Requirements Eng (2017) 22:167–190
DOI 10.1007/s00766-015-0239-x

ORIGINAL ARTICLE

Repetition between stakeholder (user) and system requirements

Richard Ellis-Braithwaite[1] · **Russell Lock**[1] · **Ray Dawson**[1] · **Tim King**[2]

Received: 15 July 2014 / Accepted: 9 September 2015 / Published online: 25 September 2015

Abstract Stakeholder requirements (also known as user requirements) are defined at an early stage of a software project to describe the problem(s) to be solved. At a later stage, abstract solutions to those problems are prescribed in system requirements. The quality of these requirements has long been linked to the quality of the software system and its development or procurement process. However, little is known about the quality defect of redundancy between these two sets of requirements. Previous literature is anecdotal rather than exploratory, and so this paper empirically investigates its occurrence and consequences with a case study from a UK defense contractor. We report on a survey of sixteen consultants to understand their perception of the problem, and on an analysis of real-world software requirements documents using natural language processing techniques. We found that three quarters of the consultants had seen repetition in at least half of their projects. Additionally, we found that on average, a third of the requirement pairs' (comprised of a system and its related stakeholder requirement) description fields were repeated such that one requirement in the pair added only trivial information. That is, solutions were described twice while their respective problems were not described, which ultimately lead to suboptimal decisions later in the development process, as well as reduced motivation to read the requirements set. Furthermore, the requirement fields considered to be secondary to the primary "description" field, such as the "rationale" or "fit criterion" fields, had considerably more repetition within UR–SysR pairs. Finally, given that the UR–SysR repetition phenomena received most of its discussion in the literature over a decade ago, it is interesting that the survey participants did not consider its occurrence to have declined since then. We provide recommendations on preventing the defect, and describe the freely available tool developed to automatically detect its occurrence and alleviate its consequences.

Keywords User requirements · Stakeholder requirements · System requirements · Duplicate detection · Redundancy

✉ Richard Ellis-Braithwaite
richard.ellis.braithwaite@googlemail.com

Russell Lock
r.lock@lboro.ac.uk

Ray Dawson
r.j.dawson@lboro.ac.uk

Tim King
tim.king@lsc.co.uk

[1] Computer Science, Loughborough University, Loughborough, Leicestershire, UK

[2] LSC Group, Lichfield, Staffordshire, UK

1 Introduction

Requirements documents written in natural language are usually the primary, and often sole means of communicating the desired capabilities of a software system to its developers or procurers [1]. Since requirements define the design problem to be solved [2], their quality has long been recognized as an important factor in the success of a software project. Researchers have recently investigated duplication *within* software requirements documents [3] in the context of minimizing the "redundancy" quality defect [4]. That is, where requirements engineers have not adhered to the "Don't Repeat Yourself" principle: "*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*" [5], i.e.,

within a project's requirements set. However, duplication between the different types of requirements documents—namely *stakeholder* (or *user*) and *system*—has yet to be investigated empirically. We propose that between-document redundancy is at least as concerning as within-document redundancy; for as well as inheriting many of the undesirable effects caused by cloning requirements or code [3, 6], engineering process-related problems may be indicated. For example, perhaps "flow down" from stakeholder needs to system capabilities could be improved [7].

Hence, the subject of this paper is a pattern of repetition that occurs between a project's requirements documents while following the generic "V" systems engineering model [8], i.e., where stakeholder requirements (problems) are transformed into system requirements (abstract solutions). In confusion over a plethora of conflicting requirements engineering (RE) terminology and guidelines [9], practitioners are sometimes led to write (or rather *re*-write) system requirements (SysR) that add insignificant information to their associated user requirements (UR), or vice versa. For example, the UR "the user shall be able to *write an email*", and the SysR "the system shall enable the user to *write an email*". From an information-theoretic viewpoint [10], these requirements could be considered duplicates, since the difference between them expresses information known about the whole set of SysRs. Such duplication renders traceability between the two statements an inefficient use of the reader's time, where it should have provided a source of rationale [11] and translation between "the voice of the customer" and "the voice of the engineer" [12]. Nevertheless, both documents (UR and SysR) must still be read by architects and developers to ensure pertinence, since it is their responsibility to "*find out the real story behind the requirements*" [13]. While doing so, reading the occasional repeated requirement statement may evoke feelings of déjà vu, but it is more concerning when a significant proportion are repeated, since:

1. Duplication between URs and SysRs indicates a misunderstanding of their role, and therefore potential omission of either the problem or solution description, indicating concerns about the SysR's "flow down";
2. Finding worthwhile non-duplicated requirements becomes akin to "finding a needle in the haystack", hence risking ignorance of important requirements (caused by readers losing the will to continue reading).

More than a decade ago, Kovitz [14] and Alexander [15] independently identified and discouraged the reportedly common practice of repeating the description between user requirements and their associated system requirements. Since this paper's authors still witness this practice in industry, the following research questions (RQs) were constructed:

RQ1 According to the literature, what are the differences supposed to be between a UR and a SysR?
RQ2 Does repetition between UR and SysR statements (still) occur, and if so, how often?
RQ3 Why does UR–SysR repetition occur, and what does it indicate about the project?
RQ4 Is UR–SysR repetition a problem, and if so, why?
RQ5 Can the problems caused by UR–SysR repetition be reduced in existing and future requirements sets?

In summary, this paper sets out to explore repetition between pairs of UR and SysR statements and to discuss whether it can be considered a problem in light of the pertinent RE literature. To provide more than anecdotal evidence in answering RQs 2–5, we present the results of a case study from a UK defense consultancy. We discuss a survey of sixteen software systems engineering consultants, and an analysis of UR–SysR requirement pairs from two independent software projects. While doing so, we describe the requirements traceability analysis tool (free to download) [16] developed:

- to enable the investigation of RQs 2 and 5;
- to support both producers and consumers of UR and SysR documentation in future projects;
- under the wider research aim of increasing the usability of requirements quality assessment.

2 Background and terminology

The following sections will introduce the concepts of requirements at the user, system, and software level (2.1 and 2.2), traceability between them (2.3), and similarity analysis (2.4).

2.1 Requirements

There is little consensus on the precise meaning of the terminology used in RE—especially on the term "requirement". Jureta et al. [9] recently summarized that it is "variously understood as (describing) a purpose, a need, a goal, a functionality, a constraint, a quality, a behavior, a service, a condition, or a capability". Fifteen years before that, Harwell et al. [17] asserted that "if it mandates that something must be accomplished, transformed, produced, or provided, it is a requirement—period". What both of these definitions share in common is that a requirement is either about a problem (i.e., a *purpose*, *need*, or *goal* to be *accomplished*) or about a solution (i.e., a *functionality*, *quality*, *behavior*, *service*, or *capability* to be *transformed*, *produced*, or *provided*). As Wieringa puts it, there are "two schools of thought" in RE [18], and confusion among practitioners about which "school" their requirements should adhere to [15] seems inevitable due to the semantic

overload of the "requirement" term. Idealistic and ambiguous guidelines, such as that requirements should describe *what* software must do rather than *how* it will do it, only complicate the issue [19], since as Kovitz [14] eloquently puts it, "*everything that a piece of software does is what it does, and everything that a piece of software does is how it does something*".

2.2 Stakeholder, system and software requirements

Qualifying adjectives, such as those in this section's heading, are often prepended to "requirement" to indicate which "school of thought" the requirement belongs. Popular requirements [11], systems [2, 20], and software [21] engineering textbooks, along with international systems and software requirements engineering standards [22–24] primarily refer to two such qualifiers: *stakeholder* and *system*. In Table 1, we distill the definitions for these types in terms of their domain (problem or solution?), role (for what purpose?), and language (whose vocabulary is it written in?), according to these sources. First, however, we describe the scope of these qualified requirement types in terms of their subtypes.

2.2.1 Stakeholder requirements subtypes

Given that a "stakeholder" is "an individual, group of people, organisation, or other entity that has a direct or indirect interest in a system" [11], then *business*, *legal*, or *user* requirements could be considered as subtypes of stakeholder requirement. For example, "business requirements" describe the "needs of the enterprise", rather than those of "a particular stakeholder or class of stakeholders" [22]. Confusingly, this terminology is occasionally used interchangeably. Throughout this paper, we continue the convention (considered to be "incorrect" [25, 26]) of referring to "stakeholder" requirements as "user" requirements, for coherence with the previous works on the topic [14, 15].

2.2.2 System requirements subtypes

Since a system is "a collection of components—machine, software, and human, which co-operate in an organised way to achieve some desired result" [11], there can be requirements on the whole system, its components, and on its interfaces. Hence, a software requirement is a system requirement by definition, but a system requirement is not necessarily a software requirement, since software requirements are enforced solely by the software agent to-be, and *"a system does not merely consist of software"* [25]. Whether or not a system requirements document contains non-software requirements (such as expectations on human behavior, hardware, or interfaces), depends on whether the software will provide "essentially all the functionality" of the envisioned system (as in "Software-Intensive Systems" [20]), or whether it will be just one part of a larger system [22].

2.2.3 Difference between stakeholder and system requirements

To summarize Table 1 and to answer RQ1, translating a UR to a SysR should involve at least:

1. **A decision on the abstract system-to-be.** Not doing so risks that the SysRs will be too ambiguous to be useful. For example, deriving specific SysRs on energy usage that would be equally applicable to burglar alarms, electric fences, and guard dogs is a difficult task [27].
2. **A transition from a stakeholder need to a system responsibility.** Not doing so risks that the need will remain unsatisfied, and indicates that a desire may not have been *clarified*, *decomposed, derived,* or *allocated* [28], such that it is clear that a component (or more generally, the system) will be causally responsible for it.
3. **Technical, rather than customer-oriented language.** Not doing so risks that software developers misinterpret requirements and waste effort on software that the customer will not accept; Rework in programming costs exponentially more time than in requirements [29].

The unanimous recommendation from Table 1 that URs should describe problems while SysRs should describe solutions cannot be interpreted as an absolute rule, since problems and solutions are not conceptually orthogonal. For example, Jackson notes that merely structuring a problem moves the description toward a solution [36], while Berry concludes that specifying a problem precisely sometimes requires reference to the solution domain, and so "for some requirements, it may be impossible to specify *what* without saying something about *how*" [37]. Furthermore, a UR or a SysR can simultaneously be a description of a problem and a solution, since what is an end in one context is a means to an end in another (as is the pragmatist's argument against "intrinsic value" [38]). Similarly, Zave and Jackson argue that "almost every goal is a subgoal with some higher purpose" [39]; adapted to this UR–SysR context is that even the most non-technical of URs will rarely not be "solutions", e.g., to achieving stakeholder happiness. (Consequently, alternative solutions to the "engineering problem" should be scoped to the SysR level to avoid requirements specifying that the stakeholder should instead "adopt religion or devotion to family" [39], for example.) Overall, the essence of the matter is not that there should be a problem–solution dichotomy, but rather

Table 1 Differences between user requirements (UR) **and** system requirements (SysR) definitions

	Domain		Role		Language	
	UR	SysR	UR	SysR	UR	SysR
Textbooks						
Requirements Engineering (Hull et al.)	"**problem** to be solved" in the "enclosing system" [11]	"abstract **solution**" [11]	"state[s] what the stakeholders **want to achieve** through use of the system" [11]	"state[s] abstractly **what the system will do** to meet the stakeholder requirements" [11]	"**stakeholder's** view" [11]	"**analyst's** view" [11]
Systems Engineering (Stevens et al.)	"**problem** domain" [20]	"**solution**…in functional terms" [20]	"**defines the results** that the system will supply to users" [20]	"defines an abstract model…**to reason about the end solution** before commitment to a specific design" [20]	"**users** can easily understand" [20]	"extra detail"—"**designers & test eng**" [20]
Software Engineering (Sommerville)	"a company… define[s]…**needs**" & "a solution is not pre-defined" [21]	"system's functions, services…to be **implemented**" [21]	"so that several contractors can bid…**offering different ways of meeting the** …**organization's needs**" [21]	"define[s] exactly what is to be implemented" … "so that the client understands and **can validate what the software will do**" [21]	"abstract… for the **customer & end-user[s]**" [21]	"precise detail for **developers & testers**" [21]
Standards & Frameworks						
ISO 29148	"provide the true picture of the **problem** to be solved" [22]	"characteristics, … and functional … requirements for a product **solution**" [22]	"**describe[s] the needs** that a given stakeholder has and how that stakeholder will interact with a solution"—"a bridge between business requirements and the … solution requirements" [22]	"provide[s] a description of what the **system should do** in terms of the system's interactions or interfaces with its external environment" [22]	"business **management**" & "business **operational level**" [22]	"understandable by both the **acquirer** & the **technical community**" [22]
SEBOK	"views of stakeholders as they relate to the **problem** (or opportunity)" [23]	"synthesis of the functions required of any **solution** system…" [23]	"enables the characterization of the solution alternatives" [23] & "the **basis of** SysR activities, **system validation** and stakeholder acceptance" [23]	"**basis of system** architecture, design, integration &, **verification**" [23] & "translates a stakeholder requirement in the language of the supplier" [23]	"users, acquirers, **customers**, and other **stakeholders**" [23]	"between the various **technical staff**" [23]
UK MoD AOF	"the **problem"** [30] "…**not**…anticipat[ing] a **particular solution**" [31]	"**solution**-focused response to the capability-focused URs" [32]	"**define[s]**… **the outcome** … that the user needs to be able to achieve" [33]	"defines what is needed…" [32] in terms of "**what the system must do** [and] how well it must perform" [34]	"understood by the **user & stakeholders"** [33]	"suitable for **contractors & stakeholders**" [35]

Table 2 Examples of user requirement–system requirement (UR–SysR) pairs from the literature

#	UR	SysR
1	"The driver shall be able to deploy the vehicle over terrain type 4A" [11]	"The vehicle shall transmit power to all wheels" [11]
2	"The <Management User> requires data to be protected <Measure of Effectiveness> from unauthorized access" [48]	"The <Security System> shall provide encryption to electronic data" [48]
3	"The maximum duration of a trip from the mother craft to the beach is 1 h, because trips over an hour will make most people too seasick to function effectively" [28]	"The landing craft shall have a minimum speed of 25 knots" [28]
4	"The elevator shall receive calls for up and down service from all floors of the building" [2]	"The elevator system shall produce digitized passenger requests" [2]
5	"Patients shall be medically aided within less than eight minutes on average" [20]	[The system shall] "Locate available ambulance" [20]
6	"The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month" [21]	"The system shall automatically generate the report for printing after 17.30 on the last working day of the month" [21]
7	"User shall be able to store grocery inventory data" [14]	"System shall store grocery inventory data" [14]

that both the positive and negative effects (and hence the pertinence and adequacy) of a SysR can be understood when it is related to one or many URs that describe the problem(s) that should be treated.

At a more philosophical level, in a discussion over what a UR or a SysR should be, it is crucial to remember that requirements in themselves are intangible and exist independently of their representations [40]. Indeed, aside from UR and SysR documents, requirements and their pertinence could be represented by other means, e.g., with goal graphs, as in GRL [41] or KAOS [42] (where system requirements "implement" stakeholder goals [43]). In RE, the important concern should not be the choice of representation or associated nomenclature, but rather that the "requirements problem" is solved [9], or put bluntly, that the "how"s satisfy the "what"s such that asking "why" of the "how" leads to the "what". Then, the wider concern should be that the "what"s will be valuable, as is the topic of value-based requirements engineering [44].

Finally, a discussion on "requirement" qualifications and representations would not be complete without addressing the distinction between the so-called Functional Requirements (FRs) that prescribe desired behaviors, and Non-Functional Requirements (NFRs) that prescribe desired qualities [45]. Regardless of the myriad of definitions proffered for these, Glinz demonstrates that the distinction lies in the requirement's representation rather than its concern [46]. For example, the different representations "*…require username and password…*" (FR) and "*…probability of unauthorized access…*" (NFR) refer to the same security concern, but provide different information about it. Similarly, a UR and a SysR may both refer to the same concern, yet their different representations should provide different information. Stated in terms of the KAOS meta-model for requirements engineering [42], if a requirement's SysR representation is missing, then information about *responsibility* may be lost (i.e., which system component will enforce the requirement?), while if the UR representation is missing, then information about which stakeholder(s) *wish* for the requirement, and why they wish for it, may be lost.

To add illustration to the so-far conceptual discussion, Table 2 provides examples of UR–SysR pairs from the literature. Examples #1–6 are intended by the respective authors to be examples of good practice, while #7 is Kovitz's example of the UR–SysR repetition bad practice. The reader will observe that these translations from UR to SysR vary in their adherence to the conceptual definitions in Table 1, but as Rost describes, compliance with RE standards by industry and even "semiofficial" authors is generally poor [47].

2.3 Traceability between user and software requirements

All of the literature cited in Table 1 recommends that relationships between SysRs and URs should be captured and maintained, e.g., with a traceability matrix such as MODAF view SV-5 [49], the House of Quality [12], or more simply as a traceability field in the spreadsheet, document, or database. The analysis enabled by such traceability can be classified as either [11]:

- **Impact analysis**—"What if this requirement is changed?" (for change analysis);
- **Derivation analysis**—"Why is this requirement here?" (for goal and cost/benefit analysis);
- **Coverage analysis**—"Are all customer concerns covered?" (for progress and accountability analysis).

The results of traceability analysis can be distilled by traceability metrics to improve the identification of risks and flaws early in the system life cycle [50]. For example, coverage metrics could be calculated to describe the percentage of URs that relate to at least one SysR. Here, complete coverage is highly desirable for convincing stakeholders that their concerns will be addressed by the system-to-be. However, the descriptive power of the coverage metric depends on the URs and the SysRs according to Table 1's guidelines. For example, a 100 % UR coverage metric would not ensure what the metric is supposed to ensure if it is calculated from a set of repeated UR–SysR pairs. Indeed, many projects have failed to derive benefit from traceability [51, 52], e.g., because of "traceability for its own sake" or for process compliance. Hence, Asuncion et al. propose that each traceability link made throughout the system's life should be questioned for its ability to be beneficial [51], and in the case of tracing between a repeated UR and a SysR, i.e., to form a UR–SysR pair, we do not see much benefit. In summary, and to contribute to answering RQ4, repetition between URs and SysRs diminishes the efficacy of the aforementioned traceability analysis, the impact of which ranges from misinforming stakeholders about problem–solution coverage, through to being unable to make value-oriented decisions in the software project, e.g., in prioritization, trade-off analysis, etc. [53].

2.4 Measuring the repetition in a UR–SysR pair (string similarity from an information retrieval perspective)

Being able to objectively quantify the degree of repetition between UR–SysR pairs is essential for understanding the extent of its existence, and therefore for the management and reduction of it. Since URs and SysRs are typically represented by textual statements, even if they were derived from models such as in [54], their similarity—and hence the degree of repetition between them—can be quantified using string similarity metrics, of which, Cosine, Jaccard, and Dice are the most popular [10]. These metrics take as input two strings and output a normalized score in the interval [0,1] for comparability of different string lengths, starting at zero when there is no commonality (no matter how different), and reaching one when there is no difference [10]. These metrics are commonly calculated (e.g., in e-mail spam detection [55]) according to the "bag of words" model where a requirement's description would be processed into a set of unordered terms (words) and their frequencies. The steps in processing a sentence to a bag of words typically include tokenization (splitting the sentence into words), and stemming (stripping words to their "base meaning", e.g., "maintainable" becomes "maintain") [56]. The contrived problem that "user uses system" and "system uses user" would be considered equal, illustrates the drawback. However, this is not likely to be an issue, since two requirement descriptions would seldom use exactly the same frequencies of terms sequenced differently to describe different information. Besides, disregarding a term's context (its surrounding terms) improves recall (i.e., more similarity is detected), since a term such as "performance", will be more common between a UR–SysR pair than sequences of terms, such as "system performance", as *n*-grams [56] would represent.

While the Jaccard and Dice metrics are intuitive (in essence, they divide the number of common terms by the total number of terms), their disadvantages for this paper's context are:

1. That term occurrence is Boolean. For example, the second occurrence of the "user" term is ignored in "the user shall be able to communicate with another user";
2. That the importance of term difference is assumed to be equal across all terms, but existence of the term "system" in a SysR is more trivial than the term "encryption".

Therefore, modern information retrieval applications such as search engines favor a vector space model of similarity using terms as dimensions, strings as vectors, TF–IDF distance (term frequency–inverse document frequency) as vector magnitudes, and the calculated cosine measure of angular similarity between vectors [56] (as visualized by the two-term example in Fig. 1). Hence, the first aforementioned problem of the Jaccard and Dice metrics (Boolean representation of term occurence) is treated by considering the frequency of terms in each of the strings.

The second problem of the Jaccard and Dice metrics (assuming equal term importance) is treated by TF–IDF by

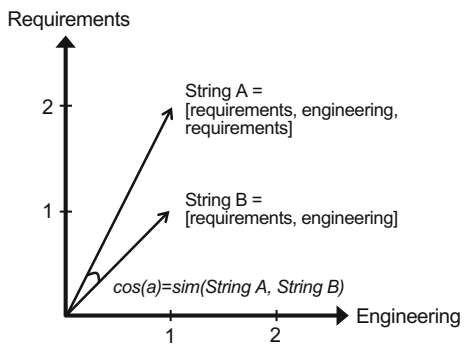

Fig. 1 Similarity between String A and String B using the cosine measure and the vector space model, as used to calculate TF–IDF (figure modified and adapted from [57], and simplified to include only two dimensions and hence only two terms)

multiplying the term frequencies by the term's inverse document frequency (the log-dampened probability that the term could be picked at random from the projects set of requirements). Hence, any non-similarity between a UR–SysR pair comprised of terms frequently occurring in the requirement set (e.g., the term "user" in a UR) is treated as less significant than non-similarity comprised of rarer terms. This weighting seems logical in the context of UR–SysR repetition. For example, the differences in UR–SysR pair #7 in Table 2 (*-user, +system, -be, -able, -to)* are likely to be common across the project's set of URs and SysRs, and they do not seem important enough to warrant reading the SysR if the UR has been read.

The usefulness of IDF weighting is underpinned by the assumption that important terms are rare while trivial terms are common. However, take for example the term "notwithstanding", which despite being generally rare, is not a significant indicator of a requirement's concern. Furthermore, specially rare terms, i.e., rare in a particular requirement set but generally common, would also be given an inflated importance by IDF. An example from this paper's case study is the replacement of the term "which" with "that" from a UR to a SysR. Hence, the impact of trivial but rare differences between UR–SysR pairs needs to be reduced by additional means. To do so, many information retrieval systems remove "stop-words" prior to calculating TF–IDF. That is, commonly occurring words such as *the*, *that,* or *with* are removed after tokenization, since while they "help build ideas", they *"do not carry any significance themselves"* [58]. The list of 33 stop-words used in Apache's Lucence (popular information retrieval software) provides a well-tested set [59]. Runeson et al. [57] also suggest that if the strings are structured (i.e., based on a template), then the template's structural words should also be removed to reduce their impact on the similarity measure. The EARS templates [60] provide such a list of candidate stop-words commonly used to structure natural language requirements. However, logical operators and other high-information-value but common words exist in both Lucene's stop-word set and the EARS template. These terms should not be removed in this paper's RE context because they are able to radically change a requirement's meaning with one term frequency change. Hence, Lucene's stop-word set and the EARS keywords comprise the stop-word set for this paper, except for the *{and, or, no, not, when, while, where, if, then}* terms.

Finally, it is worth citing Falessi et al.'s [61] recent classification and comparison of natural language processing (NLP) techniques for identifying equivalent requirements. Falessi et al. conclude that a technique comprised of the vector space model and the cosine similarity metric, "*if adopted in a traceability tool, is expected to provide the maximum benefit to the human analyst compared to other NLP techniques*". Interestingly, they observed that the use of synonymies (i.e., considering synonymous terms such as "monitor" and "screen" to be the same dimension in the vector space model) was "deleterious", "introducing more noise [i.e., false positives] rather than making a positive contribution". In addition, Runeson et al. [57] remark on the difficulty of constructing a domain-specific synonym list (for detecting duplicate software defect reports) that would be "efficient enough" to justify the effort. Indeed, they report an insignificant improvement after attempting to do so. Overall, Falessi et al. [61] find that "the simplest NLP techniques are usually adopted to retrieve duplicates".

(The reader interested in learning more about TF–IDF is directed to a one-page online tutorial [62], or to Manning and Schutze's [56] book for a more thorough treatment.)

3 Research method

This paper's subject became of interest to the authors during the first author's placement at a UK defense contractor as part of a wider research project [63]. We constructed the research questions introduced in Sect. 1 after three interviews with experienced consultants from the organization to ensure their relevance (mean 21.6 years experience; SD 2.8). Since the studied phenomenon is sociotechnical and cannot be accurately replicated in laboratory conditions, it must be studied in its natural context. Therefore, the case study research design is adopted, and is guided by Runeson and Höst's [64] guidelines for case studies in software engineering. As such, our intention is not to answer the research questions for the general population of software engineering practice, since the size and variety of samples required to do so were unaffordable within the resource constraints of this project. Instead, we intend to contribute to (or in this case, start) the "chain of evidence", and to "enable analytical generalisation where the results are extended to cases which have common characteristics" [64] (rather than generalize purely with inferential statistics [65] and its commonly "unrealistic" random sampling assumption [66]). Consequently, the sampling method used within this case study is a form of non-probability "convenience sampling", guided by aspects of "judgement sampling", "snowball sampling", and "quota sampling" [67].

As further motivation for the case study research design, Easterbrook [68] argues that it is well suited for "how and why questions" due to its depth (rather than breadth) of enquiry. Finally, falsification is afforded by single case studies, and is sufficient for providing answers to our research questions on the phenomena's incidence and consequences, especially RQ2's "does it occur?" and

RQ4's "is it a problem?" (i.e., only one non-pink flamingo is required to refute that "all flamingos are pink"). Lastly, since robust case studies require "triangulation" of conclusions using different types of data collection methods [64], Sect. 3.1 describes the questionnaire we employed (subjective data), while Sect. 3.2 describes the similarity analysis of UR–SysR pairs performed by our software tool (objective data). This mixed-methods approach is especially important since a practitioner's understanding of UR–SysR repetition (including its frequency of occurrence) will likely be distorted due to cognitive biases, such as confirmation bias or the availability heuristic [69].

3.1 Research method 1: Questionnaire

In order to answer RQs 2–4 (does UR–SysR repetition occur, why does it occur, and is it a problem?), we selected consultants from the organization who were experienced in requirements engineering ($n = 23$) and invited them to complete an online structured questionnaire. The questionnaire was constructed with compliance to Umbach's best practices in order to minimize potential for survey error [70]. We received completions from 16 participants (70 % response rate) reporting on 260 years of combined software engineering experience (mean 16.25, SD 5.32). The responses describe a combined total of 235 software development projects (mean 14.68, SD 8.26) in the last 10 years. While these projects are not likely to be wholly distinct, they are unlikely to be entirely homogenous due to the nature of the consulting business. We asked the participants to limit their responses to the last 10 years of their career in order to ensure the currency and relevance of the results. For each question in the survey, a "do not know or N/A" option was provided to prevent any uninformed responses distorting those provided with higher confidence. Where percentages are used to describe results in this paper, they are always relative to the total number of participants (16) rather than the number of non-"do not know or N/A" responses.

3.2 Research method 2: UR–SysR similarity analysis

In order to improve the robustness of our answer to RQs 2 and 5 (does it occur, and can problems be reduced?), we used our software tool [16] to analyze pairs of requirements documents from two software projects written by different authors, i.e., one author per project and four requirements documents in total. We were limited on the number of projects whose documentation we could acquire due to security restrictions, but it should be noted that this sample size is strong compared to other research on requirements documents from industry "due to nondisclosure agreements between researchers and industry", or lack thereof [61]. Crucially, in order to construct a "typical" case study (of Gerring's 9 case types [71]), we sought UR and SysR documentation that was representative of common practice, rather than of UR–SysR repetition. We were assured by the three interviewed consultants of the documents' typicality in the organization, as well as in the UK defense IT industry—afforded by the consultants' collaboration with other consultancies. (Confidence in the latter is obviously lower than in the former due to the local and isolated nature of a case study.)

We used our software tool to perform the following processes:

1. Import requirements from documents, each comprised of a global and a local (hierarchical) ID field, a description text field ("system/user shall…"), a fit criterion text field (test case), and a rationale text field (justification).
2. Reconstruct UR–SysR traceability links using textual references from SysRs to the global IDs of URs (thus forming the set of the project's UR–SysR pairs).
3. Preprocess the text fields in the requirements according to Sect. 2.4, as exemplified in Table 3.
4. Compute IDFs for the project's set of URs and SysRs (where the IDF's "document" set is comprised of only the requirement field being analyzed, e.g., the fit criterion).
5. Calculate TF–IDF distance scores for each of the project's UR–SysR links (distance score is 1-similarity score).
6. Display the UR–SysR pairs and their TF–IDF scores for identification of false positives, as shown in Fig. 2.
7. Compute statistics (TF–IDF histograms and correlations) to summarize the detected UR–SysR repetition.

Steps 1–2 required manual conversion of the MS Word documents containing the requirement statements into CSV files via MS Excel using copy and paste. Since our tool can interactively map CSV fields to the core requirement fields [72], a pre-defined requirement field structure is not required. (This mapping and import stage would not be required if the requirements were available in the requirements interchange format ReqIF [73], e.g., when using SysML models.) Steps 3–5 were implemented using the Java LingPipe linguistic analysis library [74] to ensure robust and repeatable results. The following of LingPipe's tokenizer factories were used: *IndoEuropean*, *LowerCase*, *PorterStemmer*, and *Stop* (with the stop-word list described in Sect. 2.4). Punctuation was removed using the regex *"\p{P}"*. Step 6 is assisted by our tool's use the Java diff-match-patch library [75] and Graphviz [76] to visualize differences between a UR's and a SysR's tokens, and Step 7 is assisted by our tool's use of the R statistics software [77].

Table 3 Preprocessing of a requirement description (a SysR)

Unmodified SysR description field	Tokenized, stemmed, and stop-words removed
The system shall be able to display a list of files available at each accessible location	system, abl, displai, list, file, avail, each, access, locat

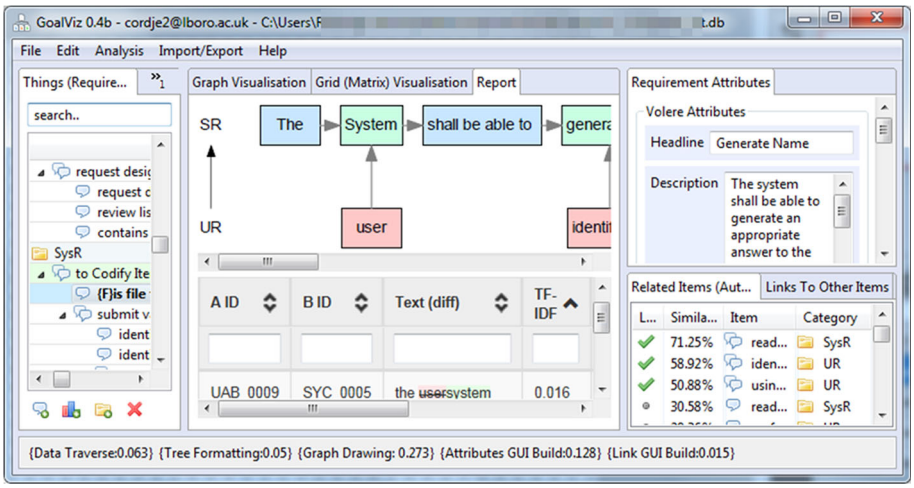

Fig. 2 Eclipse-based Java tool to import URs and SysRs, and analyze and visualize both repetition and traceability [16]

Table 4 Summary of the two software projects' requirements

	URs	SysRs	UR–SysR links (pairs)	Words per requirement (description field)
Project A	30	52	55	Mean 14.4; SD 6.7
Project B	48	60	98	Mean 21.4; SD 5.6

Table 4 describes the number of requirements in the two projects, their traceability, and their wordiness. In Project A, all but three SysRs link to one UR (there are 55 traceability links from SysRs to URs, and 52 SysRs). This indicates that the majority of repetition discovered between UR–SysR pairs in Project A can cause the repeated requirement (either the UR or the SysR) to be redundant, since it is not linked to other requirements. However, the cost of redundancy (i.e., the time to read or write) in Project B will likely be higher than it would be in Project A, since the requirement statements contain more words. All links between URs and SysRs are of positive rather than negative polarity, and so each traceability link represents a UR–SysR pair in the sense that the SysR is some part of the solution to the UR, as opposed to conflicting with it.

From the relatively low number of requirements in Project A and B, the reader might be wondering why the degree of UR–SysR repetition is investigated using TF–IDF calculated by the software tool, instead of just manual inspection by humans. Indeed, a viable strategy might be to ask people to rate the degree of repetition and then calculate inter-rater reliability. However, one of the aims of this research is to provide automated tool support to assist quality analysis of requirements documents. The aforementioned manual inspection's time consumption would be measured in hours (as elaborated in Sect. 4.3), while automated analysis' time would be measured in seconds or minutes. Furthermore, manual verification by two IT professionals external to this research, found that at the chosen TF–IDF cutoff point (0.3) for classifying a UR–SysR pair as repeated, there were no false positives (i.e., UR–SysR pairs that were not considered repeated but were classified as such), and so the TF–IDF derived results provide a confident lower bound on the proportion of repeated UR–SysR pairs in this case study. (This topic, and the lesser effects of false negatives are later discussed in Sect. 4.4.1.)

3.3 Context of the case study

An overview of the projects that the participants described in their questionnaire responses is provided in Tables 5, 6, 7, 8 and 9, where the percentages describe the distribution of the participants' responses to the questions.

Table 5 Summary of the projects whose resulting software is now, or was ever in use by the end-users

~None	Some	~Half	Most	~All
6.25 %	18.75 %	18.75 %	37.5 %	18.75 %

Table 6 Summary of Table 5's projects that achieved their intended benefits

~None	Some	~Half	Most	~All
6.25 %	31.25 %	18.75 %	43.75 %	0 %

Table 7 Summary of the projects that had easily contactable end-users

~None	Some	~Half	Most	~All
6.25 %	50 %	12.5 %	12.5 %	18.75 %

Table 8 Summary of the average proportion of the projects' functionality that was redundant

~None	Some	~Half	Most	~All
0 %	81.25 %	6.25 %	0 %	0 %

12.5 % of the respondents selected the "do not know" option

Table 9 Summary of the projects that had explicit traceability from business objectives to software requirements

~None	Some	~Half	Most	~All
37.5 %	43.75 %	6.25 %	12.5 %	0 %

The software projects' benefits and usage seem in-line, if not better than the averages reported in a recent survey of the IT industry [78]. Interestingly, difficulty in contacting end-users (Table 7) indicates that applying agile RE approaches (i.e., no UR–SysR documentation) in this organization's context could be challenging [79]. Furthermore, since requirements were rarely explicitly traced to business objectives (Table 9), it is especially important that the URs describe the SysRs' corresponding problem(s) in order to contextualize and assure their pertinence and adequacy.

The majority (75 %) of the participants had experience in mostly military projects, while 18.75 % had experience in mostly civil projects, and the remainder had approximately equally split experience. As such, most of the requirements engineering practice reported on in this case study will have adhered to the guidelines in the UK Ministry of Defence's Acquisition Operating Framework (AOF) [34]. This does not significantly limit the generalizability of the results, since as Table 1 shows, the AOF is mostly in agreement with the other standards and guidelines. Furthermore, the AOF seems opposed to redundancy in requirements documents, since it offers two redundancy-related guidelines for creating user requirements documents in [33]:

- "*Exploit the hierarchy to minimize repetition*. Statements can inherit characteristics from parent nodes."
- "*Avoid duplication.* If two users/stakeholders have a common interest, record both sponsors against a single statement."

These recommendations do not explicitly address redundancy between UR and SysR documents, e.g., for the first guideline, the hierarchy of URs may be decomposed differently than the SysRs, since at the UR stage, the system is not yet architected, i.e., split into functional components. However, when combined with the AOF's conceptual differences shown by Table 1 and the exemplary differences shown by requirement #2 in Table 2, we conclude that, in theory, URs translated to SysRs in accordance with the AOF should not contain a high degree of repetition.

4 Results

This section revisits each research question defined in Sect. 1 and provides our answers to them. RQ1 was previously answered by Sect. 2.2.3, and so this section starts from RQ2.

4.1 RQ2: Does UR–SysR repetition (still) occur?

There are two aspects to this question's answer: the proportion of projects that exhibited UR–SysR repetition (1) and the proportion of UR–SysR pairs that exhibited repetition within projects (2).

4.1.1 How many projects exhibited UR–SysR repetition?

The questionnaire asked the participants to describe how many software projects they had seen in the last 10 years that had exhibited repetition between the project's UR and SysR descriptions. To reduce the likelihood of misinterpretation, we included Kovitz's example of UR–SysR repetition (requirement #7 in Table 2) in the question. The participants were given five ordinal categories to choose from (chosen in place of interval or point estimates to minimize the survey dropout rate), spanning from "nearly none" to "nearly all" of their projects. Table 10 provides the frequencies of the participants' responses to these categories. In summary, 75 % of the respondents had seen

Table 10 Projects where requirement descriptions were repeated between URs and SysRs in the last 10 years (2003–2013)

~None	Some	~Half	Most	~All
0 %	25 %	18.75 %	50 %	6.25 %

UR–SysR repetition in at least half of the projects they had seen in the last 10 years, and all of the respondents had seen at least some projects exhibiting UR–SysR repetition.

There was a difference in the respondents' experience (number of projects in the last 10 years) between those who chose {~None, Some, ~Half} (mean 10.7 projects; SD 6.1) and those who chose {Most, ~All} in Table 10 (mean 17.7 projects; SD 8.7). Hence, it could be hypothesized that those who saw repetition in half or less of their projects would have seen more repetition if they had been exposed to more projects. However, both the high variation (SD 8.7) of the {Most, ~All} group, as well as the Wilcoxon rank-sum test result of $W = 16$, $p = 0.1$ (not requiring data to be normally distributed or on an interval scale [80]), do not provide support for this hypothesis, indicating no significant difference between stakeholder experience and frequency of observed UR–SysR repetition.

The questionnaire asked the participants if they had observed a higher or lower degree of UR–SysR repetition throughout their whole career, as opposed to the last 10 years. The majority of the respondents (87.5 %) selected "no difference", while the remainder (12.5 %) selected "less repetition", and none selected "more repetition". In other words, according to the memory of the respondents, the incidence of UR–SysR repetition has not reduced in the last 10 years, compared to their whole careers.

4.1.2 *How many UR–SysR pairs exhibited UR–SysR repetition?*

All of the literature cited in Table 1 states that URs and SysRs are primarily comprised of three core fields: the requirement's main description, a fit criterion (test case), and a rationale (justification). Hence, this section presents the analysis of repetition (i.e., TF–IDF scores for each UR–SysR pair) in our sample project's requirements' description, fit criterion, and rationale fields.

Figure 3 shows a histogram of the TF–IDF distance scores for the two projects' UR–SysR pairs' description fields, where low distance scores represent a high degree of repetition. It is apparent that while Project A has a significant number of almost identical UR–SysR pairs (31 %), the distribution is fairly symmetrical around the mean (mean 0.49; SD 0.40; median 0.56; skewness = −0.02). Hence, there is also a great deal of significantly different pairs that are at risk of being overlooked by the reader.

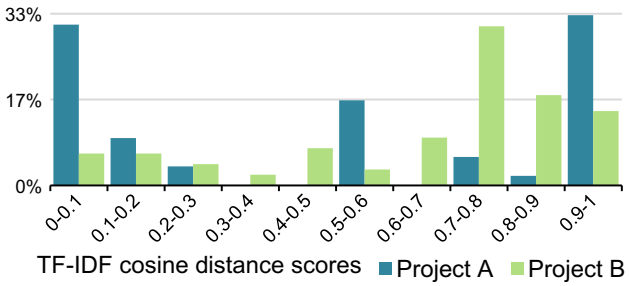

Fig. 3 Distribution of string distance (TF–IDF) scores for the *description* fields of each UR–SysR pair

Project B appears to exhibit less UR–SysR repetition since it is skewed toward a higher degree of difference (mean 0.65; SD 0.27; median 0.75; skewness = −1.02). In summary, 43 % (24) of Project A's and 16 % (16) of Project B's UR–SysR pairs' description fields have a TF–IDF distance score of up to 0.3. (The 0.3 threshold for classifying a repeated UR–SysR pair was derived from manually assessing the triviality of the differences between the UR–SysR pairs, as is later discussed in Sect. 4.4.1.) Hence, on average over the projects, nearly a third of the UR–SysR pairs' description fields were repeated with no significant textual differences.

To demonstrate the meaning of the TF–IDF distance scores shown in Fig. 3, examples of UR–SysR description fields from both projects are provided in Table 11. The requirement descriptions are presented as the output of Myer's difference algorithm with semantic cleanup applied [75] in order to improve the reader's comprehension of the differences. The visual representation of the differences allows the reader to quickly see that most of the listed UR–SysR pairs are not parsimonious (i.e., where the less data required to convey the message, the better). This edit-based model of string similarity can be used to measure the degree of similarity by counting the number of insertions and deletions required to transform one string to the other, e.g., the Levenshtein distance [81], as is used in the diff-match-patch library [75].

However, edit-based similarity metrics are highly sensitive to changes in the sequence of words, and do not attempt to distinguish between important and trivial differences. In this case study, we observed that the Levenshtein and TF–IDF distances for the description fields were more strongly correlated in Project B (Pearson's coefficient = 0.88) than Project A (Pearson's coefficient = 0.80). Hence, more of Project A's UR–SysR differences were either common, rare, grammatical, or word-order related. This information could be useful to prioritize the review of UR–SysR pairs, since disagreement between a Levenshtein and a TF–IDF score indicates that the pair may seem more similar or different to the reader

Table 11 Example UR–SysR TF–IDF distance scores and string "diff" analysis

TF–IDF	Requirement Description Field (~~UR~~/SysR)
0.02	The ~~user~~**system** shall be able to identify the manufacturer's part number that applies to the subject item of production
0.12	The system shall be able to read the content of a file on a storage medium accessible via ~~a~~**an address** ~~an address~~ **URL** identified by the user
0.15	The ~~user~~**system** shall ~~be able~~ **enable the user** to specify a single file for conversion (single file mode)
0.19	The ~~user~~**system** shall ~~be able to determine~~**indicate** the success of the conversion process through a status information return
0.21	The ~~user~~**system** shall be able to ~~review all~~**display a list of** errors that arise from the processing of a new item create
0.25	The ~~user~~**system** shall ~~be able to~~ convert one or more catalog files that are independent of the **originating** ~~cataloguing~~**catalog** management system from eOTD-r-xml to ~~Modified Segment V (~~MSV~~)~~
0.4	The ~~user~~**system** shall be able to ~~respond to situations where existing~~**display a list of** items of supply **that** potentially require cancelation
0.84	The ~~user~~**system** shall be able to ~~identify the address of a file that contains the specification for the items~~**support easy navigation through the hierarchy of accessible addresses**

Key: *strike-through:* remove to form SysR; *underline:* insert to form SysR; *plain style:* commonality

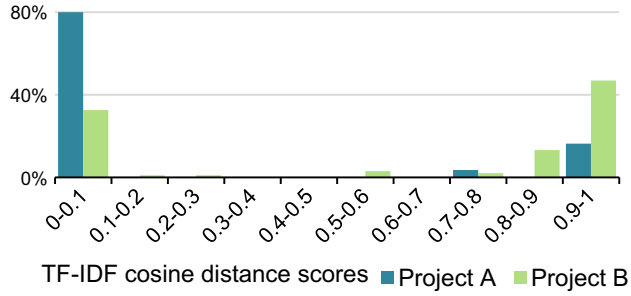

Fig. 4 Distribution of string distance (TF–IDF) scores for the *rationale* fields of each UR–SysR pair

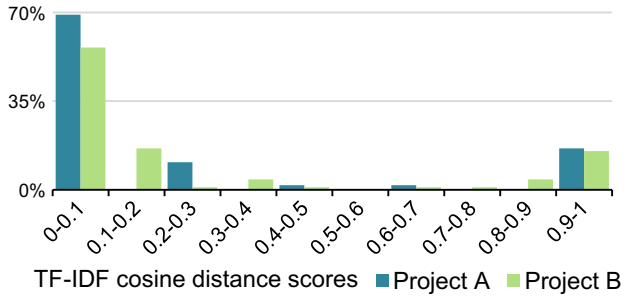

Fig. 5 Distribution of string distance (TF–IDF) scores for the *fit criterion* fields of each UR–SysR pair

than they actually are. Finally, Table 11 indicates that the scope for improving the projects' requirements likely lies in the problem domain. For example, the URs mostly refer to "user" interactions with the solution rather than problems that a particular stakeholder (individual, class, or role) wishes to be solved by the system-to-be.

Interestingly, the rationale field was repeated far more than the description field in both projects (approximately twice as many repeated UR–SysR pairs); 80 % (44) of Project A's and 35 % (34) of Project B's UR–SysR pairs have a TF–IDF distance score of up to 0.3. Furthermore, Fig. 4 shows that a majority of UR–SysR pairs were of the "copy and paste" type (i.e., exact duplication). In fact, all of Project A's rationale fields could be considered duplicated, since its non-similarities do not add valuable information, e.g., SysRs containing phrases such as: "*assumed from user requirement*". (This type of phrase could be added to a stop-phrase list so that they are not considered to be worth reading.) On average over the projects, two-thirds of the UR–SysR pairs' rationale fields were repeated with no significant differences.

Figure 5 shows that on average, the fit criterion field contained the most repetition between UR–SysR pairs; 80 % (44) of Project A's and 73 % (72) of Project B's UR–SysR pairs have a TF–IDF distance score of up to 0.3. As with the rationale field, most repetition tended to be almost exact duplication. The similar but not duplicated UR–SysR pairs (i.e., the second and third buckets in Fig. 5) were also insignificant, consisting mostly of minor corrections or updates, e.g., the UR "*every example design data file*" and the SysR *"example data file*". Additionally, the mostly different pairs (i.e., the last two buckets in Fig. 5) were also trivially different, such as where the SysR's field was empty (and so for analysis of future projects we would add an empty string to the stop-phrase list). To summarize the fit criterion field's repetition over the projects, four-fifths of the UR–SysR pairs' fields were repeated with no significant differences.

Finally, we examined correlation between the UR–SysR pair TF–IDF distance scores and the following variables:

1. UR and SysR description length;
2. UR and SysR document position (i.e., sequential IDs);
3. UR and SysR hierarchy depth, i.e., is SysR 1.1.1.1 more or less likely to be repeated from its UR than SysR 1.1?

4. UR and SysR hierarchical sibling position, i.e., is SysR 1.1.5 more or less likely to be repeated than 1.1.1?
5. Number of links to URs from the SysR.

No significant correlation for the first four variables was found in either project (the strongest Pearson's coefficient was 0.31 for Project A's SysR hierarchy depth and its description fields' TF–IDF scores). However, Project B's "number of links to URs from the SysR" was moderately correlated with the description fields' TF–IDF distance scores (Pearson's coefficient = 0.38). As could be logically derived, it appears that a significant proportion of Project B's non-repeated UR–SysR pairs (i.e., the rightmost buckets in Figs. 3, 4, 5) occurred where the SysRs were linked to more than one UR. In other words, because there were more links between URs and SysRs than SysRs (98 vs. 60), more "related but not directly derived" UR–SysR pairs existed. In support of this conclusion is Project A's weaker correlation in the same variables (Pearson's co-efficient = 0.25) and its almost equal number of UR–SysR links and SysRs (55 vs. 52). Hence, there is less risk of wasting time due to the repetition defect when reading SysRs that are linked to more than one UR.

In summary, and to answer RQ2 in light of Table 10 (answering "how many projects?") and Figs. 3, 4 and 5 (answering "how many requirements?"), we can say with confidence that redundancy-causing repetition between the textual fields of URs and SysRs has occurred in at least half of the projects in the context of this case study throughout the last 10 years. Based on the analysis of the sample requirement documents, and after confirmatory discussions with the survey participants, a conservative estimate is that, on average, one-third of the participants' projects' UR–SysR pairs exhibited redundancy-causing repetition. However, more than this proportion of a project's SysRs will have been repeated from their URs, since a project's SysRs are often linked to more than one UR, and so there are usually more UR–SysR pairs than SysRs. Furthermore, this estimate would be significantly higher if it only concerned the requirement's fit criterion and rationale fields, which are often considered to be secondary to the description field. While we cannot generalize from these results to the wider context of IT software development, discussions with the participants (who have worked with many other non-defense organizations) indicate that similar results would be not be unlikely where UR and SysR documents are produced. (As Davies observed [7], it seems that many industry projects have one generic "requirements" documents containing an undifferentiated mixture of problem descriptions and solution prescriptions.)

4.2 RQ3: Why does UR–SysR repetition occur?

Logically, there can only be three possible modes of "blame" when a UR's description is repeated across to a SysR:

1. The UR is defined too close to the solution space;
2. The SysR is defined too close to the problem space;
3. Despite analysis, the UR and the SysR are the same.

Before distributing the questionnaire, we interviewed three experienced consultants to ask why these modes of blame might occur. In total, five possible causes were elicited to explain why a UR or a SysR might be repeated to its associated SysR or UR:

- The stakeholder need was unknown;
- The solution to the need was unknown;
- There was not enough time to elicit the problem or derive a specification of the solution;
- Internal standards required both requirements sets (UR and SysR) regardless of the project's context;
- Customer standards required both requirements sets (UR and SysR) regardless of the project's context.

We then constructed a question asking the participants to rate the frequency of occurrence for each of these five causes. To mitigate researcher bias, the three consultants as well as the authors of this paper were asked to individually assess the adequacy of the causal categories, for which complete agreement was attained. An option to specify other causes was also added to the question. As with the previous questions, the ratings available to the participants were in the form of five ordinal categories (plus a "don't know" option), ranging from the cause being applicable in none of their projects to all of their projects. Figure 6 summarizes the responses to this question by showing the distributions of the respondents' ratings.

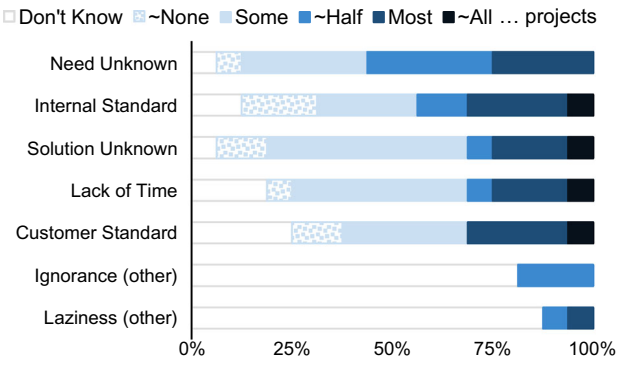

Fig. 6 Causes of UR–SysR repetition: distributions of responses for the causes' perceived frequency of occurrence

The two "other" respondent-provided causes were:

- Ignorance of the conceptual difference between what a UR and a SysR should be;
- Laziness on the project stakeholders' or requirements engineers' behalf, i.e., they were not motivated to elicit or translate the appropriate requirements.

While these "other" responses provide invaluable insights into the causes of UR–SysR repetition, their frequencies cannot be compared to the others in Fig. 6, since other participants could have been reminded of their occurrence had they been shown them. (As such, for these two "other" causes, the "Don't Know" totals represent no response, rather than explicit "Don't Know" responses.)

Among the reported causes of UR–SysR repetition, "stakeholder need was unknown" (categorized under the first mode of blame) was significantly more commonly reported than "solution was unknown" (categorized under the second mode of blame). This is not surprising, since it is well known that stakeholders and requirements engineers tend to describe solutions rather than the problems to be solved by them [11]. (In such cases that lead to UR–SysR repetition, the requirements engineer has either duplicated or slightly rephrased the description of the proposed solution in the SysR into the supposed UR.) Stevens et al. [20] observe that practitioners often believe URs to be impossible to elicit, e.g., *"Designers involved in making a sub-system, such as the engine for a car … often remark 'users don't understand what I'm building—there are no user requirements for my product'… Under these circumstances, designers will often provide functionality which is perhaps not needed by the users"*. Similarly, both Alexander's paper entitled "Is There Such a Thing as a User Requirement?" [26], and van Lamsweerde's textbook [25] provide arguments against the use of the term "User" to represent the problem description concept (as is a URs primary concern according to Table 1). Indeed, while end-users may not interact with components, those components should exist only to address stakeholder problems—even if the project was driven by means (i.e., innovation in IT) rather than by ends (i.e., problems), as Peppard et al. report is so often the case [82]. As such, requirements engineers are advised to investigate the underlying need for proposed solution capabilities by asking for their rationale [11, 20, 45], thereby assuring the pertinence of the proposed solution and enabling the evaluation of alternatives. If this good practice can be followed, then the leading cause of UR–SysR repetition (i.e., the only cause having the majority of respondents report its occurrence in at least half of their projects) could be avoided.

When a requirements engineer repeats a UR to a SysR (or vice versa), they do so out of choice or by constraint. Figure 6 shows that in this case study, the main ("by constraint") reason for UR–SysR repetition was adherence to internal standards and processes that require UR and SysR documentation regardless of context such as project complexity. This cause was closely followed by a shortage of time (though this is likely a component cause for all RE defects), and then by the customer's standards and processes requiring UR and SysR documentation regardless of context. A plausible explanation for the closeness of the "internal standard" and "customer standard" results is that the consulting company has adopted the same requirements standards and processes as their main customer, i.e., the UK MOD's Acquisition Operating Framework (AOF), and so the small differences are likely to be caused by the existence of projects not owned by that customer.

Finally, there is the case where the UR seems to be equivalent to the SysR, i.e. the third mode of blame in Sect. 4.2. To illustrate, we refer to Clements and Bass's anecdote [13] where a software project member prematurely specified the requirement of a particular database for data persistence. However, the requirement's originator was a business manager in charge of the development staff who wanted to keep an under-worked database team busy. It could then be argued that the requirement encapsulates both a business problem and a software solution. However, the two should be separated into the need for the database team's employment and the solution idea for data persistence, otherwise the requirement could be disregarded as rationale-less premature design, without describing that disregard's consequences. Many of the guidelines in the sources cited by Table 1 make reference to such situations, e.g., Stevens [20] describes encountering a requirement for all engines on an airliner to be constantly powered, rather than for the airliner always having adequate power to fly. Nevertheless, it *is* plausible that a stakeholder could solely require a solution, i.e., if they (oddly) find intrinsic value in having all engines constantly powered, or in having a database, or in having trendy technology. Where the lack of a particular solution is stubbornly declared to be the problem, and only where adequate analysis of the underlying need has been attempted, UR–SysR repetition indicates less of a risk that the stakeholders would reject the system. However, it still causes requirements specification redundancy, limits innovation (by not exploring alternatives), and worse, risks that the system would not solve the underlying (and possibly changing) problems, despite initially satisfying the stakeholder(s) (ultimately leading to dissatisfaction). Hence, this third mode of blame should be widely discouraged.

4.3 RQ4: Is UR–SysR repetition a problem?

The questionnaire asked the participants if they consider UR–SysR repetition to be a problem, and Table 12 shows that most do.

Table 12 Participant agreement on "Repetition between User Requirement and System Requirement pairs is a problem"

No strongly	No weakly	Undecided	Yes weakly	Yes strongly
0 %	12.5 %	18.75 %	31.25 %	37.5 %

The questionnaire then asked for the participants' rationale for their responses in Table 12. Of those who believed it was a problem, 56 % described generic redundancy-related consequences, e.g., the "*time wasted creating and reading them*", and the "*risk of discrepancies*". One such response described that the restatement of a UR into a SysR caused ambiguity and confusion. The next 31 % of the responses described "*deferral of systems thinking*" and decision making to the developers, leading to suboptimal and delayed decisions in the development process, and in a high number of cases, scrap and rework. The remaining 13 % of the responses can be summarized by the quote: "*very often we define the what, but not the why*", which makes the quality of existing and new decisions difficult to ascertain. For example, one respondent described that selection among alternative design options was hindered since stakeholder preferences and rationale were not adequately described.

The participants who responded "no weakly" or "undecided" (in Table 12) were interviewed to explore their opinions. The most notable responses were:

- *"Sometimes, URs and SysRs use the same terminology because it is the language that both the user and the developer use. In such cases, it is not necessarily helpful to seek to explain a clear UR in different terms for the SysR."*
- *"If the project's problem is not complex and well-tested solutions exist, then there is low risk in not translating the URs. However, the process we follow requires us to produce a UR and a SysR document regardless."*

The first quote presents an argument that UR–SysR repetition is not a problem if the project's stakeholders "use the same terminology", e.g., if they were all software developers, and so where the language dimension of difference between a UR and a SysR (Table 1) seems redundant. However, it does not follow that they *should* use the same terminology when writing URs and SysRs. While several of the guidelines in Table 1 recognize that the information [22] and "overall look, feel and structure" [35] within the different requirements documents will be similar, they stipulate that they should provide "different views for the same product" [22]. Similarly, Dorfman observes that system requirements may "closely resemble" subsystem requirements where they only need to be "allocated" to subsystems rather than translated (i.e., derived) [83]. However, this allowance concerns requirements within the solution space, rather than between the problem and solution space, which is where there is the most risk in "translating" the requirements. Hence, seeing the same terminology used between a UR–SysR pair indicates that the same information is being described, and hence that either the problem or the solution is not.

Follow-up discussions confirmed the URs were defined entirely in terms of user–system interactions, and so information about the problems to be solved by those interactions was missing. This risks the system being "unfit for purpose", or at least being suboptimal compared to the possible alternatives. Overall, pursuing only descriptions from a technical (i.e., development or engineering) viewpoint neglects the essence of RE's purpose.

The second quote presents the argument that low project risk and complexity permits not deriving a set of SysRs from the URs, instead repeating the SysRs from the URs in order to comply with a process that requires both URs and SysRs. Indeed, systems engineering standards and processes were derived by engineers experienced in complex projects such as satellites [8], which are arguably more risky and complex than the typical software development project. Instead, agile software development (which is rarely used for large or complex system development [21]) *"shun[s] formal documentation of specifications"* [79] and favors a "product backlog" consisting of "user stories". These "high-level requirements" focus on the goals of the various user roles and their rationales, and defer description of the solution to "intensive communication" between the customers and the developers "before and/or during development" [79]. Overall, the risk involved in not creating SysRs (and hence where SysRs are entirely repeated from URs) seems negligible if:

1. The project is not risky from the developer's viewpoint, i.e., there is little need for a contractually binding complete specification of system behavior;
2. "Intensive" communication between the developer and the customer is possible;
3. Creation of prototypes and/or frequent releases is economically viable;
4. The "optimal" solution to the problem is obvious and "tried and tested".

Where these points apply, and where either external or internal standards require SysRs, then creating a set of SysRs closely resembling the URs for the sake of process compliance may be a "necessary evil" from the requirements engineer's view. Indeed, since costs of deriving perhaps not-so-useful SysRs were avoided, the "Laziness" cause of UR–SysR repetition (Fig. 6) may actually benefit these projects. That being said, it is surely better to omit the SysR document than to repeat it from the URs, since repetition will lead to wasted time and inconsistency, despite

that it does not significantly risk the system being suboptimal. Unfortunately, the requirements engineer may not have the authority to make such a decision, and so UR–SysR repetition may prevail.

To conclude RQ4 (is UR–SysR repetition a problem?), the most concerning problem posed by repetition between a UR and a SysR is the increased risk that the stakeholders will not consider the system "fit for purpose" (as discussed in the summary of Table 1). The less consequential but more tangible problem is the increased redundancy within the requirements set. This will no doubt increase the reading and writing costs, in addition to increasing the risk of inconsistencies—a significant RE defect [25]. For an example of such an incurred cost, Gilb and Graham claim that requirements inspection occurs at approximately 10 words per minute [84]. According to that rate, inspecting the requirement descriptions of all UR–SysR pairs (55) in Project A ($\sim$790 words) would take 1 h and 19 min, of which, at the very least (TF–IDF distance score <0.1), $\sim$24 min could be considered wasted due to repetition. (Reading costs are emphasized since the requirements will likely be read more than they will be written.) However, the cost of those 24 min is likely to be perceived as higher than normal, since they provide no reward (i.e., new information) and so demotivate the reader. Indeed, if even some of those 24 min occur sequentially, the reader may give up reading the UR and SysR requirements as a whole. Also, we are assuming that increases in reading effort are linear, but it could be argued that effort expended is higher for very similar UR–SysR pairs, since humans are inefficient at finding subtle differences that could nevertheless be important [3]. When considering the total cost of this redundancy, it should also be considered that a typical requirements inspection meeting involves several participants (including the customer on some occasions), and that each developer working on the project is expected to read the requirements to understand the problem they are to solve [13].

Caveat to the above conclusion is the hypothetical situation where either the UR or the SysR is conflated to perform the role of both describing the problem and also the abstract solution. There is also the perhaps more likely scenario where one of the documents (UR or SysR) contains a mixture of URs and SysRs [7]. Neither occurrence was apparent in this case study, but any conclusions reached after detecting such repetition would be affected. In the former scenario, the repetition would not indicate poor problem or solution exploration, and in the latter case, the "mode of blame" (in Sect. 4.2) is reversed.

4.4 RQ5: Can the problems be reduced (current/future projects)?

This section proposes guidelines for reducing the problems associated with UR–SysR repetition in current projects, i.e., where repetition already exists (1), and in future projects (2).

4.4.1 *Reducing problems in current projects (where UR–SysR repetition exists)*

Any increased risk that the product will not be accepted by the customer can only be reliably reduced through communication with the stakeholders (and possibly developers). Attempting to translate incorrect URs or SysRs without doing so (e.g., with the use of previous experience) may lead to the correct requirements, but, the risk will not have been reduced by doing so. Hence, this section will focus on the most tangible and therefore the most reducible problem caused by redundancy: increased reading time.

Software developers do not often read requirements documents to the extent that requirements engineers would like. For example, Spolsky writes that "*the biggest complaint you'll hear from teams that do write specs is that nobody reads them*" [85]. Ambler adds to this that "*some people naively think that developers do their work based on what is in the requirements document. If this was true, wouldn't it be common to see printed requirements documents open on every programmers desk?*" [86]. Indeed, with the common management mantra that "Weeks of coding can save hours of planning" [87], and the pressure placed on developers to "get on to the real work, designing and programming" [87], this is not surprising. The incentive to read requirements documents will be even less if there is a high degree of repetition between them. Reducing the non-value-added reading tasks can only contribute toward motivating developers to read requirements. Hence, we propose that UR–SysR links can be manually or automatically tagged with an "essentiality rating" to streamline the requirements reading process.

Essentiality ratings and tracks were first proposed to treat the problem of information overload in large documents and webpages [88]. The essence of the concept is that sections of content are marked with a rating of essentiality on a numerical 1–10 scale, and/or for a specific intended audience (i.e., a track), e.g., for a "UI developer". Then, any nonessential or irrelevant content is filtered out for the reader. This concept could straightforwardly be applied to requirements documents to filter out repeated URs or SysRs from the set of requirements that the reader should read, where low UR–SysR pair TF–IDF distance scores could be translated to low essentiality scores. For example, the TF–IDF distance score of 0.3 would map to essentiality rating 3. (Note that the inverse is not applicable, since high essentiality cannot be determined by low similarity.) Alternatively, a requirements engineer who is authoring or reviewing a requirements document could manually estimate essentiality scores. For example, where

the UR is actually the same as the SysR (i.e., the third mode of blame described in Sect. 4.2), the requirements engineer would recognize that the UR is not essential reading if the SysR is read (or vice versa).

As aforementioned, manual inspection of this case study's TF–IDF scores and the triviality of the actual differences indicated that the textual differences in UR–SysR pairs represented by TF–IDF distance scores lower than 0.3 were insignificant, (i.e., reading both requirements in such pairs does not add significant information). Hence, at the 0.3 TF–IDF distance threshold (above which non-repeated UR–SysR pairs existed), 43 % of Project A's total UR–SysR description pairs and 16 % of project B's UR–SysR description pairs could be considered "trivial" reading. (The percentages refer to the number of traceability links between URs and SysRs, as defined in Table 4.) If the requirements documents are being authored or reviewed, then these "trivial" UR–SysR pairs should be checked for missing information on the problem or the solution, among the other risks outlined in Sect. 4.3. Stated in terms of individual requirements rather than pairs of requirements, this "trivial reading" statistic is likely to be higher where there are more URs than UR–SysR traceability links. For example, 17 (56 %) of Project A's 30 UR descriptions could be filtered out on the condition that the reader reads the set of SysRs. (Note that firstly, traceability between URs and SysRs is many-many [11], and so a UR is redundant only if it does not feature in a UR–SysR pair that contains non-trivial differences. Secondly, given the choice of reading a UR or a SysR in a repeated UR–SysR pair, the SysR should be chosen since it is likely to be the most recent version.)

The benefit of filtering UR–SysR pairs by essentiality ratings must be balanced with the risk of "false positives", i.e., the risk that an important but marked-as-trivial difference is filtered out. Such a risk would not likely be entertained when reading requirements for contractual purposes, and hence the threshold could be set at 0.01 to filter out only the "identical" UR–SysR pairs. Conversely, false negatives do not pose such a significant problem, since the worst effect is that a repeated requirement is read, which is equal to the as-is situation, and has not cost the user any significant time due to the automated nature of the analysis. For example, the UR–SysR pair with the TF–IDF score of 0.4 in Table 11 was considered to be a repeated UR–SysR pair by human inspection, but is not classified as such by the 0.3 TF–IDF cutoff point. The consequence that the redundant requirement would not be filtered out for the reader was not considered to be a practical problem by the three interviewed stakeholders, despite that it is a valid direction of future research.

The appropriate TF–IDF distance threshold may slightly vary between projects, depending on factors such as the size of the vocabulary, the wordiness of the requirements, and for example, whether the project is safety critical. Hence, the requirements engineer should not assume that the threshold is a universal constant between projects, otherwise important information in URs or SysRs may be disregarded. Therefore, this process of determining the TF–IDF cutoff threshold (where thresholds >0.1 are desired) represents the majority of the time required to perform the automated UR–SysR repetition analysis, since the calculation of the TF–IDF scores takes seconds. Visualizing the differences between UR–SysR pairs, as shown in Table 11 (and as performed by our tool), is likely to be useful for determining the appropriate threshold. More rigorous approaches for determining optimal TF–IDF distance thresholds to improve conclusion validity are outlined by Falessi [61]. However, their relatively high cost of application (which threatens the usability and therefore the usefulness of the automated approach) means that in practical use, thresholds tend to be "chosen by common sense and without any rigorous criteria"—but not without reason [61].

4.4.2 *Reducing problems in future projects (where UR–SysR repetition does not yet exist)*

A great deal of UR–SysR repetition is caused by unknown stakeholder needs (the top cause of UR–SysR repetition in Fig. 6), and hence where both the URs and SysRs describe solutions. If stakeholders want software that solves problems in the most innovative and cost-effective way, and if requirements engineers want software that generates value rather than merely being of "good quality", then resources from both sides should be committed to stakeholder requirements (i.e., needs) elicitation. To motivate stakeholders, evidence to show that poor requirements engineering and low stakeholder communication are top causes of project failure could be useful for motivating this change, e.g., [78, 89–91]. Motivating requirements engineers to elicit stakeholder needs rather than rushing to specify solutions, requires clarification that the "purpose of the requirements process is to add business value" [92], as well as fostering responsibility and reward mechanisms around this.

A common response from stakeholders when attempting to elicit their requirements is "I don't know how to tell you, but I'll know it when I see it" (IKIWISI) [93]. Consequently, Boehm proposes that requirements (in this context, URs) should focus on "how the new system will add value for each stakeholder" [93]. If stakeholders are unable to express their needs, then requirements engineers should attempt the use of well-established techniques for problem and goal elicitation, e.g., using scenarios, ethnography, interviews, questionnaires,

protocol/domain/task analysis, goal graph analysis, and so on [25]. If the IKIWISI problem is obscuring "what the solution should be", then requirements engineers should attempt to elicit feedback from prototypes and early releases of the software [94]. Finally, market-driven software projects have slightly different challenges (e.g., less accessible stakeholders), for which the reader is referred to Karlsson et al.'s work [95].

Internal (i.e., the requirements engineer's organization) standards or processes may stipulate that a UR and a SysR document be created regardless of the project's context (i.e., its complexity, risk, "obviousness" of solution, etc.). Only well-argued cases for the project's low complexity and a change in internal policy can reduce this source of UR–SysR repetition. If a customer's standard requires both URs and SysRs regardless of the project's context, then while the same argument applies, putting it into practice may be more difficult, due to the "customer is always right" mindset.

Finally, repetition between UR–SysR pairs caused by ignorance of the roles URs and SysRs play in the systems engineering life cycle should be treated by training the requirements engineers (or perhaps their trainers), especially with examples of good and bad practice rather than regurgitated "what not how" mantras. Crucially, motivation from management should be provided to maintain adherence to the true content of the standards (rather than misinterpreted versions sometimes found in textbooks or organizational training manuals). This will not happen quickly, since as aforementioned, requirements engineering (and indeed many other software "engineering" techniques [96]) are rarely considered a "top priority" in current software engineering practice.

5 Related work

As far as we are aware, Kovitz's discussion of UR–SysR repetition (among a considerably larger set of software requirements engineering guidelines [14]) was the first treatment of the phenomena in the literature. Kovitz implies that many organizational standards mandate that their requirements engineers wastefully create UR and SysR documents comprised of near-identical repetition, as exemplified by requirement #7 in Table 2. Alexander later writes on the topic concluding that "*you don't need to be told (do you?) that a lot of time and paper is being wasted if your organisation is duplicating the requirements in that way*" [15]. Alexander explains that in his experience, URs are often repeated from the SysRs because requirements engineers often believe that their only role is to describe system functionalities and qualities. As a consequence, he notes that the right-hand side of the V-model (verification and validation) is disrupted, since if both the SysRs and the URs describe a solution, then only verification can take place. Similarly, Davies notes that if only system requirements are captured, then "*When it comes to the validation stage, the end-user may throw out the system as not fit for purpose, even if 'It does exactly what it says on the tin'*" [7].

While both Kovitz and Alexander imply that UR–SysR repetition has frequently occurred (e.g., "…*too often* led tired engineers to write…" [15]), they provide neither an investigation nor an analysis of the real-world problem and its causes. Furthermore, neither Kovitz nor Alexander mention repetition between the different composite fields of a requirement other than the description field (i.e., the "main" field). This is significant since complete duplication in one field does not infer that the UR or SysR is completely redundant. Finally, both Kovitz and Alexander propose ways of avoiding UR–SysR repetition. Kovitz [14] proposes that requirements should be organized by their subject matter rather than by their "level of detail". However, this neglects the issues that creating separate UR and SysR documents address, i.e., for the different reasons outlined in Table 1. On the other hand, Alexander proposes that URs should be renamed to "business problems", and SysRs to "system solutions". This seems more semantically correct in light of the definitions in Table 1 as well as the criticisms in the literature of the term "User" in "User Requirement" [25, 26]. However, the "business" term is also a restrictive subtype of stakeholder requirement, and "solution" implies that the problem will be (fully) solved, rather than treated [97]. Finally, Kovitz and Alexander's discussions are over a decade old, and so prior to this research there was a need to understand the pertinence of the UR–SysR repetition problem in the current industry environment.

Juergens et al. [3] analyze the degree of "copy&paste" clones within requirements documents with their ConQAT tool. They report an average "blow-up" (i.e., "the ratio of the total number of words to the number of redundancy free words") of 13.5 % across 28 software requirements documents from different industries. In their conclusion, they propose that requirements document blow-up of more than 5 % should be considered "as a warning signal for potential future problems". However, ConQAT does not parse requirement documents into individual requirements (nor their traceability links), but rather searches through a plaintext requirements document for contiguous fixed-length sequences of words (in their study, 20 words) that occur more than once. Thus, their work is concerned with verbatim duplication of requirement document text, rather than requirements "which have been copied but slightly reworded in later editing steps" [3], which we have found to be very common in UR–SysR repetition. Furthermore,

their work does not consider traceability to other requirements, nor repetition between a project's different requirements documents, and is therefore unable to model nor analyze UR–SysR pairs.

Due to the commonality of natural language specifications and documentation in software projects, natural language processing techniques have been applied to problems in various areas of software engineering. For example, Runeson et al. [57] successfully detect two-thirds of duplicate software defect reports using vector space representation and term frequency weighting (i.e., TF rather than TF–IDF). Closer to our work, numerous researchers have tackled various requirements engineering problems with similar techniques, as the remainder of this section discusses.

Natt och Dag et al.'s ReqSimilie tool [98] suggests traceability links between requirements based on the assumption that requirements containing similar words are likely to be related since requirements engineers strive for consistent use of terminology. Regardless of the validity of that assumption for URs and SysRs, the focus of their work is the opposite to ours. ReqSimilie does not consider existing traceability links, and attempts to find requirements worth linking to, whereas our approach assumes traceability links exist and attempts to find those that do not add different information. Hence, using ReqSimilie, it is not possible to know the degree of repetition between UR–SysR pairs, since the information that defines pairs of URs and SysRs is disregarded. Furthermore, ReqSimilie's assumption that requirements are stored in a database rather than in documents means that it would not be easily usable by the majority of requirements engineers who use word processing or spreadsheet software, as is the case in our case study, and in numerous other surveys on RE practice [52, 99].

Similar to the ReqSimilie tool is Cleland-Huang et al.'s "Poirot Trace Maker" [100], which automatically suggests traceability links between requirements, design, and code artifacts, (e.g., requirement text to UML sequence diagram messages to Java method names). Poirot Trace Maker also uses vector space representation to compute the degree of artifact similarity, and so it is similar to ReqSimilie in technique but not in intent. Interestingly, Cleland-Huang et al. cite Hull et al.'s example UR–SysR pair (#1 in Table 2) as an example of requirements that cannot be automatically traced, since only the fairly generic terms *(the, shall, to, vehicle)* are shared between them. It is therefore implied that repetition of rare terms between URs and SysRs is advantageous for the sake of automated traceability. However, this would violate the guidelines on good URs and SysRs provided in Table 1, e.g., since the language used to describe a stakeholder's problem and an engineer's solution should be different. Instead, Cleland-Huang et al. propose that the use of Hull et al.'s "satisfaction arguments" [11] (i.e., a description of why the SysR satisfies the UR) could increase the recall rate, since satisfaction arguments tend to contain terminology from both the URs and SysRs. We are not optimistic about this from this paper's viewpoint (repetition), or from a pragmatic viewpoint, since:

1. A significant amount of Hull et al.'s satisfaction arguments are repeated from the UR and the SysR, introducing redundancy and hindering modifiability;
2. Satisfaction arguments are a form of manual traceability between a URs and SysRs, and so by the time a SysR could be automatically linked by using the text from a satisfaction argument, it is already manually linked.

If repetition between URs and SysRs is to be avoided, then automated traceability between them requires "*problem → solution*" knowledge. This could be inferred from traceability links within previous projects, or could be explicitly and ontologically defined (e.g., encoded with "means-end" links in GRL [41]). For example in the context of Hull et al.'s UR–SysR pair (#1 in Table 2), "power to all wheels" *is-a-solution-to* "deployment on wet mud", and "wet mud" *is* "terrain type 4A" (the latter should ideally be defined within the UR for the sake of completeness).

Lami's QuARS tool [101] automatically detects linguistic defects that could cause ambiguity, understandability, or completeness problems. However, QuARS is concerned with the quality of single requirement statements, and so completeness is assessed only in the context of each individual requirement, as opposed to the completeness of a SysR set relative to its coverage of a UR set. Similarly, Park et al. [102] propose an automated requirements analysis system to detect ambiguity and incompleteness in requirements documents. Closer to our problem is Park et al.'s claim that their system can trace dependency and reduce inconsistency between "a sentence in a high-level document and a sentence in a low level document". However, in the examples Park et al. use to explain their claim, the "level" of a requirement refers to its level of decomposition rather than its domain (problem or solution), or other dimension of UR–SysR difference listed in Table 1. As such, Park et al.'s claims are not applicable to SysR documents derived from UR documents, since *"unlike decomposed requirements, the statements of the derived requirements are different from those of the original requirements",* to quote Sage and Rouse [28].

Finally and most recently, Ferrari et al. [103] explore the use of the Sliding Head–Tail Component clustering algorithm (rather than the vector space model of similarity) to propose a new requirement document structure, in order to optimize the structure's "requirements relatedness" and

"sections independence" qualities. Despite not being directly applicable to this paper's topic, it would be interesting to apply their technique to combine UR and SysR documents, e.g., to automatically generate a requirements document that presents both sets of requirements, structured by their relatedness.

6 Conclusion and future work

Duplication within requirements documents is clearly considered to be a quality defect; international standards require that within "the set of stakeholder, system, and system element requirements … requirements are not duplicated" [22], and that software requirements "not be redundant" [4]. In this paper, we have proposed that repetition between URs and SysRs can also indicate a more serious defect: *incompleteness* of the requirement set. In other words, we find UR–SysR repetition concerning since it indicates that one set of requirements (URs or SysRs) is not complete. This is especially troubling where the solution description is repeated, since Jackson [36] makes it quite evident that the main concern in RE should be describing the application domain (i.e., problem) rather than the machine (i.e., solution). In other words, good engineers can derive solutions from problems, but the converse is less likely.

Aside from anecdotes and examples, there has been no published investigation to show if and why repetition between URs and SysRs occurs. We have presented a case study with the intention of adding to the "chain of evidence" [64]. We found that 75 % of the survey participants had seen UR–SysR repetition in at least half of their projects (Sect. 4.1.1). Then, we found that on average over our sample projects, one-third of the UR–SysR pairs had description fields that contained significant repetition (Sect. 4.1.2), while the UR–SysR pairs' fit criterion and rationale fields exhibited roughly twice as much repetition. Finally, our survey found that unknown stakeholder needs combined with the internal (i.e., the requirements engineer's) organization's context-independent stipulation for UR and SysR documentation were the most popular reasons for UR–SysR repetition (Sect. 4.2).

Based on the results of our research questions, we propose the following recommendations:

- Despite that there will likely be a significant amount of repetitious UR–SysR pairs within UR and SysR documents (especially within the non-primary attributes such as the rationales or fit criteria), stakeholders including developers should still read both, since numerous non-trivially different and informative UR–SysR pairs are also likely to exist. UR–SysR pairs where the UR is related to more than one SysR (or vice versa) are more likely to provide information to make their reading as a pair worthwhile. Where a UR has only one related SysR, the most recent of the two requirements should be read.
- Requirements engineers should ensure that stakeholder needs have been adequately elicited and analyzed if repetition between UR–SysR pairs exists. Not understanding the problem is a significant and well-established software project failure risk, that is also known to cause problems such as scrap and rework, poor innovation, suboptimal decisions, or value failure.
- Training on the role of URs and SysRs should be provided where repetition exists due to ignorance of their roles in the engineering process. Perhaps more important is that management should motivate and monitor adherence to RE standards (but not for rigor on principle).
- Checking for UR–SysR repetition can be performed with little effort using our software tool [16], and hence could be built into future requirements quality inspections. Where a significant degree of UR–SysR repetition exists, the tool could be used to streamline the reading process by filtering out trivial URs or SysRs.

Finally, it is important to remark upon the applicability of this paper to software engineering practice outside of the defense industry. Firstly, as Table 1 exemplifies by having only one source from the defense industry, the recommendation to distinguish between URs and SysRs is not limited to defense, but is advocated in general software and systems engineering practice. Indeed, in Davies' article "Ten Questions to Ask Before Opening the Requirements Document" (targeted to the general field of systems engineering), the first question is *"Is it a User Requirement, [or] a System Requirement…?"* [7], while in Wiegers' article "10 Requirements Traps to Avoid" (targeted to the general field of software requirements engineering, i.e., not defense or systems), the first trap is *"that project stakeholders refer to 'the requirements' with no qualifying adjectives … [and] as a consequence, important stakeholder expectations might go unstated and unfulfilled"* [104]. So, if one can agree that the concepts behind URs and SysRs are applicable to software and system engineering, then the main question of this paper's applicability becomes "how frequently are software systems engineered?" On this topic, and in the face of the growing Agile software development community, software engineering expert Demarco [96] recently wrote that *"I'm gradually coming to the conclusion that software engineering is an idea whose time has come and gone. I still believe it makes excellent sense to engineer software. But that isn't exactly what software engineering has come to*

mean". Indeed, as an indication of popularity, according to "Google Trends" [105] the term "user requirement" (or "stakeholder requirement") receives six times less search interest than Agile's equivalent "user story" term in 2015, whereas they were roughly equal in 2006. However, search interest in the former term has remained approximately constant over the last decade, and the use of systems engineering standards such as ISO 29148 (one of the most authoritative proponents of URs and SysRs) is still adopted in many industries where there is high complexity, expenditure, and risk (e.g., in aero, automotive, or military engineering) [21, 25]. (Common reasons for not adopting Agile include costs of frequent communication between end-users and developers [79], or a need for verification prior to coding, among others discussed in [25, 106].)

In a comparison of Systems Engineering with Agile, Turner concludes that *"there are still no silver bullets* [107]" [108]; the key concerns behind traditional "UR–SysR" requirements engineering (summarized in Table 1) still exist in modern software development. Merisalo-Rantanen et al. [109] even argue that much of modern techniques are largely a case of *"Old Wine in New Bottles"*. Indeed, an Agile user story using Cohn's [94] template of *"As a <user type>, I want to <goal> so that <reason>"* could be interpreted such that the <reason> field maps to the concerns of URs, while the <goal> field maps to user-task-oriented SysRs. Interestingly, some repetition between these "fields" of user stories is apparent, even in examples on Cohn's website [110].

Ultimately (and as the division of Sect. 4.4 into current practice and future practice makes clear), the ideal goal is to not need a tool for assessing the quality of requirements, since the requirements processes should be optimized for each software development context. However, in the past and most likely in the future, people will make mistakes and apply techniques in contexts ill-suited to them [111], either by choice or due to constraints such as organizational standards. Furthermore, the current and future trend where "commercial off-the-shelf (COTS) capabilities [i.e., solutions] determine requirements" [108], serves to increase the importance of ensuring that those capabilities are related to real stakeholder needs not described in terms of those capabilities. Overall, this paper's topic is pertinent in the current and foreseeable future state of practice, despite that it would not be in an ideal world.

As future work, a number of interesting research questions were identified during the project:

- Is there a correlation between software project success (or the amount of rework or ad hoc communication required to make it successful) and the compliance of URs and SysRs to requirements engineering standards?
- How common is UR–SysR repetition (or even the production of UR and SysR documents) in software projects within different industry contexts (i.e., non-defense)?
- Is the degree of UR–SysR repetition correlated with the number of authors creating the project's requirements documents? In other words, is better UR and SysR separation achieved when different authors write them?
- Can a greater number of trivial differences be identified in UR–SysR pairs by extracting and comparing semantic information, such as by using domain ontologies and lexical databases (e.g., Princeton's WordNet), or by comparing "Parts of Speech" (as are extracted from requirements in [112])? This would explore whether RE authors purposely replace SysR terms with synonymous terms to form a UR that appears to be different.
- Can machine learning techniques (e.g., a naïve Bayesian classifier) be used to better classify UR–SysR pairs as to whether useful information is added by a UR–SysR link?
- What are the implications of repetition of natural language within requirements engineering models, e.g., between TROPOS goals and plans, as is visible in [113]?
- How effective (i.e., credible [61]) is the proposed approach at filtering out trivial URs or SysRs (the answer to RQ5) in a wider variety of requirements documents? That is, how many trivial UR–SysR pairs (assessed manually by stakeholders) are not filtered out, and how many non-trivial UR–SysR pairs are filtered out?

Acknowledgments The authors are grateful to LSC Group for allowing data collection and publication of results. Special thanks are owed to LSC Group's Chris Lambert, Dr. James Nyambayo, and Dr. Ann Meads for the insightful discussions on the topic. Warm thanks are due to the reviewers of this paper whose suggestions resulted in the clarification of many points, and to both Springer's editors and Sachi Jain for proofreading and copy-editing.

References

1. Marinelli V, Laplante PA (2008) Requirements engineering: the State of the practice revisited. Penn State University, University Park
2. Buede DM (2000) The engineering design of systems: models and methods. Wiley, New York
3. Juergens E, Deissenboeck F, Feilkas M et al (2010) Can clone detection support quality assessments of requirements specifications? In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 2, pp 79–88
4. IEEE (1998) IEEE Std 830-1998: IEEE recommended practice for software requirements specifications. IEEE-SA Standards Board

5. Hunt A, Thomas D (1999) The pragmatic programmer: from journeyman to master, 1st edn. Addison-Wesley Professional, Boston
6. Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: 31st IEEE international conference on software engineering, pp 485–495
7. Davies P (2004) Ten questions to ask before opening the requirement document. In: 14th Annual international symposium systems engineering: managing complexity and change
8. Forsberg K, Mooz H (1995) The relationship of systems engineering to the project cycle. Eng Manag J 4:36–43
9. Jureta IJ, Mylopoulos J, Faulkner S (2008) Revisiting the core ontology and problem in requirements engineering. In: 16th IEEE international requirements engineering conference, pp 71–80
10. Lin D (1998) An information-theoretic definition of similarity. In: ICML, pp 296–304
11. Hull E, Jackson K, Dick J (2011) Requirements engineering, 3rd edn. Springer, London
12. Herzwurm G, Schockert S, Pietsch W (2003) QFD for customer-focused requirements engineering. In: 11th IEEE international requirements engineering conference, pp 330–338
13. Clements P, Bass L (2010) Using business goals to inform a software architecture. In: 18th IEEE international requirements engineering conference, pp 69–78
14. Kovitz BL (1998) Practical software requirements: a manual of content and style. Manning Publications, Greenwich
15. Alexander I (2002) Being clear: requirements are either needs or specifications. Newsl BCS Requir Eng Spec Group 25:10–12
16. Ellis-Braithwaite R (2014) GoalViz tool. In: GoalViz tool. http://www.goalviz.info/DUP/index.html. Accessed 5 Jun 2014
17. Harwell R, Aslaksen E, Hooks I et al (1993) What is a requirement? In: 3rd Annual international symposium of NCOSE, pp 17–24
18. Wieringa RJ (2005) Requirements researchers: are we really doing research? Requirements Eng 10:304–306. doi:10.1007/s00766-005-0013-6
19. Davis AM (1993) Software requirements: objects. Prentice Hall, Functions and States
20. Stevens R, Brook P, Jackson K, Arnold S (1998) Systems engineering: coping with complexity. Pearson Education, New york
21. Sommerville I (2011) Software engineering, 9th edn. Pearson Education, Boston
22. IEEE (2011) ISO/IEC/IEEE 29148:2011 Systems and software engineering—life cycle processes—requirements engineering. In: IEEE-SA Standards Board
23. Pyster A, Olwell D (2013) The guide to the systems engineering body of knowledge (SEBoK), v. 1.2. The Trustees of the Stevens Institute of Technology, Hoboken, NJ
24. IIBA (2009) A guide to the business analysis body of knowledge (BABOK guide). International Institute of Business Analysis, Whitby, ON
25. Van Lamsweerde A (2009) Requirements engineering: from system goals to UML models to software specifications. Wiley, New York
26. Alexander I (1999) Is there such a thing as a user requirement? Requirements Eng 4:221–223. doi:10.1007/s007660050022
27. Jackson MA (1995) Software requirements and specifications: a lexicon of practice, principles, and prejudices. ACM Press, New York
28. Sage AP, Rouse WB (2009) Handbook of systems engineering and management, 2nd edn. Wiley, Hoboken
29. Boehm BW, Basili VR (2001) Software defect reduction top 10 list. Computer 34:135–137
30. MoD (2012) ISO15288 Technical process: what is the relationship between requirements and design? "Engineering" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/engineering/content/tech_process/se_proc_req_design.htm. Accessed 11 Jul 2013
31. MoD (2012) User requirements principles "Requirements and acceptance" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/urprinciples.htm. Accessed 24 Sep 2012
32. MoD (2012) System requirements document (SRD) principles "Requirements and acceptance" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/srdprinciples.htm. Accessed 24 Sep 2012
33. MoD (2012) User requirements (ur): recommended core attributes "Requirements and acceptance" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/urdcoreattributes.htm. Accessed 15 Mar 2014
34. MoD (2012) System requirements principles "Requirements and acceptance" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/srprinciples.htm. Accessed 24 Sep 2012
35. MoD (2012) System requirements document (SRD) "Requirements and acceptance" acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/srdstructure.htm. Accessed 15 Mar 2014
36. Jackson MA (2000) Problem frames and methods: structuring and analyzing software development problems. Addison-Wesley, Harlow
37. Berry DM (2009) What, Not how?: the case of specifications of the New York Bagel. Ann Improbable Res 15:6–10
38. Zimmerman MJ (2010) Intrinsic vs. extrinsic value. In: Stanford encyclopedia of philosophy. http://plato.stanford.edu/entries/value-intrinsic-extrinsic. Accessed 2 Oct 2013
39. Zave P, Jackson MA (1997) Four dark corners of requirements engineering. ACM Trans Softw Eng Methodol 6:1–30
40. Kaindl H, Svetinovic D (2010) On confusion between requirements and their representations. Requirements Eng 15:307–311. doi:10.1007/s00766-009-0095-7
41. Amyot D, Horkoff J, Gross D, Mussbacher G (2009) A lightweight GRL profile for i* modeling. Advances in conceptual modeling-challenging perspectives. Springer, Berlin, pp 254–264
42. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Program 20:3–50. doi:10.1016/0167-6423(93)90021-G
43. Kavakli E (2002) Goal-oriented requirements engineering: a unifying framework. Requir Eng 6:237–251
44. Aurum A, Wohlin C (2007) A value-based approach in requirements engineering: explaining some of the fundamental concepts. In: Sawyer P, Paech B, Heymans P (eds) Requirements engineering: foundation for software quality, pp 109–115
45. Robertson S, Robertson J (2012) Mastering the requirements process, vol 3. Addison-Wesley Professional, Reading
46. Glinz M (2007) On non-functional requirements. In: 15th IEEE international requirements engineering conference
47. Rost J (2006) Are "best practices" requirements documents a myth? IEEE Softw 23:103–104
48. MoD (2012) System requirements document (SRD) part 3: System requirements "Requirements and acceptance" Acquisition operating framework (AOF). https://www.aof.mod.uk/aofcontent/tactical/randa/content/srdpart3.htm. Accessed 15 Jan 2014
49. Ministry of Defence (2005) MODAF community of interest deskbook. MoD, UK
50. Monperrus M, Baudry B, Champeau J et al (2011) Automated measurement of models of requirements. Software Qual J 21:3–22
51. Asuncion HU, Francois F, Taylor RN (2007) An end-to-end industrial software traceability tool. In: 6th Joint meeting of the

european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp 115–124
52. Bouillon E, Mäder P, Philippow I (2013) A survey on usage scenarios for requirements traceability in practice. In: Proceedings of the 19th international conference on requirements engineering: foundation for software quality. Springer, Berlin, pp 158–173
53. Boehm BW (2005) Value-based software engineering: overview and agenda. In: Biffl S et al (eds) Value-based software engineering. Springer, Berlin, Heidelberg
54. Ncube C, Lockerbie J, Maiden N (2007) Automatically generating requirements from i* models: experiences with a complex airport operations system. In: 13th international working conference on requirements engineering: foundation for software quality. Springer, Berlin, pp 33–47
55. Kanaris I, Kanaris K, Houvardas I, Stamatatos E (2007) Words versus character n-grams for anti-spam filtering. Int J Artif Intell Tools 16:1047–1067
56. Manning CD, Schutze H (2001) Foundations of statistical natural language processing. MIT Press, Cambridge
57. Runeson P, Alexandersson M, Nyholm O (2007) Detection of duplicate defect reports using natural language processing. In: 29th International conference on software engineering. IEEE computer society, Washington, DC, USA, pp 499–510
58. Rajaraman A, Ullman JD (2012) Data mining. Mining of massive datasets. Cambridge University Press, Cambridge
59. Apache (2012) StopAnalyzer (Lucene 4.0.0 API). http://lucene.apache.org/core/4_0_0/analyzers-common/org/apache/lucene/analysis/core/StopAnalyzer.html. Accessed 24 Mar 2014
60. Mavin A, Wilkinson P, Harwood A, Novak M (2009) Easy approach to requirements syntax (EARS). In: 17th IEEE international requirements engineering conference, pp 317–322
61. Falessi D, Cantone G, Canfora G (2013) Empirical principles and an industrial case study in retrieving equivalent requirements via natural language processing techniques. IEEE Trans Softw Eng 39:18–44
62. tfidf.com (2010) Tf-idf: a single-page tutorial—information retrieval and text mining. http://www.tfidf.com/. Accessed 16 Nov 2014
63. Ellis-Braithwaite R (2013) Analysing the assumed benefits of software requirements. In: 19th International working conference on requirements engineering: foundation for software quality, proceedings of the workshops and the doctoral symposium, pp 207–214
64. Runeson P, Höst M (2008) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14:131–164. doi:10.1007/s10664-008-9102-8
65. Flyvbjerg B (2006) Five misunderstandings about case-study research. Qual Inq 12:219–245. doi:10.1177/1077800405284363
66. Berk RA, Freedman DA (2003) Statistical assumptions as empirical commitments. In: Blomberg TG, Cohen S (eds) Law, punishment, and social control: essays in honor of Sheldon Messinger, 2nd edn. Aldine de Gruyter, New York, pp 235–254
67. Hair JFJ, Wolfinbarger M, Money AH et al (2015) Essentials of business research methods. Routledge, London
68. Easterbrook S (2006) Doctoral symposium keynote: you gotta have a theory. ACM SIGSOFT Foundations of Software Engineering, Portland, OR
69. Tversky A, Kahneman D (1973) Availability: a heuristic for judging frequency and probability. Cogn Psychol 5:207–233
70. Umbach PD (2004) Web surveys: best practices. New Dir Inst Res 2004:23–38. doi:10.1002/ir.98
71. Gerring J (2006) Case study research: principles and practices, 1st edn. Cambridge University Press, New York
72. Goknil A, Kurtev I, van den Berg K (2008) A metamodeling approach for reasoning about requirements. In: Model driven architecture–foundations and applications, pp 310–325
73. Adedjouma M, Dubois H, Terrier F (2011) Requirements exchange: from specification documents to models. In: 16th IEEE international conference on engineering of complex computer systems. IEEE, pp 350–354
74. Alias-i (2011) LingPipe 4.1.0. http://alias-i.com/lingpipe/. Accessed 24 Feb 2014
75. Fraser N (2012) Google-diff-match-patch—diff, match and patch libraries for plain text. http://code.google.com/p/google-diff-match-patch/. Accessed 25 Jan 2014
76. Ellson J, Gansner ER, Koutsofios E et al (2003) Graphviz and dynagraph–static and dynamic graph drawing tools. Graph Draw Softw 127:148
77. Core Team R (2014) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna
78. The Standish Group (2010) CHAOS summary for 2010. Standish Group International, Boston, MA
79. Cao L, Ramesh B (2008) Agile requirements engineering practices: an empirical study. IEEE Softw 25:60–67. doi:10.1109/MS.2008.1
80. Ott R, Longnecker M (2015) An introduction to statistical methods and data analysis. Cengage Learning, Boston, US
81. Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Sov Phys Dokl 10:707–710
82. Peppard J, Ward J, Daniel E (2007) Managing the realization of business benefits from IT investments. MIS Q Exec 6:1–11
83. Dorfman M (1999) System and software requirements engineering. In: Thayer RH, Dorfman M (eds) SEI interactive. IEEE Computer Society Press, Los Alamitos, CA, pp 7–22
84. Gilb T, Graham D (1993) Software inspection. Addison Wesley, Boston
85. Spolsky J (2000) Painless functional specifications—part 4: tips—Joel on software. http://www.joelonsoftware.com/articles/fog0000000033.html. Accessed 3 Dec 2012
86. Ambler SW (2012) Examining the "Big Requirements Up Front (BRUF) Approach." In: Agile Modeling. http://www.agilemodeling.com/essays/examiningBRUF.htm. Accessed 3 Dec 2012
87. Berry DM, Damian D, Finkelstein A et al (2005) To do or not to do: if the requirements engineering payoff is so good, why aren't more companies doing it? In: 13th IEEE international requirements engineering conference, p 447
88. Atkinson MT, Dhiensa J, Machin CHC (2006) Opening up access to online documents using essentiality tracks. In: International cross-disciplinary workshop on web accessibility (W4A): building the mobile web: rediscovering accessibility? ACM, New York, NY, USA, pp 6–13
89. Abelein U, Paech B (2013) Understanding the influence of user participation and involvement on system success—a systematic mapping study. Empir Softw Eng. doi:10.1007/s10664-013-9278-4
90. Damian D, Chisan J (2006) An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. IEEE Trans Softw Eng 32:433–453. doi:10.1109/TSE.2006.61
91. Verner J, Cox K, Bleistein S, Cerpa N (2005) Requirements engineering and software project success: an industrial survey in Australia and the US. Australas J Inf. doi:10.3127/ajis.v13i1.73
92. Favaro J (2002) Managing requirements for business value. IEEE Softw 19:15–17
93. Boehm BW (2000) Requirements that handle IKIWISI, COTS, and rapid change. Computer 33:99–102
94. Cohn M (2005) Agile estimating and planning. Prentice Hall, Upper Saddle River, NJ

95. Karlsson L, Dahlstedt ÅG, Regnell B et al (2007) Requirements engineering challenges in market-driven software development—an interview study with practitioners. Inf Softw Technol 49:588–604. doi:10.1016/j.infsof.2007.02.008
96. DeMarco T (2009) Software engineering: an idea whose time has come and gone? IEEE Softw 26:96. doi:10.1109/MS.2009.101
97. Wieringa RJ (2009) Design science as nested problem solving. In: Proceedings of the 4th international conference on design science research in information systems and technology, p 8
98. Natt och Dag J, Regnell B, Gervasi V, Brinkkemper S (2005) A linguistic-engineering approach to large-scale requirements management. IEEE Softw 22:32–39. doi:10.1109/MS.2005.1
99. Juristo N, Moreno AM, Silva A (2002) Is the European industry moving toward solving requirements engineering problems? IEEE Softw 19:70–77. doi:10.1109/MS.2002.1049395
100. Cleland-Huang J, Settimi R, Romanova E et al (2007) Best practices for automated traceability. Computer 40:27–35. doi:10.1109/MC.2007.195
101. Lami G (2005) QuARS: a tool for analyzing requirements. Carnegie Mellon Software Engineering Institute, Pittsburgh, PA
102. Park S, Kim H, Ko Y, Seo J (2000) Implementation of an efficient requirements-analysis supporting system using similarity measure techniques. Inf Softw Technol 42:429–438. doi:10.1016/S0950-5849(99)00102-0
103. Ferrari A, Gnesi S, Tolomei G (2013) Using clustering to improve the structure of natural language requirements documents. In: 19th International working conference on requirements engineering: foundation for software quality. Springer, Berlin, Heidelberg, pp 34–49
104. Wiegers K (2000) 10 Requirements traps to avoid. Softw Test Qual Eng J 2
105. Google Google Trends—Web Search interest—Worldwide (2004)—present. https://www.google.co.uk/trends/explore. Accessed 16 Jul 2015
106. Buckley K (2010) Manifesto for half-arsed Agile software development. http://www.halfarsedagilemanifesto.org/. Accessed 5 Sep 2014
107. Brooks FP (1987) No silver bullet: essence and accidents of software engineering. IEEE Comput 20:10–19
108. Turner R (2007) Toward Agile systems engineering processes. Crosstalk J Def Softw Eng 20:11–15
109. Merisalo-Rantanen H, Tuunanen T, Rossi M (2005) Is extreme programming just old wine in new bottles. J Database Manag 16:41–61. doi:10.4018/jdm.2005100103
110. Cohn M (2008) Advantages of the "As a User, I Want" user story template. http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template. Accessed 25 Feb 2014
111. Matson E (1997) Please don't feed the consultants. http://www.fastcompany.com/32476/please-dont-feed-consultants. Accessed 12 Jun 2014
112. Bhatia J, Sharma R, Biswas KK, Ghaisas S (2013) Using grammatical knowledge patterns for structuring requirements specifications. In: 2013 IEEE third international workshop on requirements patterns (RePa), pp 31–34
113. Mylopoulos J (2006) Goal-oriented requirements engineering, part II. (Slides). In: 14th IEEE international requirements engineering conference