Lightweight Requirements Annotation through Mobile Speech Recognition

Ola Petersson, Viktor Mellgren, Robert Feldt, and Emil Alegroth
Division of Software Engineering, Dept. of Computer Science and Engineering
Chalmers University of Technology, Sweden
robert.feldt@chalmers.se

***Abstract*—Requirements are crucial in software engineering and there is ample support for how to elicit and document them; however, relatively little support exist for requirements maintenance. We argue that lightweight methods for annotating requirements are needed and present a system based on speech recognition to enable it. This paper describes the system design and a set of experiments and user tests to validate its use. For more realistic evaluation our system has been adapted to the commercial requirements management tool SystemWeaver. Results show that the accuracy of free text speech input is not high enough to enable free-form addition and edits to requirements. However, requirements identification and annotation is practical by extending the system with string distance calculations to the set of requirements being matched. Lookup times on industrial requirements data was deemed practical in user tests, in particular for annotations on the go and during requirement reviews.**

I. Introduction

Many errors in software products and systems can be traced to the management of requirements [1], [2]. The fact that requirements work is in many ways focused on and dependent on people, not primarily on technical issues, often leaves more room for errors and lead to errors that are more complex. Examples of errors can be misinterpretations of customer needs or that requirements and needs change during the lifespan of a project [3], [4] and are not properly updated. A key challenge in requirements engineering is that customers typically express their needs in natural language, while the requirements engineer is tasked to elicit and translate this informal information into a more formal requirement specification. This makes room for interpretational errors and ambiguities [5], [6], where information gets lost in translation [4]. Even though companies and developers acknowledges problems associated to requirement management, many do not put enough effort and time into the task of requirement maintenance [7]. In 2012 Wnuk et al. [8] presented that 84.3% of their recipients considered obsolete requirement specifications to be either serious or somewhat serious, 76.8% of those recipients reported that they did not have any tool, method or process for handling obsolete requirement specifications. With the agile methodologies that have become more popular in the last years, more responsibility is put on the developers (and customers) to be responsive to changes [9], [10].

Since its introduction in 1952 [11], the development of speech recognition has reached a point where it has become good enough to be used with a satisfying result [12], [13], [14]. We believe that using speech recognition can be one way to lower the time needed and perceived barriers to query and update requirements during software maintenance and during the evolution of requirements understanding in a development project. This could act as a lightweight access mechanism for potentially large requirements databases and allow more active ways of working with requirements. There is a risk that much information is currently lost, and even forgotten, if it is not updated or recorded immediately. There is also a risk that navigating large requirements databases takes longer time and requires more effort than more lightweight interactions would allow.

Quality assurance of requirements are often talked about in the context of verification and validation [15], [16]. But there has been relatively little work on the quality of Software Requirement Specification (SRS) and in particular for individual requirements. The focus is still on reviews, audits and walkthroughs and primarily on the SRS as a whole. Verification of requirements is primarily described as actions taken periodically or at special points in time during a project such as gates or partial deliveries. It does not address continuous or ongoing quality assurance of requirements. Denger and Olsson [15] stresses the importance of starting with QA as early as possible in the requirement elicitation phase. They also talk about techniques to minimize the chance of introducing defects in requirements documents and the need for tool support and how that could facilitate ”other” quality aspects, not currently being addressed.

This paper will describe our design and evaluation of an annotation system for requirements databases that is based on speech recognition input from a mobile device, e.g. a so called smartphone. The article aims to evaluate if a speech recognizer is a possible tool for working towards large scale SRSs, and how such a tool is perceived by end users. Furthermore, we will evaluate if an annotation system can enable a way of continuously working with quality assurance of a requirement specifications. In order for our evaluation of the system to be more realistic this research was conducted in co-operation with the Swedish company Systemite AB. Systemite develops a tool that support the management and development of product lines of software systems, called SystemWeaver. An essential part of this tool is its requirements management and requirements engineering support. Through the collaboration with Systemite we could evaluate our annotation system on

four industrial requirements databases from a large Swedish company developing software-intensive, embedded systems.

Section II presents a background and related work while Section III describes the industrial context and the requirements tool our system is adapted to. The design of our system is then described in Section IV, followed by the different steps of the evaluation in Section V. The paper is then concluded with a discussion in Section VI and conclusions in Section VII.

II. Background and Related work

Speech recognition technology has existed since the mid 50s [11] but due to limited hardware and algorithm performance the speech recognizers have not been satisfactory in terms of quality. However, thanks to parallelized algorithms and other contributions to the corpus of speech recognition the technology has in recent years become both efficient and accurate. Smartphone technology has further advanced the technology by making speech recognition commonly available, both through cloud based and offline solutions [17]. Speech recognition is currently supplied by both Apple in the "Siri" personal assistant application for IOS and "Google Now" developed by Google for Android. These technological advances present new application areas for speech recognition but to the authors' knowledge this paper is pivotal in applying the technology to the field of requirements engineering.

Annotations are commonly used in software development to provide additional information in the source code and code development artifacts, e.g. constraint annotations (pre-/post conditions) in code or annotations to UML diagrams. However, for other types of documentation, e.g. requirements and specification documents, annotation of individual entries is uncommon. An example of such an annotation, i.e. an informal annotation, could be "the test results need to go here", which could serve as a reminder for future work or other additional information to clarify the entity. However, most annotations consists of meta data that help simplify search and filter functionality, i.e. formal annotations. Formal annotations are primarily intended to be read by machines, e.g. for Semantic-web applications [18]. An example could be the annotation "Paris", which can be related to the abstract concept "City" or the country "France" given an ontology that helps reduce ambiguity of which "Paris" the annotation refers to. A considerable body of work has been devoted to formal annotation whilst lightweight informal annotations has been left an unexplored area.

Support for the need for lightweight requirement practices is for instance given by Forward and Lethbridge who conducted a survey in 2002 about Software Documentation [7]. The survey conducted by Forward and Lethbridge showed that 54% of the 32 participants thought that word processors was a useful documentation technology and they stated that:

> "Document content can be relevant even if it is not up to date. (However, keeping it up to date is still a good objective). As such, documentation technologies should strive to become easy to use, lightweight and disposable." [7]

Hence, technological properties that annotations, recorded using speech recognition, connected to an SRS could fulfill. Further support for the need of lightweight requirement practices is given by Zhang et al. [19] who discusses the importance of lightweight requirement processes in agile development. Kaupinnen et al. claims that organizations can gain requirements engineering (RE) benefits by defining simple RE processes and focusing on a small set of RE practices, and by supporting the systematic usage of these practices [20]. Annotations are perceived to raise requirement quality and the use of annotations should therefore be integrated into the company's requirements engineering process

Software requirement quality can be captured using standards such as the IEEE830 [21]. This standard defines a set of quality attributes that a good requirement, or requirement specification, should adhere to, e.g. it should be unambiguous, correct and complete. Thus, the standards provide a taxonomy for industrial projects. This taxonomy thereby constitutes a good base for a lightweight annotation scheme since lack of adherence to a given attribute can be annotated with a single word. We also believe that if all individual requirements in a SRS meet these quality attributes it will also be reflected in the SRS as a whole. Thus, raising the quality of the SRS. Furthermore, annotations regarding the requirements quality attributes could perceivably also be used to predict defect inflow, similar to practices suggested by Staron and Meding 2008 [22].

Speech recognition, as stated, has over the recent years become more efficient and accurate, but there are still limitations. One way to mitigate these limitations, which was deployed in this research, is to use natural language processing and string distance functions, e.g. the Levenshtein or Hamming distance algorithms [23], [24]. These algorithms calculates similarities or dissimilarities between string by measuring how many operations (add, modify, delete) that are needed to transform one string to another. String distance functions were used to match inaccurate speech input to a domain of strings that included entity titles, available tags, etc.

III. Industrial Context and Requirements Tool

SystemWeaver is a product line management software application developed by Systemite AB. It is an enterprise-wide data repository that assembles design information from all components of a software system, such as internal structure, interfaces, variants, versions, requirements and component status, in models [25]. The models can be traversed and edited in a graphical interface, SystemWeaver Explorer. The foundations of all of SystemWeaver's models are *items* and *parts*.

Items are the entities in the system, e.g. components, whilst the parts are the relationships between these entities. An item or a part must be an object within a model, each with an unique identifier and a type identifier. The type identifier is called a SystemWeaver Identification (SID) that specifies the

type of a model object. For example, every object that has the SID 'RQQ' is a 'Generic requirement' whilst and object with the SID 'HWC' is a 'Hardware Component'. SystemWeaver allows item type inheritance which allows the user to perform queries to the model that do not only return items of a specific type but also items that share the same properties.

An example is a query based on 'Generic Requirement' items, with the subitems 'Functional Requirement' and 'Quality Requirement', that returns all of the items. This enables queries on specific levels at the same time as it makes it possible to aggregate data connected to items that are derived from the same type of items.

Parts, in turn, are as mentioned relationship between items. Parts always points in one direction and can define relationships such as 'Item A contains Item B' or 'Item A is derived from Item B'. The fact that the parts only points in one direction means that you would have to create two parts to make a two way relation.

The relationships an item can have to another item is within a model is defined by the meta model that was developed by Systemite AB's application engineers for the project.

Systemite AB are very flexible with their deployment of SystemWeaver to their customers.

This flexibility is made possible since all correlations and the behavior of items and parts are defined by a structured meta model that supports architectural and solution reuse as well as tailored solutions for specific customers. Therefore, each project carried out in SystemWeaver is still unique but also very adaptable. Furthermore, because of the meta model solution, items and parts can look and behave in several different ways in different models dependent on what constrictions and rules the meta model has defined.

SystemWeaver also supports task definition and tracking. Tasks in SystemWeaver are called 'issues', and are, in a simple analogy, similar to notes that can be attached to items. Issues are used for case management of the items that they are attached to and can be set to different states, which are customizable in the meta-model (e.g. started, closed, assigned to). Issues can either point to items or other issues through an issue relation. This relation, which is not considered a part but has a SID, defined as a type ID, can only be applied if the origin of the relation is an issue. Furthermore, similar to both items and parts, issues themselves all have SIDs. A graphical representation of how the artifacts in SystemWeaver fits together can be seen in Figure 1

SystemWeaver was used as a requirement database and backend in this project to facilitate the annotation scheme of the developed annotation system.

IV. Design of SpeechWeaver

The architecture of the system that was developed during this research project, SpeechWeaver, consists of a client(s) and a server (see Figure 2). The client is a smartphone application frontend, developed in Java for the Android platform, which facilitates user interaction with SpeechServer. During usage of the application, text strings are first captured using the speech

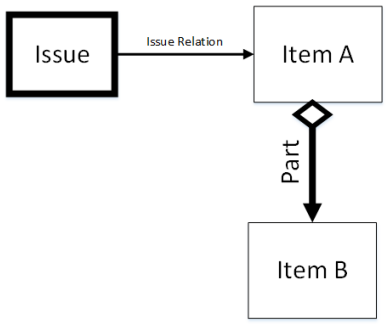

Fig. 1. An overview of the artifacts in SystemWeaver

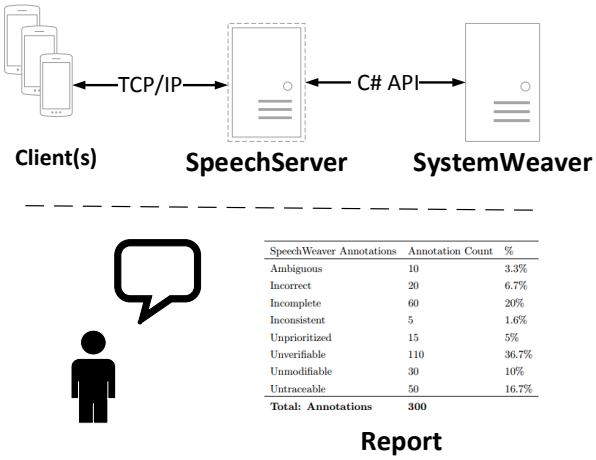

Fig. 2. An overview of the SpeechWeaver system

recognizer in the client which are then sent to SpeechServer using a custom TCP/IP protocol which is described below.

A. Client

All user interaction is performed with the client application in SpeechWeaver system, which is developed for Android 4.1 and above to facilitate offline voice typing [26]. However, because of the string based communication, the system does not require an Android based client, only that the client can send and receive TCP/IP messages. Hence, it is possible to develop clients for different platforms, e.g. desktop applications, in order to take advantage of their respective advantages in different use cases. However, in this project the focus has been on smartphones in order to take advantage of the mobility that they provide. Therefore all further references in this article to "the client" will refer to the developed Android client.

During usage of SpeechWeaver all user interaction is handled through Android's speech-to-text for input and text-to-speech for output which make them auditory. Furthermore, since this application is tailored for industrial practitioners, information security was also a requirement for the system. Hence, before the user can interact with the requirement database, the user is required to type in and send user credentials (username and password) to the SpeechServer. Once user credentials have been verified, the user is presented with

a list of to the user available projects that he/she can select to work with. After selection of a project, the user can perform three types of actions, which are:

1) Annotate a requirement - Performed by providing the client with a requirement ID and an annotation through auditory input.
2) Annotate a requirement with additional data - Performed by selecting a file from the smartphone (e.g. a picture or a recorded sound file) followed by the acton described in 1.
3) Generate a report - Performed through client interaction that triggers the SpeechServer to generate a report over all annotated requirements.

The use of auditory interaction with SpeechWeaver removes all need for the user to focus on the application. However, to initiate one of the above mentioned actions the user must interact with client through the Smartphone's touchscreen. Action 1, *Annotate a requirement* can however also be initiated through the button on a bluetooth headset. Thus facilitating concurrent usage of the system during, for instance, requirements review.

The Android application captures the users voice using the built in API for speech recognition [27]. When launching the speech recognizer, a pop up dialog appears and a sound is generated to inform the user that the speech recognizer is ready to capture input. The speech recognizer will then record the users input until it does not receive any new input for one second, i.e. until the user goes silent after stating the desired phrase. This event will cause the pop up dialog to close and a sound is generated to to inform the user that the speech recognizer has caught the input and is processing it. When the speech recognizer has processed the input, it returns up to five results. These results are based on how confident the recognizer is that the returned result was correct, where the first of the five results is what the recognizer interpreted as the most likely input from the user.

All five alternative user input strings are then sent to the SpeecServer for post-processing.

Android's standard speech recognizer supports different languages. However, through out this project, English US was selected used as the primary input language, meaning that the domain of possible interpretations for the speech recognizer were words from the american english dictionary. In addition to this dictionary, the speech recognizer also provides the commands "period", "comma", "exclamation mark/point" and "question mark" which are translated into their corresponding characters [28].

The communication between SpeechServer and the clients is handled by a custom protocol built on tags that represent different states in the communication. The tags are sent over a TCP/IP-socket and are interpreted at the receiving end. The tags attached to the messages are dependent on which state the system currently is in, i.e. when the client trigger the annotation by sending a requirement title, the messages is tagged with ID (Identification). When the server has found a requirement title (the ID), it returns a message with the tag WFT (Waiting For Tag) and the next output containing the annotation from the client to the server is tagged with ANOT (Annotation).

During the setup-phase, after a connection has been established between the client and the server and the user has logged into the system, the server acts as the active part to form a context for the whole system to work within. Once the setup-phase is over, SpeechServer takes a passive role where different operations are triggered based on the what is received from the client. This implementation allows the communication states to be decided from the client and also means that they don't need to be followed if the client decides to abort the current communication state.

Once the setup-phase is over, operations (sending an ID, sending an annotation, create a report etc) are always triggered by the client. The different operations can at any point be aborted and re-selected by the client.

B. SpeechServer

The server is written in C# and runs one thread for each client thereby allowing it to handle multiple clients.

Clients connect to SpeechServer by sending input to a predefined socket that the server listens to. User authentication, in turn, is handled by checking with SystemWeaver that the provided username and password are both valid.

As soon as a client connects to the server, the client is given a broker-object by the C# API to be able to interact with SystemWeaver. The broker provides the client with all necessary operations to interact with SystemWeaver, such as authentication, fetching items/parts and writing to SystemWeaver through a connection that is kept alive as long as the client stays connected.

If the client has not provided a specific project to work on, the server fetches all the projects from the server and sends them to the client and asks which project the user wants to work in. The *contexts* from a selected project, which is a delimited part of a project or an abstraction level (e.g. Analysis Level, Design Level, Implementation Level), are treated in the same way by the user, but are traversed to from the project node. This project retrieval is performed during the *Setup Phase*.

After the setup phase, the *Annotation Phase* begins, this is where the user adds an annotation. This phase starts with the client providing an input string with the name of a requirement that the SpeechServer then tries locate. When the requirement is found on the server it stores the name temporarily and asks the client what annotation to annotate the requirement with. When the annotation has been chosen by the user it is processed and classified into one of the predefined annotations on the server. The classified annotation is then added to the requirement via the C#-API of SystemWeaver.

The server look-up is performed by first asking SystemWeaver for all projects. When the user has selected a specific project, the look-up continues by traversing that project to find the project *contexts* (as mentioned above). After the *context* has been identified, all further look-ups are performed

through traversal from that context node. This implementation makes the look-up process quite fast since all server data is cached. The time to perform the look-ups depend on the size of the SRS and the structure of the items the SRS.

C. String Processing

When the server processes the string input it does it in several steps that are dynamic depending on the accuracy of the results. First the client provides the five best results from the speech recognizer (as described above) for an interpreted spoken input. First the server tries to find a perfect string match to the five results. If no perfect matches are found, the Levenshtein distance algorithm is used to find the closest matches on the first of the users input strings. The reason to use all of the five inputs when searching for perfect matches is that the latter results can hold small differences that are crucial for perfect matches, e.g. When a user says"Requirement five", the speech recognizer might interpret the most likely input to be "requirement five", whilst a latter of the results might be "requirement 5". To run all of the five results is an effective way to find more perfect matches, resulting in a better user experience. However, if all five results where run towards the Levenshtein distance algorithm, not enough information would be accessible to say that the closest matches from one of the results from the speech recognizer would be better than another. Therefore, we have chosen to only run the most likely interpreted input against the Levenshtein distance algorithm. Then the server returns the closest matches sorted by their Levenshtein distance from the input string so the user can choose the correct alternative.

There are a few reasons why not to handle the entire input string in one message. The first reason is that there are problems breaking the input string down in its parts since it's complicated to find which part of the string is the id, and what parts are annotations. If the system would have listened for special syntax (example highlighted in bold) such as:

Requirement: *"System should not create a file."* ***Annotation: "Ambiguous"***

there is a possibility that the requirement could contain these keywords or other syntax words which would result in a faulty delimitation of the string. The second reason is that a single string solution requires more from the user since he/she needs to remember and formulate the syntax, keywords, requirement ID and the annotation before speaking [29]. Having separated the input into different stages lessens the burden for the system to compensate for these issues, as well as lessen the burden for the user since the user now only needs to answer the questions stated by the application. The question-response communication implementation also makes failure mitigation easier if an operation fails since the user can quicker just abort or retry the operation.

A way of improving the speech recognizer is to add an algorithm which gives us an indication of what input the user *wanted*, as opposed to what the speech recognizer interpreted. When searching for entities within a finite interval, as in the example with requirement names in a requirement database, we improve the searches by measuring an edit-distance between the interpreted input from the speech recognizer and the values in the target domain. This allows us to narrow down the number of possible results and thereby identify the correct result with higher certainty.

The distance between two strings x, y given by $\mathrm{lev}_{x,y}(|x|, |y|)$ where

$$\mathrm{lev}_{x,y}(i,j) = \begin{cases} \max(i,j) & , \min(i,j) = 0 \\ \min \begin{cases} \mathrm{lev}_{x,y}(i-1,j)+1 \\ \mathrm{lev}_{x,y}(i,j-1)+1 \\ \mathrm{lev}_{x,y}(i-1,j-1)+[x_i \neq y_j] \end{cases} & , \text{ else} \end{cases}$$

Fig. 3. The edit-distance (Levenshtein distance) between two strings, *x*,*y*, where all operations has a cost of one

Edit-distance uses the number of operations needed to transform a string *x* to a string *y*. In Levenshtein's string distance function all operations has the same cost, i.e. regardless if you add, delete or modify a character within the string, it is still calculated as a cost of one. An edit-distance where this property is fulfilled is often called a Levenshtein distance [30].

In our system we use Levenshtein distances when a given input *x* does not have a a requirement name that perfectly matches *x*. We then calculate the distance between *x* and all requirements in the given database and returns a set of requirements with the lowest Levenshtein distance, and the user can then chose which requirement the user originally sought.

D. Annotations and Report Generation

In order to create high quality software requirements specifications (SRS), IEEE have developed a recommended practice for SRSs [21]. The guideline also defines how to connect the requirements engineering practices to the software lifecycle processes [31]. We decided to use the chapter *4.3 Characteristics of a good SRS* from IEEE Recommended Practice for Software Requirements Specifications as the basis for the annotations.

The provided annotations in SpeechWeaver are the complementary antonyms of the characteristics defined in the IEEE 830 standard. These annotations are just a suggestion of annotations that could be used. However, they were chosen because they can capture many of the problems a requirement can have. This is customizable per customer and organization, to suit problems that a particular organization encounters, this is also customizable in run time and can be changed and perfected continuously during a project to define more narrow definitions of the annotations. It is also possible to have synonyms to the annotation tags, so that a user can say *"Not clear."* or *"Not well defined"* instead of *"Ambiguous"*. This reduces the burden to remember exact semantics, but the synonyms map to the same tag within the system, that is *Ambiguous*.

The annotations made by the users are stored in SystemWeaver as individual entities, this allows several users to annotate the same requirement with the same annotation several times. These annotations are stored in an *issue* connected to the requirement, and the connection can be traversed in any direction. The report generation uses these annotations while iterating over the report structure. The output of the report generation are the statistics for the requirements and how many have been annotated with the different annotations. An example can be seen in Figure 2. Note that in the first table many requirements are tagged with different annotations, this means that if a requirement is annotated several times with the same tag by different users, i.e. several people think that a requirement is *ambiguous*, then the annotation is only counted once for that requirement, to see the status over all requirements. This also means that one requirement can be represented in all categories. In the other table we see the total number of annotations, i.e all annotations are counted.

The generated report also groups annotations by headline so the reader can see all requirements annotated with a specific annotation. Furthermore there is headlines for each requirement so the reader can see what problems a particular requirement has. The output shows the description for each requirement so the review process becomes as simple as possible.

The annotations can also be shown for specific time periods or iterations, depending on the project setup. This means that statistics can be calculated over time and the data can be plotted to see the history and development of the annotations.

V. Evaluation

The design research project to develop SpeechWeaver was complemented with four evaluation steps as shown in Figure 4. Steps 1 and 2 focused on evaluating to what degree output from speech recognition is useful for identification (step 1) and free-text annotation (step 2) of requirements. Scaling to realistic requirements database sizes is critical for industrial applicability and since the response time of the system was considered a key characteristic by the company and the identification of requirements dominated the round-trip time, evaluation step 3 measured this time on the server for four different requirement sets. Finally, step 4 evaluated the SpeechWeaver through a user test and a follow-up interview. Below we describe the evaluation steps and their results in detail.

A. Identifying Requirements Through Mobile Speech Input

Even though speech recognition algorithms have improved in recent years the error rates can still be high. This is especially true for the speech recognition available in mobile devices which is typically speaker-independent (which has higher error rates) and has a limited compute power (which makes it impossible to use the best known algorithms). Our initial results also confirmed that there was often a large discrepancies between the intended requirements supplied as speech input and the output from the speech recognizer. To

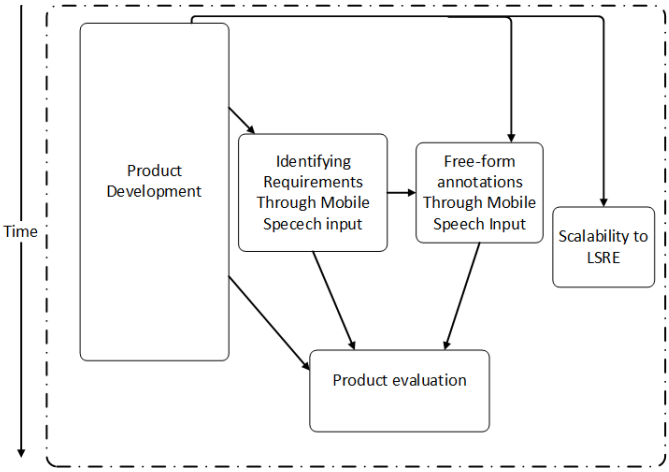

Fig. 4. An overview of the design research

make requirements identification possible we extended the system with the Levenshtein string distance as described above. To evaluate how well this solution performed an experiment was performed.

The experiment had 10 subjects, all who were students at Chalmers University of Technology, and all with English as their second language. Each student was given 10 randomly selected requirement titles from an SRS with 4544 requirements and were asked to speak the titles into the speech recognizer. The most likely output from the speech recognizer was then saved and used to evaluate lookup of requirements in that SRS through the Levenshtein edit-distance algorithm. For each input we noted the input length (string length of output from speech recognizer), the target requirement length, distances from input to all requirement titles in the SRS, as well as the rank of the target requirement in the ranking created by the distance function. An example of the data saved for each input can be found in Table I

The algorithm was extended to also check for perfect matches (i.e. the interpreted input was exactly the same as the requirement title), and if it was found, the position of the returned result from the speech recognizer were saved. This extension was made to utilize that the speech recognizer returned five different interpretations of the spoken input. If a perfect match was found in any of the five results, the input was never run through the distance function. If a perfect match was not found, only the most likely result from the speech recognizer was run through the distance function.

To evaluate the performance of our ranking algorithm we consider the lookup a success if the target requirement is ranked among the top five (5) requirements. Presenting up to five possible matches to the user was deemed feasible if there is no exact match. Figure 5 shows a graph for the percentage of requirements that was ranked among the N top requirements, for increasing numbers of N from 1 to 30. From the graph we can see that for 73% of the requirements the target requirement was top ranked and in 93% of the cases the target rank was among the top five among the 4544 requirements in this SRS.

TABLE I
TABLE SHOWING HOW LEVENSHTEIN DISTANCE AND RANKING IS CALCULATED FOR A GIVEN INPUT IN A CLOSED DOMAIN OF REQUIREMENT TITLES.

Input	Requirement titles	Distance	Rank
And its allies buffet	Initialize buffer	9	1
	Erroneous APLM buffer	15	3
	State monitoring buffer	14	2
	Transaction: Buffer on - Buffer off	26	5
	Buffer out of range	18	4

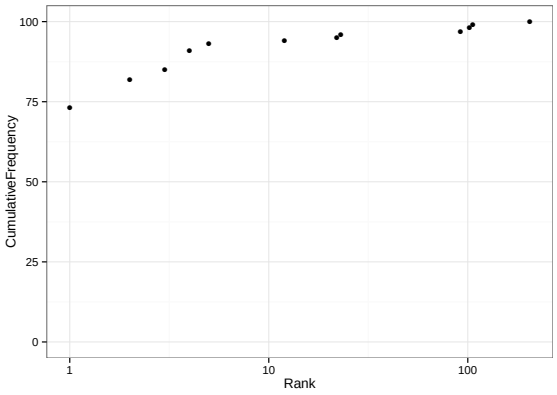

Fig. 5. Graph showing the cumulative frequency and the edit distance rank

The mean rank was 6.96 but this is dominated by a few requirements where the speech recognizer gave output very far from the target requirement; the median rank was 1.

The speech recognizer delivered 15 perfect matches, and of these 12 where at the first position of the returned results from the speech recognizer. In two of the cases the perfect match was at the speech recognizer's second result, and in one of the cases the perfect match was at the fifth result. In all of the cases, the input to the Levenshtein edit distance algorithm (i.e. the first result returned from the speech recognizer) had the Levenshtein rank 1. The ranges of the inputs that resulted in perfect matches were between 15 to 43 characters, this is most likely because longer strings are more vulnerable to error since any word can be misinterpreted, shorter requirement titles seemed to contain more abbreviations and symbols such as "-¿" or parentheses/brackets, presumably a consequence of that the recognizer's expected input is words of natural language. None of the given input returned a false perfect match.

The correlation between Levenshtein distance to the sought requirement title and the difference in length between input and what the speech recognition interpreted can be seen in figure 6. This figure shows that the Levenshtein distance gives better results for strings with length similar to the target requirement. Since the input and interpreted string length often is quite close, this gives good results. This also gives a hint that it's better for the user to try to pronounce the whole title, even hard to pronounce symbols and abbreviations, so that

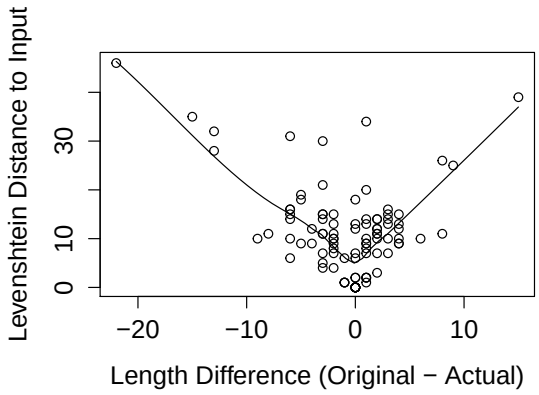

Fig. 6. The Levenshtein distance and it's correlation with the difference in input length, the line shows a locally fitted regression

the length of the interpreted input is as close to the target requirement as possible, thus making the matching better.

The requirement matching experiment showed that a string edit distance algorithm such as Levenshtein's edit distance can give satisfiable results for matching natural language to name conventions used in an SRS.

B. Free-form annotations Through Mobile Speech Input

Experiment 2 was conducted to evaluate how the speech recognizer behaved with free text input. Even if this is a typical use of speech recognition we are targetting a specific, technical domain and not everyday speech/text. Thus, the performance of speech recognition can be expected to be worse. Since free-form speech recognition would be needed in allowing arbitrary annotations to requirements this is a key concern for our application.

From the same SRS as mentioned above, 100 sentences were randomly selected. The same 10 students as mentioned above were then asked to speak 10 of the sentences into the speech recognizer.

The best interpreted result from the speech recognizer was then saved in a text file and were later given a value between 0-3, were the numeric defined the level of severeness of the error, as can be seen in Table II

TABLE II
TABLE SHOWING INTERPRETATIONS OF SEVERITY

Severity	Meaning
0	No error
1	Minor error, still comprehensible
2	Error, hard/impossible to comprehend
3	Severe Error, meaning of text has changed, but has a viable meaning in the context

The results from the Recognition Accuracy Experiment in table III shows that free text speech input is not good enough to provide accurate descriptions as of now. As many as 64% are not comprehensible, and 10% can introduce errors if interpreted as written. The number of samples was 98 rather

than the stated 100, this was because the experiment equipment failed to record two of the inputs from two different subjects.

TABLE III
NUMBER OF DESCRIPTIONS IN EACH SEVERITY RANKING

Severity	Frequency (n=98)	Explanation
0	8	No error
1	17	Minor error, still comprehensible
2	63	Error, hard/impossible to comprehend
3	10	Severe Error, meaning of text has changed, but has a viable meaning in the context

C. Scalability to large requirements databases

Software projects in the industry can be very large and complex and often involve several thousand requirements. To evaluate the scalability of our solution we selected four (4) different requirements databases of medium to large scale from an industrial partner and recorded the execution time for a requirement lookup [32]. All SRS are from the automotive industry and use domain-specific terminology.

For each of SRSs, a requirement was identified. A user then gave the name of the identified requirement as input to the system through speech. Each SRS was queried 30 times and the different values for the above mentioned variables where then saved for analysis. Since the mobile connection speed as well as latency might vary a lot over time and with the position of the user we measured only the execution time on the server side. However, this is also realistic since the processing time on the client is minimal and was not considered a factor in user testing. For this experiment SpeechServer and SystemWeaver is running on a laptop with an dual core Intel i5-350 processor with hyper-threading and 4GB RAM.

The times shown are from the start of the look-up on the SpeechServer, until the SpeechServer has returned the items and is ready to send to the client. Results show that even for large scale requirements engineering the lookup times are manageable.

The time to look up requirements can be seen in Figure 7. In practice the look up items is sometimes unpredictable since SystemWeaver as well as SpeechServer does caching internally. Performance can depend on what has been searched before and other factors, such as other users and how much have and can be cached. These results we got here are close to what can be expected in actual use (excluding the network latency and speed as discussed above). The lookup times are dominated not by the ranking through string distance but by the lookup of the requirements in the db. Since these lookups go through the existing API it was not feasible to propose changes to the underlying system. For speedier look ups on frequently performed tasks SpeechServer could cache the requirements titles locally. The maximum observed processing time for the Levenshtein ranking was in the order of 300ms, even for the largest SRS. Thus, even though many data

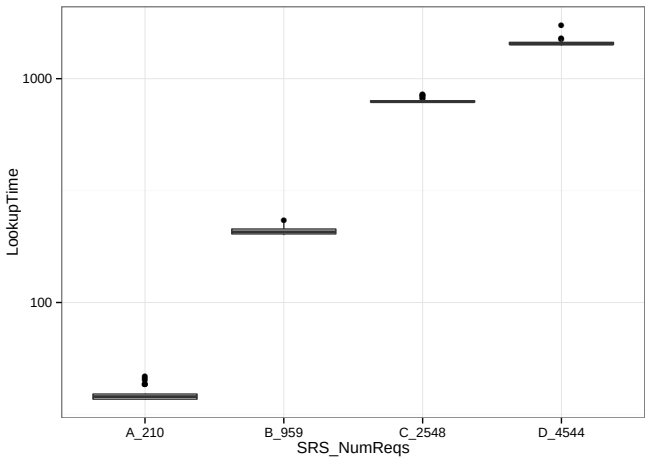

Fig. 7. Time to look up requirements

structures and solutions, such as for example suffix filters [33], for fast and approximate string matching could be used they are not warranted unless the requirements db's are very large (on the order of 10,000 to 100,000 requirements).

D. User testing and interviews

The end user evaluation of SpeechWeaver contained two parts, monitored user tests followed by interviews. The subjects for the user test was five requirement- and application-engineers at Systemite AB.

The subjects was given a short introduction about the features and the design of the application and was then given five randomly selected requirements within a chosen context, and was asked to create an issue to each of the requirement with the application. Out of these five, two of the issues required an attached file. After the requirements were annotated the user sat down and explored the result of the actions by looking at the updates that had been added in SystemWeaver.

Finally the user was interviewed about the experience of using the application. The main focus of the interview was the experience of using speech as input, but also overall impressions of the usability and ease of use of the product as a whole.

Below we present results from this interview in two parts, (1) the annotations and the usefulness of those, and (2) the overall experience of the application. All of the interviewed subjects used TODO-lists in the format of analog or digital notes to their ongoing projects. What they found to be a huge profit with the annotations on the requirements was that there were a direct connection between the TODO-entity (the annotation) and the actual requirement.

A factor that arose as very important for all of the interviewees were the content of the annotations. They found it very important that the naming of the annotations fitted the way they worked with maintenance of projects, and that there needed to be a clear definition and distinction between each and one of the annotations. The interviewees expressed

that the annotations without any more descriptive information (a file or a description field) were better for peer-reviewing purposes (i.e. letting users go through the SRS and go through the annotations in a group format), where as an annotation that contained descriptive information were better for actual modifications and improvements on the requirement.

The key notes that the subjects expressed as very positive was the accuracy and the response time of the application.

Many subjects expressed the usefulness of the mobility in the application. To be able to query an SRS with several thousand of requirements anywhere, from the car or the bar, was considered a huge advantage. The solution was also considered very light weight, and the flow combined with the low amount of time to create an issue were expressed as an easy solution to directly ventilate their thoughts towards the SRS, instead of a tidy operation when they would otherwise have to make several operations to navigate and interact with SystemWeaver.

Some felt uncomfortable using speech as the main input for the application. Moreover, requirement titles that contained ”special” characters (e.g. ”#”, ”->”, ”=”) or abbreviations were found hard to translate to spoken words. Another problem that was pointed out was the sound that was produced during the operation, and several of the subjects stated that they would not use it in an office environment.

VI. Discussion

Speech recognition have developed over the years, but as shown by our results, it is still not good enough to recognize longer, natural language free form sentences. This limits the range of maintenance actions for which our system can be used. For example, we cannot allow the substitution of longer sequences of text, such as in requirements descriptions or rationales often found in specifications.

With the use of simple string distance measures and by constraining the set of sentences being matched against the output from the speech recognizer can be used to query, and thus annotate, text entities, such as requirements, even in large software development projects. In our experiment the constrained set was all the requirement titles of the specification. But future work could consider other constraining sets. For example, once a requirement has been selected for edits, which attribute of it, and potentially which sentence within the value of an attribute, to change or annotate could be found through a similar use of constrained string distance matching. Thus we consider our results an important first step towards a more extensive requirements annotation and maintenance system.

Our results and user testing also show that mobile devices enables speech recognition as a tool for querying requirement specifications, and that this opens up new possibilities for when and where you can ‘reach’ the specifications. This increased accessibility together with the ease of use that the users expressed when testing the system could lead to a change in how software engineers view maintenance tasks. Today it is often considered a disturbance and chore and as being troublesome.

To annotate SRS could possibly be used in a similar manner to how defect inflow can be predicted as described by mstaronmetrics. This would allow for requirement engineers and other people associated to the software documentation to prepare and revise their work with the quality of the SRS.

A threat to the validity of this research is that only five requirements engineers was used in the user testing and that they were all from the same company. However, since they work with a software development and requirements support tool and also work with customers they have a broad experience from many different software development contexts. Another threat to the generalizability of our results is that all four requirements specifications used in the experiments was from one and the same company, in the automotive domain. Since our system is not adapted in any way to the specific domain or vocabulary used in these specifications we do not consider this a major threat. In our data collection and experiments we have used randomization and replicate measurements to avoid threats to internal and conclusion validity.

VII. Conclusion

This paper presents the design, implementation and evaluation of a lightweight system for annotating requirements specifications. By connecting the default speech recognition available in a present-day (Android-based) mobile device to a commercial software development management tool, we can allow access and annotation of large-scale requirements databases from anywhere where you can reach the Internet.

Speech recognition has improved considerably in recent years and its accuracy has increased. However, one of our experiments show that the accuracy of speech recognition available in mobile devices today is not yet adequate to allow free text input, such as comments or larger edits, in the often specialized-vocabulary, technical domains of industrial requirements specifications (SRSs). Even when used for shorter inputs, such as the lookup of a requirement prior to annotating it, accuracy is often poor. By extending our system with a simple string distance measurement we could improve accuracy considerably for the more limited lookup and annotation tasks. With the off-the-shelf speech recognizer and input from 10 non-native English speaking test subjects (with English as their second language) the system ranked the targeted requirement among the top five requirements returned 93% of the time for 98 requirements randomly sampled from a SRS containing 4544 requirements.

Moreover, a scalability experiment showed that with a mobile device, you can query and work with large scale SRSs with feasible round-trip times. Results from the database lookups were presented to the users within seconds even for SRSs with several thousand requirements. Together with speech recognizers within mobile smart devices this enables users to reach and query large scale requirement databases anytime, anywhere.

In user testing with five requirement engineers and managers at our industrial collaborator the system was considered surprisingly accurate and responsive. The users considered

the system a suitable addition to their workflow, in particular as a substitute for their present day lists of requirements to change and updates to be made. They also noted that the system was useful in requirements reviews. For more extensive maintenance work it would need to allow longer input of more free form text. Some users also was concerned about the noise they would generate while using the application; they said they would be reluctant to use it in an open office layout.

Annotations on 'softer' software engineering artifacts, such as requirements, is an unexplored area. We have presented a way to use these annotations as a lightweight maintenance- and peer-review tool for marking requirements during development and maintenance of large-scale industrial requirements specifications. Future research should investigate the long-term effects of such annotations on the quality and efficiency of requirements evolution and maintenance. Our system can also be used to collect metrics on requirements churn, quality and change frequencies. To improve speech recognition accuracy future research could build up a specific speech recognition locales and dictionaries specific to the technical terms commonly seen in the technical areas related to software engineering and development.

ACKNOWLEDGMENT

The authors would like to thank Systemite AB for their cooperation during this research.

REFERENCES

[1] M. Tseng and J. Jiao, "Computer-aided requirement management for product definition: a methodology and implementation," *Concurrent Engineering*, vol. 6, no. 2, pp. 145–160, 1998.

[2] K. E. Wiegers, *Software requirements*. Microsoft press, 2009.

[3] H. Saiedian and R. Dale, "Requirements engineering: making the connection between the software developer and customer," *Information and Software Technology*, vol. 42, no. 6, pp. 419–428, 2000.

[4] E. Bjarnason, K. Wnuk, and B. Regnell, "Requirements are slipping through the gapsa case study on causes & effects of communication gaps in large-scale software development," in *Requirements Engineering Conference (RE), 2011 19th IEEE International*. IEEE, 2011, pp. 37–46.

[5] A. Al-Rawas and S. Easterbrook, "Communication problems in requirements engineering: a field study," *COGNITIVE SCIENCE RESEARCH PAPER-UNIVERSITY OF SUSSEX CSRP*, 1996.

[6] D. Ross and K. Schoman Jr, "Structured analysis for requirements definition," *Software Engineering, IEEE Transactions on*, no. 1, pp. 6–15, 1977.

[7] A. Forward and T. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*. ACM, 2002, pp. 26–33.

[8] K. Wnuk, T. Gorschek, and S. Zahda, "Obsolete software requirements," *Information and Software Technology*, 2012.

[9] J. Highsmith and A. Cockburn, "Agile software development: The business of innovation," *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[10] N. Ibrahim, "An overview of agile software development methodology and its relevance to software engineering," *Jurnal Sistem Informasi*, vol. 2, no. 1, 2012.

[11] K. H. Davis, R. Biddulph, and S. Balashek, "Automatic recognition of spoken digits," *The Journal of the Acoustical Society of America*, vol. 24, no. 6, pp. 637–642, 1952.

[12] B. Ballinger, J. Schalkwyk, M. Cohen, C. Allauzen, and M. Riley, "Speech to text conversion," Sep. 29 2011, uS Patent App. 13/249,181.

[13] B. Ballinger, J. Schalkwyk, M. Cohen, and C. Allauzen, "Language model selection for speech-to-text conversion," Dec. 22 2010, uS Patent App. 12/976,920.

[14] J. Schalkwyk, D. Beeferman, F. Beaufays, B. Byrne, C. Chelba, M. Cohen, M. Kamvar, and B. Strope, "Google search by voice: A case study," *Visions of Speech: Exploring New Voice Apps in Mobile Environments, Call Centers and Clinics*, vol. 2, pp. 2–1, 2010.

[15] "Quality assurance in requirements engineering," in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds., 2005.

[16] M. Chemuturi, *Mastering Software Quality Assurance : Best Practices, Tools and Technique for Software Developers*. Ft. Lauderdale, FL, USA: J. Ross Publishing Inc., 2010.

[17] H. Barra. (2012, 6) Google i/o 2012 - keynote day 1. http://www.youtube.com/watch?v=VuC0i4xTyrI. Visited January 2013.

[18] V. Uren, P. Cimiano, J. Iria, S. Handschuh, M. Vargas-Vera, E. Motta, and F. Ciravegna, "Semantic annotation for knowledge management: Requirements and a survey of the state of the art," *Web Semantics: science, services and agents on the World Wide Web*, vol. 4, no. 1, pp. 14–28, 2006.

[19] Z. Zhang, M. Arvela, E. Berki, M. Muhonen, J. Nummenmaa, and T. Poranen, "Towards lightweight requirements documentation," *Journal of Software Engineering and Applications*, vol. 3, no. 9, pp. 882–889, 2010.

[20] M. Kauppinen, M. Vartiainen, J. Kontio, S. Kujala, and R. Sulonen, "Implementing requirements engineering processes throughout organizations: success factors and challenges," *Information and Software Technology*, vol. 46, no. 14, pp. 937–953, 2004.

[21] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, 1998.

[22] M. Staron and W. Meding, "Predicting weekly defect inflow in large software projects based on project planning and test status," *Information and Software Technology*, vol. 50, no. 78, pp. 782 – 796, 2008.

[23] V. Levenshtein, "Binary coors capable or correcting deletions, insertions, and reversals," in *Soviet Physics-Doklady*, vol. 10, no. 8, 1966.

[24] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[25] Systemite products and services. http://systemite.se/content/products-services/. Visited February 2013.

[26] C. Davies. (2012, 6) Android 4.1 jelly bean adds offline voice typing. http://www.slashgear.com/android-4-1-jelly-bean-adds-offline-voice-typing-27235871/. Visited April 2013.

[27] (2013) Recognizerintent. Visited April 2013. [Online]. Available: http://developer.android.com/reference/android/speech/RecognizerIntent.html

[28] (2013) Type text by speaking. Visited May 2013. [Online]. Available: http://support.google.com/android/bin/answer.py?hl=enanswer=1650147

[29] J. Wilding, "Over the top: Are there exceptions to the basic capacity limit?" *Behavioral and Brain Sciences*, vol. 24, no. 01, pp. 152–153, 2001.

[30] H. Blockeel, B. Demoen, E. Duval, N. Lavrac, and N. JACOBS, "Relational sequence learning and user modelling," 2004.

[31] "Ieee/eia standard industry implementation of international standard iso/iec 12207: 1995 (iso/iec 12207) standard for information technology software life cycle processes," *IEEE/EIA 12207.0-1996*, pp. i–75.

[32] B. Regnell, R. B. Svensson, and K. Wnuk, "Can we beat the complexity of very large-scale requirements engineering?" in *Requirements Engineering: Foundation for Software Quality*. Springer, 2008, pp. 123–128.

[33] J. Kärkkäinen and J. C. Na, "Faster filters for approximate string matching," in *Proc. ALENEX*, vol. 7, 2007, pp. 84–90.