Automated Software Engineering 4,375-412 (1997)

AbstFinder, A Prototype Natural Language Text Abstraction Finder for Use in Requirements Elicitation*

LEAH GOLDIN AND DANIEL M. BERRY dberry@cs.technion.ac.il
Faculty of Computer Science, Technion, Haifa 32000, Israel

Abstract. Abstraction identification is named as a key problem in requirements analysis. Typically, the abstractions must be found among the large mass of natural language text collected from the clients and users. This paper motivates and describes a new approach, based on traditional signal processing methods, for finding abstractions in natural language text and offers a new tool, AbstFinder as an implementation of this approach. The advantages and disadvantages of the approach and the design of the tool are discussed in detail. Various scenarios for use of the tool are offered. Some of these scenarios were used in case study of the effectiveness of the tool on an industrial-strength example of finding abstractions in a request for proposals.

Keywords: abstraction finder, natural language text, requirements elicitation, evaluation of tool, tool use method

1. Introduction

1.1. The Problem

Requirements are often ill-defined, fuzzy, ambiguous, incomplete, or simply incorrect with respect to the users' needs. Problems in the system caused by deficiencies in software requirements are often not identified until well after the system is deployed (Martin, 1988), or are thought to be caused by bad design or limitations of computing technology.

It is well known that as much as 60% of the errors that show up during a system's life have their origin in the requirements gathering and specification stage (Davis, 1990; Schach, 1992). It is also well known that the cost to correct an error found in the development and later stages of system development is orders of magnitude higher than to correct the same error found during the requirements gathering and specification stages (Boehm, 1981). The importance of getting the requirements right cannot be underestimated.

On the other hand, it appears that the least understood step of systems development is the requirements elicitation and specification stage, and that within this stage, elicitation is less understood than specification. The difficulty of elicitation also cannot be underestimated.

The problem is that there is a tremendous gap between the client's needs and the software engineer's understanding of the client's needs. The gap is widened by the fact

* This paper is an expansion of a paper published by the same authors in the *Proceedings of the First International Conference on Requirements Engineering* (Goldin, 1994a)

that the client may not even be able to verbalize his or her own needs. The client speaks with fuzzy sentences replete with tacit assumptions, and the software designers are just not able to identify his or her intentions.

Many system design or programming methods, e.g., those of Jackson (Jackson, 1975), Parnas (Parnas, 1972), Booch (Booch, 1986), Myers (Myers, 1979), Orr (Orr, 1977; Orr, 1981), etc., 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. It is clear that writing of the requirements is a *major* part of the problem solution, and that when this writing is done properly, many pitfalls in the path of delivering the required system may be avoided.

Large E type (Lehman, 1980) 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 deceptively similar to each other through 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.

Full discussions of the problems of obtaining good requirements and of the effect of the failure to obtain them may be found in a textbook by Davis (Davis, 1990) and in a paper by Krasner (Krasner, 1988).

1.2. Brief History of Requirements Engineering Methods and Tools

The early work in requirements engineering was focussed on requirements specification and analysis (Davis, 1990). The tools and methods of the time permitted and assisted in the organization of the requirements, refinement of details, consistency checking, preparation of the specification, and in some cases, formalization of these requirements (Alford, 1977; Ross, 1977; Teichroew, 1977; Alford, 1978; Zave, 1982; Burstin, 1984; Alford, 1985; Borgida, 1985; Sievert, 1985; Estrin, 1986). Some of these tools also provided a means to relate design and implementation artifacts to their individual requirements to assist in the eventual validation of the realization of the requirements. The starting point for these tools and methods is a written statement of the requirements, usually a list of sentences, perhaps in a highly constrained subsets of English. However, none of these tools and methods give much help in actually obtaining the sentences in the first place and in recognizing the relevant abstractions, especially in the context of a large client organization. Methods and tools are needed to assist in getting sentences, writing them down, and finding the abstractions from which the requirements can be written.

More recently the software engineering community has been paying attention to the problem of eliciting the raw information from the clients (Lubars, 1993). Some of this work, e.g., contextual inquiry (Holtzblatt, 1993), has focused on observing the client's organization in action and modeling what it does and why. Goguen and Linde survey a variety of methods used in requirements elicitation as well as their own ethnometric approach (Goguen, 1993). Goguen himself has pointed out the necessity of considering the entire social situation in eliciting requirements, because there are some requirements that simply do not come from anyone's mind; they have to be invented by the stakeholders working together explicitly for that purpose (Goguen, 1994). Leite and others have devised methods and tools for building the vocabulary of the problem domain and of resolving different viewpoints among the stakeholders (Leite, 1991; Leite, 1993). Several have considered the problem of resolving the myriad, possibly inconsistent viewpoints of the stakeholders in the problem (Leite, 1991; Finkelstein, 1992). Others have attempted to work with problem descriptions in a restricted natural language to produce formal descriptions (Saeki, 1987; Ishihara, 1993). Ryan discusses the general role of natural language processing in requirements engineering (Ryan, 1993). A number of people have considered prototyping as a vehicle for requirements discovery and validation (Luqi, 1992; Bowers, 1994) Zahniser has shown that a paper prototype consisting of self-sticking paper stuck to a storyboard is also an effective means to discover and validate requirements (Zahniser, 1993). Further information can be found in a special issue of *IEEE Software* (IEEE, 1994a) and in three recent requirements engineering conference proceedings (IEEE, 1993; IEEE, 1994b; IEEE, 1995)

It is interesting to observe that the software engineering methodology literature abounds with papers and books that use a small example to illustrate a program development method. The example starts with a brief problem statement, i.e., the requirements. The second author's personal experience writing such examples is that getting that problem statement to say what it should took as much time as carrying out the development and doctoring it to look like it followed the method. Of course, the final paper or book never mentions this fact.

1.3. Envisioned Requirements Gathering Environment

This work is aimed at producing an essential part of an envisioned integrated environment for gathering, sifting, and writing requirements. This environment may very well be part of a large environment used for software development, deployment, and maintenance (IEEE, 1988). For now, the environment is described as helping the human requirements analyst (RA) massage transcripts of interviews with members of a client organization into a consistent, complete, unambiguous, coherent, and concise statement of what the organization wants. We do not care what language is being used either for the interview transcripts or for the final requirements. The environment should support any possibility. Usually the input to the environment will be a natural language transcript, possibly with pictures (Harel, 1987), but the environment should support any language possibility. However, the output language in which the requirements are written, can be anything from natural language with pictures, to any of the requirements expressing languages mentioned in Section 1.2.

Since we do not know enough about effective requirements writing to be able to codify the process, it is our feeling that a completely expert-system approach is out of the question, at least for now. We therefore envision an environment consisting of clerical tools that help with the tedious, error-prone steps of what a human RA does.

The goal of the environment is to organize the whole collection of requirements information as a network of nodes each denoting an abstraction and containing a description of all that is known and required about the abstraction, such as described by Dardenne, van Lamsweerde, and Fickas (Dardenne, 1993).

Among the tools the environment needs is one to help identify the abstractions that will make the nodes from the transcripts of the interviews. Abstraction identification is performed on any and all information that can be gathered from the client and its representatives, including users. We call this mass of information the *client information* (CI). It is assumed first that all the CI to be considered in abstraction identification is available online in simple ASCII form. Ideally, this CI should be what the client believes is a complete description of the system to be built. This description is written mostly in some natural language, but it can contain pictures if the text of the picture ends up being easily found, e.g., as in a PostScript description of a line-drawn figure. It will be necessary to enter transcripts of any verbal interactions that are considered relevant.

While the person doing requirements engineering is generally called a *requirements analyst*, the person doing elicitation, including abstraction identification, is called an *elicitor*. This is the term used in the rest of the paper.

1.4. Outline of the Rest of the Paper

Section 2 gives and operational definition of “abstraction” and describes previous and the current approach to finding them in natural language text. It ends with a description of a new clerical support tool, AbstFinder, for requirements text. Section 3 describes scenarios for the use of AbstFinder in the requirements elicitation process. Section 4 considers the evaluation of the effectiveness of AbstFinder through a case study of its application to an industrial strength requirements engineering problem. Finally, Section 6 draws conclusions. More details can be found in the Ph.D. thesis on which this paper is based (Goldin, 1994b).

2. Abstraction Identification

This section describes past and a current effort to establish automatic assistance for identifying abstractions. First, however, it is necessary to attempt to define *abstraction* so that it will be understand exactly what is supposed to be identified.

2.1. Operational Definition and Assumptions

Abstraction, in general, is ignoring details. When people try to understand a written requirements document, they usually abstract the contents. In this case, abstracting means ignoring enough details to capture the main ideas or concepts in the document. What details are ignored cannot be defined formally, or even informally. However, everyone involved with a project seems to know an abstraction when it is presented. Such a definition is not workable. Therefore, operationally, abstraction identification is defined as identifying some words from the written requirements document, and it is hoped that the scheme for selecting the words yields words that help humans to understand the abstraction. For example, if the words “book” and “flight” are selected as identifying an abstraction, it is hoped that the humans involved in an airline reservation system will understand that the concept of booking (reserving) a flight, and not of books (literature) about flight, has been identified. Therefore, an operational definition of *abstraction identifier* is given and it is hoped that what is yielded does help the human spot abstractions.

The *abstraction identifier* of an abstraction in a document is defined as a set (in the non-ordered sense of the word) of *chunks* within one sentence from the document. A chunk is a sequence of arbitrary consecutive characters from the document, which may include blanks. It is intended that when a human being reads an abstraction identifier, he or she will have no trouble understand the identified abstraction. For example, {“book”, “flight”} is an abstraction identifier from sentences containing “booking a flight” and “the flight was booked”. The human being, in reading these sentences in a document about an airline reservation system should have no trouble recognizing the concept of booking a flight as the abstraction identified. Of course, whether in fact, the human understands the identified abstraction remains to be demonstrated in practice.

Each abstraction identifier is used for retrieval of the abstraction’s *contents*. An abstraction’s contents is the set of all sentences in the transcript that contain all elements of the abstraction identifier. The contents of an abstraction derived from the initial text received from the customer may be ambiguous, incomplete, and inconsistent. Negotiation with the customer will be needed in order to resolve inconsistencies and to add more information in order to obtain useful requirements. Obtention of requirements from abstractions is a laborious activity and lies outside of the scope of this paper.

In any case, an abstraction is not equal to a requirement. According to IEEE Standard 610.12-1990, a requirement is defined as “condition or capability needed by a user to solve a problem or achieve an objective”. Thus, an abstraction can be thought of as higher level than requirements. The correspondence between requirements and abstractions is many to many. The importance of the abstractions is that they can serve as an *initial* list for requirements, and be used for the negotiation with the customer.

Until now, elicitors have identified abstractions manually. An elicitor scans the CI, trying to note important subjects and objects of sentences, i.e., nouns and noun phrases, and determines the abstractions from them. However, humans get tired and overlook relevant ideas. So it is useful to have tools that do the clerical part of the search for abstractions without overlooking anything. The elicitor still has to do all of the *thinking* with the output of the tools, but he or she will have confidence that no relevant bit of information has been overlooked in the gathering of input for abstraction identification.

No matter what, the elicitor must read all the input at least once. The larger this input, the more that must be digested in the elicitor'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 relying only on reduced input is that something important might be overlooked. Therefore, the tools must engender confidence that nothing important has been lost in the reduction.

The identifiers of the abstractions can also serve as titles of sections of the requirements specifications. Each of these sections has to be filled with details in order to produce a well-defined requirement. For instance, the section titled "navigation" might be filled in as follows, "The system shall navigate according to the parameters, how, when, where". Actually, the most refined abstractions are needed for the requirements, in order to give each individual requirement the most accurate title. For example, in the RFP transcript given later, "Unmanned Air Vehicle" is a well-defined abstraction and is mentioned in almost every paragraph. However, this phrase is the title of the entire document, which identifies the whole project, and it does not help much in capturing the detailed requirements needed to develop the system. So, a more refined abstraction identifier such as "navigation", "launch recovery", or "communication", which identifies some function or data, is much more useful for a well-defined individual requirement.

The list of the abstraction identifiers does not replace the original transcript. Reading only this list does not lead to understanding the client's needs. This list, however, assists the elicitor in two ways. First, it helps the elicitor keep the important concepts in focus. Second, it is used as a check list in order to keep the elicitor from overlooking anything.

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 (Luhn, 1958). On the other hand, these assumptions have been seriously challenged by the work on automatic abstracting (Johnson, 1993; Wilensky, 1986). Since the evidence is not conclusive either way, we prefer to let each situation and tool addressing it speak for itself. If the results indicate that the approach works for that situation, then approach should be used; if not, then the approach should not be used. Moreover, the tool presented in this paper makes no pretence to doing anything close to *automatic* abstracting. Its job is to extract enough information from the input to allow a *human* elicitor to identify the abstractions. It does not matter if some of what it presents is meaningless, so long as it does not lose anything that is meaningful. Perhaps with less strict requirements for automation, the approach works well enough. Ultimately, the proof is in the eating; if the tool proves to be helpful, then the approach must have some merit.

The repetition-based approach requires that important abstractions for requirements are discussed more than once among the sentences obtained from the clients. If an abstraction is mentioned only once in the input, it will not be found by this approach. This fact might be considered a fundamental flaw in the approach. However, in the case of abstraction identification for requirements engineering, it is quite reasonable to regard any once-mentioned idea as not important to implement. In any case, the tool is probably no worse than no tool at all, in the sense that in manual abstraction identification, an only-once-mentioned idea will be lost in the sauce. The validation of this assumption will come retroactively as a result of the case-study demonstrated effectiveness of the tool.

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 elicitor 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.

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 elicitor 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.

Abstraction identification for requirements elicitation differs from abstraction identification for indexing and information retrieval, e.g., for libraries, in one important way that affects many trade-off decisions. Since requirement elicitation abstraction identification is done only once in the lifecycle of a software-based system, speedy algorithms are not essential and algorithm speed can easily be sacrificed to other concerns, such as effectiveness, coverage, etc.

2.2. History Leading to AbstFinder

A description of the sequence of increasingly better tools developed by the second author and his colleagues serves to motivate the design of AbstFinder, as AbstFinder directly addresses many of the weaknesses of these earlier tools. For each of these tools, some case studies were done to evaluate the effectiveness of the tool, and these case studies showed that the tools had promise. Normally, more experiments would be needed to measure this effectiveness, but the flaws noted for each tool led to the decision to abandon the approach embodied in the tool, making further experimentation moot.

2.2.1. Grammatical Parsers

An early idea for abstraction identification, reported in (Berry, 1987) 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 the words. In the same example phrase, the key concept is “flight booking” and not just “flight”, the only real noun found in the phrase. Other authors, e.g., Smeaton (Smeaton, 1995), have observed the same weaknesses with grammatical-tagging-based approaches to other text-understanding problems. Smeaton goes so far as to suggest that algorithmically simpler low-level string-based processing may yield better results overall than more complex, even syntax-based processing.

2.2.2. Repeated Phrase Finder

A second idea (Aguilera, 1987; Aguilera, 1990) 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.

In small experiments with text-book sized examples, findphrases was found to be effective in aiding the elicitor to identify abstractions in all stages of the lifecycle. That is, in polished requirements documents of up to five pages in length, while findphrases failed to find all of the independently identified abstractions directly, from what it found all abstractions could be found by all people participating in the experiment. 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. In many cases, concepts do not appear as adjacent words but rather a set of words separated not more than a few words. Most of these concepts appear as closely separated pairs of words standing for an agent-object relation. Moreover, this relational information often allows distinguishing between semantically distinct uses of the same word, i.e., homonyms, by showing the context from which the word comes.

2.2.3. Lexical Affinities

A third idea (Maarek, 1988) 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 (Cruse, 1986). 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 a bit more effective in finding abstractions than the repeated phrase finder, but not much more, i.e., it found directly more of the independently found abstractions than did findphrases from the same documents used to evaluatefindphrases. At present, the LA finder does not find LAs consisting of more than two words of common grammatical structure, verb-noun, adjective-noun, etc. Of course, findphrases has no problem in finding phrases longer than two words.

2.2.4. Remaining Weaknesses

findphrases and the LA finder each have weaknesses that the other does not have. findphrases finds long phrases but identifies only fixed patterns, whereas the LA finder identifies nonadjacent words but is limited to precisely pairs of words. findphrases cannot identify phrases written in differing orders, for example, "book a flight" and "flight booking" which are not the same phrase but do belong to the same abstraction. The basic LA finder cannot handle phrases of length two whose elements are in different order, but which are grammatical variants of the same root, such as "book a flight" and "flight booking". Some help to these problems can be obtained using the traditional approach of stemming to identify grammatical roots (Salton, 1989; Frakes, 1992). Neither of them identifies synonyms as belonging to the same abstraction. The traditional approach of synonym replacement or use of a synonym dictionary during matching can be used. Moreover, synonym replacement can be used to replace pronouns by their referents in order to increase the frequency of these nouns.

The new approach described in the next section is an attempt to get the best of both findphrases and the LA finder and to render stemming as unnecessary.

2.3. New Approach

This section describes a new approach that eliminates many but not all of the weaknesses of the older tools. First, a formal statement of the approach is given motivated by a description of what is desired. While the new approach solves most of the weaknesses of the older tools, there are a few remaining.

2.3.1. Motivation

In general, an abstraction identifier may be a phrase within a sentence. This phrase may be composed of an arbitrary number of words, distributed within the sentence, with arbitrary sized gaps, and may appear in different orders in different sentences. For example, after examining the two sentences, "book ... a ... night flight" and "... flight ... booking", an elicitor should recognize the common concept of booking a flight as an abstraction, with {"book", " flight"} as an identifier. The tool should find the identifier, and leave it to the elicitor to recognize the concept from the idenitifier.

It is therefore, 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 not to 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

and

> book funny ,

the matching chunk is "book f". Fortunately, a human elicitor can ignore the "f" as meaningless. By experimentation, it was determined that attempting to algorithmically remove 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 recognition may be difficult. Among

> impossible to see

and

> a possibility seems ,

the matching 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. Again, algorithmic attempts to avoid these problems, particularly the latter, are fraught with the danger of losing information.

2.3.2. Formal Description

AbstFinder takes the novel approach of considering each sentence as a stream of bytes without any semantics. The problem of finding common phrases between sentences that identify abstractions reduces to the problem of finding possibly discontinuous common substreams, i.e., chunks, among the streams.

Finding common chunks that are in different orders in the sentences may be achieved by comparing one sentence against each of the circular shifts[1] of the other, searching, in each case, for possibly disjoint runs of consecutive matching characters in the two.

Let S denote a sentence. Then $length(S)$ is its length, and for $1 \le i \le length(S)$, $S[i]$ is the i^{th} character of S. For $1 \le i < j \le length(S)$, $S[i..j]$ is the substring of S stretching from $S[i]$ through $S[j]$. Additionally, if $length(S) \ge 1$, then $head(S) = S[1]$ and $tail(S) = S[2..length(S)]$; if, however, $length(S) = 0$, then $head(S)$ is undefined and $tail(S)$ is the empty string, ε. BLANK is the blank character. Finally, $S \,||\, T$ is the concatenation of S followed by T.

If $length(S) \ge 1$, the i^{th} circular shift of S, $CS_i(S)$ is defined recursively.

$$CS_1(S) = tail(S) \,||\, head(S)$$
$$CS_i(S) = tail(CS_{i-1}(S)) \,||\, head(CS_{i-1}(S)), \text{ for } 2 \le i \le length(S) .$$

It is necessary to put a blank at the end of S before circularly shifting S in order that the end of S not form a bogus word with the concatenated beginning of S.

In comparing two sentences it will be necessary to pad the shorter one with blanks to the length of the longer one. Therefore,

$$\text{for } n \ge length(S),\ pad^n(S) = S \,||\, \text{BLANK}^{n - length(S)} .$$

The special case of padding by one more character is denoted as simply $pad(S)$,

$$pad(S) = pad^{length(S)+1}(S) = S \,||\, \text{BLANK} .$$

A run in two sentences S and T of the same length is a string of consecutive characters that appears in both sentences in exactly the same positions of each 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. Each run obtained from comparing two sentences, one of them a circular shift, is called a *phrase*, because it can contain several words, which are common to the two sentences.

Suppose that $length(S) = length(T) = n$, $1 \le i < j \le n$, and $j - i \ge WordThreshold$. Then,

$$
\begin{aligned}
run_{i,j}(S,T) \;=\; \bigcup \{ a \mid\ & 1 \le i < j \le n \textbf{ and } j - i \ge WordThreshold \textbf{ and} \\
& a = S[i..j] \textbf{ and } a = T[i..j] \textbf{ and} \\
& \textbf{if } i \ne 1 \textbf{ then } S[i-1] \ne T[i-1] \textbf{ fi and} \\
& \textbf{if } j \ne n \textbf{ then } S[j+1] \ne T[j+1] \textbf{ fi } \} .
\end{aligned}
$$

The right hand side yields a nonempty set only when $S[i..j]$ is a run of significant length in *S* and *T*.

Suppose that $length(S) = length(T) = n$. Then,

$$runs(S,T) \;=\; \bigcup_{i=1}^{n} \bigcup_{j=1}^{n} run_{i,j}(S,T) .$$

The abstraction in common between two sentences *S* and *T* may be defined to be the set of runs in common in their circular shifts after padding each by one blank, as is mentioned above. However, as explained below, for simplicity, the runs are those found by comparing the shorter sentence padded to one more than the length of the longer with the circular shifts of the longer.

Let *S* and *T* be two sentences. If they are of unequal length, then let *L* be the longer of the two and *s* be the shorter of the two. Otherwise, let *L* be *T* and *s* be *S*. Let $n = length(L)$. Then,

$$Abst(S,T) \;=\; \bigcup_{i=1}^{n} runs(pad^{n+1}(s), CS_i(pad(L))) .$$

Consequently, even if *S* and *T* are of unequal length, $Abst(S,T) = Abst(T,S)$. The abstraction of a particular sentence *S* is taken as the union of all $Abst(S,X)$ for all other sentences *X*. Thus, there cannot be more abstractions than there are sentences.

From the sentences (not really, but the example has to be kept short!)

file to ignore
the ignored files

the working of the definition causes the sentences to be padded to

```
file to ignoreXXXX
the ignored filesX
```

where “X” represents a padding blank, which is really indistinguishable from an ordinary blank. The definition causes the circular shifts of “`the ignored filesX`” to be matched for runs against the padded “`file to ignoreXXXX`”, as shown in Figure 3.

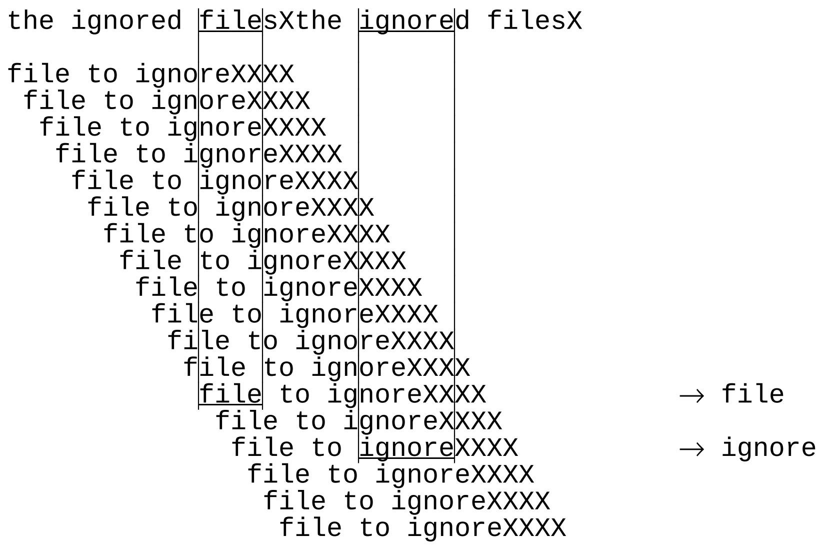
```
file to ignoreXXXX

the ignored filesX
he ignored filesXt
e ignored filesXth
 ignored filesXthe
ignored filesXthe 
gnored filesXthe i
nored filesXthe ig
ored filesXthe ign
red filesXthe igno
ed filesXthe ignor
d filesXthe ignore
 filesXthe ignored
filesXthe ignored        → file
ilesXthe ignored f
lesXthe ignored fi       → ignore
esXthe ignored fil
sXthe ignored file
Xthe ignored files
```

Figure 3. Comparing shorter sentence to circular shifts of the longer

The formal description admits of a very straightforward implementation that completely avoids generating and storing of the circular shifts. Basically, the sentence that would be circularly shifted, the longer one, is concatenated to itself after the one blank padding and the 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 4 shows the algorithmic rendition of the formal run search shown in Figure 3. Note that it is neither necessary to pad the second occurrence of the longer sentence, nor to pad the shorter sentence.

This algorithm will be recognized as the traditional signal processing algorithm to find commonality in two signal streams (Sklar, 1988). Perhaps, the power of this approach comes from its treatment of a sentence as a stream of arbitrary characters with the substreams appearing anywhere rather than being constrained to fall on word boundaries because the sentence is considered a string of words.

It is clear that searching for runs by comparing the shorter sentence padded to the length of the longer with the circular shifts of the longer is different from searching for runs by comparing the longer sentence with the circular shifts of the shorter padded to the length of the longer. However, the difference in the set of runs produced is strictly in what the human would regard as noise (Recall the discussion at the beginning of Section 2.3.). The set of meaningful words and word roots among these runs are the same. The implementation of searching for runs by comparing the longer sentence with the circular



```
the ignored filesXthe ignored filesX

file to ignoreXXXX
 file to ignoreXXXX
  file to ignoreXXXX
   file to ignoreXXXX
    file to ignoreXXXX
     file to ignoreXXXX
      file to ignoreXXXX
       file to ignoreXXXX
        file to ignoreXXXX
         file to ignoreXXXX
          file to ignoreXXXX
           file to ignoreXXXX
            file to ignoreXXXX            → file
             file to ignoreXXXX
              file to ignoreXXXX          → ignore
               file to ignoreXXXX
                file to ignoreXXXX
                 file to ignoreXXXX
```

Figure 4. Comparing doubled longer sentence to shifts of the shorter

shifts of the shorter padded to the length of the longer would require concatenating more than two copies of the shorter sentence in order to simulate its circular shifts; indeed the exact number of copies needed depends on the ratio of the lengths of the two sentences. Given that human intelligence is needed anyway to interpret the runs, and different noise is still noise, for simplicity in the algorithm it was decided to always compare the shorter sentence padded to the length of the longer with the circular shifts of the longer. This implies that in a set of sentences, only half of all total possible comparisons are done since $Abst(S,T)$ equals, by definition, $Abst(T,S)$.

2.3.3. How the New Approach Avoids Weaknesses of Previous Approaches

The new approach provides an effective way of identifying abstractions in natural language transcripts of client interviews, which allows

- unlimited phrase length, within the confines of a sentence,
- phrases with unlimited gaps between the words within a sentence,
- arbitrary permutations of a phrase to be recognized as the same phrase,
- 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. Treating a sentence not as a list of words but as a signal stream frees the algorithm from any phrase constraints. The cyclical sliding of the sentences enables identifying similar words in whatever order they appear in each sentence.

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.

There is one advantage of findphrases and the LA Finder which is preserved in AbstFinder. Their algorithms are independent of the language of the requirements transcripts. Even information that is language dependent, such as ignored words, suffixes, etc., is provided in a user’s input file.

The tool is purposely non-intelligent so it can guarantee that it considers all of the input and the elicitor can have confidence that none of the input has been overlooked as is required.

2.3.4. Possible New Weaknesses of the New Approach

One problem 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 real abstractions to be identified. With too much noise, the elicitor 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 very meaningful phrases, the threshold may be set too high and not all abstractions will be found. So, it is necessary to experiment with threshold values, and these values may prove to be different for each problem. It may also be necessary to run the same problem with different thresholds.

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 language there appears to be a characteristic set of common words, and for each application area there appears to be a characteristic set of application-dependent keywords.

1. The common words, e.g., “a”, “on”, “the”, “in”, etc. obviously do not identify any abstraction. When looking for similarity, these words will skew the list of correlated phrases that identify an abstraction, and will populate it with too much noise for

humans to easily find the real abstractions identifiers. One should fill an *ignored-phrases-file* with common words, in order to mark them for not taking part in the calculation for runs. 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 RFP text, which is entitled "Unmanned Aerial Vehicle (UAV)" and which was used as a case study (See Section 4.2.2), the words "unmanned", "aerial", "vehicle", and "UAV" appear in almost every sentence. When looking for commonality, these words will also skew the list of abstractions making it harder for the elicitor to find other abstractions. So, when using AbstFinder, the elicitor should fill an *ignored-application-phrases-file* with these frequent application keywords, which identify larger abstractions.

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

A third noise problem can be caused by common long suffixes. For instance, "cation" in "application" and in "communication", or "ance" in "accordance" and in "appearance". In order to avoid these suffixes being counted as possible abstractions by AbstFinder, the elicitor should fill the *ignored-suffixes-file* with the recognized common suffixes, in order to mark them for not taking part in the calculation for runs.

An enhanced operator $Abst(S,T)$, $Abst'(S,T)$ is applied in AbstFinder program, which uses $run'_{i,j}(S,T)$, instead of $run_{i,j}(S,T)$, where

$$\begin{aligned} run'_{i,j}(S,T) \;=\; \bigcup \{a \mid\ & 1 \le i < j \le n \ \textbf{and}\ j-i \ge WordThreshold \ \textbf{and}\ a = S[i..j]\ \textbf{and} \\ & a = T[i..j]\ \textbf{and}\ a[i..j] \text{ is not an ignored suffix } \textbf{and} \\ & \textbf{if}\ i \ne 1\ \textbf{then}\ S[i-1] \ne T[i-1]\ \textbf{fi and} \\ & \textbf{if}\ j \ne n\ \textbf{then}\ S[j+1] \ne T[j+1]\ \textbf{fi}\ \} . \end{aligned}$$

In general, a little noise that sometimes causes useless information to appear among the valid abstractions, does not harm abstraction identification. If there is not too much noise, the elicitor can easily distinguish the the noisy strings from the meaningful words and ignore them. Also an ambiguous phrase, without a common word that was ignored, poses no problem for a human to interpret if it is reported as a repeated phrase.

There is also the problem of inconsistency resulting from using the same concept for different abstractions or using different concepts for the same abstraction. As an example, consider a communication system in which the concept "frequency" refers to both the frequency of hopping for anti-jamming purpose and the frequency of a clock. These are two completely different abstractions. Those inconsistencies originating in the client's transcript show up in the output of AbstFinder as very strange abstractions in which clocks appear to have a Hertz as a unit and helps find an open channel for communication. The human elicitor is expected to note these strange abstractions and manually split them.

2.3.5. AbstFinder Program

The AbstFinder program incorporates the algorithm described in the previous section. 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. Indeed, such duplicates should strengthen the frequency of the abstractions embodied in the sentences, especially if they come from different sources. The set of runs returned by an invocation of *Abst* on one pair of sentences, one being a circular shift, is called the *phrases* of that pair of sentences. The meaning that can be ascribed to these phrases is the *abstraction* embodied by the phrases in common in the two sentences. The main data structures of the program are corr_phrases and corr_lines. Each entry in corr_phrases is the set of phrases obtained by comparing one sentence to all circular shifts of all of the other sentences; any sentence for which no phrases are found does not have an entry in corr_phrases. corr_lines is an array indexed the same as corr_phrases, such that for each entry in corr_phrases, its entry in corr_lines contains the line numbers of the sentences that compose the abstraction that is identified by the corr_phrases entry.

```
Accept four input files:
     a punctuation-keyword-file, an ignored-phrases-file,
     an ignored-suffixes-file, an ignored-application-phrases-file,
     and a synonyms-file;

Partition the text into sentences one per line, where a sentence is
     the text lying between two consecutive elements of the
     punctuation-keyword-file;

comment line and sentence are used interchangeably from now on tnemmoc

Remove from the text strings found in the ignored-phrases-file
     and strings found in the ignored-application-phrases-file, and
     mark suffixes according to the ignored-suffixes-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 := 0; comment number of abstractions accumulated so far
     which must always be less than or equal to 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

    comment sort the NA identified abstractions so that the most
        refined ones are at the top of the list 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/sentences from which
        the phrases came;

    Prepare and print the output as described below;
```

The output of AbstFinder comes in two parts. The first part is a table summarizing the identified abstractions, and the second part gives a full description of each of the abstractions.

The appendix shows the first part of a run of AbstFinder that features in a later discussion, specifically of a description of the findphrases program. 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 to its first phrase (the NA of the algorithm). The “phrases#” 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. This truncation is a design feature, and its purpose is to signal to the elicitor reading the summary part, that this abstraction is identified by too many phrases, and may be too broad (See Section 3). Even if the phrases are truncated, the full list of phrases may be found in the corresponding entry in part 2. The abstractions in the table are listed in order of increasing numbers of correlated phrases, and within any particular number of correlated phrases, in order of decreasing numbers of sentences from which the phrases came. Note that the elicitor uses only the “correlated-phrases” field in order to decide on abstraction identifiers. All the other parameters are for providing data for the measurement of the effectiveness of AbstFinder.

An abstraction identified by one phrase is more distilled than one that is identified by more phrases. Obviously, there exists an abstraction that identifies the whole document and contains every sentence in the document, but we are not interested in it. So, the first criterion for ordering the abstractions 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 come. The more sentences contained in an abstraction the more significant it probably is.

As a result of this ordering, the more refined abstractions are higher in the list. In reality, an abstraction that appears at the end of the list may not be any less important. On the contrary, an abstraction appearing near the end of the list, which therefore does not appear many times in the CI, gets lost among many other concepts and it may be difficult to distinguish it from the others. Thus, it is very important to identify that abstraction as quickly as possible in order to get back to the client and obtain more information about it.

The second part of the output of AbstFinder is a full description of the abstractions in order of their serial numbers in part 1. Figure 5 shows one of them. The output itself is in the Courier font and the commentary is in the Times Roman font.

```
{1} abst_id=42
====================
correlated phrases of abstraction are
(#=1)                                 number of correlated phrases
       punctuation keyword file|      phrases themselves

correlated sentences of abstraction are
(#=11)                                number of correlated sentences
       44 2 8 14 20 21 23 24 30 35 49 identity numbers of sentences
```

Figure 5. Second part of AbstFinder output

Another byproduct of AbstFinder is the *corr-phrases-file*. The *corr-phrases-file* contains a list of all and only the abstraction identifiers, i.e., the correlated-phrases of part 1 of the output.

The full program can be thought of as a kind of clustering (Salton, 1989). In clustering, one starts with each object in a separate class. Then a distance measure is selected. The next step is to group into one class all objects whose distance is according to a predefined criterion. This repeats until intra-class distances are low and inter-class distance is high.

In AbstFinder, as with normal clustering techniques, there is a similarity measure (the length of the runs among two sentences) and a criterion for deciding when two items are similar (the sum of lengths of the runs being greater or equal to *WordThreshold*). However, true clustering puts each object in one and only one class, as it is a partitioning. In AbstFinder a sentence is allowed to be in several abstractions. Moreover, true clustering starts with an arbitrary classification, and then moves objects from class to class until the criterion is fulfilled. The final result of classification can be heavily influenced by that arbitrary first classification. Generally, in requirements elicitation, there is no *a priori* classification available. Moreover, it is desirable to avoid being influenced by any initial prejudices.

Moreover, AbstFinder can be compared only to flat clustering. Hierarchical clustering is not relevant to abstraction identification. The hierarchical clustering may be good for search in libraries (Maarek, 1989) in which one starts at a top level and wants to get to the lowest level with clusters consisting of single elements in order to fetch a matching library item. In abstraction identification, one is looking for some middle level in which an abstraction is defined by a good phrase, consisting of a few words in a few sentences. An abstraction should not be at too high a level, because it has to be specific enough to have requirements. When using AbstFinder, hierarchies are generated only when the elicitor wishes to do so. The elicitor expresses this wish by *zooming* into an abstraction that he or she thinks that is too broad (See the first author's Ph.D. thesis (Goldin, 1994b) for details on the zooming process).

2.3.6. Performance Analysis of Program

As mentioned in Section 2.3.4, the *WordThreshold* parameter must be set very carefully. An abstraction is identified by a concept, and a concept is composed of natural language words. Thus, if we assume that the minimum length of a meaningful word is three characters, then *WordThreshold* has to be set to at least three in order to be able to capture common concepts as abstraction identifiers according to AbstFinder.

However, while prototyping the tool, it was decided to keep one space between words, and to take these spaces into consideration while calculating similarity. So, a threshold of three characters was found to be too low. It happened often that a string of form "x y" was found as a match. This match is meaningless because "x" is the last character of one word and "y" is the first character of the successive word. When the threshold was raised to five characters, AbstFinder appeared to capture only meaningful phrases. Of course, each application can have its own *WordThreshold*, and it will be necessary to experiment with the value of the threshold. Fortunately, there is no reason that several different activations of AbstFinder, each with a different *WordThreshold*, cannot be used by the elicitor for abstraction identification. Remember, the goal is a list of abstractions and not the phrases, whose only purpose is to help identify the abstractions.

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. While, in principle, sentence lengths are unbounded, since they are natural language sentences they can be regarded as bounded, say at about 100 characters. Moreover, as the elicitor follows the iterative process of using AbstFinder, the sentences are getting shorter as the ignored files are getting bigger (See Section 4.2.2.). There are faster algorithms based on the use of tries or Patricia trees. These can be made $o\,(c \times N)$ if desired (Knuth, 1973). However, experience with

AbstFinder on an industrial-sized example shows that its real performance problem is squeezing the data into the faster main memory. This problem is only exacerbated when the faster algorithms that require an order of magnitude more storage are used. In any case, it is no problem for the elicitor to go out to lunch while waiting for AbstFinder to report on a large input.

3. Scenarios for Usage of AbstFinder

With any scheme of automated assistance, scenarios for usage should be defined. Even the most intelligent elicitor cannot abstract a full CI (recall: client information) all at once. Usually, the elicitor reads some pages in order to learn the terminology of the CI. Then he or she reads the CI several times, in an iterative learning process, capturing another set of abstractions in each pass. Thus, an automated assistance for abstraction identification has to be compatible with human limitations in absorbing new material.

This section describes typical scenarios that an elicitor might follow in order to have AbstFinder help identify the abstractions in a new problem given to him or her by a client. These scenarios were found by the authors in the course of their use of AbstFinder for the case studies and actual work.

3.1. Learning What Words to Ignore

First, some trial runs need to be done on small parts of the CI, taken from different sections of it, in order to learn the language of the document. Learning here consists in identifying an initial set of ignored words, putting common words into the *ignored-phrases-file*, special application words into the *ignored-application-phrases-file*, and suffixes into the *ignored-suffixes-file*. The *ignored-phrases-file* and the *ignored-suffixes-file* can be accumulated from one application to another. The *ignored-application-phrases-file* is specific to an application. Actually, the *ignored-application-phrases-file* may contain very important high level abstractions that have to be taken into consideration by the elicitor, but which have been recognized, noted, and put into that file in order not to clutter up the output. After each run of AbstFinder, the ignored words and suffixes files are updated, because after any change, new noise appears. The authors’ experience is that the process converges after a few runs.

Typically, the *ignored-application-phrases-file* is quite short, because it contains mostly the highly repeated terms that hide other terms, and the *ignored-phrases-file* is longer, because it is accumulated from many applications. For the 100-page RFP case study described in Section 4.2.2, each of the three files was, in fact, about two-thirds of a full single-spaced page, i.e., about 300 words.

3.2. *What to Do with a Well-Organized Document*

If the CI to be analyzed is a well-organized document, submitting it as is to AbstFinder may yield abstractions that belong to the table of contents or the meta-language of document writing. These included organizational concepts such as “summary”, “confidential”, and “base-line configuration” as well as common chapter titles such as “Introduction”.

The table of contents itself, although it looks like a good classification of the material, is only an organizational list. It is not necessary that each title in the table imply an abstraction. For example, the title “Characteristics” does not identify an abstraction because it is too broad and unfocused. The title “Introduction” is purely organizational. We aim to have abstractions of only functional or informational strength, the two highest module strengths, according to Myers (Myers, 1979).

Therefore, the titles in the table of contents do not necessarily have to appear in the AbstFinder result list. In fact, it is suggested to remove the table of contents from the CI before applying AbstFinder. Unless the table of contents is removed, every title of it will appear in the abstraction list, because each title appears at least twice; once in the table and again at the beginning of the named section.

On the other hand, for an unorganized collection of documents, the abstractions produced make good candidates for sections of an organized document produced from their contents and the phrases of these abstractions might very well end up being the section titles that show up in the table of contents, along with the organizational titles such as “Introduction”.

3.3. *Iteration to Final List of Abstractions*

When using AbstFinder with a huge CI, the elicitor should read the output list of abstractions and note the abstractions identified by fewer than four or five phrases. Abstractions identified by more than five phrase are difficult to understand[2]. 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 CI, and that abstraction is clearly not very useful. Therefore, the elicitor has to stop at some point, a point before 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 elicitor has to settle for a point at and beyond which they are not useful.

The main purpose of the clerical tool is to identify *all* meaningful abstractions. Without full coverage, the elicitor will never trust the tool to not overlook something important. 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 iterative procedure described below should be carried out.

This procedure uses the Strainer program, an auxiliary tool designed to strain out words from a text file according to a list of words contained in another file. The extraneous abstraction elimination is done by activating Strainer using the logged *corr-phrases-file* as the input list of words to be removed from the original document. Strainer removes a chunk from the text file if and only if it has white space on both sides. Being surrounded by white space means that the chunk is a whole word in the text file and not part of a word. Note that AbstFinder is designed to find common portions of sentences, even if they are only parts of words. Strainer however, is designed to remove only whole words. This feature is very important for the iterative usage of AbstFinder, in order not to ruin the text and keep what remains after straining potentially meaningful.

The following describes the iterative procedure:

1. Activate AbstFinder once on the original document.
2. With the Strainer program, remove from the original document the abstractions already recognized and logged by the elicitor, leaving what is left in another file *f*.
3. 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 elicitor is left with *f* containing a list of abstractions that are all meaningless. That all of the abstractions left are meaningless indicates that all the meaningful abstractions were already identified and have been strained out from the CI. Note that it does not matter if this list is large, as it was for the case study, so long as it is easy for the human elicitor to scan it and determine that there is nothing meaningful in it.

This iterative way of applying AbstFinder and then Strainer, is suitable for a human elicitor 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 elicitor is confident that nothing will be overlooked, because abstractions 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.

Indeed, here is the way that the elicitor can catch words, including pronouns, that should have been replaced by others in the *synonym-file*. They stand out among the meaningless material in the final *f*.

Recall that AbstFinder generates also the *corr-phrases-file* file that contains the abstraction identifiers, each abstraction per line. The elicitor, by using Part 1 of the formatted output of AbstFinder, decides which abstractions are meaningful, and stops at abstraction *n*. The elicitor keeps only the *n* first lines of *corr-phrases-file* file, i.e., the identifiers of the *n* recognized abstractions, and removes the rest of *corr-phrases-file*. These *n* abstraction identifiers are also logged in the accumulative *abstraction-phrases-file*.

This accumulated list of meaningful abstractions provides the desired full coverage. The iterative process is illustrated in Figure 6. In this figure, each box labeled “Corr-Phrases” represents one output from AbstFinder and has an iteration number, the

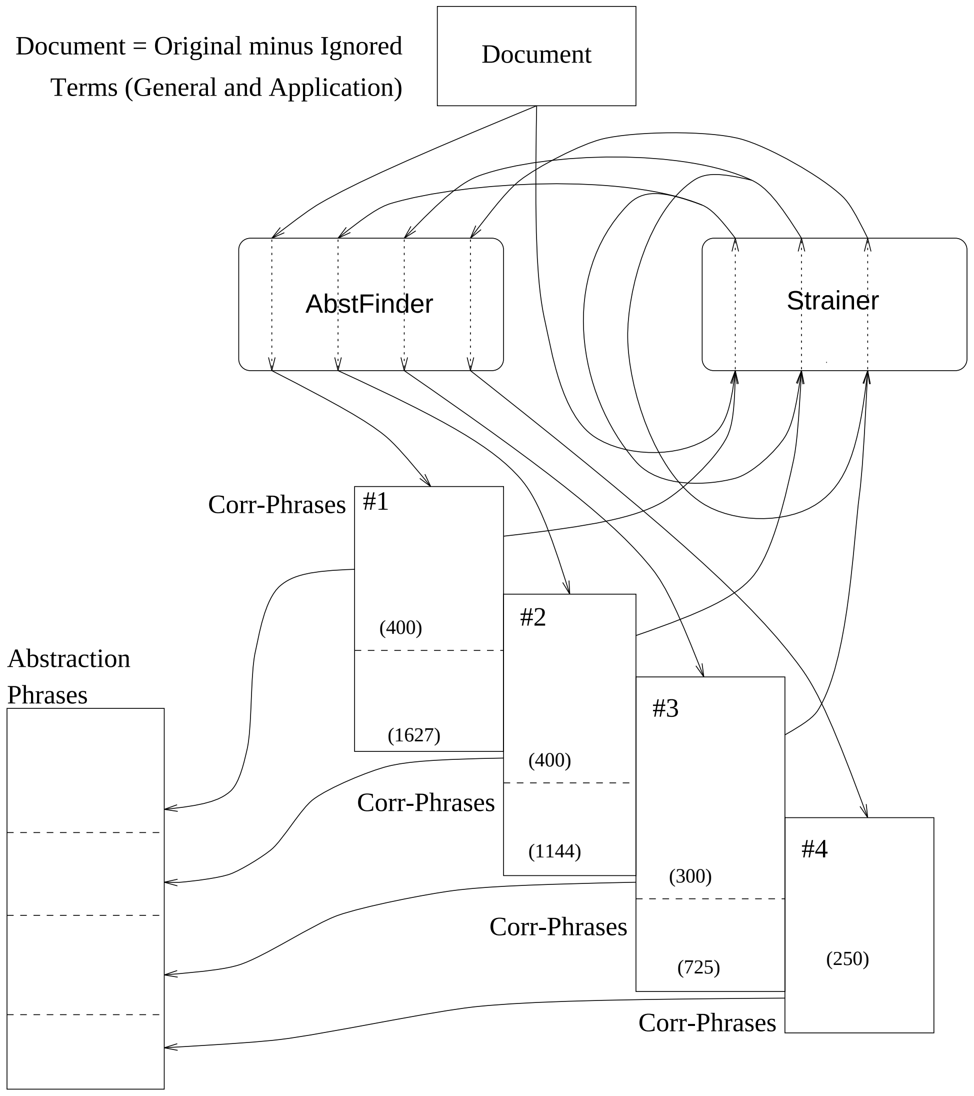


Figure 6. Iterative abstraction identification

number of abstractions accepted by the elicitor as meaningful above the dotted line cut-off point, and the total number of abstractions in the file, reported by AbstFinder, below the dotted line. In the case of the box numbered "#4", no abstraction was accepted as meaningful, so there is no dotted line, and the number of abstractions there is the total reported. The numbers in the figure are from the 100-page RFP case study described in Section 4.2.2. The number of meaningless abstractions is about 250 and this amounts to a bit less than 20% of the total number. Therefore, the amount of text to examine for meaninglessness is considerably smaller than the original.

3.4. Summary of Scenarios

The scenarios can be used in different combinations for different purposes. When using AbstFinder, the user has to be cognizant of the purpose or objective for the use, i.e., requirements, indexing (Section 4.3 of the first author's Ph.D. thesis (Goldin, 1994b) considers using AbstFinder for indexing.), etc., and follow an appropriate combination of scenarios. Let

- "learning" denote the activity of *learning what words to ignore*,
- "well-organized" denote the activity of *dealing with a well-organized document*,
- "zooming" denote the activity of *zooming* (Recall that zooming is building a hierarchy of subabstractions for a given abstraction that is found to be too broad. Due to space limitations, it is described only in the first author's Ph.D. thesis. However, the activity is mentioned in the scenarios for completeness of their specifications), and
- "iteration" denote the activity of *iteration to a final list of abstractions.*

All of these activities were described in previous sections. The flow of typical scenarios for use of AbstFinder in requirements abstractions identification are embodied in the following regular expression ("[]" means optional, "*" means zero or more repetitions, and "{}" is for grouping).

```
[well-organized], learning, iteration, {[learning], iteration}*, zooming* .
```

The learning step is completely dependent on the person that does it. The person will decide to ignore terms, mostly application ignored words, according to his or her objective. For instance, when doing abstraction identification for requirements, many details, such as names of people, will not be considered as abstraction for requirements, and will come to be ignored during the learning step by their having been put in the *ignored-application-phrases-file*.

In the findphrases case study described in (Goldin, 1994b), whose output is given in the appendix, the scenario followed was

```
learning, iteration, learning, iteration ,
```

and for the industrial case study described in Section 4.2.2, the scenario followed was

well-organized, learning, {iteration, learning}^{3}, zooming^3 .

Identifying abstractions in order to generate input for some requirements analysis method, such as OOA or SA may involve a different scenario according to a different objective, specifically, identifying objects or functions.

One could argue, as a referee did, that "the ignored-application-phrases-file makes [this] approach lexicon dependent and reliant on domain experts.... Also [there is] the need for the synonyms file.... How do you know what to ignore when you are still discovering requirements? This gives ... a sort of circular argument; you need to know what you want, to know what to throw away!" This question gets right to the heart of the importance of the human in this process. The tool is not expected to find the abstractions; the human user of the tool does that. The tool *is* expected to find sufficient clues that prompt an intelligent human, in the process of learning the domain, to find all the abstractions. As for what to ignore, it almost does not matter. Anything not ignored that should be begins to show up prominently in the output and that alone should prompt ignoring. Since anything in the ignored phrases files is at the decision of the human, that human can re-examine it at any time for significance. As for synonyms, it has been the authors' experience that the human intervention inherent in the scenarios is sufficient to teach the human what should be synonyms. If something is missed one time, it can be caught in a later run through the straining activity. In any case, the proof is in the using. Section 4.2.2 details a case study in which a single AbstFinder-assisted elicitor following a scenario presented above outperforms three expert analysts working completely manually in two orders of magnitude more time.

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.

4.1. How to Evaluate a New Method or a Tool

The effectiveness of any new method or tool must be evaluated. There is very little work in use of such a tool beyond that of findphrases and the Lexical Affinities Finder. So, the new tool must be compared to these old tools to see that the new tool does at least as well or better than the old ones. This is not enough, because the old tools were tested against only toy examples.

Such a tool must really be tested against a human effort, since heretofore requirements elicitation has been done manually by a humans. Testing human efforts is very difficult and is more difficult than building alternatives to purely human efforts. As with 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 (Schach, 1992). Moreover, individual differences can dominate the controlled variable of the experiment (Sackman, 1968).

For a tool that is aimed to support humans, the critical question is whether the abstractions found by the tool are meaningful to the human elicitors that have to approve them. Meaningfulness can be confirmed only by humans and is very much affected by the *WordThreshold*. As described in Section 2.3.6, a *WordThreshold* of 5 was chosen, but may be changed according to the situation at hand.

Testing against human effort must show that the new tool does at least as well and possibly better than expert human elicitors in less time or with fewer people. Time and people power are easy to measure, but how to measure the concept of doing at least as well and possibly better than a human or other tool? A new tool should find at least all abstractions and maybe some not found by the humans or the old tool. Still, a criterion should be established for tools which involves comparing only input to expected output.

The key 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:

Heretofore, abstraction identification has been done manually by a human elicitor with the danger of overlooking relevant ideas. Therefore, the desire is for a clerical tool that helps with the tedious, error-prone steps of what a human elicitor does. It must be that this clerical tool 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 elicitor will not be willing to be assisted by any tool unless he or she is confident that it is covering, that no critical piece of information will be overlooked in the process of abstraction identification.

Clearly, the identity function is a covering tool. However, presenting all the input does not help the elicitor either. The other main requirement for the tool is that it reduce the amount of text that the elicitor must look at. A human elicitor still has to do the *thinking* with the output of the tool, in order to approve the abstractions found. The elicitor will not be effective if the amount of information that must be examined is not significantly reduced from the original volume. A tool whose output is significantly smaller than its input is said to be *summarizing*. Since a key goal of the tool is to reduce the sheer volume of text that the elicitor must examine in detail, comparing byte lengths of texts is an acceptable measure.

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 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, because one must measure the list generated by AbstFinder to that made by a known expert (and pray that in fact the expert is good). There is no better measure than experience, and ultimately the proof will be in acceptance of tool by the elicitor community.

4.2. Case Studies

Case studies were used to evaluate AbstFinder. Two of them are described below.

4.2.1. Findphrases Case Study.

The findphrases decomposition was used as a case study because the decomposition was already known. So, it could be used to check AbstFinder's results against already known results. The document that served as the requirements was the manual page of findphrases, because in fact, the manual page was written as a requirements document before the program was written. The already known abstractions were taken from Aguilera's (Aguilera, 1987) program decomposition, and her own list of abstractions identified by findphrases, and Maarek's (Maarek, 1988) list of abstractions identified by lexical-affinities (See Table 1).

As shown in the appendix, 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.

The abstractions of findphrases		AbstFinder results	
Data Abstraction	Repeated phrases	AbstFinder's Corr-Phrases	Abst#
string_type_file	strings, characters	character|symbolcharacter|	7
argument_line	argument,option	argument |optional	3,14
output_file	output, tables of the output	output|tables	25
chunk_file	file(s), free format	free format |files	9
punc_keyword_table	punctuation keyword(s) file	punctuation keyword file	1
multi_tokens_table	multi tokens file	token file	5,17
text_file	text,input,arbitrary text	arbitrary text|input	12
phrases	phrase, repeated phrase, ignored phrase	phrase	4
sentences	sentence(s)	sentence	6

Table 1. The abstractions of findphrases

4.2.2. RFP Case Study

The Request For Proposal (RFP) document for the Unmanned Aerial Vehicle-Short Range (UAV-SR) system (IAI, 1989) is a large industrial-strength RFP, about 100 pages long, containing about 2200 sentences, which we were lucky to get. The RFP transcript had already been analyzed by three experts over a month, for a total effort of three person-months, and they had produced from it a list of sentences, each giving one requirement, functional or non-functional. The experiment consisted of the first author, called the elicitor 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 elicitor did not see this human-made document until after she had finished generating her list with the help of AbstFinder. The hope was that the elicitor, assisted by AbstFinder, 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.

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 draws conclusions without giving the full introspection found in the thesis. (See Section 4.1).

4.2.2.1. Meaningfulness

After the elicitor finished generating what she thought was a complete list of abstractions, the phrases in this list were examined by three expert analysts of the RFP transcript. The three professional requirements analysts, Mr. Kudish, Dr. Winokour, and Mr. Engel, are highly skilled and have nearly sixty years of cumulative experience in real-time system and software requirements analysis. They all said that they found all of theAbstFinder-generated phrases to be meaningful to them. They confirmed that the abstraction identifiers generated by AbstFinder contained terms and phrases that identify real functions and objects of the RFP that they already knew. One of the IAI people, Dr. Michael Winokur, was very impressed to see in the beginning of the AbstFinder-generated list some abstractions, such as "surrogated training", that the analysts had overlooked for a long time until the customer finally pinned the abstractions on the analysts noses.

4.2.2.2. Summarizing

The output of AbstFinder was summarizing. The original document RFP was 214,654 bytes long while the final AbstFinder-assisted list was only 47,105 bytes, about 21% of the size of the original data. The original transcript contained full sentences of text. The AbstFinder result, via *corr-phrases-file*, contained only the abstraction identifiers, one per line, that were recognized and logged by the elicitor. Recall that AbstFinder output contains

- noise of parts of words that cannot be algorithmically eliminated without risking losing non-noise and
- repetitions, both between abstractions and within an abstraction.

4.2.2.3. People and Computer Power

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 machine time, three hours operating time, and about two hours of elicitor 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 most frequent abstractions on the list, took about 30 minutes. The last run took about 5 minutes. However, note that the elicitor was doing other things while the machine was running.

4.2.2.4. 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 has 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 using Strainer to strain 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. (It is interesting that one of the auxiliary tools became a tool for evaluating the main tool!) The result of the subtraction of *corr-phrases-file* from the human-made document was 3019 bytes. The RFP was 214,654 bytes (about 100 pages) long and the human-made requirements document was 83 pages (about 140K bytes) long. The phrases of the remainder were analyzed very carefully in order to see if AbstFinder missed any abstraction. The phrases of the remainder were separated to several categories according to their characteristics.

1. Most of the phrases originate from the strict meta-language of the requirements specification format of the human-made document, such as “activate”, “allow”, “deactivate”, “herein”, “include”, “integrate”, “must”, “only”, “provide”, which are not abstractions and were used only in the human-made document for stating requirements and not in the RFP original transcript.
2. Some concepts were in different grammatical forms such as “transmit” in the AbstFinder abstraction list, and “transmitting”, “transmitters”, and “transceiver” in the human-made document. Those words in the leftovers do not carry any new concept,

they actually describe the same abstraction. The same is for:

"calibrate" and "calibration" "assigned" and "assignment".

While AbstFinder is designed to classify all the "transmit..." words as a single abstractions, Strainer is designed to remove only whole words and does not remove words that properly contain a recognized root. If Strainer were to remove parts of words, then the remainder of the document will be a mass of unreadable text. For instance, suppose that "inter" were found by AbstFinder as a common part among "interchangeability" and "interfaces". Removing "inter" as part of word would leave in the leftovers "changeability" and "faces". Both of these accidentally generated words are garbage relative to the application.

3. 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. Many acronyms are shorter than the *WordThreshold*, and a full phrase if appears only once it is not going to be caught by any frequency-based scheme. Actually, only the "NBC" was not found, all the others were found since the term of the acronyms were repeated in the text more than once. 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 elicitor.

   a. The synonym dictionary can be used to replace the acronyms by their full phrases for the purpose of abstraction identification.

   b. Recognize all the acronyms as important abstractions, log them as abstractions, and then add them to the *ignored-application-phrases-file*.

   Only after recognizing the abstractions, the elicitor may switch to using acronyms as abstractions identifiers.

4. Ten concepts appeared in the leftovers because they appear in the RFP only once, and AbstFinder identifies only concepts that appear more than once, at least once for definition and once for use. Of these, five phrases were synonyms in the context of the system that was defined in the RFP, such as "contour" and "elevation", and "enemy" and "threats", that occurred because the *synonym-file* was not implemented yet.

5. The remaining five phrases were specific examples of some already captured abstractions and appeared in the text with linguistic clues, "i.e.", "e.g.", and "for example". These are not abstractions, they are details that will be put inside the abstraction.

To sum up, after some generally applicable modifications that should be part of a standard scenario for use of AbstFinder, full coverage was achieved. This, by the way, is how each case study led to to refinement of the use scenarios. This, by the way, is how each case study led to to refinement of the use scenarios.

4.2.2.5. Does Better than Human Experts

We were interested to see if AbstFinder found some concept that the human elicitor overlooked. This meant to check if the list of requirements in the human-made document cover the list of abstractions found by AbstFinder. That question was answered by removing 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. Again, the subtraction was done by Strainer.

The result was about 35,402 bytes long. There were very meaningful concepts concerning “communications”, “ordnance”, “weather conditions”, etc. Perhaps some of these did not appear in the human-made document because they were hidden in the classified requirements appendix of the RFP document. This appendix is competition sensitive and was not submitted to the research case study. Note, that the RFP specifies the whole system, hardware and software, while the human-made document specifies the software only. So, most of these 35,402 bytes concern other requirements than software. A great portion of these leftovers was noise, i.e., parts of words, and did not contribute any new concept.

We also found that the people of the project were not happy to hear the results of this investigation, because the project was already in progress, and they felt, incorrectly, that it was not the right time for them to find things that they might have missed. Note again, that according to the project people, some of the abstractions such as “surrogated training” appeared very clearly at the output list of AbstFinder while the project people overlooked it for long time. So, we got the impression that an elicitor operating AbstFinder can do better than a group of human elicitors.

4.2.3. Results

For the specific case studies carried out, 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 at the top of the output list of AbstFinder.
- at least as good as three human experts on the RFP. In fact, AbstFinder found some abstractions that they did not find.

Moreover, in the second case study, the AbstFinder output amounted to 21% of its input, and the runs of AbstFinder on the RFP to determine its abstractions took one day, while the three human experts took three months to do their analysis of the RFP.

The conclusion is that for the case studies presented, AbstFinder is good, it has coverage and it is summarizing. Once again, note what is claimed. AbstFinder helps find abstractions, i.e., modelling concepts and *not* requirements *per se*. Then, it is up to the elicitor to use these abstractions to help invent the requirements.

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. As these were the first case studies, their evaluation was designed as they were being carried out, only after it became apparent what the key issues were. These were, in effect, pilot case studies. Future case studies should have the evaluation planned in advance in accordance with suggestions offered by Pfleeger (Pfleeger, 1994; Pfleeger, 1995a; Pfleeger, 1995b) To encourage such experiments, the authors are making the source code of the tool available. Please contact the second author at the electronic mail address listed in his affiliation 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.

5. Comparison to Other Natural Language Processing Work

There is a wealth of material representing years of work in automatic indexing, abstraction, and thesaurus generation, all under the rubric of document processing and information retrieval (Rau, 1989; Cavazza, 1992; Damerau, 1993; Salton, 1986; Salton, 1989; Frakes, 1992; Baeza-Yates, 1992; Srinivasan, 1992; Fox, 1992). It is interesting to compare our approach, of this paper, to that of this older work. Among this older work, the closest to ours in terms of goals is automatic abstraction, but, all of it deals with automatic identification of key concepts in large collections of documents of any subject. A key difference between this older work and ours is the degree of human participation. Automatic indexing, abstracting, and thesaurus generation aim to automate enough to eliminate the need for human participating, as they are being used to assist in dealing with an ever growing body of documents. Our work not only permits human interaction, but even insists on it and takes advantage of it to overcome weaknesses in the approach. This dependency leads to different choices being made in functionality, algorithm, and use scenarios. This difference is possible because AbstFinder is applied *once* at the beginning of a project's lifecycle, and it is no problem if the tool has to be run multiple times on the same document. There is no sense of losing ground agains a growing body of literature. The fact that the list of abstractions found is not expected to be in final form means that it is OK that there is some noise and word fragments among the list.

Perhaps, because of this difference, all the algorithms used in automatic indexing, abstraction, and thesaurus generation have to be fairly smart with no human intervention and with a minimum expenditure of space and time resources.

In addition, there are those who say that abstraction identification can be considered *inverse data retrieval*. Data retrieval is the activity by which one retrieves data according to a known keyword. Abstraction identification is the activity by which one looks for the key concepts to be used for retrieving information about the concepts. One of the main concerns in information retrieval (Salton, 1983) is the automatic indexing of documents, which consists in producing for each document a set of indices that form a *profile* of the document. A profile is a short-form description of a document that is more easily manipulated than the document and plays the role of the document's surrogate during the retrieval. In abstraction identification for requirements elicitation, understanding is needed to be able to state the raw requirements. In indexing for information retrieval, a profile is a list of keywords that do not necessarily have any meaning and cannot be used as abstraction identifiers. However, a list of abstraction identifiers is a good list of index terms for retrieval.

6. Conclusions

This paper has described the rationale for and the design of a prototype tool, AbstFinder, to help find abstractions in natural language text to be used in requirements elicitation. It has described a single case study of the tool's use on an industrial strength example of abstraction identification in an RFP, a typical input document for requirements analysis. The results of the case study were promising, as they showed that for the example at hand, with the people involved, a single person, non-expert in the domain, but expert in using the tool, using the tool was faster and more effective in abstraction identification than three domain experts working on the same input.

This paper has *not* demonstrated any tool for automated abstraction identification or extraction, and it has *not proved* that the tool it does present is effective for its purpose. It has merely shown that AbstFinder shows some promise as a tool that helps a human elicitor find clues that help him or her identify the abstractions, provided that he or she is prepared to do all the thinking.

The paper has also described scenarios for the use of AbstFinder to help a requirements elicitor identify the abstractions embodied in natural language text purporting to be a complete description of a system to be built. In the case study, these scenarios allowed the elicitor to work with small chunks consisting as a whole of less than the full set of documents while retaining confidence that the entire set of documents has been considered.

The results of this case study, while tantalizing, are not enough. It is now time for more extensive case studies and perhaps even controlled experiments. Only as the tool proves effective in practice, does it have any real value. It is also time to begin building production quality versions of the tool, based on what has been learned by the use of the tool.

Acknowledgments

The authors thank Yossi Gil, Yona Lavi, Janos Makovsky, and Amiram Yehudai for their constructive criticism of the first author's Ph.D. thesis on which this paper is based. The authors also thank Avner Engel, Yossi Kudish, and Michael Winokur for participating in the case study and for comments on the thesis. They thank Bashar Nuseibeh, the journal editor, and the anonymous referees for their hard questions and revision requirements which led to a better paper.

Notes

1. The first *circular shift* of a sentence s is s with its first character removed and appended to the end. For n less than or equal to the length of s, the n^{th} circular shift of s is the $n-1^{\text{th}}$ circular shift of s with its first character removed and appended to the end.

2. We are talking about key phrases devoid of the normal connective material needed to make a coherent complete sentence or thought.

References

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

Aguilera, C.S. 1987. Finding Abstractions in Problem Descriptions using findphrases. M.S. Thesis, Computer Science Department, UCLA, Los Angeles, CA.

Alford, M.W. 1977. A Requirements Engineering Methodology for Realtime Processing Requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69.

Alford, M.W. 1978. Software Requirements Engineering Methodology (SREM) at the Age of Two. *COMPSAC 78 Proceedings.*

Alford, M.W. 1985. SREM at the Age of Eight; The Distributed Computing Design System. *Computer*, 18(4):36–46.

Baeza-Yates, R.A. 1992. Introduction to Data Structures and Algorithms Related to Information Retrieval. (W.B. Frakes and R. Baeza-Yates Eds.) *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, pp. 1–12.

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

Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Booch, G. 1986. *Software Engineering with Ada*. San Francisco, CA: Benjamin-Cummins. Second Edition.

Borgida, A., Greenspan, S., and Mylopolous, J. 1985. Knowledge Representation as the Basis for Requirements Specifications. *Computer*, 18(4):82–91.

Bowers, J. and Pycock, J. 1994. Talking Through Design: Requirements and Resistance in Cooperative Prototyping. *CHI '94*.

Burstin, M.D. 1984. Requirements Analysis of Large Software Systems. Ph.D. Dissertation, Department of Management, Tel Aviv University, Tel Aviv, Israel.

Cavazza, M. and Zweigenbaum, P. 1992. Extracting Implicit Information from Free Text Technical Reports. *Information Processing and Management*, 28(5):609–618.

Cruse, D.A. 1986. *Lexical Semantics*. Cambridge: Cambridge University Press.

Damerau, F.J. 1993. Generating and Evaluating Domain-Oriented Multi-Word Terms from Texts. *Information Processing and Management*, 29(4):433–447.

Dardenne, A., Lamsweerde, A. van, and Fickas, S. 1993. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20:3\n50.

Davis, A.M. 1990. *Software Requirements: Analysis and Specification*. Englewood Cliffs, NJ: Prentice-Hall.

Estrin, G., Fenchel, R.S, Razouk, R.R., and Vernon, M.K. 1986. SARA (System ARchitect's Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311.

Finkelstein, A., Kramer, J., and Nuseibeh, B. 1992. Viewpoints: A framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57.

Fox, C. 1992. Lexical Analysis and Stop Lists. (W.B. Frakes and R. Baeza-Yates Eds.) *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, pp. 102–130.

Frakes, W.B. and Baeza-Yates, R. 1992. *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall.

Goguen, J.A. and Linde, C. 1993. Techniques for Requirements Elicitation. *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA, pp. 152–164.

Goguen, J.A. 1994. Requirements Engineering as the Reconciliation of Technical and Social Issues. (J.A. Goguen and M. Jirotka Eds.) *Requirements Engineering: Social and Technical Issues*. Academic Press, pp. 165–199.

Goldin, L. and Berry, D.M. 1994. AbstFinder: A Prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology, and Evaluation. *Proceedings First International Conference on Requirements Engineering*. Colorado Springs, CO: IEEE Computer Society, pp. 84–93.

Goldin, L. 1994. A Method for Aiding Requirements Analysts in Requirements Elicitation for Large Software Systems. Ph.D. Dissertation, Faculty of Computer Science, Technion, Haifa, Israel.

Harel, D. 1987. On Visual Formalisms. *Communications of the ACM*, 30(6)

Holtzblatt, K. and Jones, S. 1993. Contextual Inquiry: A Participatory Technique for System Design. (A. Namioka and D. Schuler Eds.) *Participatory Design: Principles and Practice*. Hillsdale, NJ: Erlbaum.

IAI 1989. System Specification for Unmanned Air Vehicle — Short-Range (UAV-SR) System (RFP). Internal Report, Israeli Aircraft Industries, Ltd.

IEEE 1988. *IEEE Software*, 5(2)special issue on Computer Aided Software Engineering.

IEEE 1993. *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA: IEEE Computer Society.

IEEE 1994. *IEEE Software*, 11(2)special issue on Requirements Engineering.

IEEE 1994. *Proceedings of the First International Conference on Requirements Engineering*. Colorado Springs, CO: IEEE Computer Society.

IEEE 1995. *Proceedings of the Second IEEE International Symposium on Requirements Engineering*. York, England, UK: IEEE Computer Society.

Ishihara, Y., Seki, H., and Kasami, T. 1993. A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies. *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA, pp. 232–239.

Jackson, M.A. 1975. *Principles of Program Design*. London: Academic Press.

Johnson, F.C., Paice, C.D., Black, W.J., and Neal, A.P. 1993. The Application of Linguistic Processing to Automatic Abstract Generation. *Journal of Document and Text Management*, 1(3):215–241.

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

Krasner, H. 1988. Requirements Problems in Large Software Projects: New Directions for Software Engineering Technology. Technical Report, MCC, Austin, TX.

Lehman, M.M. 1980. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

Leite, J.C.S.P. and Freeman, P. 1991. Requirements Validation through Viewpoint Resolution. *IEEE Transactions on Software Engineering*, SE-17(12)

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

Lubars, M., Potts, C., and Richter, C. 1993. A Review of the State of Practice in Requirements Modeling. *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA, pp. 2–14.

Luhn, M. 1958. The Automatic Creation of Literature Abstracts. *IBM Journal of Research and Development*, 2(2):159–165.

Luqi 1992. Computer-Aided Prototyping for a Command-and-Control System Using CAPS. *IEEE Software*, 9(1):56–67.

Maarek, Y. and Berry, D.M. 1988. The Use of Lexical Affinities in Requirements Extraction. Technical Report, Faculty of Computer Science, Technion, Haifa, Israel.

Maarek, Y. 1989. Using Structural Information for Managing Very Large Software Systems. Ph.D. Dissertation, Faculty of Computer Science, Technion, Haifa, Israel.

Martin, J. and Tsai, W.T. 1988. An Experimental Study in Upstream Software Development. Technical Report, University of Minnesota, Minneapolis, MN.

Myers, G.J. 1979. *Composite/Structured Design*. New York, NY: van Nostrand Reinhold.

Orr, K.T. 1977. *Structured Systems Development.* New York: Yourdon.

Orr, K.T. 1981. *Structured Requirements Engineering.* Topeka, KS: Ken Orr & Associates.

Parnas, D.L. 1972. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(2):1053–1058.

Pfleeger, S.L. 1994. Experimental Design and Analysis in Software Engineering, Part 1. *Software Engineering News*, 4(19):16–20.

Pfleeger, S.L. 1995. Experimental Design and Analysis in Software Engineering, Part 2. *Software Engineering News*, 20(1):16–21.

Pfleeger, S.L. 1995. Experimental Design and Analysis in Software Engineering, Part 3. *Software Engineering News*, 20(2):14–15.

Rau, L.F., Jacobs, P.S., and Zernik, U. 1989. Information Extraction and Text Summarization Using Linguistic Knowledge Acquisition. *Information Processing and Management,* 25(4):419–428.

Ross, D.T. 1977. Structured Analysis (SA): A Language for Communicating Ideas. *IEEE Transactions on Software Engineering,* SE-3(1):16–33.

Ryan, K. 1993. The Role of Natural Language in Requirements Engineering. *Proceedings of the IEEE International Symposium on Requirements Engineering*. San Diego, CA, pp. 240–242.

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

Saeki, M., Horai, H., Toyama, K., Uematsu, N., and Enomoto, H. 1987. Specification Framework Based on Natural Language. *Proceedings of the Fourth International Workshop on Software Specification and Design*. Monterey, CA, pp. 87–94.

Salton, G. and McGill, M.J. 1983. *Introduction to Modern Information Retrieval.* New York: McGraw-Hill.

Salton, G. 1986. Another Look at Automatic Text Retrieval Systems. *Communications of the ACM*, 29(7):648–656.

Salton, G. 1989. *Automatic Text Processing: The Translation, Analysis, and Retrieval of Information by Computer*. Reading, MA: Addison-Wesley.

Schach, S.R. 1992. *Software Engineering.* Boston, MA: Aksen Associates & Irwin. Second Edition.

Sievert, G.E. and Mizell, T.A. 1985. Specification-Based Software Engineering with TAGS. *Computer*, 18(4):56–66.

Sklar, B. 1988. *Digital Communication Fundementals and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Smeaton, A.F. 1995. Low Level Language Processing for Large Scale Information Retrieval: What Techniques Actually Work. *Proceedings of a Workshop: Terminology, Information Retrieval, and Linguistics*. Rome.

Srinivasan, R. 1992. Thesaurus Construction. (W.B. Frakes and R. Baeza-Yates Eds.) *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, pp. 161–218.

Teichroew, D. and Hershey, E.A. III 1977. PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, SE-3(1):41–48.

Wilensky, R. 1986. Points: A Theory of the Structure of Stories in Memory. (B.J. Grosz, K. Sparck Jones, and B.L. Webber Eds.) *Readings in Natural Language Processing.* Los Altos, CA: Morgan Kaufman.

Zahniser, R.A. 1993. Design by Walking Around. *Communications of the ACM,* 36(10):115–123.

Zave, P. 1982. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269.

Appendix

```
# 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|
```