AbstFinder, A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology, and Evaluation

Leah Goldin Daniel M. Berry

Computer Science, Technion, Haifa 32000, Israel

Abstract

In order to help solve the problems of requirements elicitation, this paper motivates and describes a new approach, based on traditional signal processing methods, for finding abstractions in natural language text. The design of AbstFinder, *an implementation of the approach, and the evaluation of its effectiveness on an industrial-strength example are described.*

1 Introduction

It appears that the least understood step of systems development is the requirements gathering and specification stage, and that within this stage, gathering is less understood than specification.

Many system design or programming methods start from an assumed clear statement of requirements and show how to arrive at a design of a program meeting those requirements. However, none of these methods really explain how these requirements are obtained in the first place.

Large, complex software, for which it is difficult or even impossible to obtain clear requirements, is usually developed for a client organization in which there are many people who have some view or say as to what the desired system should do. These views range from being totally unrelated to each other to being totally inconsistent with each other. It is no wonder that the distillation of these views into a consistent, complete, and unambiguous statement of the requirements, albeit in natural language, is a *major* part of the problem of developing software which meets the client's needs. Therefore, it is essential to have methods and tools that help in distilling these many views into coherent requirements.

Extant production quality tools, methods, and systems for requirements engineering focus on the analysis stage and by and large ignore the problem of eliciting requirements from the client.

More recently the software engineering community has been paying attention to the problem of eliciting the raw information from the clients [6, 4, 8].

There are a number of environments envisioned or being built, e.g., REGEE [5] and AMORE [15] for gathering, analyzing, and writing requirements. Many of these provide a way to organize a domain model into a collection of abstraction nodes into which all information about these abstractions is stored. One tool that these environments need is to help identify the abstractions that form the domain model.

2 Abstraction identification

An early stage in requirements engineering is abstraction identification. Heretofore, abstraction identification has been done manually by an RA. The RA scans all the transcripts, trying to note important subjects and objects of sentences, i.e., nouns. The problem is that humans get tired, get bored, fall asleep, and overlook relevant ideas. So it is proposed that REGEE contain tools that do the clerical part of the search without getting tired, falling asleep and overlooking anything. The human RA still has to do all of the *thinking* with the output of the tools, but he or she will be confident that no piece of information has been overlooked in the process of gathering input to the human process of abstraction identification.

Note that we are excluding expert-system approaches. We do not believe that enough is understood about requirements identification to codify the process.

That is, no matter what, the RA must read all the input at least once. The larger this input, the more that must be digested in the RA's process of abstraction identification. There is the danger of information overload in gathering this input. To avoid information overload, it is useful to somehow reduce the size of the input that must be digested. The danger in reducing the size of the input and relying only on the reduced input is that something important might be overlooked. Therefore, confidence is needed that nothing important is overlooked.

It is useful to have a definition of abstraction so that it is understood what must be identified. An *abstraction* is a concept left after ignoring irrelevant details. This definition is hard to pin down because the terms "concept" and "irrelevant" defy precise definition. However, most competent software engineers and RAs recognize an abstraction when they see one. Therefore, we are reduced to finding implementable syntactic definitions with the

0-8186-5480-5/94 $03.00

hope that they match the semantic reality.

2.1 Assumptions

Underlying all the approaches attempted in the past and finally taken here are some assumptions that ultimately have to be validated. Their validation will come retroactively as a result of the success of the resulting tools. The assumptions are that

1. at least some manifestation of all abstractions is expressible within the confines of a single sentence and
2. each individual abstraction is discussed in more than one sentence.

If these assumptions hold, then a repetition-based approach, such as proposed below, should work. The main idea behind such an approach is that the importance of a term in the text is proportional to its frequency of occurrence within the text. It has been empirically verified that a writer repeats important words in the text as he or she tries to explain or verify them [9].

The assumptions seem to overlook a high-level abstraction that consists of a concept spread out over several sentences that individually do not expose the concept. Either these do not occur or if they do occur, it is assumed that the human RA will notice them as an aggregate of several identified concepts. For this identification to be possible, it must be that each individual subconcept is mentioned more than once so that all of them show up and can be recognized. Our experience has shown that these high level abstractions are not a problem to identify.

2.2 Existing abstraction identification tools

An early idea for abstraction identification, reported in [2] was to use a parser in order to find the nouns. The result was that the few errors it made were distracting and it was more comfortable to find the nouns manually. Ultimately, the idea of using a parser in order to find the nouns for abstraction identification was abandoned, because it did not inspire confidence that it found everything. More importantly, the parser would overlook an important noun because it appears to the parser as a verb. For example, in the phrase "book a flight", "book" is a verb and not a noun as thought to be by many parsers. Even a better, but still ultimately imperfect, parser does not solve this confidence problem. Finally, the abstractions are often noun *phrases* and not just words. In the same example phrase, the key concept is "flight booking" and not just "flight", the only real noun found in the phrase.

A second idea [1] was to use findphrases, a repeated phrase finder, a repetition-based approach. Counting isolated words in the text is not sufficient, because a lot of information is lost. In particular, information on the relationships in which words are involved is lost. Therefore, it is necessary to consider the phrases in which the words appear.

findphrases was found to be effective in aiding the human RA to identify abstractions in all stages of the life-cycle. However, one particular weakness was noticed. A repeated phrase finder fails to count as a repetition of "book a flight" the phrase "book the flight" since it looks for fixed patterns. Were each of these phrases to appear only once, the concept of "booking a flight" would not show up at all in the list of repeated phrases, even though the concept shows up twice. In many cases, concepts do not appear as adjacent words but rather a set of words separated but not more than a few words. Most of these concepts appear as closely separated pairs of words standing for an agent-object relation.

A third idea [10] was to use *lexical affinities* (LAs) as the atomic unit for identifying major abstractions within a text. An LA stands for the correlation of the common appearance of two items in sentences of the language [3]. For the purposes of the LA finder, the definition was restricted, by observing LAs within a finite document rather than on the whole language. For instance, in the present paper, "abstraction" and "identification" are bound by a lexical affinity.

The LA finder was found to be as a bit more effective in finding abstractions than the repeated phrase finder, but not much more. At present, the LA finder does not find LAs consisting of more than two words of common grammatical structure, verb-noun, adjective-noun, etc.

findphrases and the LA finder have each weaknesses that the other does not have. findphrases finds long phrases but identifies only fixed patterns, whereas the LA finder identifies nonadjacent words in possibly differing order but is limited to precisely pairs of words. Neither of them identifies synonyms.

One key point that emerged in the consideration of the past work is that it is critical for the tool to have guaranteed coverage, even if it is less intelligent. The lack of intelligence is no real drawback since the human RA has to analyze the output of the tool anyway. He or she will provide the missing intelligence. Indeed, there are some advantage to forcing the human to think carefully. However, to be sure that the thinking is supplied with full information, full coverage by the tool is critical. Particularly disastrous is a so-called intelligent tool that makes mistakes and leaves things out in its attempt to be intelligent.

2.3 New approach

This section describes a new approach that eliminates many but not all of the weaknesses of the older tools. While the new approach solves most of the weaknesses of the older tools, there are a few remaining.

2.3.1 Motivation and informal description: It is desired to determine for any pair of sentences, the set of chunks that they have in common independently of the order of these chunks in the sentences. The chunks in general will be words. However, many times, it is desired that these chunks be words sans suffixes and prefixes in order to capture the commonality in the form of the grammatical root of two occurrences of the same word in different parts of speech. Therefore, it is necessary to allow these chunks to not begin and end at word boundaries. That is, in the two sentences

The flights are booked
He is booking a flight

we wish to find the two chunks "flight" and "book", neither of which is a full word in both sentences. (The fact that they are in different orders in the two sentences is dealt with below.) The upshot of this desire is that the sentences are considered streams of characters with no particular status accorded to the usual word-ending characters such as blanks and punctuation.

One side effect of ignoring word boundaries is that *noise* can creep into the matching chunks. For example, among

book flight
book funny

the matching chunk is "book f". Fortunately, the human RA can ignore the "f" as meaningless. During prototyping, it was determined that attempting to algorithmically excise the noise caused significant material to be lost, e.g., in formulae, variables are significant single-character chunks. Also, we are counting on the intelligence of the human user of the program to recognize meaningful words from the chunks. Sometimes this may be difficult. Among

impossible to see
a possibility seems

the common chunks are "possib" and "see". The two main problems are illustrated here. Will a human be able to connect "possib" to the correct root "possible"? Will the human be misled to believing that "to see" is a common concept. To assist the human in finding abstractions and avoiding being misled, it will be necessary to print with an abstraction at least a pointer to the sentences involved.

A *run* in common in two sentences S and T is a string of consecutive characters that appears in both such that the character before the run in each differ and the character after the run in each differ. For a run to be significant, it is required that its length be greater than *WordThreshold*, a value that has to be set experimentally as described below. From the sentences (not really, but the example has to be kept short!)

file to ignore
the ignored files

the runs are "file" and "ignore". $Abst(S,T)$ is the set of all runs in common in S and T.

To find runs, a cyclic shifting algorithm is used. First, each sentence is padded by an extra blank to prevent the beginning of a sentence concatenated to its end from forming a spurious word. Then the shorter sentence is padded with more blanks to the length of the padded longer one. Finally, the padded longer sentence is concatenated to itself and the padded shorter sentence is compared for runs with the doubled sentence after positioning its beginning at each successive character of the first half of the doubled sentence. Figure 1 shows the steps of the run search for the example sentences. Note that it is not really necessary to pad the second occurrence of the longer sentence.

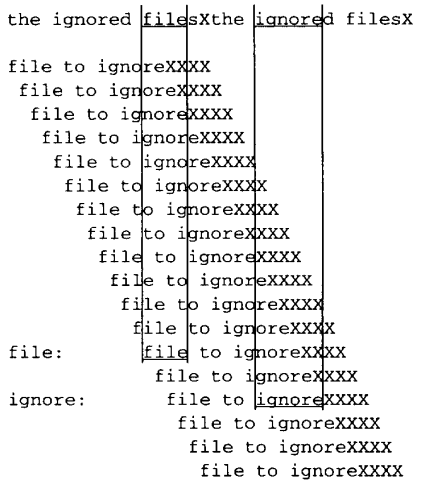

Figure 1: Finding runs in two sentences

This algorithm will be recognized as the traditional signal processing algorithm to find commonality in two signal streams [14]. In the new approach, a sentence is regarded as a stream of characters rather than a string of words. Perhaps the power of this approach comes from its treatment of a sentence as a stream of arbitrary characters with the subsignals appearing anywhere rather than

being constrained to fall on word boundaries.

2.3.2 Avoiding weaknesses of previous approaches: The new approach provides an effective way of identifying abstractions in natural language transcripts of client interviews, which allows

1. unlimited phrase length, within the confines of a sentence,
2. phrases with unlimited gaps between the words within a sentence,
3. arbitrary permutations of a phrase to be recognized as the same phrase,
4. automatic matching of subwords that share a common root, when the variation to other parts of speech is regular, e.g., as for "purchased" and "purchase".

The new approach solves the weaknesses of findphrases and the LA finder algorithms, of being unable to deal with phrases with arbitrary numbers of words, with arbitrary gaps between words of the phrases, and with arbitrary permutations of the words in the phrases.

One weakness of all previous methods remains, namely that of identifying as a single concept phrases that have nothing textual in common. There are two manifestations of this, irregularity in changes to other parts of speech, e.g., the past tense of "buy" is "bought", and synonyms. People use different words, called synonyms, for the same thing, and a particular word might appear less used than its concept actually is. Synonyms are used particularly when the requirements are written by more than one person. Both of these problems can be regarded as that of replacing one word by another. Therefore, the program, AbstFinder, containing the basic algorithm, has been provided a facility for synonym replacement, according to a dictionary that can be enhanced by the user.

2.3.3 Possible weaknesses of new approach: One problem of the new approach is to set the *WordThreshold* parameter. If it is not set high enough, then parts of words—called noise in signal processing terminology—might hide the the real abstractions to be identified. With too much noise, the human RA will not see the trees in the forest and will not find the abstractions. If the *WordThreshold* is set too high, then abstractions that are identified by a word shorter than the *WordThreshold* will be missed. The risk is that to get only meaningful phrases, the threshold may be set too high and not all abstractions will be found. So, it will be necessary to experiment with threshold values, and the values may prove to be different for each problem and possibly even different for different portions of a problem. Fortunately, based on our experience using AbstFinder, after a few uses of the systems, the RA learns to estimate a good setting for *WordThreshold* after one run and to recognize sections of the input that would be badly handled by any particular setting. In any case, coverage is not lost because the same job can be run with different settings of *WordThreshold*.

A second noise problem can be caused by words or phrases which are meaningful but do not contribute to the abstraction identification process. For each application area, there appears to be characteristic sets of (1) common words and (2) application-dependent keywords, which appear often enough to skew the list of abstractions, making it harder for the RA to find real abstractions.

1. The common words, e.g., "a", "on", "the", "in", etc. obviously do not identify any abstraction. One should fill an *ignored-phrases-file* with common words, in order to mark them for not taking part in the similarity calculation. The *ignored-phrases-file* can also accumulate application-independent words that can be used for any project.
2. The application-dependent keywords are actually important and repeat a lot in the text. For example, in the text of the RFP case study (See Section 4.2), entitled "Unmanned Aerial Vehicle (UAV)", the words "unmanned", "aerial", "vehicle", and "UAV" appear in almost every sentence. One should fill an *ignored-application-phrases-file* with these frequent application keywords, which identify larger abstractions than are useful.

Filling these ignored phrases files requires experimentation and is basically a learning process. This process is described in Section 3.

2.3.4 AbstFinder program: The AbstFinder program incorporates the algorithm described in the previous section. A pseudo code rendition of the program is shown in Figure 2 at the end of the paper. AbstFinder's algorithm uses the information yielded by $Abst(S,T)$ for all distinct combinations of two sentences S and T. A sentence is not compared with itself, but no attempt is made to avoid comparing a sentence to another sentence that happens to be a duplicate. Recall that the set of runs returned by an invocation of *Abst* on one pair of sentences, is called the set of *abstractions* for that pair of sentences. The main data structure of the program is corr_phrases, an array with at most one element per sentence. Each entry in corr_phrases is the set of abstractions obtained by comparing one sentence to all of the other sentences; any sentence for which no phrases are found does not have an entry in corr_phrases.

The output of AbstFinder comes in two parts. The first part is a table summarizing the identified abstrac-

tions, and the second part gives a full description of each of the abstractions. Appendix 1 shows the first part of a run of AbstFinder that features in a later discussion. There is one row in the table per identified abstraction. The first field, labeled "#)", gives a serial number for the abstraction. The field labeled "Abst#" gives the abstraction number assigned by AbstFinder as it found its first phrase (the NA of the algorithm). The "phrase#" field gives the number of distinct phrases that were united into the abstraction by AbstFinder. The "lines#" field gives the number of distinct lines or sentences that contain these phrases. Finally, the "correlated-phrases" field shows the phrases themselves with vertical bars in between them and after the last one. Each blank starting from the second column after the beginning of the field is significant and is part of its run. This field is truncated by its flowing beyond the physical width of the paper. Even if the phrases are truncated, the full list may be found in the corresponding entry in part 2.

An abstraction identified by one phrase is more distilled than one that is identified by more phrases. So, the first criterion for ordering the abstractions in the output is in order of increasing numbers of correlated phrases. Then, when two abstractions have the same number of correlated phrases, the second criterion for ordering is in order of decreasing numbers of sentences from which the phrases came. The more sentences contained in an abstraction the more significant it probably is.

The AbstFinder program can be thought of as a kind of clustering [12] with the length of a run as the criterion. However, the abstraction classes are not predetermined and do not form a partition.

2.3.5 Performance analysis of program: As mentioned in Section 2.3.3, the *WordThreshold* parameter must be set very carefully. In order to be able to identify single word matches, *WordThreshold* has to be set to 3, if we assume that the minimum length of a meaningful word is 3 characters. However, while testing the tool, It was decided to keep all input spaces between words, and to take them into consideration while calculating similarity. Therefore, a threshold of 3 characters was found to be too low, as it yields meaningless runs of the form "x y" from around interword gaps. So, the threshold was raised to 5 characters, and the result was that AbstFinder appeared to capture only meaningful phrases.

Another important concern is the performance of AbstFinder. The time complexity of AbstFinder is $o(c \times N^2)$, where N is the number of sentences in the document, and c depends on the length of the sentences, which can be regarded as bound by natural limits. There are faster algorithms based on the use of tries or Patricia trees. These can be made $o(c \times N)$ if desired [7]. However, experience with AbstFinder on an industrial-sized example shows that its real performance problem is space; and this problem is only exacerbated when the faster algorithms are used. In any case, it is no problem for the RA to go out to lunch while waiting for it to report on a large input. Moreover, generally speaking AbstFinder is run only on early documents only for the purpose of assisting in identifying abstractions. Therefore, the slow runtime of AbstFinder on large files is no real burden.

3 Usage of AbstFinder

This section describes a typical scenario that an RA might follow in order to have AbstFinder help identify the abstractions in a new problem given to him or her by a client. It is assumed that the RA has on-line what the client believes is a complete description of the system to be built, written mostly in some natural language.

First some trial runs need to be done on small parts of the transcript, taken from different sections of it, in order to learn the language of the document. Learning here consists in identifying the ignored words, putting the common words into the *ignored-phrases-file*, and the special application words into the *ignored-application-phrases-file*. Besides words to ignore, the *ignored-application-phrases-file* may later contain very important high level abstractions that have been recognized, noted, and put into the *ignored-application-phrases-file* in order not to clutter up the output.

When using AbstFinder with huge transcripts, the RA should read the output list of abstraction and note the abstractions identified by fewer than four or five phrases. Abstractions identified by more than five phrase are difficult to understand. They are also often extraneous because they capture concepts that are too general to be useful. The extreme example is the one abstraction that identifies the whole transcript, and that abstraction is clearly not very useful. Therefore, the RA has to stop at some point, at which the abstractions are still useful, and beyond which they are not useful. It may be impossible to find a single point meeting both criteria, so often the RA has to settle for a point beyond which they are not useful.

The main purpose of the clerical tool is to identify *all* meaningful abstractions. Without full coverage, the RA will never trust the tool to not overlook something. So if the list is cut off just before the point at which abstractions are identified by six phrases, the concern is whether there are any abstractions that are not recognized because they are identified by more than five phrases. In order to eliminate any worry about a possible lost of abstractions, the following iterative procedure should be carried out.

Activate AbstFinder once on the original document. Then, with the strainer program, remove from the origi-

nal document the abstractions already recognized and logged by the RA, leaving what is left in another file *f*. Then, activate AbstFinder on *f*. The result of AbstFinder is a new list of abstractions, without the ones that were recognized before, but with some that had been buried in the first abstraction list after the cut-off point.

The process repeats until finally the RA is left in *f* with a very short list of abstractions, which are all meaningless. That meaningless list indicates that all the meaningful abstractions were identified previously and strained out from the transcript. The accumulated list of meaningful abstractions provide full coverage.

This iterative way of applying AbstFinder and then strainer, is suitable for a human RA to capture large amounts of information. Doing it step by step allows him or her to look each time over a limited, readable, and understandable amount of information and to accumulate it. The RA is confident that nothing is overlooked, because things that have not been seen yet will pop up in some later iteration. The iterations continue until finally she or he is sure that the document has been wrung dry of abstractions.

4 Evaluation of AbstFinder

This section considers the evaluation of the effectiveness of AbstFinder for finding abstractions in natural language text. It is first necessary to explain how such a tool can be evaluated with the help of case studies. Then two of the case studies are described. These lead to the conclusion that for them, AbstFinder is indeed effective.

With any new idea of a method or a tool one must evaluate its effectiveness. First, it is useful to compare the new tool to old tools, such as findphrases and the LA Finder, to verify that the new tool does at least as well or better than the old ones.

One must really test such a tool against a human effort, since heretofore requirements elicitation has been done manually by a humans. There is no simple analytic method for testing human efforts. As for many software engineering issues, controlled experiments are out of the question. Running sufficient numbers of instances to obtain significant results is prohibitively expensive when the instances involve industrial-sized problems [13]. Moreover, too often, individual differences dominate the controlled variable of the experiment [11].

An important issue is the question of whether the abstractions found by the tool are meaningful to the human RA that has to approve them. Meaningfulness can be confirmed only by humans, and is very much affected by the *WordThreshold*.

The key objective measures of the effectiveness of AbstFinder are: (1) its coverage, and (2) how summarizing it is. A tool that is not covering or which does not summarize is not good, for the following reasons:

It must be that this tool does not overlook any important abstraction that will need to be present in the requirements specification. A tool that does not overlook important abstraction is said to be *covering*. An RA will not be willing to be assisted by any tool unless he or she is confident that it is covering.

Clearly, the identity function is a covering tool. However, presenting all the input does not help the RA either. The other main requirement for the tool is that it reduce the amount of text that the RA must look at. A human RA still has to do the *thinking* with the output of the tool, in order to approve the abstractions found. The RA will not be effective if the amount of information that must be examined is too big. A tool whose output is significantly smaller than its input is said to be *summarizing*.

Note finally, that a tool that is only summarizing is no good either. The most summarizing tool is that which outputs nothing. The tool must summarize while preserving coverage.

Measuring the ability to summarize is easy. It is done by simply comparing the ratio of sizes between the input transcript to the output of AbstFinder. Coverage is much harder to measure. One must compare the list generated by AbstFinder to that made by a known expert (and pray that in fact the expert is good) and judge whether all concepts found in the latter are present in the former. There is no better measure than experience, and ultimately the proof will be in acceptance of tool by the RA community.

To put the evaluation in context, it is important to understand typical scenarios of abstraction identification with and without AbstFinder to help the RA.

Without AbstFinder, the RA

1. reads all the documents once to get a sense of what is there, and
2. then repeatedly reads individual documents and parts thereof in order to find and verify abstractions until no more new abstractions are found.

With AbstFinder, the RA

1. reads all the documents once to get a sense of what is there, and
2. follows the iterative procedure of Section #iteration# until no new abstractions are found.

Thus, in traditional abstraction identification, the full set of documents are read over and over, with no prior limit. In AbstFinder-assisted abstraction identification, if the output is covering, the full set of documents is read only once. Thereafter, only the much smaller summaries need to be examined over and over. Besides being smaller than

the full set, the summaries gather the most important concepts to the top of the list for a better focus.

For the evaluation, since the first steps of the two scenarios are the same, the comparisons focuses on the differences in what must be examined for the second steps.

Two of the case studies used to evaluate AbstFinder are described below.

4.1 Findphrases case study

The findphrases decomposition was used as a case study because the decomposition was already known. Moreover, it had been the subject of case studies for abstraction identification with the help of findphrases and the the LA Finder. Therefore, it could be used to check AbstFinder's results against already known results and against previous tools. The document that served as the requirements was the manual page of findphrases, because in fact the manual page was written before the program was written as a requirements document. The already known abstractions were taken from Aguilera's [1] program decomposition, and her own list of abstractions identified by findphrases, and Maarek's [10] list of abstractions identified by lexical-affinities (See Table 1).

As shown in Appendix 1, the first 25 of the 48 entries of the AbstFinder output list includes all the abstractions found by Aguilera in implementing findphrases, all abstractions found by findphrases, and all abstractions found by Maarek with lexical affinities. So, for this case study, AbstFinder was found to be at least as covering as findphrases and the LA finder and was found to cover all abstractions found by a human programmer.

4.2 RFP case study

The Request For Proposal (RFP) document for the Unmanned Aerial Vehicle-Short Range (UAV-SR) system is a large industrial-strength case, about 100 pages long, containing about 2200 sentences, which we were lucky to get. The RFP transcript was already analyzed by three experts over a month, for a total effort of three person-months. The experiment consisted of the first author, called "the RA" below, analyzing the RFP with the help AbstFinder, and comparing the resulting abstractions list to the list of requirements produced by the three experts. The list of requirements produced by the three experts is called "the human-made" document below. The RA did not see this human-made document until after she had finished generating her list with the help of AbstFinder. The hope was the RA would find *meaningful* abstractions in a *summarizing* output list of AbstFinder while providing full *coverage* of the client's requirements, and with a lot less effort than three person-months.

The abstractions of findphrases	
Data Abstraction	Phrases
string_type_file	strings, characters
argument_line	argument, option
output_file	output, tables of the output
chunk_file	file(s), free format
punc_keyword_table	punctuation keyword(s) file
multi_tokens_table	multi tokens file
text_file	text, input, arbitrary text
phrases	phrase(s), repeated phrases, ignored phrases
sentences	sentence(s)

Table 1: The abstractions of findphrases

In the thesis of the first author, the effectiveness of AbstFinder for the RFP case study is evaluated. Space limitations do not permit presentation of the full details of this evaluation. Instead, this section indicates how the evaluation was carried out and the draws only the specific and general conclusions.

4.2.1 Meaningfulness: After the AbstFinder user was finished generating what she thought was a complete list of abstractions, the phrases in this list were examined by the three expert analysts of the RFP transcript. They all found all of the AbstFinder-generated phrases to be meaningful to them. One of them was very impressed to see in the beginning of the AbstFinder-generated list some abstractions, such as "surrogated training", that they had overlooked for a long time until finally the customer pinned it on their noses.

4.2.2 Summarizing: The output of AbstFinder was summarizing. The original document RFP was 169,323 bytes long whereas AbstFinder's final result was only 47,105 bytes, about 25% of the size of the original data.

4.2.3 People and computer power: Using AbstFinder helped the RA to get the list of abstractions faster. The list of requirements generated by the three experts required one month of concentrated work for a total of three person-months. Running AbstFinder took about five hours total CPU time, three hours operating time, and about two hours of RA overview, which is about one day of work. The first run of AbstFinder on RFP took about two hours. The second run, after straining out the top, most frequent abstractions on the list, took about 30 minutes. The last run took about 5 minutes. However, note that the RA was doing other things while the CPU

was running.

4.2.4 Coverage: The problem with evaluating coverage is that someone must sit down and see that all abstractions in the human-made document show up in the AbstFinder-generated list. The high probability of error in this tedious job makes any claimed "yes" answer highly suspect. In addition, the person doing the job had a vested interest in finding a "yes" answer. Therefore a more systematic way to evaluate coverage had to be found.

The coverage question can be answered by straining from the human-made document all abstractions that appear in AbstFinder's result and seeing if there are any leftovers. No leftovers means full coverage. The smaller size of the leftovers and the greater visibility of meaningless text increases the credibility of the answer. The result of that subtraction was 3019 bytes. The RFP is 169,323 bytes (about 100 pages) long and the human-made requirements document is 83 pages (about 140K bytes) long. The phrases of the remainder were analyzed very carefully in order to find out if AbstFinder missed any abstraction. The phrases of the remainder were separated into several categories according to their characteristics.

Most of the phrases were in the meta-language for requirements specification such as "activate", "allow", "deactivate", "herein", "include", "integrate", "must", "only", "provide", which are not meaningful abstractions. Perhaps these should be added to the *ignored-phrases-file*.

Some concept were in different grammatical forms such as "transmit" in one document and "transmitting", "transmitters", and "transceiver" in the other one, or "calibrate" and "calibration", or "assigned" and "assignment". Those are actually the same abstractions. While AbstFinder is able to classify all the "transmit..." words, all the "trans...", etc. as single abstractions, strainer works on whole words and is not able to remove words that properly contain a recognized root.

Acronyms such as "NBC" are introduced to replace a longer full phrase such as "Nuclear, Biological, Chemical"; the full phrase appears only once at the introduction of the acronym or in a dictionary of acronyms and the acronym appears many times throughout the document. The acronyms are used to save the writing of the longer full phrase. AbstFinder did not identify many acronyms and failed to find most of the full phrases that the acronyms replaced. Many acronyms are shorter than the *WordThreshold*, and a full phrase that appears only once is not going to be caught by any frequency-based scheme. Given that reducing *WordThreshold* causes generation of too much noise, there are two solutions, both general enough to be made part of a standard scenario for the RA.

1. The synonym dictionary can be used to replace the acronyms by their full phrases for the purpose of abstraction identification.
2. Recognize all the acronyms as important abstractions, log them as abstractions, and then add them to the *ignored-application-phrases-file*.

The latter solution is useful for the RA who actually prefers to work with the acronyms.

To sum up, after some generally applicable modifications that should be part of a standard scenario for use of AbstFinder, it was clear that full coverage was achieved in the AbstFinder-assisted abstractions list.

4.2.5 Does better than human experts: We were interested to see if AbstFinder found some concept that the human RA overlooked. This meant checking if the list of requirements in the human-made document cover the list of abstractions found by AbstFinder. That question was answered by having strainer remove from the AbstFinder abstraction list all that appears in human-made document to see if there are any leftovers in the AbstFinder results that the humans overlooked.

The result was about 35,402 bytes long. In these bytes were found concepts that were hidden in the Classified Requirements Appendix of the RFP document, non-software requirements, and concepts, such as "surrogated training", that the human experts did overlook. Therefore, we got the impression that a human RA assisted with AbstFinder can do better than several unassisted human RAs.

4.3 Results

For the specific case studies carried out,

1. AbstFinder was found to be at least as good as findphrases and the LA finder on the findphrases requirements. All the abstractions found by findphrases and the LA finder were found on the top of the output list of AbstFinder.
2. a human RA assisted by AbstFinder is at least as good as three human experts on the RFP, and in fact found some abstractions that the experts did not find.

The conclusion is that for the case studies presented, AbstFinder is good, it has coverage and it is summarizing.

5 Conclusions

More experiments on industrial sized examples must be carried out. With each such experiment, it is important to have a qualified, independent analysis available with

which to compare the AbstFinder-generated list of abstractions. To encourage such experiments, the authors are making the source code of the tool available. Please contact the second author at *dberry@cs.technion.ac.il* for more details.

Also now that the prototype has successfully proved a concept, it is time to consider scrapping the oft-modified prototype in favor of a freshly written production version, in which better algorithms and data structures are used.

References

[1] Aguilera, C. and Berry, D.M., "The Use of a Repeated Phrase Finder in Requirements Extraction," *Journal of Systems and Software* **13**(9), p.209–230 (1990).

[2] Berry, D.M., Yavne, N.M., and Yavne, M., "Application of Program Design Language Tools to Abbott's Method of Program Design by Informal Natural Language Descriptions," *Journal of Software and Systems* **7**, p.221–247 (1987).

[3] Cruse, D.A., *Lexical Semantics,* Cambridge University Press, Cambridge (1986).

[4] Goguen, J.A. and Linde, C., "Techniques for Requirements Elicitation," pp. 152–164 in *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA (January, 1993).

[5] Goldin, L., "An Environment for Aiding Requirements Analysts in Requirements Elicitation for Large Software Systems," Ph.D. Dissertation, Faculty of Computer Science Department, Technion, Haifa, Israel (December, 1993).

[6] Holtzblatt, K. and Jones, S., "Contextual Inquiry: Principles and Practice," Technical Report DEC-TR 729, Digital Equipment Corporation (October, 1990).

[7] Knuth, D.E., *The Art of Computer Programming: Sorting and Searching,* Addison-Wesley, Reading, MA (1973).

[8] Leite, J.C.S.P. and Franco, A.P.M., "A Strategy for Conceptual Model Acquisition," pp. 243–246 in *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA (January, 1993).

[9] Luhn, M., "The Automatic Creation of Literature Abstracts," *IBM Journal of Research and Development* **2**(2), p.159–165 (April, 1958).

[10] Maarek, Y. and Berry, D.M., "The Use of Lexical Affinities in Requirements Extraction," Technical Report, Technion, Haifa, Israel (November, 1988).

[11] Sackman, H., Erickson, W.J., and Grant, E.E., "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM* **11**(1), p.3–11 (January, 1968).

[12] Salton, G., *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer,* Addison-Wesley, Reading, MA (1989).

[13] Schach, S.R., *Software Engineering,* Aksen Associates & Irwin, Boston, MA (1992). Second Edition.

[14] Sklar, B., *Digital Communication Fundementals and Applications,* Prentice-Hall, Englewood Cliffs, NJ (1988).

[15] Wood, D.P., Christel, M.G., and Stevens, S.M., "A Multimedia Approach to Requirements Capture and Modeling," in *Proceedings of the International Conference on Requirements Engineering*, Colorado Springs, CO (April, 1994).

Read a *punctuation-keyword-file*, an *ignored-phrases-file*, an *ignored-application-phrases-file*, and a *synonyms-file*;
Partition the text into one sentence per line, a sentence being the text lying between two consecutive elements of the *punctuation-keyword-file*;
Remove from the text substrings found in the *ignored-phrases-file* and substrings found in the *ignored-application-phrases-file*, and replace words by their synonyms according to the *synonym-file*;

declare N := number of lines; **comment** = number of sentences **tnemmoc**
  declare corr_phrases[1:N], corr_lines[1:N];
  declare NA := 1; **comment** number of abstractions accumulated so far ≤ N **tnemmoc**
  for i **from** 1 **to** N **do**
    corr_phrases[NA] := ∅; corr_lines[NA] := {i};
    for j **from** i+1 **to** N **do**
      if *Abst*(line[i],line[j]) ≠ ∅ **then**
        corr_phrases[NA] := corr_phrases[NA]∪*Abst*(line[i],line[j]); corr_lines[NA] := corr_lines[NA]∪{i}∪{j} **fi od**;
    if corr_phrases[NA] ≠ ∅ **then** NA := NA + 1 **fi od**;
  NA := NA – 1; **comment** correct overshoot **tnemmoc**

Sort both corr_phrases and corr_lines so that correspondence between corr_phrases[i] and corr_lines[i] is preserved and the elements of corr_phrases are ordered mainly by increasing numbers of phrases in the elements and within the group for any number of phrases, by decreasing numbers of lines from which the phrases came;
Prepare and print the output as described in Section 2.3.4;

Figure 2: The AbstFinder **program**

Appendix 1: (Output is edited to get more of it to fit on this page)

```
# of lines read from input file is 54      # of abstractions found is 48

 #)     | abst# |phrase#|lines# | correlated-phrases
------  |------ |------ |------ |--------------------
 1      | 42    | 1     | 11    |  punctuation keyword file|
 2      | 14    | 1     | 2     |  whitespace|
 3      | 6     | 1     | 2     | argument |
 4      | 38    | 2     | 25    |  phrase| phrase |
 5      | 44    | 2     | 14    | s file|tokens file|
 6      | 23    | 2     | 12    | tokens |s sentence|
 7      | 16    | 2     | 9     | character|symbolcharacter|
 8      | 43    | 2     | 8     | multi |r multi |
 9      | 2     | 2     | 6     |  free format |files |
 10     | 4     | 2     | 5     |  blank| blank tab newline|
 11     | 41    | 2     | 3     | files | configuration |
 12     | 11    | 2     | 3     | arbitrary text|input |
 13     | 26    | 3     | 29    |  phrases |file phrases file | phrases file phrases |
 14     | 5     | 4     | 28    | phrase|argument |optional | phrase |
 15     | 24    | 4     | 26    | phrases |s sentence| phrase|s sentences|
 16     | 45    | 4     | 26    | phrases a| phrase| phrases | configuration |
 17     | 20    | 4     | 17    | s file | multi tokens file|e token| multi tokens file |
 18     | 33    | 4     | 12    | keyword|keyword |keyword p|d prev|
 19     | 12    | 4     | 12    | character|words |symbolcharacter|characters|
 20     | 25    | 5     | 31    |  phrase|tokens| phrase |e tokens| phrase t|
 21     | 0     | 5     | 26    | phrases |arbitrary text|d phrase| phrases |phrases a|
 22     | 29    | 5     | 25    | phrases| phrase| phrases|table phrases| table|
 23     | 35    | 5     | 25    | phrases| phrase| phrase | phrases| phrase s|
 24     | 36    | 5     | 25    |  phrase|e phrase|phrase | phrase | phrase phrase |
 25     | 39    | 5     | 25    |  phrases |output| tables | phrases s|tables output|
 26     | 9     | 5     | 14    | tokens f|free format| line |e token|e tokens|
 27     | 15    | 5     | 11    |  blank| blank tab newline|file f|character|wordcharacter|
 28     | 17    | 5     | 10    |  blank |character | character |wordcharacter|wordcharacter |
 29     | 3     | 5     | 10    | file m| free format | blank| line |blanks|
 30     | 21    | 6     | 20    |  punctuation keyword file |multi token|character |symbolcharacter multi token|listed |e sentence|
 31     | 22    | 7     | 37    | d phrase|punctuation keyword |tokens | list |s delimited |keyword p| punctuation keyword |
 32     | 48    | 7     | 28    | phrase|phrase |option| option| option |e sentence| phrase |
 33     | 34    | 7     | 26    | phrase|e phrase|phrase |s sentence|s sentences| phrase |e phrase |
 34     | 30    | 7     | 25    | phrase|phrase | phrase |s phrase | table|consists | tables |
 35     | 7     | 7     | 19    |  punctuation keyword file |free format | free format |file free format list character strings |words |punctuation keyw
 36     | 32    | 8     | 27    | phrase| phrase|blanks| blank| phrase | phrase t|table | table|
 37     | 37    | 8     | 27    | phrase|phrase |option|option | phrase | b option |s phrase |option phrase|
 38     | 8     | 9     | 30    | phrases |phrases file |optional | line | optional | phrase| phrases file |e list |al phrase|
 39     | 18    | 9     | 17    |  punctuation keyword|character |symbolcharacter |characters| whitespace|wordcharacter| wordcharacter|e symbolcharacter
 40     | 1     | 10    | 39    | phrases |file m| punctuation keyword file |phrases file |tokens f|multi tokens |multi tokens file| multi tokens file|
 41     | 40    | 10    | 30    | phrases |e phrases |phrases f|s file|phrases file | phrases file |s phrase|se phrase|d prev| phrases s|
 42     | 19    | 11    | 18    |  punctuation keyword file |multi token|keywords |character strings |character strings punctuation |symbolcharacter |wo
 43     | 28    | 12    | 34    | phrases| punctuation keyword|punctuation keywords |input | phrase| phrases|table phrases| table|consists |t lines |out
 44     | 27    | 12    | 29    | phrases |phrases file |file phrases |phrase |option| option|option | phrase | phrases file phrases |s phrase |es phras
 45     | 31    | 12    | 25    | phrase|e phrase|phrase |e list |al phrase| phrase |es phrase|t lines |e phrase | phrase s| phrase phrase |phrase lis|
 46     | 47    | 13    | 39    | phrases |tokens | punctuation keyword|multi tokens |option|punctuation keywords | option|option multi tokens |listed |
 47     | 10    | 13    | 29    | multi tokens |multi tokens file|s file |free format | free format |optional |file free format list character strings |
 48     | 13    | 13    | 28    |  punctuation keyword|file m|multi tokens file| punctuation keyword file |s file |option|punctuation keywords | option|
```