Published in Natural Language Engineering, Cambridge University Press, Vol. 5, Issue 1, pp.1-18, 1999

An Approach to Program Understanding by Natural Language Understanding

Letha H. Etzkorn, Lisa L. Bowen, Carl G. Davis

Computer Science Department

The University of Alabama in Huntsville

Huntsville, AL 35899

letzkorn@cs.uah.edu, lbowen@cs.uah.edu, cdavis@cs.uah.edu

Abstract

An automated tool to assist in the understanding of legacy code components can be useful both in the areas of software reuse and software maintenance. Most previous work in this area has concentrated on functionally-oriented code. Whereas object-oriented code has been shown to be inherently more reusable than functionally-oriented code, in many cases the eventual reuse of the object-oriented code was not considered during development.

A knowledge-based, natural language processing approach to the automated understanding of object-oriented code as an aid to the reuse of object-oriented code is described. A system, called the PATRicia system (Program Analysis Tool for Reuse) that implements the approach is examined. The natural language processing/information extraction system that comprises a large part of the PATRicia system is discussed and the knowledge-base of the PATRicia system, in the form of conceptual graphs, is described. Reports provided by natural language-generation in the PATRicia system are described.

Keywords: Natural Language Processing, OO Software, Software Reuse, Knowledge-based systems.

1 Introduction

The reuse of software products on projects subsequent to that project during which the software products were developed is a long term goal of software developers, since such reuse can save many of the costs associated with redevelopment (Hooper and Chester 1991; Basili and Abd-El-Hafiz 1992). One of the difficulties that tends to hinder software reuse lies in the numerous paradigms by which software is organized and developed. The object-oriented paradigm itself tends to result in code that is inherently more reusable than code developed using the functionally-oriented paradigm. However, much object-oriented legacy code exists for which eventual reuse of the code was not considered during the development process. Thus identification of reusable components in such code can still be a difficult process.

This research addresses the identification of reusable components in legacy code that was developed using the object-oriented paradigm. This approach uses knowledge about existing object-oriented code to assist in preparing portions of that code, primarily classes (with their associated methods) or class hierarchies, for reuse in a subsequent software project, thus providing a technique that can significantly enhance the reusability of object-oriented software.

2 Background

Program understanding approaches are generally divided into three categories (Basili and Abd-El-Hafiz 1992; Abd-El-Hafiz and Basili 1996). The algorithmic

approaches annotate programs with formal specifications. These programs rely on the user to annotate the loops, and provide assistance only in proving the correctness of the annotations. The knowledge-based approaches annotate programs with informal, English text specifications. The knowledge-based approaches typically require less user interaction than the algorithmic approaches. The transformational approaches are similar to the transformational paradigm of automatic program synthesis but with the application direction of the transformation rules reversed.

The knowledge-based program understanding approaches can further be divided into three major areas: 1) the graph-parsing approach, 2) the heuristic concept-recognition approach, and 3) the informal tokens approach.

The graph-parsing approach (Rich and Wills 1990) translates a program into a flow graph (the graph employs both data and control flow). The domain base contains a library of graphical grammar-rule plans. The program's graphs are compared versus the plans in the library, and thus recognition becomes a graph-parsing problem.

The heuristic concept-recognition approach (Ning 1989; Ning, Engberts, and Kozaczynski 1994) provides a knowledge-base of programming events, such as statement, control, inc-counter, dec-counter, bubble sort, etc. The lowest level events are matched to code statements, then the low level events are combined to form higher level events. Ning, Engberts and Kozaczynski (Ning et al. 1994) use an approach that is primarily a heuristic concept-recognition approach, but that also employs control flow and data dependency knowledge in the recognition of the most primitive events.

The informal tokens approach is that taken by Biggerstaff (Biggerstaff, Mitbander, and Webster 1994). Biggerstaff argues that a parsing oriented approach based on

structural patterns of programming language features is necessary, but not sufficient for solving the general concept assignment problem. Parsing approaches return programming-oriented concepts such as searches, sorts, numerical integration, etc. However, human-oriented concepts (acquire target, reserve airplane seat) are not allied directly with programming language features or algorithms. Thus, in order to extract the higher level, human oriented concepts it is necessary to extract informal information from the source code in terms of natural language tokens from comments and identifiers, and some heuristics related to occurrences of closely related concepts and the overall pattern of relationships.

3 Object-Oriented Program Understanding Approach

An approach based on informal tokens is particularly appropriate for object-oriented code. For object-oriented code, more so than for functionally-oriented code, much understanding can be performed by simply looking at comment and identifier names (Etzkorn, Davis, Bowen, Etzkorn, Lewis, Vinz, Wolf 1996; Etzkorn and Davis 1996; Etzkorn and Davis 1997). This is true since object-oriented code is organized in classes, with everything required to implement a class at least mentioned (if not defined) in the class definition. For example, consider Figure 1. Figure 1 represents a C++ class whose purpose is serial communications. The member functions associated with this class would largely consist of writes-to and reads-from various hardware locations. The hardware writes and reads would be extremely difficult to understand without a knowledge of the schematic of the underlying hardware. Obviously in this case a maintenance person, or a software engineer considering inclusion of the class and its member functions in a

software reuse library, would be heavily dependent on the information received from comments and identifiers.

This research employs a knowledge-based, heuristic, concept-driven approach to object-oriented program understanding, that focuses heavily on informal tokens such as comments and identifiers. Natural language parsing techniques are used in this approach.

Note that previous automated reuse component identification systems have employed an algorithmic approach to program understanding (Caldiera and Basili 1991) (as well as focusing on functionally-oriented code). Algorithmic approaches to program understanding can be difficult to automate, largely due to the difficulties inherent in loop analysis. In the CARE (Computer Aided Reuse Engineering) system developed by Caldiera and Basili (Caldiera and Basili 1991), two aspects of their approach were unautomated. First, as each potentially-reusable component was packaged for reuse, an engineer with knowledge of the application domain in which the component was developed was required to analyze each component to determine the service that component could provide. Second, the Component Qualifier portion of the CARE system was an interactive assistance tool for a software engineer which provided a 'specifier' that supported the construction of a formal specification to be associated with each component. However, the Component Qualifier was unable to completely generate a formal specification of a component automatically, since it required human intervention in loop annotation.

Abd-El-Hafiz and Basili discussed some approaches that developed algorithmic techniques for finding the invariants or functions of specific simple classes of loops

(Abd-El-Hafiz and Basili 1996). Abd-El-Hafiz and Basili mentioned 'These algorithmic approaches analyze loops through the use of formal, semantically sound, and unambiguous notation. Although some of them provide guidelines on how to mechanically generate loop invariants or functions, no algorithmic techniques were actually used to implement automatic analysis systems.'

Due to these difficulties with an algorithmic approach, a heuristic, knowledge-based approach can be more automatable. One of the major goals of this research is to automate reusable component identification as much as possible.

The use of informal tokens in this approach goes beyond that of Biggerstaff (Biggerstaff et al. 1994). Biggerstaff's approach primarily concentrated on the simple matching of comment keywords. This approach not only uses additional information from identifier names, it also employs information extraction/natural language processing techniques.

Information extraction systems attempt to answer certain pre-defined questions, while ignoring extraneous information (Etzkorn et al. 1996). While information extraction systems are more commonly applied to texts such as newspaper and journal articles, many techniques derived from those systems are applicable to comment and identifier understanding.

Another difference between our approach and that of Biggerstaff is that it also employs structural information about the code, such as the class hierarchy, to guide the understanding process.

4 Comment and Identifier Understanding

If comments are considered as a sublanguage, that is, as a language forming a subset of natural language in terms of words allowed and grammar allowed, then the difficulty of parsing comments is greatly diminished. This allows the choice of a simpler natural language parser than would be required for a more generalized natural language parsing problem. It also allows for the employment of heuristics based on the common formats of comments and identifiers.

A sublanguage is a subset of natural language (Kittredge and Lehrberger 1982). It is not known how many sublanguages exist in a given language . Some sublanguages are the 'language of biophysics,' the 'language of car repair manuals,' and the 'language of naval transmissions.' In some cases in a particular sublanguage, the norms of usage may conflict with those of natural language. Sublanguage structures that do not conform to standard language are referred to as 'deviant.' One fairly common example of deviant language structures is the telegraphic style that is used in weather bulletins or Navy telegraphic messages (Fitzpatrick, Bachenko, and Hindle 1986; Lehrberger 1986).

In order to show that a particular corpus contains a sublanguage of natural language it is necessary to show that the language in the corpus is either grammatically restricted (a grammatical sublanguage) or restricted in semantic domain (a subject-matter sublanguage), or both. The grammatical restrictions in a sublanguage may involve commonality in the use of the imperative or the indicative, uniformity of tense, restriction of modality, and the deletion of articles (telegraphic style). The restriction in semantic domain means that the language in the corpus will contain some words and word senses based on the intended use of the corpus, whereas other words and word senses will typically not be used.

An initial examination of typical comments led to the following observations (Etzkorn and Davis 1994):

1) Comments are usually written in the present tense, with either indicative mood or imperative mood. For example, 'This routine reads the data' is present tense, indicative mood, whereas 'read the data' is present tense, imperative mood.

2) The set of verbs typically used for comments is much restricted over the set of all English verbs. Verbs often used in comments include: is, uses, provides, implements, accesses, prints, inputs, outputs, reads, writes, supplies, defines, retrieves, gets, etc. Verbs seldom used in comments include: smiles, frowns, laughs, rides, flies, jumps, sings, fights, electrocutes, falls, punishes, hires, fires, pats, throws, pitches, calms, etc.

The set of comment verbs, while still very large, is still in general much smaller than the set of natural language verbs. Also, since the domain for analysis will also be restricted, the selection of verbs to support becomes much smaller.

Note that comments often have a telegraphic quality—certain determiners are left out. For example, a typical comment would be 'Open file,' rather than 'Open the file.' This telegraphic quality can be considered a 'deviant' rule of grammar, that is common to a sublanguage.

An initial feasibility study of comments in three C++ graphical user interface packages, GINA[1] 37,429 lines of code, wxWindows[11] 37,439 lines of code, Watson (Watson 1993) 3,722 lines of code, was performed to determine whether comments can be considered to be a grammatical and subject-matter sublanguage of English. Later,

another study was performed that examined comments in C++ packages drawn from four different implementation areas:

- Real time packages: DOSThread[5] 1,361 lines of code, ISC[9] 3,126 lines of code, Serial[10] 784 lines of code.
- Text analysis packages: JPEG[7] 1,552 lines of code, String++[8] 3,126 lines of code, DOC++[13] 28,237 lines of code.
- Database packages: Combits[2] 13,850 lines of code, Quick Database[3] 12,432 lines of code.
- Mathematical packages: NEWMAT08[4] 17,112 lines of code, MFLOAT[6] 2,079 lines of code, BLITZ++[12] 32,698 lines of code.

In these studies, a degree of randomness was achieved by examining comments from files selected by directory order. In both these studies, the comment analysis was performed manually. In the first study of the C++ GUI packages, the first one hundred and eight comments in each package were examined (comments from .H and .CPP files were examined together. Typically this included comments from several different files. Comments that consisted of code were ignored, since it was felt that in this case the comment characters were performing a different duty—that of removing code from the compilation process rather than documenting the code.

In the study of the comments from the C++ packages drawn from the four implementation areas, comments from .H files and .CPP files were examined separately. Again, a degree of randomness was achieved by examining comments from files selected by directory order. The first 100 comments from the .H files in each C++ package were examined, and the first 100 comments from the .CPP files in each C++ package were

examined. Then the results were examined overall for .H and .CPP files in each implementation area. Finally, overall results for .H and .CPP files for all packages and implementation areas were examined

Comments may conveniently be divided into those which are in sentence form, and those which are not. When in sentence form, it was found that comments are usually in either present tense, simple past tense, or simple future tense, but most often present tense. A description of the common formats for sentence form comments is shown in Figure 2. When not in sentence form, it was found that comments still tended to have common formats. A description of the common formats for non-sentence form comments is shown in Figure 3. Note that mathematically intensive code would be more likely to contain mathematical formulas. Comments were also examined as to content. A description of the common content of comments is shown in Figure 4.

Figure 5 shows the results of an analysis of tense in sentence style comments, from the first study of three graphical user interface packages (Etzkorn and Davis 1997). Note that 57% of all comments are in sentence style. Of the sentence style comments, 82% are in present tense. Only 3% are in a non-standard format. Figure 6 shows the content of comments in the three graphical user interface packages studied. This shows that 51% of comments in the first study provided an operational description. Note that only 3% of all comments were not directly related to the immediate description of computer software.

Figure 7 shows the tense of comments in sentence form in .H files from the second study, while Figure 8 shows the tense of comments in sentence form in .CPP files in the

second study. Note that comments are overwhelmingly in present tense (greater than 75%).

From the above studies, it can be seen that comments typically are grammatically restricted when compared to natural language as a whole, in relation to the use of tenses, mood, and voice employed. It can therefore be concluded that comments can be treated as a grammatical sublanguage. Although the no standard format category was larger in the second study, which may indicate that other comment categories could be defined; still, it is clear that comments are quite grammatically restricted when compared to natural language as a whole.

Figure 6 shows the content/usage of comments in the first study of the three graphical user interface packages. In this study, 51% of comments provided an operational description of the code. Figures 9 and 10 show the content of comments in the second study. In this study, between 53% and 67% of comments provided an operational description of the code. Since the comments that provided a definition are nearly all in non-sentence format, this means that most of the comments in sentence form are intended to provide an operational description of the code.

Therefore, since comments are grammatically restricted when compared to natural language as a whole, comments can be considered to be a grammatical sublanguage of natural language. Also, since comments are used to describe the operation of computer software, and to define portions of computer software, comments can be treated as a subject-matter sublanguage related to the operation of computer software.

5 The PATRicia System

A tool that employs the natural language processing approach to program understanding discussed earlier has been implemented. This tool is called the **PATR**icia system, for **P**rogram **A**nalysis **T**ool for **R**euse. The portion of the **PATR**icia system that deals with the understanding approach is called **CHRiS**, for **C**onceptual **H**ierarchy for **R**euse including **S**emantics. Figure 11 is a level 2 data flow diagram of **CHRiS**. In addition to the approach described above, the **PATR**icia system also has sections that deal with the calculation of software quality metrics from various object-oriented metrics.

Although the overall program understanding approach is applicable to many object-oriented languages, **CHRiS** currently analyzes C++ code. C++ was chosen for the initial **CHRiS** prototype partly because there are large established bases of C++ code where the need for such a tool as **CHRiS** is obvious, and partly because C++ parsing tools were readily available.

5.1 PATRicia System—Description of Operation

CHRiS is a combination of shareware tools, new C and C++ code, and lex-generated parsers. The C++ parser employed is the Brown University C++ parser, which generates a parsable abstract syntax tree output in a file. The natural language parser is the Sleator and Temperley natural language parser (Sleator and Temperley 1991).

The Sleator and Temperley natural language parser is a word-based, link grammar parser, which has a power similar to that of a context-free grammar. However, most

common comment-style sentences can be parsed with the Sleator and Temperly parser, as well as a large variety of more complicated sentences. The Sleator and Temperly parser handles: noun-verb agreement, questions, imperatives, complex and irregular verbs, many types of nouns, past-or-present participles in noun phrases, commas, a variety of adjective types, prepositions, adverbs, relative clauses, possessives, coordinating conjunctions, and more (Sleator and Temperley 1991).

Prior to the selection of the Sleator and Temperley parser, a study of how comments were handled by the Sleator and Temperley parser was performed. For comments taken from 5 different C++ sources, the Sleator and Temperley parser correctly handled 93% of comments in the first parse. This high percentage of comments handled can be attributed to the fact that comments tend to possess restricted grammar and dictionary (part of the sublanguage aspect of comments) (Etzkorn et al. 1996; Etzkorn and Davis 1997).

This high percentage of comments handled in the first parse means that **CHRiS** can be quite fast and accurate in its analysis of comments—for most comments, the semantic analysis which follows the initial syntactic analysis will not result in a conflict between word usage according to the parse and word sense within the **CHRiS** knowledge base. Additionally, for another 4.82% of comments in this study, only minor errors, such as a single adverb modifying the wrong verb or a prepositional phrase that is attached to the wrong word, occurred in the first parse. These errors are not important since the **CHRiS** tool does not currently look at the prepositional phrase attachment provided by the parser; rather, prepositional phrase attachment is handled semantically within the inferencing engine of the **CHRiS** knowledge base. Also, there is very little adverbial usage in the **CHRiS** knowledge-base since adverbs qualify verbs, and this is generally not often

needed for a knowledge-base whose purpose is in examining computer software. Thus for nearly 98% of comments **CHRiS** is able to use the first parse.

5.2 Difficulties in Parsing Comments

Some difficulties in comment parsing arose since comments tend to have a telegraphic quality (some common connecting words such as 'the' are often missing). Some other difficulties in parsing due to the format of comments were also seen. It was possible to use some heuristics based on common forms of comments that were identified in the sublanguage study discussed earlier to make certain comments more parsable prior to the application of the Sleator and Temperley natural language parser. For example, a fairly common type of comment is the following:

Read_Data: Reads the data.

Or

Read_data--Reads the data.

Neither of these comments is acceptable as written to the natural language parser. Both of the above are handled by having the sentence replaced by the following sentence:

Read_data reads the data.

The software that implements this heuristic looks to see if a colon or dash is found after the first word in the sentence, then removes the colon or dash, and converts the first character in the following word to lower case. The Read_Data identifier itself will be understood separately, as part of a separate identifier understanding process (discussed below) (Etzkorn and Davis 1997).

Another heuristic relates to sentences of two words. First the natural language parser is applied to the sentence. If the sentence is rejected, then the determiner 'the' is inserted between the two words in the sentence, and the natural language parser is again applied. This is identical to a heuristic that is used in the identifier understanding process. One other heuristic employed is placing the word 'This' in a sentence that begins with a plural verb, and uncapitalizing the verb. ('Reads the data.' Thus becomes 'This reads the data.') This is determined in a very simple manner, e.g., if the original sentence is rejected by the natural language parser, then the software looks at the sentence to see if the first word ends with an 's.' If this is so, the word 'This' is added to the beginning of the sentence, and the natural language parser is again applied to the sentence.

Other heuristics involve the removal of parenthetical information from a comment before the application of the natural language parser, and the appending of each comment sentence with a period character before the application of the natural language parser. Many comments in computer software do not have a sentence terminated with a period. Also, many comment sentences do not begin with a capital letter, and ensuring that they do is another heuristic.

This leads to another comment processing difficulty, which is the difference between comments that occur on a single line, and comments that continue between several lines. Consider a C style comment of the following type:

> /* The following routine extracts sentences from the header buffer and sends them to the natural language parser. If parsable, additional information about the usage of keywords is included in the facts sent to CLIPS. Certain types of sentence based heuristics are handled here. */

In this case, where a sentence is extended over multiple lines, each sentence must be extracted separately (a sentence here is a sequence of characters terminated by a period), and sent separately to the natural language parser.

A harder challenge comes from C++ style comments that extend across multiple lines. For example,

```
//We begin the
//sentence on one line and
//complete it on another. Worse
//yet, we have two sentences in
//the same manner.
```

This type of sentence is handled by a preprocessor that is applied to the software prior to the **CHRiS** tool being run on the software. C++ style comments that appear sequentially, on lines that did not include actual code, have the C++ comment characters removed, and a C style comment character added at the beginning and at the end of a sequence of comments. They can then be processed by **CHRiS** in the same manner as the C style comments discussed above.

5.3 Syntactically Tagging Identifiers

Empirical information about typical identifier formats is used by **CHRiS** to correctly syntactically tag subkeywords extracted from identifiers (Etzkorn and Davis 1997). For example, a typical member function name would be OpenFile. A typical member variable name would be FileName. A study of identifiers in C++ code showed that member function names tend to be verb related, and can often be put in sentence form, whereas variable names tend to consist of an adjective or adjectives followed by a

noun. The subkeywords of an identifier can usually be extracted by the use of capitalization information and underscores.

5.4 Understanding Process

Traverse Graph guides the understanding process through the class hierarchy graph (base classes are understood first, followed by derived classes). Parse Abstract Syntax Tree primarily extracts class member variable identifiers and member function identifiers from the abstract syntax tree file that was provided by the C++ parser. Facts are built from parsed comment sentences, and from the syntactically-tagged identifier keywords, and are presented to the **CHRiS** knowledge base.

The knowledge-base, which is a form of semantic net based on conceptual graphs, and the inferencing engine are implemented using the CLIPS version 6.0 expert system shell, as objects and messages.

6 The CHRiS Knowledge-base

The **CHRiS** knowledge-base is a weighted, hierarchical semantic net. Concepts in the semantic net are stored as conceptual graphs (Sowa 1984) or as part of a conceptual graph. Figure 12 illustrates the **CHRiS** knowledge-base.

The semantic net has an interface layer of syntactically-tagged keywords. A keyword in the interface layer that is an adjective, for example, is a different keyword than a keyword which is a noun. Parsed comment sentences, and syntactically tagged keywords are compared by the CLIPS inferencing engine to the interface layer of the semantic net.

A form of spreading activation is then employed within the semantic net. Identified keywords in the interface layer 'fire' concepts in individual conceptual graphs. Those concepts, and the entire individual conceptual graphs, then fire other, higher level concepts.

The semantic net is itself object-oriented; it is implemented in CLIPS version 6.0 objects and messages. Inferencing within the semantic net is performed entirely by messages between objects. The 'firing' of concepts is performed by sending messages between CLIPS objects.

Keywords in the interface layer are stored in lower case. Capitalization information is extracted from keywords and used to break them into subkeywords prior to the presentation of these syntactically-tagged subkeywords to the interface layer.

Interior concepts (non-interface nodes), and interior conceptual graphs have associated English text defining the operation of that concept within the current domain. This text gives a high-level definition of the concept, without as much low-level detail as is found in the code itself. Both interior concepts and interface nodes allow for information regarding the location (the name of the class or classes) where that concept was identified.

7 CHRiS Reports

CHRiS provides multiple output reports. These include:

1) a concepts-recognized report, along with English text descriptions of the concepts found in each class.
2) An English language description of the functionalities found in each class.
3) a list of recognized keywords
4) a full report listing all nodes fired in the knowledge-base, that can serve as a simple explanation feature, since it is possible to trace the firing path from the interface layer to any concept or conceptual graph in question. Future plans for this report include a facility to reduce the scope of the report to designated nodes and their related nodes only.

The natural language generation is performed very simply. It is based on the low-level conceptual graphs in the semantic net. If an OBJ (object) conceptual relation connects from concept A to concept B, then concept A is considered to be a verb, and concept B a noun that is the object of the concept A verb. In the case of a single AGNT (agent) link, the verb is normally considered to be singular (defaults to singular). If Concept A is connected to two or more concepts with AGNT (agent) conceptual relations, then those concepts are considered to form a compound subject, and the verb becomes plural. The functionality report is generated in present tense only.

Each noun concept (concepts in a conceptual graph that are pointed to by an AGNT (agent) or OBJ (object) conceptual relation) has a determiner generated. Usually, when the determiner is generated as part of a sentence that also contains a verb, then the determiner generated is indefinite, either ‘a’ or ‘an,’ and the selection of ‘a’ versus ‘an’ is based on whether the following concept begins with a consonant or a vowel.

The report produced using natural language generation is presented as a list of the functionalities provided by a class. Figure 13 is an example of part of a **CHRiS** natural language report. The production of this natural language report was completely automated, using the natural language generation facilities within **CHRiS**.

8 CHRiS Results

The domain of graphical user interfaces was chosen as an initial test domain. **CHRiS** has now been applied to class hierarchies drawn from three different C++ software packages, the GINA package (Backer et al. 1991), the Watson XWindows package (Watson 1993), and the wxWindows package (Smart 1994).

A set of natural language metrics adapted with some changes from the area of information extraction were employed to measure the operation of **CHRiS** (Etzkorn and Davis 1997). This includes a metric for recall, which is a measure of the completeness of concept generation, a metric for precision, which is a measure of the accuracy of concept generation, and a metric for overgeneration, which is a measure of spurious generation of concepts.

An experiment to evaluate the operation of **CHRiS** was performed. Several C++ and GUI experts, most with advanced degrees in either computer science or computer engineering and all with several years of industrial experience (5 to 17 years) were used. These experts identified important concepts that occurred in classes from three different GUI packages. Then they compared a merged version of these concepts to the concepts identified by **CHRiS**. The measure of recall over the three packages examined varied

from 70% to 90%, precision varied from 70% to nearly 85%, while overgeneration over the three packages was always under 10%.

The approach followed by **CHRiS**, which is part of the **PATR**icia system, has been shown to be a workable solution to the problem of automatically determining the functionality of potentially reusable components in existing object-oriented software. This determination of functionality is used to identify a software component that is useful and thus potentially reusable in a particular domain of interest. A software component that can have its functionality substantially identified within the current domain is considered to be useful in that domain.

The remainder of the **PATR**icia system provides a quantitative analysis of quality attributes related to the reusability of a software component. Thus the **PATR**icia system as a whole provides a complete determination of the reusability of a software component within a particular domain of interest. A future area of research related to **CHRiS** would be to use identified concepts within the **CHRiS** knowledge-base to automatically classify those software components which were previously determined to be reusable for insertion into a software component library.

References

Abd-El-Hafiz, S.K., and Basili, V.R. 1996. A knowledge-based approach to the analysis of loops. *IEEE Transactions on Software Engineering* 22: 339-60.

Basili, V.R., and Abd-El-Hafiz, S.K. 1991. Packaging reusable components: the specification of programs. Technical Report CS-TR-2957, Department of Computer Science, University of Maryland at College Park.

Biggerstaff, T.J., Mitbander, B.G, and Webster, D.E. 1994. Program understanding and the concept assignment problem. *Communications of the ACM* 37: 72-82.

Caldiera, G., and Basili, V.R. 1991. Identifying and qualifying reusable software components. *IEEE Computer* 24: 61-70.

Etzkorn, L.H., and Davis, C.G. 1994. A documentation-related approach to object-oriented program understanding. In *Proceedings of the IEEE Third Workshop on Program Comprehension*, pp. 39-45. Los Alamitos: IEEE Computer Society Press.

Etzkorn, L.H., Davis, C.G., Bowen, L.L., Etzkorn, D.B., Lewis, L.W., Vinz, B.L., and Wolf, J.C. 1996. A knowledge-based approach to object-oriented legacy code reuse. In *Proceedings of the Second International Conference on Engineering of Complex Computer Systems*. pp. 493-6. Los Alamitos: IEEE Computer Society Press.

Etzkorn, L.H., and Davis, C.G. 1996. Automated object-oriented reusable component identification. *Knowledge-Based Systems* 9: 517-24.

Etzkorn, Letha H., and Davis, Carl G. 1997. Automatically identifying reusable OO legacy code. *IEEE Computer* 30: 66-71.

Fitzpatrick, E., Bachenko, J., and Hindle, D. 1986. The status of telegraphic sublanguages. In R. Grishman and R. Kittredge (eds.), *Analyzing Language in Restricted Domains*, pp.39-51. New Jersey: Lawrence Erlbaum and Associates.

Hooper, J.W., and Chester, R.O. 1991. *Software Reuse Guidelines and Methods*. New York: Plenum Press.

Kittredge, R., and Lehrberger, J. 1982. *Sublanguage: Studies of Language in Restricted Domains*. New York: Walter de Gruyter.

Lehrberger, J. 1986. Sublanguage analysis. In R. Grishman and R. Kittredge (eds.), *Analyzing Language in Restricted Domains*, pp. 19-38. New Jersey: Lawrence Erlbaum and Associates.

Ning, J.Q. 1989. *A knowledge based approach to automatic program analysis*. Doctoral dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.

Ning, J., Engberts, A., and Kozaczynski, W. 1994. Automated support for legacy code understanding. *Communications of the ACM* 37: 39-51.

Proceedings of the Third Message Understanding Conference (MUC-3). San Mateo: Morgan Kaufmann.

Rich, C. and Wills, L.M. 1990. Recognizing a program's design: a graph-parsing approach. *IEEE Software* 7: 82-89.

Sleator, D. and Temperley, D. 1991. Parsing English with a link grammar. Technical Report CMU-CS-91-196, Department of Computer Science, Carnegie-Mellon University.

Sowa, J.F. 1984. *Conceptual Structures*. Reading: Addison-Wesley.

Watson, M. 1993. *Portable GUI Development with C++*. New York: McGraw-Hill.

Electronic Source Material

The author has attempted to provide full reference-style details for the software packages listed. Where the details are not made available it is to protect the privacy of the individuals concerned.

1. Backer, Andreas, Genau, Andreas, and Sohlenkamp, Markus. 1991. The generic interactive application for C++ and OSF/MOTIF user manual and tutorial, version 2.0. Human-Computer Interaction Research Division, Institute for Applied Information Technology, German National Research Center for Computer Science. Anonymous ftp at ftp.gmd.de, directory gmd/ginaplus.

2. Combits. 1997. The CS libraries: a database kit. P.O. Box 3303, 2280 GH Rijswijk,The Netherlands. http://www.combits.nl.

3. Curtis, David. 1996. Quick database. Joint Academic Department of Psychological Medicine, London Hospital Medical College. dcurtis@hgmp.mrc.ac.uk.

4. Davies, R. 1995. NEWMAT08: a matrix library in C++. 16 Gloucester Street, Wilton, Wellington, New Zealand. robert.davies@vuw.ac.nz.

5. English, J. 1993. Class DOSThread: a base class for multithreaded DOS programs. Department of Computing, University of Brighton. je@unix.brighton.ac.uk.

6. Kaufmann, Friedrich, and Meuller, Walter. 1995. MFLOAT 2.0. Technical University of Graz. fkauf@fstgds06.tu.graz.ac.at.

7. Lane, Tom, Gladstone, Phil, Ortiz, Luis, Boucher, Jim, Crocker, Lee, Minguillon, Julian, Phillips, George, Rossi, Davide, and Weijers, Ge'. 1996. JPEG. Independent JPEG Group. ftp.uu.net, jpeg-info@uunet.uu.net.

8. Moreland, C. 1994. String++ V. 3.1. 4314 Filmore Road, Greensboro, NC 27409. carl.moreland@analog.com, http://oak.oakland.edu.

9. Laor, Ofer 1992. ISC: Interrupt Service Class V.3.66. 27 KKL St., Kiriat Bialik, Israel, S5919325@techst02.technion.ac.i1. S5919325@techst02.technion.ac.il.

10. 1992. Serial. http://www.rt66.com/ftp/pc/windows.

11. Smart, Julian. 1994. Reference manual for wxWindows 1.60: a portable C++ GUI toolkit. Artificial Intelligence Applications Institute, University of Edinburgh. http://www.aia..ed.ac.uk/~jacs/wx.

12. Veldhuizen, T. 1997. BLITZ++ users manual. Department of Computer Science, University of Waterloo, Canada., tveldhui@seurat.uwaterloo.ca, http://monet.uwaterloo.canada/blitz.

13. Wunderling, Roland and Zoeckler, Malte. 1996. DOC++: a C++ documentation system for LaTeX and HTML. Electronic Library for Mathematical Software, Berlin. doc++@zib.de. http: //www.zib.de/ Visual/software/ doc++/ index.html.

```
/* This class provides serial, polled I/O drivers
*/

class com {
     unsigned num_data_bits;    // Number of data
bits

      unsigned parity;         // none=0, even=1,
odd = 2

  public:

      unsigned  in_data();      // Serial I/O read

   void out_data( unsigned val ); //  serial I/O
write
};
```

Present Tense

- Indicative mood, active voice. e.g., 'This routine reads the data.'
- Indicative mood, active voice, missing subject. e.g.., 'Reads the data.'
- Imperative mood, active voice. e.g., 'Read the data.'
- Indicative mood, passive voice. e.g.., 'This is done by reading the data.'
- Indicative mood, passive voice, missing subject. e.g., 'Done by reading the data.'

Past Tense

- Indicative mood, either active or passive voice, occasional missing subject. e.g., 'This routine opened the file.' 'Opened the file.'

Future Tense

- Indicative mood, either active or passive voice, occasional missing subject. e.g., 'This routine will open the file.' 'Will open the file.'

Definition Format

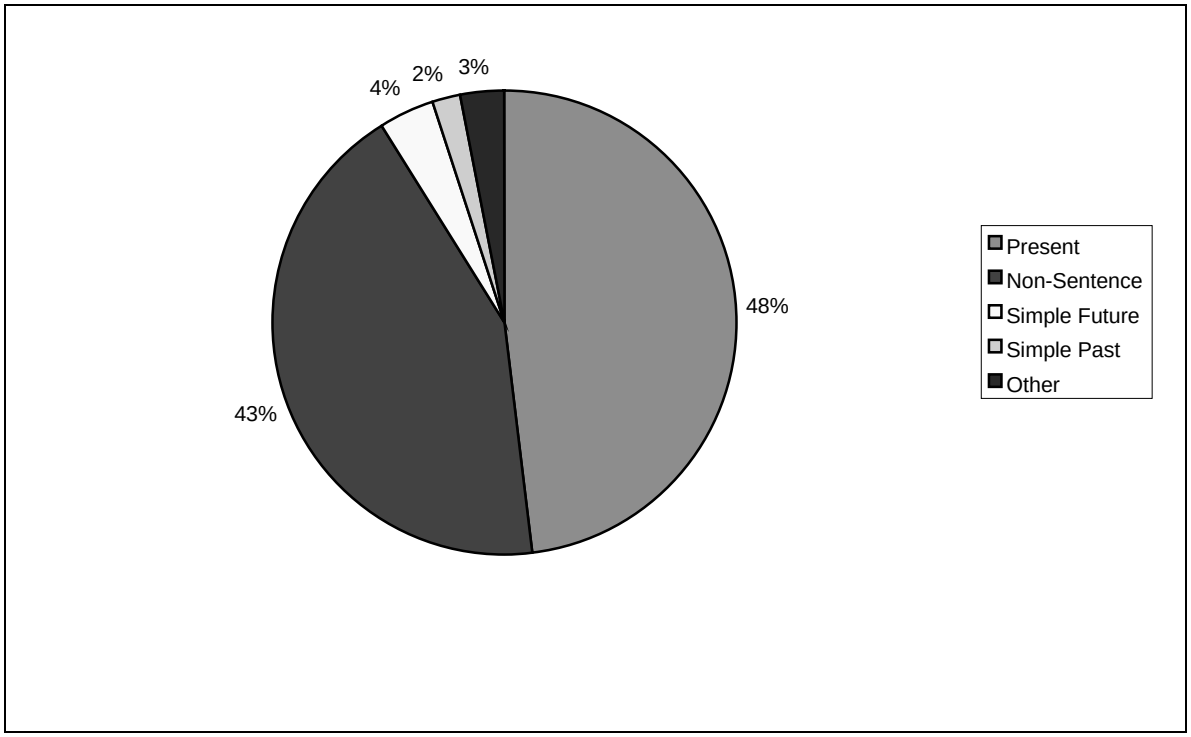
- Itemname—Definition. e.g., 'MaxLength—Maximum CFG Depth.'
- Definition. e.g., 'Maximum CFG Depth.'
- Unattached prepositional phrase. e.g., 'To support scrolling text.'
- Value Definitions. e.g., '0= not selected, 1 = is selected.'
- Mathematical Formulas. Can be Boolean expressions.

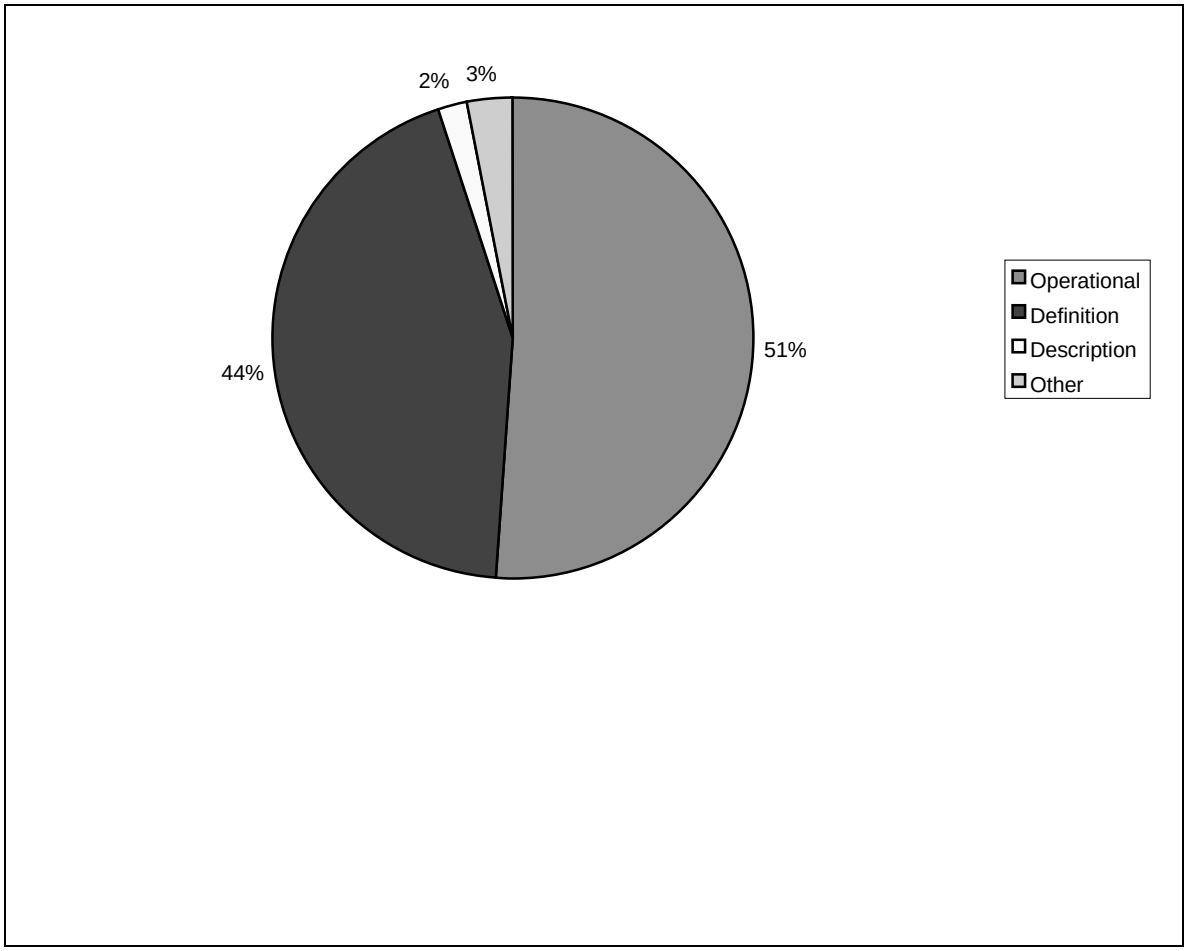
Operational Description. e.g., 'This routine reads the data. Then it opens the file.'

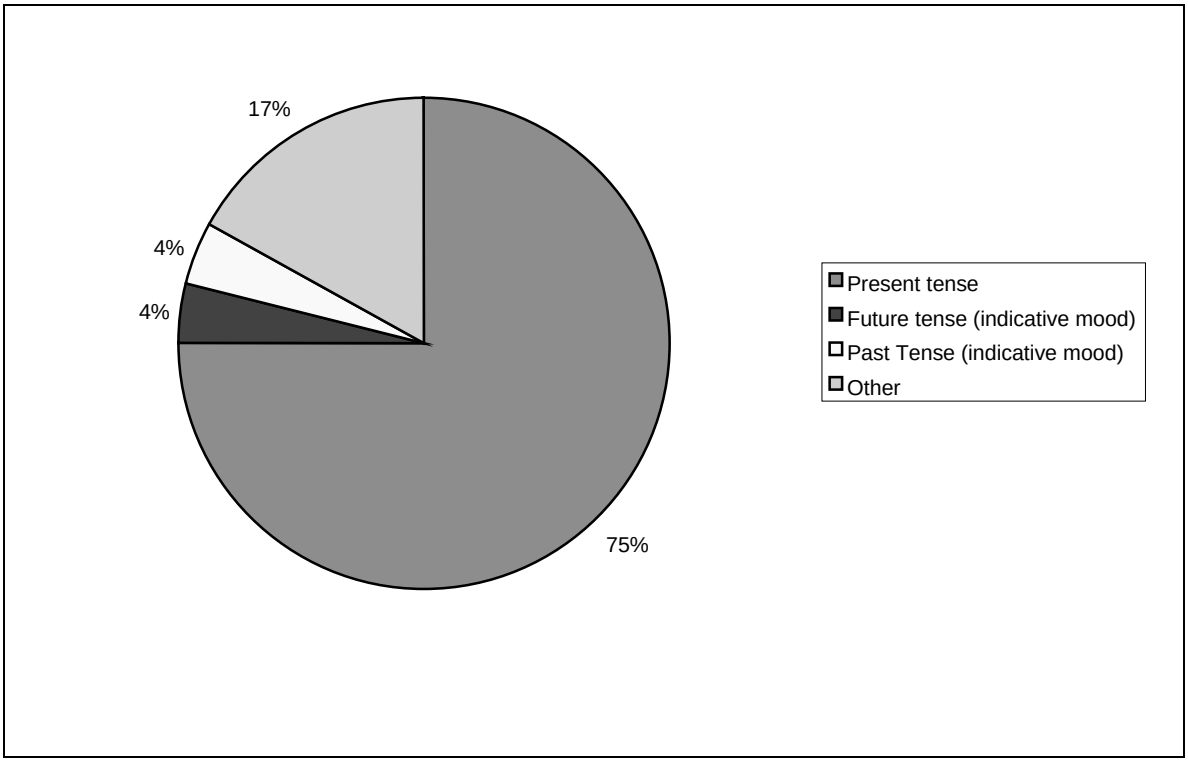
Definition. e.g, 'General Matrix—rectangular matrix class.'

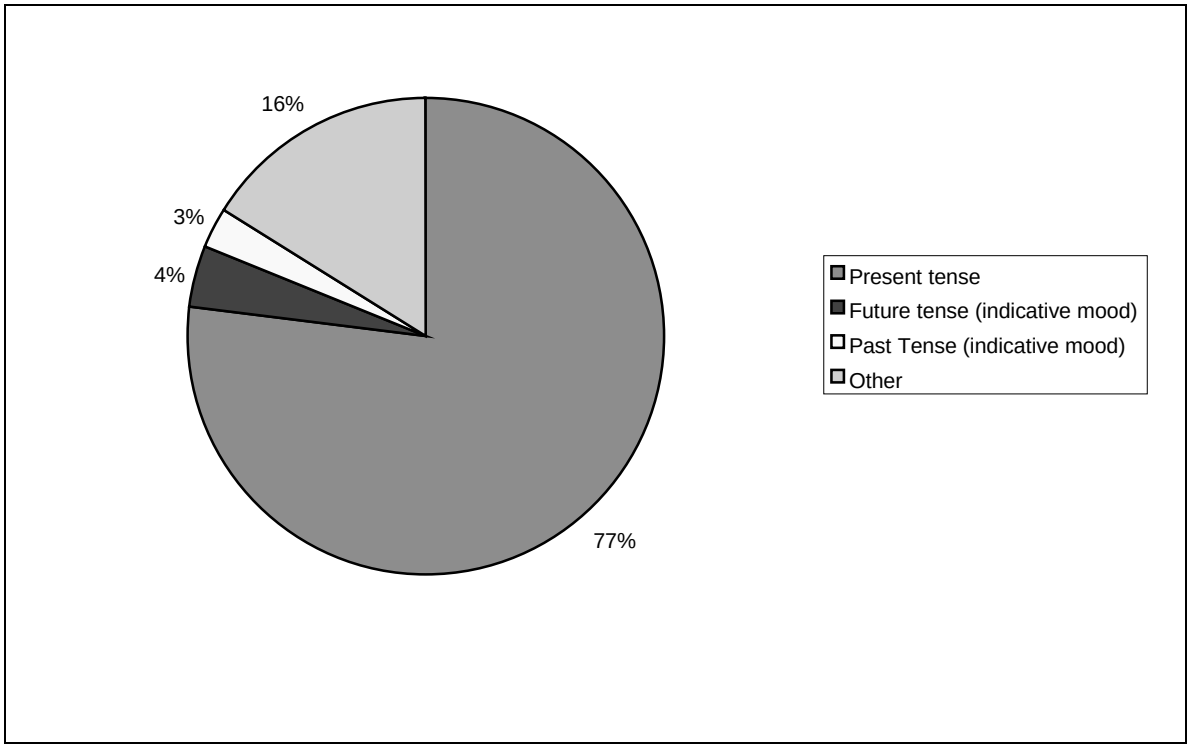
Description of Definition. e.g., 'This defines a NIL value for a list.'

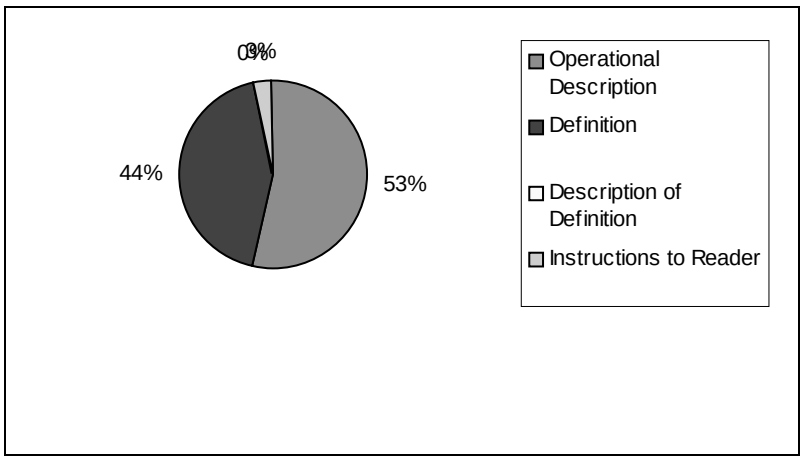
Instructions to reader. e.g., 'See the header at the top of the file.'

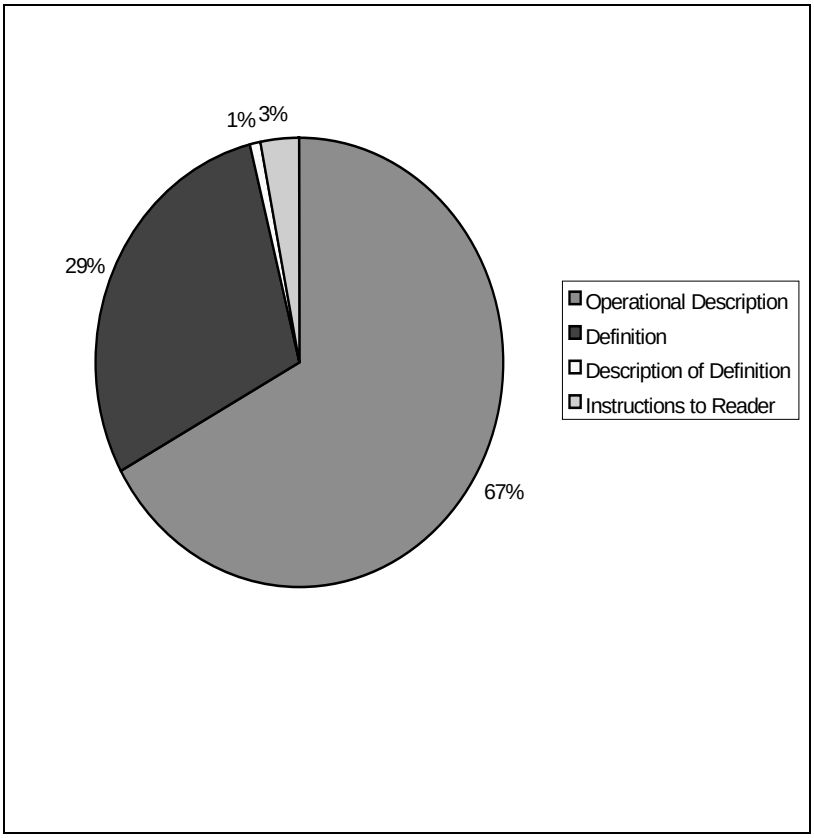

4%
2%
3%
48%
43%
Present
Non-Sentence
Simple Future
Simple Past
Other

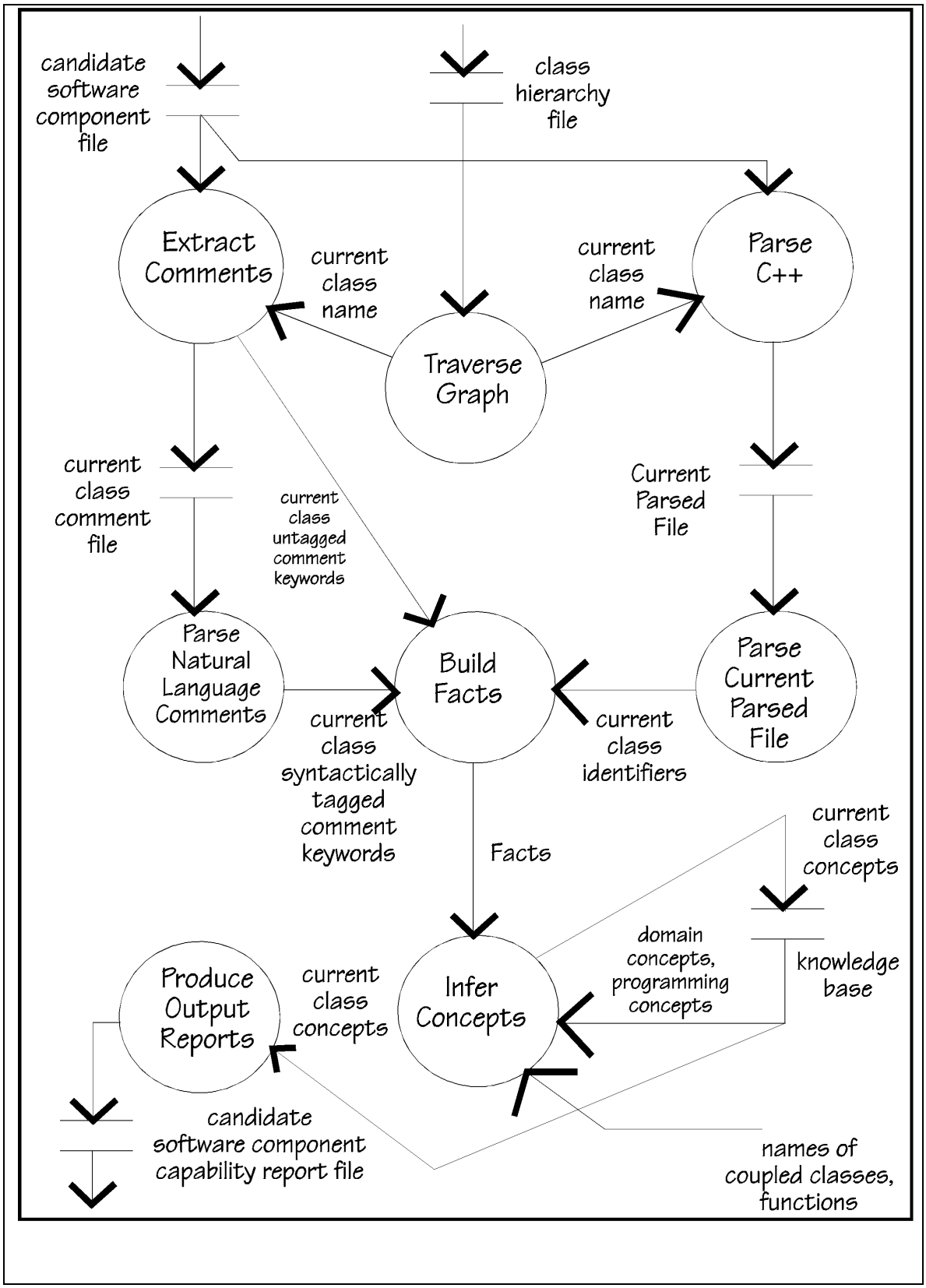

2%
3%
51%
44%
Operational
Definition
Description
Other

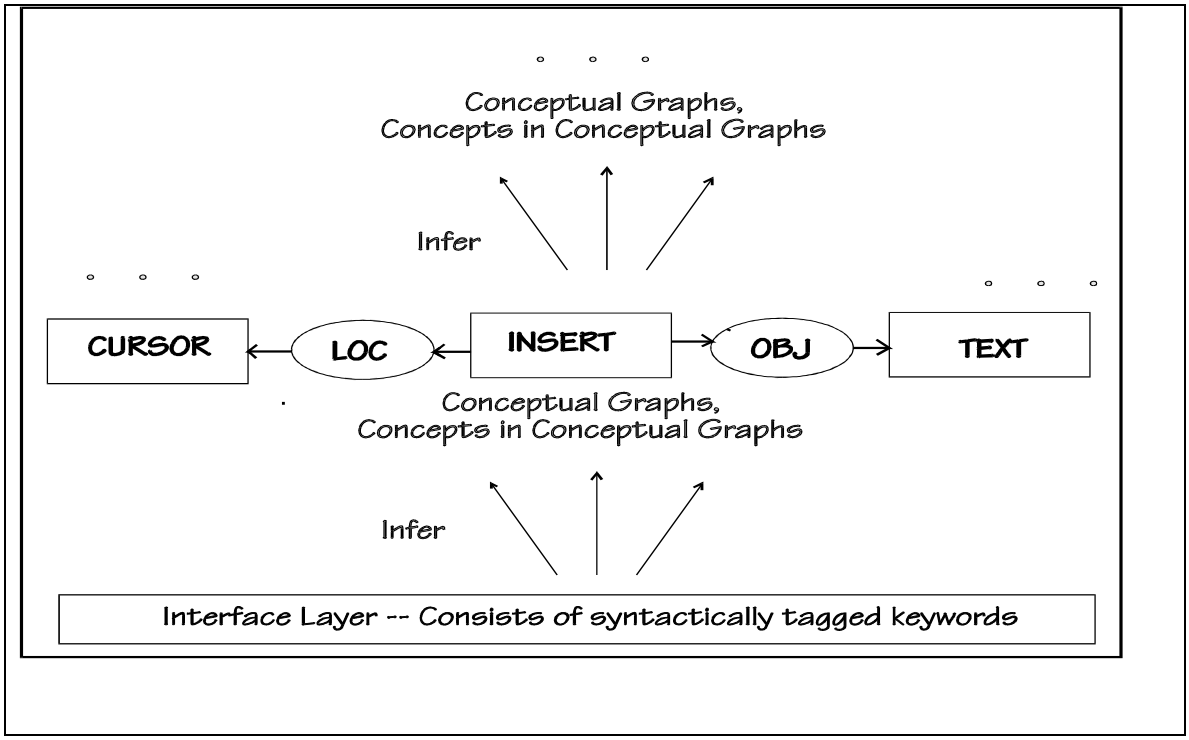

17%
4%
4%
75%
Present tense
Future tense (indicative mood)
Past Tense (indicative mood)
Other


16%
3%
4%
77%
Present tense
Future tense (indicative mood)
Past Tense (indicative mood)
Other


0%
3%
44%
53%
Operational Description
Definition
Description of Definition
Instructions to Reader


1%
3%
29%
67%
Operational Description
Definition
Description of Definition
Instructions to Reader


candidate software component file
class hierarchy file
Extract Comments
current class name
Traverse Graph
current class name
Parse C++
current class comment file
current class untagged comment keywords
Current Parsed File
Parse Natural Language Comments
Build Facts
Parse Current Parsed File
current class syntactically tagged comment keywords
current class identifiers
Facts
current class concepts
domain concepts, programming concepts
knowledge base
Produce Output Reports
current class concepts
Infer Concepts
candidate software component capability report file
names of coupled classes, functions


Conceptual Graphs,
Concepts in Conceptual Graphs
Infer
CURSOR
LOC
INSERT
OBJ
TEXT
Conceptual Graphs,
Concepts in Conceptual Graphs
Infer
Interface Layer -- Consists of syntactically tagged keywords

Class wxbItem: graphs

-- contains the concept ‘text’ that is described by a font descriptor and a height descriptor and a width descriptor.

-- minimizes a window.
HINT: A button that can be described by a color descriptor and a left descriptor can minimize a window.

-- focuses an <object>.
HINT: It is possible to focus an area.