The Use of Lexical Affinities in Requirements Extraction

Yoëlle S. Maarek* and Daniel M. Berry§

Computer Science Department, Technion, Haifa 32000, Israel

Abstract

The use of lexical affinities to help a human requirements analyst find abstractions in problem descriptions is explored. It is hoped that a lexical affinities finding tool can be used as part of an environment to help organize the sentences and phrases of a natural language problem description to aid the requirements analyst in the extraction of requirements. An experiment to confirm its effectiveness is described.

1. INTRODUCTION

The first steps in the development of any computational system should be the writing of requirements with the client's help. It may be necessary to build a prototype first, but ultimately before building a production-quality version, it is necessary to agree upon what is to be in the system. Winchester and Estrin [34] list a number of requirements for the requirements themselves. The main of these from the programmer-client perspective are that the requirements must be understandable to both the customers and the designers and builders; the parts of the requirements must be consistent with each other; and the requirements must be complete so that the designers and builders do not have to make unintended value judgements during their work.

This paper deals ultimately with, describes, and determines the effectiveness of one tool designed to assist in one part of the process of writing requirements. It is essential that the reader understand the context in which this tool is expected to operate. Hence, Sections 2 through 5 are devoted to briefly describing this context.

2. THE PROBLEM

Many system design or programming methods, e.g, those of Jackson [19], Parnas [26], Booch [10], Myers [23], Orr [24,25], etc., start from a clear statement of the requirements and show how to arrive at a design of a program or even at a program meeting these requirements. However, none of these methods really explain how these requirements are obtained in the first place. It seems clear to us (at least) that the writing of the requirements is a *major* part of the problem, and that once these are available, the arrival at an implementation, by comparison, is relatively straightforward.

Large E type [20] software, for which it is difficult or even impossible to obtain clear requirements, is usually developed for a client organization in which there are many people who have some view or say as to what the desired system should do. These views range from being totally unrelated to each other to being totally inconsistent with each other. It is no wonder that the distillation of these views into a consistent, complete, and unambiguous statement of the requirements, albeit in natural language, is a *major* part of the problem of developing software which meets the client's needs. Therefore, it is essential to have methods and tools that help in distilling these many views into coherent requirements.

3. PAST WORK

There are already a variety of systems, tools, and methods for dealing with requirements. These include SADT [28,27], IORL [31], PSL/PSA [32], RDL [34], RSL [5,6,7], RML [11] and Burstin's prototype [12] tool. The first two are graphically oriented, and the second of these is automated. The remainder work from highly constrained subsets of English consisting of sentences, each of which states one requirement to which the final implementation must adhere. These sentences can be considered as relations in a database. Those which are automated have tools for working with the sentences and abstractions of the requirements document once these sentences and abstractions have been recognized and stated. Due to space limitations, only those having a direct impact on this work are described in detail herein. A more complete discussion can be found in [3]

Burstin's prototytpe tool allows tuples of a relation, i.e., sentences, each with a verb and objects, to be organized into a hierarchy of abstractions. Each abstraction contains those sentences sharing a common collection of objects, with the verbs representing procedures of the abstraction. There are tools for introducing and moving sentences to and from abstractions and for placing and moving abstractions in the hierarchy. There is also a rudimentary application-oriented expert system that helps recognize when two or more phrases of sentences may be talking about the same thing, e.g., "plane" and "airplane" or "passenger" and "flier."

All of the mentioned systems are useful for working with sentences and abstractions of a requirements document, once they are recognized and formed. Organizations implementing and using two of these, PSL/PSA and SREM, report much user satisfaction [32,7,30].

It is interesting that all of the above requirements analysis systems deal with relations and that all but the first two, which are not picture-oriented, have gone to the use of a relational database for storing the relations. All can be used to support an abstraction-based requirement development which leads naturally to an abstraction-based software development [9].

However, none of the methods and tools 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. The descriptions of all of the methods either fail to mention how to get the sentences or say something to the effect of "get them and write them down" as if there were nothing to it.

Teichroew and Hersbey [32] offer that "since most of the data must be obtained through personal contact, interviews will still be required." PSA does help this gathering process in that its "intermediate outputs ... also provide convenient checklists for deciding what additional information is needed and for recording it for input."

Alford [5] says that the "SREM steps address the sequence of activities and usage of RSL and REVS to generate and validate the requirements. It

Address correspondence to the second author.

§ This work was supported in parts by the University of California MICRO program, Unisys Corporation, and NCR Corporation.

* Current Affiliation: IBM T.J. Watson Research Center, P.O.B. 704, Yorktown Heights, NY 10598

©1989 ACM 0-89791-305-1/89/0500/0196$00.75

assumes [italics are not in the original] that system function and performances have been allocated to the data processor, and have been collected into a Data Processing Subsystem Performance Requirement or DPSPR."

Even eight years later, Scheffer, Stone, and Rzepka [30], from a completely different company which had been using SREM, state only that the "initial input to SREM is a system specification that is translated into RSL and interpreted to determine the interfaces with the outside world, the messages across these interfaces, and the required processing relationships and flows."

The first step of the TAGS method [31] is the conceptualization step, "User concepts and requirements are used to develop a conceptual model that is the basis for subsequent engineering." This conceptual model is the top level SBD. In the cited article, there is advice on the issues that should be dealt with in arriving at it. However, no tools are provided, since the TAGS method deals with activities that follow the production of this first SBD.

Therefore, we feel that the gap between the initial fuzzy natural language statements from the individuals in the client organization to the sentences, i.e., relations, with which these tools work is still too large. Methods and tools are needed to close this gap.

4. ENVISIONED REQUIREMENTS GATHERING ENVIRONMENT

We ultimately envision an integrated environment, REGEE, for gathering, sifting, and writing requirements. This environment may very well be part of a larger environment used for software development, deployment, and maintenance [1,2]. REGEE is in the very rudimentary prototyping stage, as we do not understand the process it is supposed to assist. For now, REGEE 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. REGEE should support any possibility. Usually the input, transcript language will be some natural language, possibly with pictures [17]. However, the output language, in which the requirements are written, can be anything from natural language, possibly with pictures, to predicate calculus; in particular, it should be possible to use any of the requirements expression languages mentioned in Section 3.

We do not know enough about effective requirements writing in order to be able to codify the process. Thus at least for now, a completely automated expert-system approach is out of the question. We therefore envision an environment consisting of clerical tools that help with the tedious, error-prone steps of what one particular human RA, the second author, does.

We view a requirements document both in the process of being written and in final form as a network, very often a hierarchy, of nodes each denoting an abstraction and containing a description of all that is known and required about the abstraction. The arcs between the nodes can be used to describe the "uses" relation or any other basis for organizing the set of abstractions.

REGEE needs two basic kinds of tools,

1. to help identify the abstractions that will make the nodes from the transcripts of the interviews, and
2. to help organize the abstractions into a network of abstraction-nodes, each to contain a consistent, complete, unambiguous, coherent, and concise description of that abstraction.

This paper deals with a proposed tool of the first kind, a lexical affinities finder, to be described below. To understand the rationale behind the lexical affinities finder, it is useful to understand the envisioned tool of the second kind. We are building a significant enhancement of the Burstin tool mentioned in Section 3. This tool provides a medium in which nodes, implemented as windows on a work station screen, can be organized into a network, as suggested by Figure 1. Each window can be made to hold arbitrary text, including text that causes displaying of a picture. Any arbitrary element of the text of any window can be given links connecting the element to any window or to any element, possibly in another window. Figure 2 shows two windows from a description of an airline reservation system.

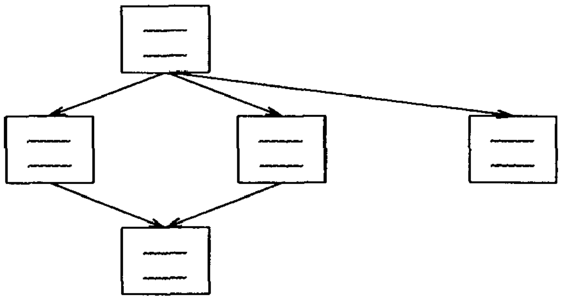


Figure 1: Network of Nodes

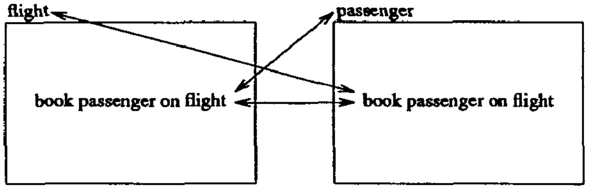


Figure 2: Links

The links connect an element to windows giving more details about the element or to other elements talking about the same or related concepts, as the human RA desires. The RA can use these links to navigate through the windows as he or she is tracking down the information that allows the contents of each window to be refined into a consistent, complete, unambiguous, coherent, and concise description of the window's abstraction.

This description of the tool of the second kind suggests building it on top of some existing hypertext system [35,13,15]. Indeed, Garg and Scacchi have suggested maintaining all life-cycle documents as hypertext [16].

5. ABSTRACTION IDENTIFICATION

The way identification of abstractions is done now is that the human RA scans the transcripts trying to note important subjects and objects of sentences, i.e., nouns. The problem is that humans get tired, get bored, fall asleep, and overlook relevant ideas. So we want a tool that does the clerical part of the search without getting tired, getting bored, falling asleep, and overlooking anything. The human RA still does all the *thinking* with the output of this tool, confident that no occurrence of any noun has been overlooked.

Our first idea, reported in [8], was to use a parser to find the nouns. However, we tried it and found that the few errors it made were so distracting that it was more comfortable to do it by hand. Moreover, we realized that even a better, but still imperfect, parser would still be distracting. Thus, the program did not inspire confidence that it found everything. Maybe there was an important noun that was overlooked because it appeared to the parser as a verb. Even a better, but still ultimately imperfect, parser does not solve this confidence problem. We want something with guaranteed coverage, even if it is less intelligent. The lack of intelligence in the tool is no problem because a human is applying his or her intelligence to the output of the tool.

6. REPEATED PHRASE FINDER

A second idea, reported in [4,3], (Most of this introductory material is borrowed from this paper.) is to use findphrases, a repeated phrase finder. The idea is based on the observation that the frequency of occurrence of a term within a text carries much information on the importance of this term in the text. Indeed, it has been empirically verified that a writer repeats words of importance in a text as she explains or varies her argument [21]. Just counting repeated words within a document is not sufficient for identifying its major abstractions. In counting isolated single words, 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 its simplest application, the user provides findphrases with the text to be analyzed and a file containing punctuation and keywords. The punctuation and keywords are used by findphrases to break the text into sentences. findphrases processes the phrases of the sentences and produces a series of reports. The basic output contains: (1) the input file as is with lines numbered and the punctuation and keywords overstruck, (2) a frequency ranked table of repeated phrases, and (3) an alphabetically ordered table of repeated phrases. Each entry in these tables gives the numbers of the lines in which the phrase occurs, so that each phrase may be examined in its original context to decide which abstraction is really represented by the phrase. A number of options are provided that the user may use to controll the parsing of the input text into tokens and phrases, to control the printing of the phrases in the tables of the output, and to indicate which additional tables are to be printed.

Observe that there is a learning process involved in using findphrases effectively. First, it appears that there are different characteristic punctuation-keyword, multi-token and initial ignored-phrases files for each language. These can be catalogued for general use. In addition for each class of applications, there appears also to be a characteristic set of additional ignored phrases. Finally, as one is doing a particular application, one finds it useful to extend the ignored phrases file with common words that are actually important abstractions, but whose presence skews the list and populates it with too much noise for finding the other abstractions.

The works [4, 3] describe tests of effectiveness of findphrases in helping the human RA identify abstractions.

The tests involved four examples of program development, each of which had multiple versions of the same program ranging from natural language descriptions, through designs, decompositions, etc., to code. Three of these had been published in the literature and one had not.

It was desired to determine if findphrases is effective in helping the human RA to find, in the natural language transcripts of interviews about a system under development, *all* of the abstractions that serve as the basis for the requirements, design, and implementation. It was deemed effective if we, as humans, do indeed recognize the same set of abstractions in the outputs of findphrases run (with the appropriate parameter files in each case) on all versions of the same problem. Finding the same set of abstractions in all versions says that the abstractions found in the first version, the natural language description, are sufficient to cover all abstractions that will be needed for all subsequent versions, including the code and that no other abstractions will need to be invented.

The first experiment is Abbott's example of programming with the help of natural language. This example is the focus of a paper [8] that points to the need of this phrase finding tool. For this experiment, three versions of the program solution are compared. The first version was written in standard English, the second in an Ada™[2]-based program design language, and the third in Ada. The second experiment is the problem of writing the phrase finder itself. In writing the phrase finder, the manual page served as the requirements document. findphrases was run with its own manual page to see if the same abstractions that formed the basis for the modular decomposition used in writing the code are identified from the information provided by findphrases. The third experiment takes Mitchell's text book [22] example of writing a sorting program starting from the English statement of the requirements and ending with a Pascal program developed with a structured programming method. Four versions of the program solution are compared, the initial English description, two program design language descriptions, and the final Pascal program.

The fourth experiment takes Wiener and Sincovec's text book [33] example of writing a spelling checker program. They start with a statement of the requirements, develop a modular decomposition for the solution, and produce an Ada program.

In these experiments, findphrases was found to be effective in aiding the human RA to identify abstractions. 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. 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 by 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 by showing the context from which the word comes.

7. LEXICAL AFFINITIES

We propose to take *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 [14]. For our purposes, we restrict this definition, by observing LAs within a finite document rather than on the whole language. For instance, in this paper, *requirement* and *analysis* are bound by a lexical affinity. For our purpose, we consider only LAs involving *open-class words* as meaning bearing. Open classes gather nouns, verbs, adjectives and adverbs whereas *closed-class words* are represented by pronouns, prepositions, conjunctions and interjections [18]. The LA finder's output is a list of lexical affinities with their associated frequency of appearance within the considered text. For instance, the analysis of the *rm* manual page in the UNIX™[3] environment, returns as the most frequent LAs, the list (delete file), (file file), (file permission), etc. which all appear three times within the one-page document. Were the manual page taken as a statement of the requirements of rm, we believe that this list of LAs would be of great help for assisting the human RA in his or her process of extracting requirements. As we see in the above example, among the three LAs cited, two represent major abstractions. This can be much improved by accounting for the general context or universe to which the document belongs. This would allow filtering out such LAs as (file file) cited above.

In order to account for the general context or universe, LAs need to be scaled according to their specific contribution in the given document. As a measure of their contribution, we propose evaluating the ***resolving power*** of every LA. We define the resolving power ρ of an LA is a function of its quantity of information and its frequency of appearance within the considered text. The quantity of information represented by a word *w* in a given textual universe is defined as [29]

$$\textsc{Info}(w)=-\log_2 P\{w\}.$$

Thus, if a word asterisk occurs once in every 20 000 words, its quantity of information is estimated to be

$$\textsc{Info}(\texttt{asterisk})=-\log_2 5\times10^{-5}=14.29.$$

In contrast, the word the that occurs once in every 15 words, has its information contents estimated to be

$$\textsc{Info}(\texttt{the})=3.9.$$

Drawing on this definition of the quantity of information for single words, we define the resolving power of an LA within a document *d* as follows.

Let (w_1,w_2,f) be a tuple retrieved while analyzing a document *d*, where (w_1,w_2) is an LA appearing *f* times in *d*. The resolving power of this LA in *d* is defined as:

$$\rho((w_1,w_2,f))=f\times\textsc{Info}(w_1)\textsc{Info}(w_2)$$

The higher the resolving power of an LA is, the more characteristic it is of the considered document. The best LAs, in terms of resolving power, within a document, represent key concepts of the considered document. These LAs may therefore provide valuable assistance to the human RA in the process of extracting requirements.

In order to conduct a first test of the effectiveness of LAs in helping the human RA to find abstractions, we have tried to use LAs to find the abstractions in one of the examples used to test findphrases, namely the findphrases manual page.

8. THE findphrases MANUAL PAGE

One of the original experiments for the repeated phrase finder considered the problem of developing a program for findphrases. That is, findphrases was tested using its own manual page, which is found in the appendix. The data abstractions identified by findphrases from this

[2] Ada was a trademark of the U.S. Dept. of Defense (AJPO).

[3] UNIX is a registered trademark of AT&T Bell Laboratories.

application were compared with the abstractions used in the decomposition for the final program. Space limitations prevent showing the actual output of the running of findphrases on the manual page. However, it can be found in [3].

The decomposition used to write the findphrases program includes fifteen different modules, as illustrated in Figure 3. (See Chapter 3, Section 3 of [4] for a description of the modules.) The modules with dashed outlines are built into the implementing programming language. Nine of the non-built-in modules, i.e., those with the thicker outlines, are data abstraction packages. Table 1 below shows the correspondence between these packages and some of the repeated phrases identified by findphrases from the manual page description:

THE DATA ABSTRACTION PACKAGES	THE REPEATED PHRASES
string_type_file	`strings, character strings`
argument_line	`argument, option`

output_file	`output,`
	`tables of the output`

chunk_file	`file(s), free-format`

punct_keyword_table	`punctuation, keyword(s),`
	`punctuation/keyword(s),`
	`punctuation-keyword-file`

multi_tokens_table	`multi-tokens,`
	`multi-tokens-file`

text_file	`text, input,`
	`arbitrary text`

phrases	`phrase(s),`
	`ignored phrases,`
	`repeated phrases`

sentences	`sentence(s)`

Table 1:Packages and Phrases

The name of each abstraction, except for chunk_file, directly corresponds to at least one repeated phrase appearing in the manual page. Although the phrase `chunk` does not appear in the description, the chunk_file abstraction is implied by the phrases indicated above. During the program's development, chunk_file was recognized as the abstraction to be used by the procedures that read the various free-format files. Thus, this abstraction is indirectly identified by repeated phrases appearing in the tables. Note that it does not bother us that we had to do some thinking to make these connections. We believe that this kind of thinking is precisely what an RA is doing when presented with such a list of concepts.

In this experiment, the repeated phrases tables produced by findphrases include phrases that identify the abstractions that were used in the program decomposition. The tables act as guides to the user when looking for the abstractions. The user's attention is first focused on the phrases with the highest frequency and then on possibly related phrases. The line numbers printed in the tables enable the user to locate the phrases in the text. By analyzing the context of the sentences containing the phrase, the importance of the phrase as a potential abstraction can be determined.

In addition to finding the list of abstractions, as the user analyzes the context of each phrase usage, a list of statements related to that phrase can be extracted. For example, consider the phrases `multi-token`, `multi-tokens` and `multi-tokens-file`. By examining the context of the lines that contain these phrases, one finds the following information:

1. the optional *multi-tokens-file* contains in free format the list of character strings to be taken as multi-tokens;
2. a multi-token is a string consisting of more than one symbolcharacter (non-word character);
3. if the –m option is present, the input text is parsed using the information provided by the *multi-tokens-file*;
4. a token may be a multi-token;
5. if the –v and the –m options are present, the list of multi-tokens is printed.

These statements represent requirements for the data abstraction multi-token. Located by the manual method, these same statements were used when the initial decomposition and implementation were developed. The final program uses a procedure to read the *multi-tokens-file* and a package to handle a multi-tokens table. These statements also guided the development of the text parsing routines.

Thus, in this experiment, the human RA finds in the repeated phrases generated from the manual page the very abstractions used in the modular decomposition and implementation.

9. THE LAs for THE findphrases MANUAL PAGE

We have extracted the LAs from the findphrases manual page and ranked them according to their resolving power. The whole UNIX manual has been used as the textual universe, in order to determine the quantity of information of each word. The 30 LAs with the highest resolving power are given in Table 2.

LEXICAL AFFINITY	ρ
`ignore-phrase`	699.123546
`phrase-repeat`	696.600983
`multi-token`	483.353302
`file-phrase`	418.820829
`contain-`**`phrase`**	299.996707
`file-keyword`	281.335816
`keyword-punctuate`	271.657352
`file-token`	268.032537
`file-punctuate`	263.292488
`format-free`	232.582511
`option-present`	228.395449
`begin-`**`phrase`**	208.569285
`phrase``-table`	207.914468
`repeat-table`	186.066941
`file-ignore`	163.864523
`file-multi`	160.532580
`phrase``-token`	138.637142
`entry-table`	132.656315
`contain-file`	131.339831
`list-token`	128.602978
`phrase``-tally`	124.433003
`contain-ignore`	122.670819
`number-supply`	119.521321
`list-phrase`	116.148446
`occurrent-phrase`	112.433009
`contain-list`	112.236757
`phrase-sentence`	111.806985
`phrase``-provide`	106.772850
`length-`**`phrase`**	105.449181
`free-token`	102.949931

Table 2: Most significant LAs

As a first step, the human RA quickly selects, among the top LAs, the main abstractions which are marked in boldface in Table 2. In our example, these abstractions are `ignore-phrase`, `phrase-repeat`, `multi-token`, `keyword-punctuate` and `phrase-sentence`. Single words can also be selected when they are involved in many of the most significant LAs, for instance, in our example, the word `phrase` appears to be a major abstraction. Some other abstractions can also be found after a finer analysis[4], such as chunk, represented by `free-format`, argument-line represented by `option-present` or text_file by noticing all the occurences of `file-phrase` and `text`.

In a second step, the human RA obtains more information on the main abstractions by performing a keyword-based search on the whole list of the LAs. The correspondence between the abstractions, selected at the first step, and the LAs selected during this search is given in Table 3. The most significant LAs leading identification of abstractions are in italics. These might form an initial list of operations, attributes, or methods of the abstractions.

[4] Please note that at this stage, the human RA is required to *understand* the transcript.

ABSTRACTIONS	LEXICAL AFFINITIES
ignored phrase	*ignore-phrase*
	file-ignore
repetition	*phrase-repeat*
	repeat-table
	find-repeat
multi_tokens	*multi-token*
	file-token
	phrase-token
	list-token
punctuation_keyword	*keyword-punctuate*
	file-punctuate
	file-keyword
	keyword-phrase
sentence	*phrase-sentence*
	occurring-sentence
	consecutive-sentence
	sentence-token
	present-sentence
	entry-sentence
phrase	ignore *phrase*
	phrase-repeat
	file-*phrase*
	contain-*phrase*
	begin-*phrase*
	phrase-table
chunk	*format-free*
argument_line	*option-present*
	multi-option
	argument-present
	argument-option
text_file	*file-phrase*
	arbitrary-text
	phrase-text
	repeat-text

Table 3: Packages and LAs

10. CONCLUSIONS

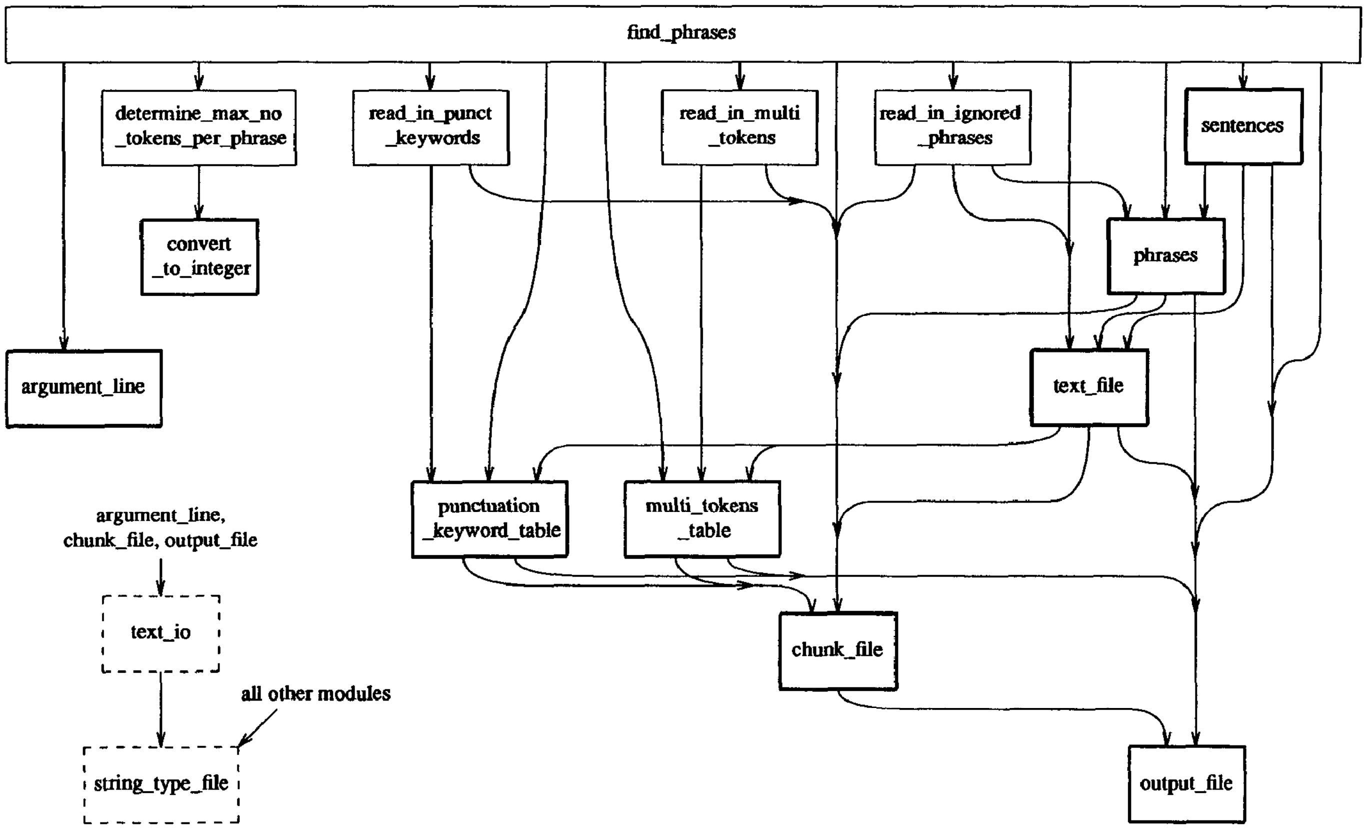
The initial attempt to use LAs to assist a human RA in finding abstractions looks promising. The LAs help the RA identify the same abstractions that were used to build the implementation. In addition, the LAs were helpful in organizing the information in the descriptions and allowed the user to focus on the important phrases. It will now be necessary to try out the LAs on other problems.

It is not clear at this time which is better, LAs or repeated phrases. The use of LAs finds repeated phrases whose key elements are not necessarily adjacent and are not necessarily appearing in the exact same form each time. However, at present the LA finder does not find LAs consisting of more than two words. It finds only verb-noun, adjective-noun, and other common grammatical structure pairs. LAs of more than two words must be inferred as joins of binary relations. Of course, the repeated phrase finder has no problems finding phrases of more than two words. Perhaps both should be used, and tools for dealing with both should be available to the users of REGEE. Perhaps a combined tool should be built in which non-adjacency, synonyms, and varying parts of speech are tolerated in finding repeated phrases.

REFERENCES

1. "Requirements for the Ada Programming Support Environment: STONEMAN," Technical Report, U.S. Department of Defense (1981).
2. *IEEE Software* 5(2) (March, 1988).
3. C. Aguilera and D.M. Berry, "The Use of a Repeated Phrase Finder in Requirements Extraction," *Journal of Systems and Software*(9) (1990 (To appear)).
4. C.S. Aguilera, "Finding Abstractions in Problem Descriptions using findphrases," M.S. Thesis, Computer Science Department, UCLA, Los Angeles, CA (October, 1987).
5. M.W. Alford, "A Requirements Engineering Methodology for Realtime Processing Requirements," *IEEE Transactions of Software Engineering* SE-3(1), pp. 60-69 (1977).
6. M.W. Alford, "Software Requirements Engineering Methodology (SREM) at the Age of Two," in *COMPSAC 78 Proceedings* (November, 1978).
7. M.W. Alford, "SREM at the Age of Eight; The Distributed Computing Design System," *Computer* 18(4), pp. 36-46 (April, 1985).
8. D.M. Berry, N.M. Yavne, and M. Yavne, "Application of Program Design Language Tools to Abbott's Method of Program Design by Informal Natural Language Descriptions," *Journal of Software and Systems*(7), pp. 221-247 (1987).
9. V. Berzins, M. Gray, and D. Naumann, "Abstraction-Based Software Development," *Communications of the ACM* 29(5), pp. 402-415 (May, 1986).
10. G. Booch, *Software Engineering with Ada*, Benjamin-Cummins, San Francisco, CA (1986). Second Edition.
11. A. Borgida, S. Greenspan, and J. Mylopolous, "Knowledge Representation as the Basis for Requirements Specifications," *Computer* 18(4), pp. 82-91 (April, 1985).
12. M.D. Burstin, "Requirements Analysis of Large Software Systems," Ph.D. Dissertation, Department of Management, Tel Aviv University, Tel Aviv, Israel (1984).
13. J. Conklin, "A Survey of Hypertext," MCC Technical Report No. STP-356-86, Rev. 1, MCC, Austin, TX (February 9, 1987).
14. D.A. Cruse, *Lexical Semantics*, Cambridge University Press, Cambridge (1986).
15. N.M. Delisle and M.D. Schwartz, "Contexts — A Partitioning Concept for Hypertext," *ACM Transactions on Office Information Systems* 5(2), pp. 168-186 (April, 1987).
16. P.K. Garg and W. Scacchi, "Maintaining Software Life Cycle Documents as Hypertext: Issues, Analysis, and Directions," Technical Report, University of Southern California, Los Angeles, California (1987).
17. D. Harel, "On Visual Formalisms," *Communications of the ACM* 30(6) (June, 1987).
18. R. Huddleston, *Introduction to the Grammar of English*, Cambridge University Press, Cambridge (1984).
19. M.A. Jackson, *Principles of Program Design*, Academic Press, London (1975).
20. M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE* 68(9), pp. 1060-1076 (September, 1980).
21. M. Luhn, "The Automatic Creation of Literature Abstracts," *IBM Journal of Research and Development* 2(2), pp. 159-165 (April, 1958).
22. W. Mitchell, *A Prelude to Programming: Problem Solving and Algorithms*, Reston Publishing, Reston, VA (1984).
23. G.J. Myers, *Composite/Structured Design*, van Nostrand Reinhold, New York, NY (1979).
24. K.T. Orr, *Structured Systems Development*, Yourdon, New York (1977).
25. K.T. Orr, *Structured Requirements Engineering*, Ken Orr & Associates, Topeka, KS (1981).
26. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* 15(2), pp. 1053-1058 (December, 1972).
27. D.T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering* SE-3(1), pp. 16-33 (January, 1977).
28. D.T. Ross and K.E. Schoman, Jr., "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering* SE-3(1), pp. 6-15 (January, 1977).
29. G. Salton and M.J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York (1983).
30. P.A. Scheffer, A.H. III Stone, and W.E. Rzepka, "A Case Study of SREM," *Computer* 18(4), pp. 47-54 (April, 1985).
31. G.E. Sievert and T.A. Mizell, "Specification-Based Software Engineering with TAGS," *Computer* 18(4), pp. 56-66 (April, 1985).
32. D. Teichroew and E.A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information Processing Systems," *IEEE Transactions of Software Engineering* SE-3(1), pp. 41-48 (January, 1977).
33. R. Wiener and R. Sincovec, *Software Engineering with Modula-2 and Ada*, John Wiley & Sons, New York (1984).
34. J. Winchester and G. Estrin, "Requirements Definition and Its Interface to the SARA Design Methodology for Computer-Based Systems," *AFIPS Conference Proceedings* 51, pp. 369-379 (June, 1982).
35. N. Yankelovich, N. Meyerowitz, and A. van Dam, "Reading and Writing the Electronic Book," *Computer* 10(18), pp. 15-30 (October, 1985).

Figure 3: findphrases DECOMPOSITION



APPENDIX findphrases MANUAL PAGE

FINDPHRASES (LOCAL) UNIX Programmer's Manual FINDPHRASES (LOCAL)

NAME

findphrases – find repeated phrases in an arbitrary text

SYNOPSIS

findphrases [**−n***number*] **−p***punctuation-keyword-file* [**−x***ignored-phrases-file*] [**−m***multi-tokens-file*] [**−u**] [**−b**] [**−s**] [**−t**] [**−v**] [**−c**]

DESCRIPTION

All files mentioned in the synopsis provide their data in what is referred to as free format subject to particular restrictions to be described for each case. In free format, the items of the file may be entered zero or several per line with a mixture of blanks and tabs before, in between, and after the items. Consequently, no item can include a blank, a tab, or a newline.

The **−n** argument is optional and if present provides a number *number* serving as the maximum length phrase (to be described later) to be tallied. If this argument is not present, if it does not supply a number, or if the supplied number is outside the reasonable range of greater than zero and less than or equal to 50, then *number* is taken as 10.

The *punctuation-keyword-file* contains in free format a list of those character strings to be taken as punctuation/keywords (see below). The optional *ignored-phrases-file* contains one-per-line a list of phrases to be ignored in the tallying (see below). In each line, the tokens (see below) are in free format. The optional *multi-tokens-file* contains in free format a list of those character strings consisting of more than one symbolcharacter (see below) which are to be taken as multi-tokens (see below).

No assumptions are made about the standard input, thus it may be an arbitrary text. The program parses the text into words and symbolcharacters. These in turn are formed and classified into tokens and punctuation/keywords based on the information provided by the *punctuation-keyword-file* and, when the **−m** option is present, the *multi-tokens-file*.

First some definitions are necessary:

Whitespace: blank, tab, newline, beginning-of-file, end-of-file

Wordcharacter: letter, digit, _

Symbolcharacter: any printable character which is neither a wordcharacter nor a blank

Word: any sequence of wordcharacters delimited on each side by whitespace or a symbolcharacter

Punctuation/Keyword: whatever is in the *punctuation-keyword-file*; the symbolcharacter strings are called punctuation and the wordcharacter strings are called keywords

Multi-token: whatever is in the *multi-tokens-file*

Token: any word, symbolcharacter, or multi-token which is not listed in the *punctuation-keyword-file*

Sentence: list of tokens delimited on each side by punctuation/keyword

Phrase: one or more consecutive tokens occurring within one sentence

The main job of this program is to tally the occurrence of all phrases in all sentences. The maximum length phrase that has to be considered is that of *number* tokens. If the *ignored-phrases-file* is provided, then the phrases given in the file are to be ignored in the tallying. If the **–b** option is used along with the *ignored-phrases-file*, then phrases which begin with an ignored phrase are also ignored in the tallying.

The standard output consists of:

a copy of the input as is, with the lines numbered and the punctuation/keywords overstruck two times (i.e., printed three times in place) so that they can be spotted easily,

a frequency ranked table of the repeated phrases. i.e., those appearing more than once among the sentences; that is the entries of the table are given in order of decreasing frequency, and

an alphabetically ordered table of the repeated phrases.

In the two tables, the entry for a repeated phrase consists of:

a sequence of asterisks indicating the phrase's frequency as a percentage of the maximum frequency; in this one asterisk represents 10%,

the actual number of occurrences of the repeated phrase,

the repeated phrase itself, and

a list of the numbers of all lines containing the beginning of the repeated phrase.

In printing the repeated phrase itself in a table entry, the underscores, i.e., "_", are printed as blanks. This means that an underscore can be used immediately preceeding or following a word that looks like a keyword to prevent it from being considered a keyword.

Note that the definition of "phrase" is independent of the number of times it occurs in the sentences. An *ignored phrase* is simply one to be ignored in the tallying but not in searching for phrases. A phrase which contains an ignored phrase which itself is not ignored is to be tallied. When the **–b** option is present, a phrase which begins with an ignored phrase is not to be tallied. A *repeated phrase* is one whose final tally is greater than one. Only the repeated phrases show up in the tables of the output.

Typically, the *ignored-phrases-file* will contain so-called noise phrases such as "a", "an", "the", "of", "of the", etc. plus any useless phrases found in previous runs of the program.

One particular configuration of the files is as follows:

Punctuation-keyword-file: ; [ ] **abort accept access all and array at begin body case constant declare delta digits do else elsif end entry exception exit for function generic goto if in is limited loop mod new not null of or others out package pragma private procedure raise range record rem renames return reverse select separate subtype task terminate then type use when while with xor**

Multi-tokens-file: ** := <= >= /= .. <> << >>

This configuration is suited for finding repeated phrases in Ada™ (Ada is a trademark of the U. S. Department of Defense.) or in an Ada-based program design language.

If the **–u** option is present, then only the unique phrases that are not wholly and everywhere contained in another phrase are listed in the tables of the output. In addition to the already specified output, if the **–s** option is present, then all the sentences are listed; if the **–t** option is present, then all the tokens are listed; if the –v option is present, then the output is verbose with the punctuation/keywords listed, and when the **–m**, and respectively the **–x**, option is present, the multi-tokens, and respectively the ignored phrases, are listed. If the **–c** option is present, then upper and lower case distinctions are to be applied in determining whether a phrase is in a sentence. The default is to ignore case distinction in the comparisons.

DIAGNOSTICS

They are good, of course.

BUGS

There are none, of course.

As an example, when running findphrases against its own manual page, the following findphrases options were used:

i. The *punctuation-keyword-file* consisted of a standard set of punctuation: period, comma, colon, semi-colon, question mark, and exclamation point.

ii. The *ignored-phrases-file* consisted of a list of sixty-seven phrases to be ignored: apostrophe, opening and closing double quotes, opening and closing parentheses, opening and closing brackets, dash, colon, underscore, 10, a, ada, all, an, and, any, are, as, based, be, begin, beginning, below, by, called, can, configuration, course, described, e, each, end, entry, file, for, i, if, in, into, is, it, items, may, not, number is, of, on, or, respectively, see, so, synopsis, taken, than, that, the, then, this, those, thus, times, to, tokens, (see below), when, which, and with. These phrases were found in prior runs.

iii. The *multi-tokens-file* consisted of the following symbols: '' `` ** := <= >= /= .. <> << >>

iv. The –u and –b options were used to print the Tables of Repeated Phrases. The –b option was used to ignore in the tallying of repeated phrases those phrases which began with an ignored phrase. The number used with the –n option was 11.