we identified in January 2022. Table 5 illustrates the resulting antonyms of the top-10 abstractions for Webex's side-by-side mode for dual camera support feature released by Cisco in August 2021 [76]. The NL description of this feature is: *"Users can share live content through the back camera of their device while remaining a video participant through the front camera of their device. When users use Side-by-Side mode for dual camera, their front and back camera video is combined to one video side by side. The other meeting participants see the participant video that placed the user's front video and back video side by side at the same time."* As shown in Table 5, more antonyms are found for values than for keys; this trend generally holds in our experimentation of Webex's new and changing features. For each identified antonym, we automatically form $\neg\,\mathcal{E}$ in the <key, value>

Table 6 Characteristics of Webex features released over a four-month period from Aug 2021 to Nov 2021, and $\neg\mathcal{E}$ formation summaries

		Aug	Sept	Oct	Nov	Average
Feature characteristics	# of features released by Cisco	27	16	22	25	22.5
	# (%) of experimented features	14 (52%)	8 (50%)	10 (45%)	12 (48%)	11 (49%)
	average # of sentences per feature	7.3	10.9	8.6	7.9	8.4
	# (%) of features with pictorials	8 (57%)	4 (50%)	7 (70%)	5 (42%)	6 (55%)
	# of features with key's antonyms	3	1	3	2	2.3
$\neg\mathcal{E}$ formation summaries	# of features with value's antonyms	14	8	10	12	11
	average # of antonyms per key	2.0	1.0	1.6	1.8	1.8
	average # of antonyms per value	2.2	1.8	2.0	1.8	2.0
	total # of $\neg\mathcal{E}$ formed as <key, value> pairs	342	125	189	242	224.5

Fig. 13 Feature testing results



pair. For example, only one $\neg\mathcal{E}$ exists for the 6th pair in Table 5, which is <"video camera", "analog camera">. Note that our $\neg\mathcal{E}$ construction replaces key or value, but not both simultaneously. The 1st abstraction of Table 5, thus, has five $\neg\mathcal{E}$ pairs: <"bottom", "human side">, <"top", "human side">, <"side", "nonhuman side">, <"side", "human bottom">, and <"side", "human top">. Having antonyms on both key and value, such as <"bottom", "nonhuman side">, deviates too much from $\mathcal{E}$ and is therefore not considered as a $\neg\mathcal{E}$ pair.

We performed a quantitative study on the recently released Webex features. Table 6 lists some characteristics of the features, as well as some summary statistics of the $\neg\mathcal{E}$ formation results. Altogether, Cisco released an average of 22.5 Webex features over the period of four months from Aug 2021 to Nov 2021; however, we experimented about a half of those features as a result of our manual feasibility analysis. In particular, we excluded features related to hardware upgrade (e.g., *support available for Apple iOS 15* as part of Oct 2021's release) and software integration (e.g., *Microsoft Teams integration: starting and scheduling meetings from message extension* as part of Sept 2021's release), because our acceptance testing currently focuses on a specific functional aspect of one software system, i.e., Webex. As shown in Table 6, our feature testing approach is applicable to 49% of the new and changing features, among which 55% included pictorials in their descriptions. Considering all the 44 features that we experimented with, the NL description parts contain an average of 8.4 sentences per feature, indicating the lightweight documentation of the external release notes. While our approach takes into account only the NL parts of the features descriptions, processing the pictorials in an automatic way seems to be an interesting direction for future research.

The bottom of Table 6 shows the results of antonym searching for the *top-20 ranked* abstractions. This decision was informed by the accuracy results presented in Fig. 9. As mentioned earlier, less antonyms were found for keys than for values. In fact, all the features in our experiment resulted in $\neg\mathcal{E}$ based on the abstractions' values. In terms of the number of antonyms found, each key returned 1.8 and each value returned 2.0 on average. As listed in Table 6, the total number of <key, value> pairs representing $\neg\mathcal{E}$ is 342, 125, 189, and 242 for Aug, Sept, Oct, and Nov, respectively. Each $\neg\mathcal{E}$, along with the corresponding $\mathcal{E}$, serves as a unit of analysis for our feature testing. We classify the results as follows:

– **Case 1: untestable** means that the formed $\neg\mathcal{E}$ is not viable to be a test oracle, and hence no feature testing could be performed. For example, the $\neg\mathcal{E}$ of <"video interpreting", "dead video"> formed on the basis of the 10th abstraction shown in Table 5 is untestable. Case 1 thus offers little value to the software developers or testers.

- **Case 2: boundary condition confirmed** suggests that not only is $\neg\mathcal{E}$ testable, but also the test oracle of Eq. (7) is supported by the actual feature testing. For instance, the 3rd abstraction identified for Nov's *support Q &A and chat during a practice session in Webex Events* feature is $\mathcal{E}$=<"chat", "online chat">. Running antonyms() on the value of this $\mathcal{E}$ returns $\neg\mathcal{E}$=<"chat", "offline chat">. Our feature testing did confirm that $\mathcal{E}$, feature $\neq \neg\mathcal{E}$, feature, since an offline chat was disabled to be sent in Webex. Therefore, Case 2 helps to test the feature fails *as expected*, increasing the confidence toward the decision of releasing the feature.
- **Case 3: unexpectedly failed**, such as the testing results shown in Fig. 12, provides valuable information for the software development team to catch buggy implementations, improve the release notes, and create potentially innovative features as new requirements.

Figure 13 shows our feature testing results in which the number of analysis units driven by $\neg\mathcal{E}$ is annotated under each month. Overall, our current use of antonyms() has 64–82% untestable results, making the increase in precision an important target for future work. As antonyms() works only for a single word, building a mechanism to handle phrases could potentially reduce the untestable $\neg\mathcal{E}$. The boundary condition confirmed case ranges from 16 to 35% in Fig. 13. If we consider the testable $\neg\mathcal{E}$'s, then a dominant proportion have led to the testing results confirming the oracle. On one hand, such findings indicate the viability of our test oracle formulated in Eq. (7). On the other hand, the Webex features released by Cisco are indeed of high quality when tested both positively and negatively in different environments. Finally, unexpectedly failed cases appear in a few occasions, demonstrating the additional value offered by our feature testing approach.

6.5 Discussions

Our feature testing approach is motivated by the couple of practical challenges presented in Sect. 6.1. Because choosing the right feature from an entire software system to test is difficult, we will not choose but instead test each new and changing feature to be released. Because reasoning about requirements satisfaction by only observing software testing results is difficult, we will perform testing according to a well-defined oracle given in Eq. (7).

Such an oracle takes advantage of *metamorphic testing*, which is a property-based software testing technique that has been successfully applied to scientific software systems, search engines, AI-based applications, among others [81, 82]. In essence, metamorphic testing relies on some domain properties, such as sine(x)=sine(π---x), to test a software's implementation. In many occasions, one may not know the exact output before testing the software with a single test case, e.g., the exact value of sine(x) for an arbitrary x might be unknown due to the implementation's numerical decisions, the exact search results and their correct ranking of an arbitrary query may be unknown due to subjectivity and the lack of knowledge about the user's search intent, etc. Executing a software's implementation with *multiple* test cases and then checking whether a desired property would hold is what metamorphic testing embodies. Our work extends the body of work on metamorphic testing by asserting the software property relative to a coupled dual environmental conditions. This new way of focusing on requirements-level features to structure metamorphic testing addresses a critical challenge of metamorphic relation construction [81, 82].

The implications of our work also relate to A/B testing, which can be regarded as a type of controlled experiment that compares two variants: A (a control) and B (a treatment) [77]. Modern tech companies including Amazon, Google, LinkedIn, and Microsoft use A/B testing as a way to align RE and software testing, as designing and running a control experiment with actual end users could inform whether a deployed feature, e.g., integrating Bing's search results with social media, influences the overall evaluation criterion. Auer et al. [83] reported that the least common type of treatment encountered in the A/B testing literature was new features, which our work addresses. Our feature testing approach is complementary to A/B testing. While A/B testing constructs two different conditions: one with a new feature and the other without it, our work can be used to better design the conditions of the treatment, e.g., under which environmental circumstances the feature is really experimented and what other factors need to be controlled or interpreted with caution. The A/B testing results could also inform the $\mathcal{E}$ and $\neg\mathcal{E}$ design of our feature testing approach.

As for the connection between abstraction identification and ontology engineering, early-RE work such as Sawyer et al. [2] used the identified abstractions as candidate class names in UML modeling. In contrast, we focus on identifying key-value pairs to assist requirements-based testing and, hence, favor attributes such as testability and bug revealingness.

7 Conclusions

Automatically finding abstractions that are of particular significance in a given domain has attracted much attention in RE, though the primary focus has been on supporting early-RE activities such as requirements elicitation and modeling [1–4]. In this paper, we have presented an automated approach built on five novel NLP patterns to identifying abstractions in the form of <key, value> pairs. We regard

such a form to be testable and also select relevant Wikipedia pages as an indicative corpus. Evaluating our approach with six software applications in two different domains shows that the <key, value> pairs are more accurate than the abstraction candidates generated by contemporary techniques. Initial findings also indicate our abstractions' capabilities of revealing bugs and matching the environmental assumptions created manually.

Building on the identified abstractions, we introduce a new way of performing feature testing to overcome a couple of practical challenges: choosing the right feature to test and bridging the reasoning about requirements satisfaction from observing software executions. We extend the automated support with WordNet's antonyms() function to help construct an opposing environmental condition for an abstraction's key and value. Analyzing the recently released Webex features shows the viability of the test oracle underlying our feature testing approach and illustrates the potential innovation that the unexpectedly failed cases could provoke to the software development project.

Our work can be extended toward several avenues. Empirical studies, including theoretical replications [84, 85], with more software systems and more application domains are needed to lend strength to the findings reported here. Moreover, grouping related abstractions can be explored for uncovering more bugs. Finally, developing a mechanism to find antonyms for a phrase and even a hash structure, rather than only for a single English word, might improve the precision and the utility of our feature testing approach.

Acknowledgements We thank Sarah Sturmer and Sreelekhaa Nagamalli Santhoshkumar from the University of Cincinnati for their preliminary work on related research topics and their insightful comments on this work. The research is partially supported by the National Natural Science Foundation of China under Grant No. 62192731, 61802374, 62002348, and 62072442, the National Key Research and Development Program of China under Grant No. 2018YFB1403400, and Youth Innovation Promotion Association CAS.

Data availability The datasets generated during and/or analyzed during the current study are available in the Zenodo repository, https://doi.org/10.5281/zenodo.5858384.

Declaration

Conflicts of interest The authors have no conflicts of interest to disclose.

References

1. Gacitua R, Sawyer P, Gervasi V (2011) Relevance-based abstraction identification: technique and evaluation. Requir Eng 16(3):251–265
2. Sawyer P, Rayson P, Cosh K (2005) Shallow knowledge as an aid to deep understanding in early phase requirements engineering. IEEE Trans Softw Eng 31(11):969–981
3. Gacitua R, Sawyer P, Gervasi V (2010) On the effectiveness of abstraction identification in requirements engineering. In: Proceedings of the international requirements engineering conference (RE), Sydney, Australia, September–October 2010, pp 5–14
4. Goldin L, Berry DM (1997) AbstFinder, a prototype natural language text abstraction finder for use in requirements elicitation. Autom Softw Eng 4(4):375–412
5. Dwarakanath A, Ramnani RR, Sengupta S (2013) Automatic extraction of glossary terms from natural language requirements. In: Proceedings of the international requirements engineering conference (RE), Rio de Janeiro, Brazil, July 2013, pp 314–319
6. Arora C, Sabetzadeh M, Briand LC, Zimmer F (2017) Automated extraction and clustering of requirements glossary terms. IEEE Trans Softw Eng 43(10):918–945
7. Gemkow T, Conzelmann M, Hartig K, Vogelsang A (2018) Automatic glossary term extraction from large-scale requirements specifications. In: Proceedings of the international requirements engineering conference (RE), Banff, Canada, August 2018, pp 412–417
8. Yu E (1997) Towards modeling and reasoning support for early-phase requirements engineering. In: Proceedings of the international symposium on requirements engineering (RE), Annapolis, MD, USA, January 1997, pp 226–235
9. Ryan K (1993) The role of natural language in requirements engineering. In: Proceedings of the international symposium on requirements engineering (RE), San Diego, CA, USA, January 1993, pp 240–242
10. Jin Z (2018) Environment modeling-based requirements engineering for software intensive systems. Morgan Kaufmann
11. Jackson M (1997) The meaning of requirements. Ann Softw Eng 3:5–21
12. van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley
13. Tun TT, Lutz RR, Nakayama B, Yu Y, Mathur D, Nuseibeh B (2015) The role of environmental assumptions in failures of DNA nanosystems. In: Proceedings of the international workshop on complex faults and failures in large software systems (COUFLESS), Florence, Italy, May 2015, pp 27–33
14. Knight JC (2002) Safety critical systems: challenges and directions. In: Proceedings of international conference on software engineering (ICSE), Orlando, Florida, USA, May 2002, pp 547–550
15. Bhowmik T, Chekuri SR, Do AQ, Wang W, Niu N (2019) The role of environment assertions in requirements-based testing. In: Proceedings of the international requirements engineering conference (RE), Jeju Island, South Korea, September 2019, pp 75–85
16. Peng Z, Rathod P, Niu N, Bhowmik T, Liu H, Shi L, Jin Z (2021) Environment-driven abstraction identification for requirements-based testing. In: Proceedings of the international requirements engineering conference (RE), Notre Dame, IN, USA, September 2021, pp 245–256
17. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: a survey. IEEE Trans Softw Eng 41(5):507–525
18. Niu N, Mahmoud A (2012) Enhancing candidate link generation for requirements tracing: the cluster hypothesis revisited. In: Proceedings of the international requirements engineering conference (RE), Chicago, IL, USA, September 2012, pp 81–90
19. Wang W, Gupta A, Niu N, Xu LD, Cheng J-RC, Niu Z (2018) Automatically tracing dependability requirements via term-based relevance feedback. IEEE Trans Ind Inf 14(1):342–349
20. Wang W, Niu N, Liu H, Niu Z (2018) Enhancing automated requirements traceability by resolving polysemy. In: Proceedings

of the international requirements engineering conference (RE), Banff, Canada, August 2018, pp 40–51
21. Wermter J, Hahn U (2005) Finding new terminology in very large corpora. In: Proceedings of the international conference on knowledge capture (K-CAP), Banff, Canada, October 2005, pp 137–144
22. Leveson NG (1995) Safeware: system safety and computers. Addison-Wesley
23. Hull E, Jackson K, Dick J (2010) Requirements engineering. Springer
24. Rahimi M, Xiong W, Cleland-Huang J, Lutz RR (2017) Diagnosing assumption problems in safety-critical products. In: Proceedings of the international conference on automated software engineering (ASE), Urbana, IL, USA, October–November 2017, pp 473–484
25. Alenazi M, Niu N, Savolainen J (2020) A novel approach to tracing safety requirements and state-based design models. In: Proceedings of international conference on software engineering (ICSE), Seoul, South Korea, June–July 2020, pp 848–860
26. Alrajeh D, Cailliau A, van Lamsweerde A (2020) Adapting requirements models to varying environments. In: Proceedings of international conference on software engineering (ICSE), Seoul, South Korea, June–July 2020, pp 50–61
27. Yang C, Liang P, Avgeriou P (2018) Assumptions and their management in software development: a systematic mapping study. Inform Softw Technol 94:82–110
28. Garlan D, Allen R, Ockerbloom J (2009) Architectural mismatch: why reuse is still so hard. IEEE Softw 26(4):66–69
29. Jin X, Khatwani C, Niu N, Wagner M, Savolainen J (2016) Pragmatic software reuse in bioinformatics: how can social network information help? In: Proceedings of international conference on software reuse (ICSR), Limassol, Cyprus, June 2016, pp 247–264
30. Bhuta J, Boehm B (2007) A framework for identification and resolution of interoperability mismatches in COTS-based systems. In: Proceedings of the international workshop on incorporating cots software into software systems: tools and techniques (IWICSS), Minneapolis, MN, USA, May 2007
31. Bazaz A, Arthur JD, Tront JG (2006) Modeling security vulnerabilities: a constraints and assumptions perspective. In: Proceedings of the international symposium on dependable, autonomic and secure computing (DASC), Indianapolis, IN, USA, September–October 2006, pp 95–102
32. Wang W, Dumont F, Niu N, Horton G (2020) Detecting software security vulnerabilities via requirements dependency analysis. IEEE Trans Softw Eng 48(5):1665–1675
33. Kroll P, Kruchten P (2003) The rational unified process made easy: a practitioner's guide to the RUP. Addison-Wesley
34. Laplante PA (2007) What every engineer should know about software engineering. CRC Press
35. Unterkalmsteiner M, Gorschek T, Feldt R, Klotins E (2015) Assessing requirements engineering and software test alignment—five case studies. J Syst Softw 109:62–77
36. Uusitalo EJ, Komssi M, Kauppinen M, Davis AM (2008) Linking requirements and testing in practice. In: Proceedings of the international requirements engineering conference (RE), Barcelona, Spain, September 2008, pp 265–270
37. Flammini F, Mazzocca N, Orazzo A (2009) Automatic instantiation of abstract tests on specific configurations for large critical control systems. Softw Test Verif Reliab 19(2):91–110
38. Miller T, Strooper PA (2012) A case study in model-based testing of specifications and implementations. Softw Test Verif Reliab 21(1):33–63
39. de Santiago Júnior VA, Vijaykumar NL (2012) Generating model-based test cases from natural language requirements for space application software. Softw Test Verif Reliab 20(1):77–143
40. Siegl S, Hielscher K-S, German R (2010) Model based requirements analysis and testing of automotive systems with timed usage models. In: Proceedings of the international requirements engineering conference (RE), Sydney, Australia, September–October 2010, pp 345–350
41. Garousi V, Bauer S, Felderer M (2020) NLP-assisted software testing: a systematic mapping of the literature. Inform Softw Technol 126:106 321:1–106 321:20
42. Fischbach J, Vogelsang A, Spies D, Wehrle A, Junker M, Freudenstein D (2020) SPECMATE: automated creation of test cases from acceptance criteria. In: Proceedings of the international conference on software testing, validation and verification (ICST), Porto, Portugal, October 2020, pp 321–331
43. Skoković P, Rakić-Skoković M (2010) Requirements-based testing process in practice. Int J Ind Eng Manag 1(4):155–161
44. Meneely A, Smith B, Williams L (2012) iTrust electronic health care system: a case study. In: Cleland-Huang J, Gotel O, Zisman A (eds) Software and systems traceability. Springer
45. Niu N, Brinkkemper S, Franch X, Partanen J, Savolainen J (2018) Requirements engineering and continuous deployment. IEEE Softw 35(2):86–90
46. Zave P, Jackson M (1997) Four dark corners of requirements engineering. ACM Trans Softw Eng Methodol 6(1):1–30
47. Gunter CA, Gunter EL, Jackson M, Zave P (2000) A reference model for requirements and specifications. IEEE Softw 17(3):37–43
48. Mahmoud A, Niu N (2015) On the role of semantics in automated requirements tracing. Requir Eng 20(3):281–300
49. Ezzini S, Abualhaija S, Arora C, Sabetzadeh M, Briand LC (2021) Using domain-specific corpora for improved handling of ambiguity in requirements. In: Proceedings of international conference on software engineering (ICSE), Madrid, Spain, May 2021, pp 1485–1497
50. Chernov S, Iofciu T, Nejdl W, Zhou X (2006) Extracting semantics relationships between wikipedia categories. In: Proceedings of the workshop on semantic Wikis (SemWiki), Budva, Montenegro
51. Beautiful S. A Python library for pulling data out of HTML and XML Files. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/
52. Abualhaija S, Arora C, Sabetzadeh M, Briand LC, Vaz E (2019) A machine learning-based approach for demarcating requirements in textual specifications. In: Proceedings of the international requirements engineering conference (RE), Jeju Island, South Korea, September 2019, pp 51–62
53. Lin X, Peng Z, Niu N, Wang W, Liu H (2021) Finding metamorphic relations for scientific software. In: Proceedings of international conference on software engineering (ICSE) companion volume, Madrid, Spain, May 2021, pp 254–255
54. spaCy. Industrial-strength natural language processing in Python. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://spacy.io/
55. Kübler S, McDonald R, Nivre J (2009) Dependency parsing. Morgan & Claypool Publishers
56. Dalpiaz F, Dell'Anna D, Aydemir FB, cCevikol S (2019) Requirements classification with interpretable machine learning and dependency parsing. In: Proceedings of the international requirements engineering conference (RE), Jeju Island, South Korea, September 2019, pp 142–152
57. Cohen KB, Christiansen T, Hunter LE (2011) Parenthetically speaking: classifying the contents of parentheses for text mining. In: Proceedings of the annual symposium on biomedical and health informatics (AMIA), Washington, DC, USA, October 2011, pp 267–272
58. Klaussner C, Zhekova D (2011) Pattern-based ontology construction from selected Wikipedia pages. In: Proceedings of the

international conference on recent advances in natural language processing (RANLP) Student Research Workshop, Hissar, Bulgaria, September 2011, pp 103–108
59. Hearst MA (1992) Automatic acquisition of hyponyms from large text corpora. In: Proceedings of the international conference on computational linguistics (COLING), Nantes, France, August 1992, pp 539–545
60. Klaussner C, Zhekova D (2011) Lexico-syntactic patterns for automatic ontology building. In: Proceedings of the international conference on recent advances in natural language processing (RANLP) Student Research Workshop, Hissar, Bulgaria, September 2011, pp 109–114
61. Finin TW (1980) The semantic interpretation of nominal compounds. In: Proceedings of the annual national conference on artificial intelligence (AAAI), Stanford, CA, USA, August 1980, pp 310–312
62. Peng Z, Niu N (2021) Co-AI: a Colab-based tool for abstraction identification. In: Proceedings of the international requirements engineering conference (RE), Notre Dame, IN, USA, September 2021, pp 420–421
63. Liu H, Shen M, Zhu J, Niu N, Li G, Zhang L (2022) Deep learning based program generation from requirements text: are we there yet? IEEE Trans Softw Eng 48(4):1268–1289
64. Nyamawe AS, Liu H, Niu N, Umer Q, Niu Z (2019) Automated recommendation of software refactorings based on feature requests. In: Proceedings of the international requirements engineering conference (RE), Jeju Island, South Korea, September 2019, pp 187–198
65. Nyamawe AS, Liu H, Niu N, Umer Q, Niu Z (2018) Recommending refactoring solutions based on traceability and code metrics. IEEE Access 6:49–475
66. Nyamawe AS, Liu H, Niu N, Umer Q, Niu Z (2020) Feature requests-based recommendation of software refactorings. Empir Softw Eng 25(5):4315–4347
67. Niu N, Savolainen J, Bhowmik T, Mahmoud A, Reddivari S (2012) A framework for examining topical locality in object-oriented software. In: Proceedings of the annual IEEE computer software and applications conference (COMPSAC), Izmir, Turkey, July 2012, pp 219–224
68. OpenEMR. A medical practice management software system supporting electronic medical records (EMR). Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://en.wikipedia.org/wiki/OpenEMR
69. OpenMRS. A collaborative open-source project on medical record systems (MRS). Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://en.wikipedia.org/wiki/OpenMRS
70. OpenEMR Features. Features of OpenEMR. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://www.open-emr.org/wiki/index.php/OpenEMR_Features
71. OpenMRS User Guide. A complete user guide for OpenMRS. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://wiki.openmrs.org/display/docs/User+Guide
72. Maier D (1978) The complexity of some problems on subsequences and supersequences. J ACM 25(2):322–336
73. Lin X, Simon M, Peng Z, Niu N (2021) Discovering metamorphic relations for scientific software from user forums. Comput Sci Eng 23(2):65–72
74. Hsia P, Kung DC, Sell C (1997) Software requirements and acceptance testing. Ann Softw Eng 3:291–317
75. Haugset B, Hanssen GK (2008) Automated acceptance testing: a literature review and an industrial case study. In: Proceedings of the agile development conference (AGILE), Toronto, ON, Canada, August 2008, pp 27–38
76. Cisco. What's new for the latest channel of Webex meetings. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://help.webex.com/en-US/article/xcwws1/What's-New-for-the-Latest-Channel-of-Webex-Meetings
77. Kohavi R, Tang D, Xu Y (2020) Trustworthy online controlled experiments: a practical guide to A/B testing. Cambridge University Press
78. NLTK Project. Natural language toolkit. Last accessed on 2022/09/09 16:36:31. [Online]. Available: https://www.nltk.org/
79. Lilian JF, Sundarakantham K, Rajashree H, Shalinie SM (2019) SSE: semantic sentence embedding for learning user interactions. In: Proceedings of the international conference on computing, communication and networking technologies (ICCCNT), Kanpur, India, July 2019, pp 1–5
80. Killawala A, Khokhlov I, Reznik L (2018) Computational intelligence framework for automatic quiz question generation. In: Proceedings of the international conference on fuzzy systems (FUZZ-IEEE), Rio de Janeiro, Brazil, July 2018, pp 1–8
81. Segura S, Fraser G, Sánchez AB, Cortés AR (2016) A survey on metamorphic testing. IEEE Trans Softw Eng 42(9):805–824
82. Chen TY, Kuo F-C, Liu H, Poon P-L, Towey D, Tse TH, Zhou ZQ (2018) Metamorphic testing: a review of challenges and opportunities. ACM Comput Surv 51(1):4:1-4:27
83. Auer F, Ros R, Kaltenbrunner L, Runeson P, Felderer M (2021) Controlled experimentation in continuous experimentation: knowledge and challenges. Inform Softw Technol 134:106 551:1–106 551:16
84. Niu N, Koshoffer A, Newman L, Khatwani C, Samarasinghe C, Savolainen J (2016) Advancing repeated research in requirements engineering: a theoretical replication of viewpoint merging. In: Proceedings of the international requirements engineering conference (RE), Beijing, China, September 2016, pp 186–195
85. Khatwani C, Jin X, Niu N, Koshoffer A, Newman L, Savolainen J (2017) Advancing viewpoint merging in requirements engineering: a theoretical replication and explanatory study. Requir Eng 22(3):317–338