Semantic Decomposition by Covering

Shankar B. Sripadham

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Dr. Walling R. Cyre, Chair
Dr. Hugh Van Landingham
Dr. Robert Broadwater

August 4, 2000
Blacksburg, Virginia

Key Words:
Covering, Sentence interpretation,
Branch and bound.

Semantic Decomposition by Covering

Shankar B. Sripadham

ABSTRACT

This thesis describes the implementation of a covering algorithm for semantic decomposition of sentences of technical patents. This research complements the ASPIN project that has a long term goal of providing an automated system for digital system synthesis from patents.

In order to develop a prototype of the system explained in a patent, a natural language processor (sentence-interpreter) is required. These systems typically attempt to interpret a sentence by syntactic analysis (parsing) followed by semantic analysis. Quite often, the technical narrative contains grammatical errors, incomplete sentences, anaphoric references and typological errors that can cause the grammatical parse to fail. In such situations, an alternate method that uses a repository of pre-compiled, simple sentences (called frames) to analyze the sentences of the patent can be a useful back up. By semantically decomposing the sentences of patents to a set of frames whose meanings are fully understood, the meaning of the patent sentences can be interpreted.

This thesis deals with the semantic decomposition of sentences using a branch and bound covering algorithm. The algorithm is implemented in C++. A number of experiments were conducted to evaluate the performance of this algorithm. The covering algorithm uses a standard branch and bound algorithm to semantically decompose sentences. The algorithm is fast, flexible and can provide good (100 % coverage for some sentences) coverage results. The system covered 67.68 % of the sentence tokens using 3459 frames in the repository. 54.25% of the frames identified by the system in covers for sentences, were found to be semantically correct. The experiments suggest that the performance of the system can be improved by increasing the number of frames in the repository.

Acknowledgement

I would like to thank Dr. Walling Cyre for making this thesis possible and for all the guidance he provided. I take this opportunity to thank Dr. Van Landingham and Dr. Broadwater for serving as members of my committee.

I am also thankful to my parents and the rest of my family for all the support they provided me. Lastly, I wish to thank all my friends and all others who have directly or indirectly helped me in my efforts.

Table of Contents

List of Figures

Chapter 1. INTRODUCTION

1.1. Motivation

At the turn of a new millennium, the rapidly advancing world of digital systems is all set to leap frog into a whole new world of innovations, designs, prototyping, synthesis and system development. Efforts are underway to automate the product cycle, starting at the design and ending with the product development and testing, with the ultimate goal of reducing product cycle time. Although automation does reduce the overall cycle time, other factors such as the time the designers, modelers and engineers spend in understanding the documentation given to them can increase the product cycle time. These documents are often in the form of technical reports, patents or specification, written in natural language. An automated process to interpret these technical narratives is a critical need for reducing product cycle time.

There exist many tools for automating natural language interpretation. Conventional systems, such as the ASPIN system developed here at the ADRG lab of Virginia Tech, use a parser and rely heavily on the grammatical analysis of the sentences of the technical narrative for its interpretation. The writer of the narrative is not always constrained to write sentences that satisfy the limited ASPIN grammar and dictionary. In such cases the parser fails, warranting the need for an alternate approach to interpretation, that relies less heavily on the grammar. Such an alternative may not be a replacement for the conventional approach, but may be a good complement for the same.

1.1. System Overview

This thesis provides an alternative approach to semantically decompose sentences. Sentences are decomposed to semantic entities called frames. A *frame* is an abstract canonical sentence whose semantics is fully understood. The frames are stored in a repository. A sentence is decomposed into a set of frames that best *cover* the sentence. However the sentences are not directly used to generate the cover. Instead, a sentence is first reduced to a sequence of head words and their connectives. Such a sequence is called a *skeleton* of the sentence. A skeleton contains the core meaning of the sentence. The overview of the system used to generate a cover is presented in this section.

The architecture of the system implemented in this thesis is as shown in Figure 1. All the components of the architecture are software modules, which may be objects or methods of a class. This author has not implemented some of the blocks in the diagram. These blocks have been borrowed from previous research works. These blocks are shown in lighter color in Figure 1. They have been shown here merely to complete the picture of the various stages of processing.

The input sentences are read from an input file and passed through a partial parser called *Chunker*. *Chunker* generates an abstraction of the sentence by replacing phrases in a sentence by their main words. The abstraction of the sentence is then passed through an *Abstraction Processor*.

The *Abstraction Processor* processes the abstraction to generate a skeleton of the sentence. The processing is carried out in three stages. Each stage uses a knowledge base to carry out the process. These knowledge bases are shown as *bins* in Figure 1. In the first stage, the "noisy" tokens are eliminated from the abstraction. Such tokens include punctuation characters, conjunctions and abbreviations. Since such tokens do not contribute to the core meaning of the sentence, they are called "noisy" tokens.

In the next stage of processing, the nouns and the verbs of the abstraction are normalized to their root morphology. For instance, a noun such as "**processors**" will be normalized to its root morphology "**processor**." Likewise, verbs such as "**written**," "**wrote**,"

SYSTEM ARCHITECTURE

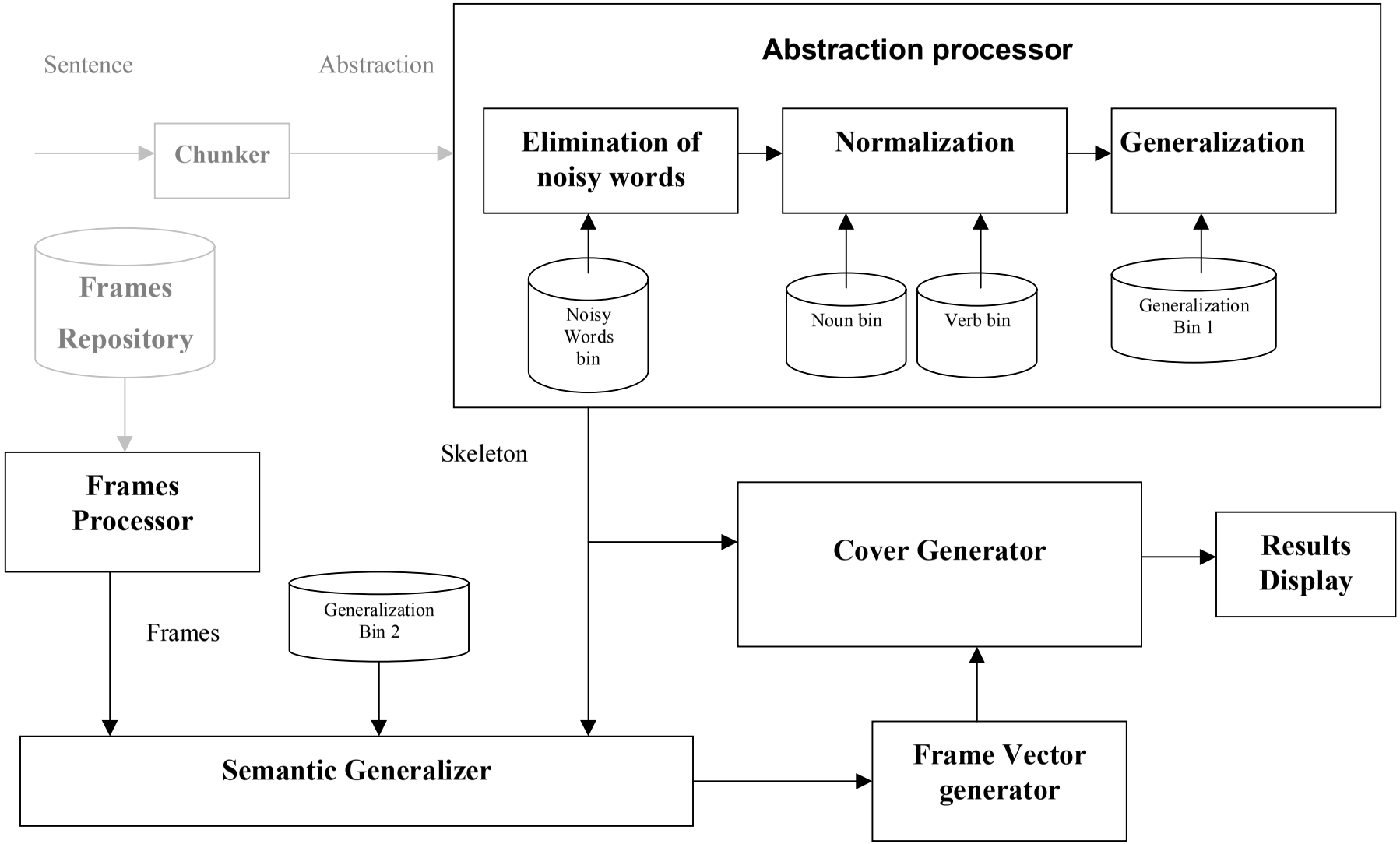

Figure 1 System Architecture.

"**writes**" and "**writing**" will be normalized to the root morphology "**write**." Normalization is important because the repository of frames that is used to cover the skeleton contains words in their root form.

In the third stage of processing, the words of the abstraction are generalized to their higher-level concept types as dictated by a **type hierarchy**. A type hierarchy defines the relationship among various semantic concept types. Most semantic concepts (such as "**microprocessor**") can be viewed as a subtype of a more general concept type (such as "**processor**"). This relationship can be used to generalize the subtype concepts to their corresponding supertype. The advantage of using such semantic generalization is that the repository can have fewer frames containing supertype concepts instead of a large number of frames containing subtype concepts. The semantic generalization can then be used to generalize the subtype tokens in the skeleton to their corresponding supertypes to effect a cover. Semantic generalization thus improves the overall coverage results. The generation of a skeleton from a typical sentence is shown in Figure 2.

The skeleton that is generated is covered using the frames from a repository. The frames are first processed using a *Frames Processor* that generates an alternate representation of the frame. This alternate frame form facilitates easier processing. The processor then filters all the frames whose semantic content correlates poorly with the semantics of the skeleton. In order to improve covering results, the tokens of the frames are also subject to semantic generalization using a *Semantic Generalizer*. The *Semantic Generalizer* uses a type hierarchy that is more elaborate than the one used for skeleton generation to generalize the tokens of the frames.

While the frames and skeleton can be directly used to generate the semantic cover for the skeleton, such an approach would be memory inefficient. Hence, the frames are mapped to a bit vector representation that uses less memory. The *Frame Vector Generator* is used to generate these vectors.

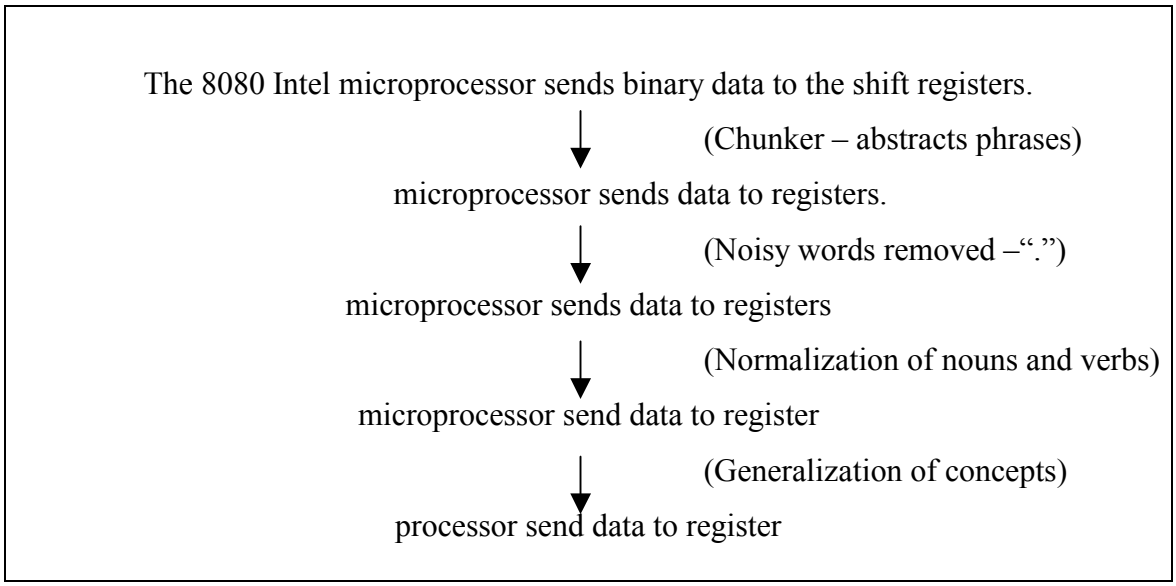

Figure 2 Evolution of a skeleton.

The skeleton and the frame vectors are given to the *Cover Generator*. The *Cover Generator* uses a covering algorithm to generate the cover. The covering algorithm used in this thesis is a branch and bound algorithm. The output of the *Cover Generator* is a set of frames that best cover the skeleton. Such a set is a semantic decomposition of the sentence.

1.2. Contributions

This section lists the research contributions made by this author.

Frames Processor: The *Frames Processor* reads frames from an input file and stores them as a linked list. The frames read in from the input file are of the form *subj verb obj, list of cases.* These frames are processed and stored in a linked list in the form v*erb(subject subj , object obj, list of cases).* The *Frames Processor* also rejects frames that have poor semantic correlation with the skeleton.

Frame Vector Generator: For a given sentence skeleton, the *Frame Vector Generator* generates a vector for each frame in the frames input file. The *Frame Vector generator* is implemented as a class, CFrameVector.

Semantic Generalizer: In order to improve covering results, the tokens of the frame are semantically generalized using type hierarchy. The *Semantic Generalizer* compares each skeleton token with all the frame tokens. If the skeleton token is a supertype concept of any of the frame tokens, then the corresponding frame token is generalized so that it matches the skeleton token type. The *Semantic Generalizer* is implemented as a function MultiHierarchy().

Abstraction Processor: The *Abstraction Processor* converts the abstract sentence generated by the *Chunker* into a skeleton. The abstract sentence is processed in three stages. In the first stage the noisy words are eliminated. A file containing a list of noisy tokens identifies the noisy tokens in the skeleton. In the next stage, the *Abstraction Processor* normalizes the verbs to their root form and the nouns to their singular form. These processes are also carried out as dictated by a verbs file and a nouns file respectively. In the last stage, any token that is a subtype of other concept types in the type hierarchy, is replaced by its corresponding supertype. A file containing the subtype - supertype relation is used for this purpose. The *Abstraction Processor* is implemented as a class, CHierarchy.

Cover Detector: The *Cover Detector* forms the core of this thesis. It finds a covering solution for a skeleton using the frames from the repository. The covering algorithm uses a branch and bound algorithm. The covering algorithm operates on a set of frame vectors called the table. The *Cover Detector* is implemented as a class, CTable. The branch and bound algorithm is implemented as a member function of this class.

Results Display: The *Results Display* is an html file that contains the output generated by the covering algorithm. The output is in the form of a skeleton followed by the set of covering frames. Color-coding is used to distinguish the covered tokens from the tokens that are not covered. In the skeleton, the color “Red” is used to identify the tokens that are not covered. The color “Blue” is used to identify the tokens that are covered. In the frames, the color “Blue” is used to identify the tokens that cover a skeleton token and the color “Black” is used to identify tokens that do not cover any of the skeleton tokens.

Frames File: In the automatically generated frames file, “frames2.txt”, more than 80% of the frames required cleaning. The frames contained numerals that identified nouns (devices, components etc.) in the place of subjects and objects. In order to clean the frames, the numerals had to be replaced by the corresponding nouns that they identified. The patents in which the numeric conventions were defined were used for this purpose. The numerals were replaced using simple C++ programs and a spreadsheet.

1.3. Document Organization

This section describes the organization of this thesis.

Chapter1 formally introduces the thesis. The motivation and the scope of the thesis are presented in this chapter. This chapter also presents the overview of the system developed in this thesis.

Chapter 2 summarizes the background and some other works and their relationship to this project. The reader is introduced to the notion of frames that have been used to cover the sentence abstractions. Works relevant to this thesis, are also presented in this section.

Chapter 3 deals with the theoretical aspects of this thesis. This chapter explains the algorithm used in this thesis from a theoretical perspective. In chapter 4, we talk about the actual implementation of the system. Chapter 5 presents the results of this thesis. In chapter 6, we briefly explore the limitations of the system and the scope for further improvement of this system. This chapter formally concludes the thesis.

Chapter 2. RELEVANT WORK

2.1. Introduction

This chapter introduces the background material vital for the understanding of this thesis. In section 2.2 the idea of frames is introduced. Sections 2.3 through 2.5 discuss some of the approaches that have been proposed for Boolean covering problems. In section 2.6, a method to disambiguate sentence words in order to interpret a sentence is presented. Section 2.7 discusses an approach to information extraction using template-filling.

2.2. Case Frames

Charles J. Fillmore [Fillmore68] uses the notion of a case to identify the underlying syntactic and semantic relationships in a sentence. Fillmore defines six different types of cases: agentive (A), instrumental (I), dative (D), factitive (F), locative (L) and objective (O).

For example, in the sentence "*the processor transmits a value to the register using the bus"* the preposition *to* identifies the *locative* case, and *using* identifies the *instrumental* case. The noun *processor* is the *agentive* case, and the noun *value* is the *objective* case.

It is important to note that none of these cases can be interpreted as a subject or object, although, A appears to play the subject role and O, the object role in active sentences. For example, in the two sentences *"processor transmits data"* and *"data was transmitted by the processor,"* the token *processor* is A in both the sentences, whereas *processor* is the subject in the first sentence and direct object in the second.

Since all verbs do not accept the same set of cases, Fillmore uses the terms "case frames" and "case features" to describe the cases a verb may have. A case frame describes the cases used in a sentence, and a case feature of a verb describes the types of case frames into which a verb may be inserted.

For example, the case frame for the sentence *"the processor sends the data to the memory through the bus"* would be [——A O L I], indicating that there are nouns in the agentive, objective, locative, and instrumental cases. The case frame for *"the controller interrupts the processor"* would be [——A O]. The verb *send* can be inserted into either case frame, though in the latter case a nonsensical sentence, "*the controller sends the processor*" results. However, the verb *interrupt* can only be inserted into the second case frame.

In this thesis, a variation of case features is used to define the meaning of words and to identify the roles the various nouns play in relation to the verb. The prepositions that identify these roles are termed *syntactic markers*. The subject and object markers identify the subject and object respectively. Such a variation of the case feature is called a frame in this thesis.

2.3. Quine-McCluskey Method for Boolean minimization

The Quine-McCluskey Method [McCluskey56] for Boolean minimization provides an exact solution for the sum of products minimization of Boolean functions. This tabulation method was first proposed by Quine and later simplified and extended by McCluskey. Firstly, a prime implicants table is formed, which has the minterms along the columns and the prime implicants (PIs) along the rows. Figure 3 shows an example PIs table in

which C_i identifies the minterms and R_i identifies PIs. An "X" at the intersection of the corresponding row and column indicates that the PI covers the minterm. An essential PI is one that covers a minterm not covered by any other PI. For example, in the table shown in Figure 3, the minterm C_5 has only one "X". Hence, it identifies R_1 as an essential PI. The significance of an essential PI is that it must be a part of the solution since it is the only way to cover a particular minterm.

	C_1	C_2	C_3	C_4	C_5
R_1	X	0	X	0	X
R_2	X	X	X	X	0
R_3	X	X	0	X	0

Figure 3 Example of a Table.

Once all the essential PIs have been selected, the dominant PIs that cover the remaining minterms are identified. If two PIs of a table T, i and j, are such that i has an "X" for all the minterms for which j has an "X," then i is said to dominate j. For example, in Figure 3 prime implicant R_2 dominates R_3, as it has an "X" for all the minterms (C_1, C_2 and C_4) for which prime implicant R_3 has an "X." Dominated PIs can be discarded. The set of essential PIs and some dominant PIs forms the final cover.

McCluskey proposed that in the absence of essential or dominant PIs, for each minterm, the PI with the maximum number of "X" entries might be chosen as part of the solution set. If all minterms are covered, then the set of selected PIs forms the first approximate solution to the covering problem. The covering problem is now fragmented into two sets of PIs: the set of PIs that belongs to the solution and the second set of PIs that does not belong to the solution set, called the non-solution set. McCluskey proposed to replace a set of PIs from the solution set with a set of PIs from the non-solution set. This replacement set should have fewer PIs than the sub set of the solution set that it seeks to replace. However, it should cover all the minterms that the original set of PIs covers. If such a set can be found then the approximate solution can be improved. This process is repeated until a minimal solution is found. This method is useful to get a good approximate solution for the covering problem.

By mapping the tokens of the skeleton to the minterms, and the frames to the PIs, we can select a set of frames that best cover the skeleton in terms of the number of word matches. We do incorporate some ideas of the Quine-McCluskey algorithm in our covering algorithm. We form a table with the skeleton words as the column headers and the frames as the row headers. Word matches between the skeleton tokens and the frame tokens are identified and marked at the corresponding intersection. Dominant frames are treated similar to dominant PIs and essential frames are treated similar to essential PIs. Using an approach similar to McCluskey's, a set of frames that best covers the skeleton can be found.

While the Quine-McCluskey approach is useful to identify a covering solution, it does not provide a computational algorithm. Rudell and Sangiovanni present a computational algorithm for the covering problem. The next section discusses the approach adopted by Rudell and Sangiovanni.

2.4. Multiple Valued Minimization for PLA Optimization.

The paper by Rudell and Sangiovanni [Rudell87] provides an exact Boolean minimization algorithm. A new covering technique using a branch and bound algorithm was presented. The covering problem is viewed as a table containing rows and columns. Since the covering problem is NP hard, the worst case complexity of any algorithm can be expected to be poor. But the average case complexity can be greatly improved with the use of efficient heuristics. Two heuristics that are used in this approach are borrowed from the Quine-McCluskey approach. These are identification of the essential elements and identification of dominant rows. The algorithm also makes use of other heuristics that are:

Partitioning: If table **A** can be reordered into block diagonal form, then it represents two or more sub-problems, each of which may be solved by the application of the algorithm on the corresponding independent blocks. The results can then be unified.

Dominating columns: If a column i dominates column j of **A**, then if column j is covered, column i will automatically be covered. Column i thus provides redundant information for the covering problem and can be removed from **A**.

A routine to identify the "maximal independent set" forms the core of the algorithm. This routine finds the maximal set of columns of **A**, all of which are pair wise disjoint. The number of columns in this independent set is a lower bound on the solution to the covering problem.

The branch and bound algorithm explores the solution space recursively, by assuming that each row is included or excluded from the cover. If a row is included, the recursive call has an argument **A'**, which is the table **A** from which the row being included is deleted as well as the columns incident to that row. The included row is added to the solution. If the row is excluded, then the row being excluded is deleted from **A** to form **A'**. The excluded row is also deleted from the solution. The algorithm is then applied to the reduced table **A'**. The algorithm maintains two solutions that are the current solution and the best solution. If the accumulated set of rows in the current solution exceeds the bound that is the number of rows in the best solution, that search path is abandoned. However, if the current solution is better than the best solution, it replaces the best solution. In this way the best solution is improved to find a solution that is close to the optimal solution.

In this thesis the skeleton covering problem has been mapped to this branch and bound algorithm. However, while Rudell and Vincentnelli's algorithm is for Boolean minimization for PLA optimization, this thesis seeks to cover the skeletons. Consequently, the two papers address a number of dissimilar constraints.

While the Rudell and Vincentnelli's algorithm does not discriminate among the columns, in this algorithm only those frames whose verb is found in the skeleton are candidates for the solution. Partitioning and elimination of dominant columns as a means of reducing the matrix size, was not implemented in the algorithm used in this thesis. Partitioning provides a lower bound on the solution to the covering problem. A solution can be terminated if the size of the current solution plus the size of the independent set is greater

or equal to the best solution seen so far. However, reordering the table into a block diagonal form is NP hard. Approximate lower bounds can be computed in less time. However the calculation of the lower bound was considered not very useful as the size of the problem was reducing quickly even without the use of a lower bound.

Reduction of table size using column dominance was not implemented for the following reasons:

(1) Unlike table size reduction using row dominance that uses bit vector operations, column dominance would involve accessing individual bits using a loop implementation. This can be costly (in terms of time).

(2) The table size reduction is fast even without the use of column dominance for table size reduction.

2.5. Efficient Heuristic Algorithm for Set Covering Problem

Karp [Karp72] has shown the set covering problem to be NP-hard. Presently, covering algorithms finding optimal solutions in polynomial time do not exist. Baker [Baker81] presents two simple, polynomial bounded procedures to produce a near optimal solution.

The paper attempts to cover a set **F** using a set of sets $\{P_1,P_2,...,P_n\}$, where P_i is a subset of **F**. The first of the two algorithms starts with the empty set as an initial solution and successively adds to the solution, the set P_i which covers previously uncovered elements at a minimum average cost per new cover. The process continues until all elements of the set **F** are covered by the solution set.

The second algorithm provides a method for merging the different solutions generated by the first algorithm in different trials. From two candidate solutions, the algorithm identifies sets of P_i that cover mutually exclusive subsets of **F**. When all of the sets P_i have been assigned to a set, the cost of the corresponding sets of each solution are compared and the lowest cost subsets are used to form the composite solution.

A parallel may be drawn between the sentence skeleton and the set **F** and between the frames and the set of sets $\{P_1,P_2,\ldots,P_n\}$. However, the frames in our problem are not necessarily subsets of the sentence skeleton.

The trade off is between optimality and complexity. While this paper trades optimality for complexity, in our case, a solution which is near optimal is sought with an exponential worst case bound.

2.6. Logical Form Generation as Abduction

Dasigi ([Dasigi94a], [Dasigi94b]) proposes a new approach to disambiguate word senses in a given sentence. A sentence is viewed as a set of assertions. An assertion is an attribute-value pair or an object-attribute-value triple. The goal of Dasigi's paper was to draw a domain specific interpretation for a given text. Such an interpretation is construed from a domain specific network that contains a set of all possible assertions.

The domain specific network consists of syntactic categories such as Value, Object, Attribute and Conjunctor, and their relationships such as PossibleValue, HasAttribute, HasPart and Instance. Sub-graphs are formed with the syntactic categories as the nodes and their relationships as arcs. The syntactic categories are also attached to semantic categories such as Object-*processor*, Attribute-*speed* and Value-*slow*, where *processor*, *speed* and *slow* are the semantic categories. The semantic categories are also nodes and have relational arcs between them. Such nodes and arcs also form sub-graphs. The syntactic and the semantic sub-graphs inter-link to form the semantic network.

A word of a sentence is associated with one or more categories in the domain specific network. These categories may be syntactic or semantic categories. Dasigi's paper uses a covering approach to sentence interpretation, which involves choosing a set of assertions that is both syntactically and semantically appropriate. Such an "appropriate" set is chosen as dictated by the criterion of parsimony.

The approach seeks to cover both the syntactic and the semantic aspects of the words along two routes, the syntactic route and the semantic route. The syntactic route to covering addresses the issues of word ordering and structure of the sentence. Along this route, "minimality" is chosen as the constraint that satisfies the notion of parsimony.

For example, given a set of words W_1, W_2, W_3; If W_1 evokes the syntactic categories $\{S_2, S_3\}$, W_2 evokes the categories $\{S_1, S_3\}$ and W_3 evokes the categories $\{S_1, S_2\}$, then if the input is W_2, W_3; the possible covers are $\{S_1\}$ and $\{S_2, S_3\}$. But using the minimality constraint we conclude that $\{S_1\}$ is the parsimonious cover along the syntactic route.

Along the semantic route, irredundancy and cohesiveness are chosen as the constraints that satisfy the criterion of parsimony. Irredundant covers are those covers whose subsets do not yield a cover. Cohesive covers are those covers whose sub-trees consist of the smallest number of nodes (semantic categories).

The two routes are applied such that they aid each other in narrowing the search for the most parsimonious cover. In contrast to Dasigi's conceptual structures of O-A-V triplets, this thesis uses a conceptual structure called the frame. Also the knowledge base used here is a repository of frames unlike Dasigi's semantic network. The frames inherently contain in them, both semantic and syntactic information required to cover the skeletons.

Like Dasigi's approach, this covering algorithm also uses the minimality constraint to identify the parsimonious solution. However, unlike Dasigi's approach, covering is sought along a single route without distinguishing between the syntactic and the semantic route.

2.7. PIE system for Template Filling

This paper [Lin95] discusses a parser for information extraction. The text is broken up into sentences. A lexical analyzer turns a stream of tokens into a sequence of lexical items. The lexical items may be combined or deleted by lexical rules. The system takes the sequence of lexical items and outputs a dependency tree between the words of the

sentence. A parse for the full sentence is attempted. However, if this fails, parse fragments that cover the complete sentence are retrieved. The system uses a lexicon, which contains domain specific syntactic information such as head words and a list of their usage. For each word from the sentence that is found in the lexicon, a lexical item is created for each of its usages. The lexical item contains the position of the word in the sentence, root form of the word, and its *semantic features*. For instance the word "100" has the semantic feature *number* and the word "Dollar" has the semantic feature *money*.

After all the words of the sentence have been retrieved, a set of lexical rules is matched against the lexical items. The lexical rule identifies the relevant semantic entities in the sentence. A lexical rule consists of a pattern and an action. If the words of the sentence match the pattern of a certain rule then the corresponding action fires.

The lexical items are given as input to a parser. The parser uses grammar rules to output a dependency tree. Figure 4 shows an example of a dependency tree.

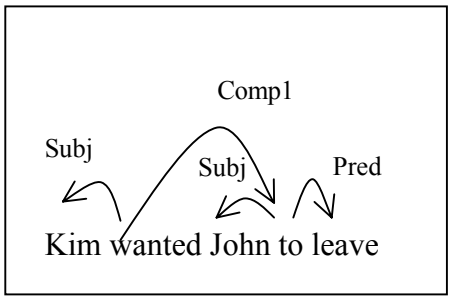

Figure 4 Evolution of a skeleton.

The dependency tree is constructed using the dependency grammar. When a word modifies two or more words in a sentence, dependency grammars face serious problems. Unlike the Lin's approach, in this thesis a grammar-free covering approach is presented.

In this thesis the end result is not a dependency tree but rather a set of frames that cover the skeleton sentence. The dependency tree provides more semantic information compared to the frames. However for the purposes of this thesis, the semantic information provided by the frames is considered adequate.

Chapter 3. Covering Algorithm

The ASPIN system developed at the ADRG lab analyzes English sentences using a parser with a context-free, phase structured grammar (PSG) followed by a compositional semantic analyzer. This approach will fail when the grammar or semantic rules to analyze a sentence are inadequate. Such performance degradation warrants the need for an alternate approach to language analysis.

This thesis provides an alternative. The view taken here is that a sentence is composed of semantic patterns called frames. The notion of frames was developed by Filmore in his case grammar. Unlike phase structure grammar, case grammar does not depend on the word order of a sentence, but identifies the roles of words in a sentence by their relationship in a predicate. Filmore calls his frames case features. These case features have slots that identify their fillers as agent, object or instrument of a particular verb in a particular usage or meaning. A slot and filler comprise a *case*.

This thesis borrows the notion of a frame from Filmore's case features. A frame is viewed an abstract canonical sentence. Each frame contains the semantic information that sentences or sentence phrases usually comprise. It is assumed that the semantics of the frames are fully understood. As an alternative to parsing, a sentence is decomposed to a set of frames. The semantic content of the sentence is then interpreted through the semantic content of the individual frames in the set. Such a set of frames is called a cover. These frames are selected by correlating the verb and the cases of the frames with the sentence. Frames with strong correlation form good candidates for the cover. The

covering approach to semantic decomposition of sentences is robust, as it does not use a parser. The next section provides an overview of the covering approach.

3.1. Approach

The sentences from the input text are first "chunked". The process of chunking abstracts the phrases of a sentence to their head words. Such an abstract sentence contains the core meaning of the original sentence. However, if the chunking process is not implemented perfectly, there may still be some "noisy" tokens in the abstract sentence that need to be filtered. These tokens are considered "noisy", as they do not contribute to the core meaning of the sentence. Such tokens include punctuation characters, conjunctions and abbreviations. Once these tokens are removed the abstract sentence can be further processed to normalize the nouns and the verbs in the sentence to their root morphology. For instance verbs such as "*written*", "*write*", "*wrote*" and "*writes*" will be normalized to the root form "*write*". This is essential because the verbs and nouns in the frame are in the root form. The normalization of nouns and verbs of the abstract sentence helps to improve the correlation between its tokens and the verb and the cases of the frames.

The abstract sentence is further processed to abstract its head words to more generalized concept types. For instance, the head word **microprocessor** is abstracted to **processor.** This is again useful in improving the correlation between the tokens of the abstraction and the verb and cases of the frames.

The result of abstracting a sentence this way is called a sentence skeleton. In order to decompose the sentence semantically, the skeleton is covered by a set of frames that are drawn from a repository of such frames. The tokens of the frames are also subject to semantic generalization in order to improve covering results. The frames are mapped to a bit vector representation called the frame vector. A bit vector representation is preferred, as it is more memory efficient. The covering algorithm uses the bit vectors to generate a cover. A branch and bound algorithm is used as the covering algorithm. The goal of the algorithm is to find a minimal set of frames that best cover the skeleton. The following sections describe the various stages of processing in greater detail.

3.2. Chunking

The input sentences are abstracted using Chunker software. The Chunker uses phase structure grammar for grammatical analysis of the sentences. Phase structure grammar (PSG) is widely used for sentence generation and sentence interpretation [Nagao96]. PSG views a sentence as a sequence of phrases. Each of these phrases in turn is viewed as a sequence of sub phrases and so on. PSG defines the type of sub phrases that a phrase can comprise and their order in a phrase. Using PSG, words can be combined to form phrases, which in turn can be combined to form sentences.

Sentence interpreters use grammatical analyzers such as Chunker to analyze sentences. Contrary to constructing sentences, grammatical analyzers use grammar rules to decompose sentences. Chunker decomposes a sentence by abstracting its phrases. The noun phrases and the verb sequences are reduced to their head words. For instance, consider the example sentence shown in Figure 5.

The Intel 8086 microprocessor completes the routine execution cycle and sends the binary data to the static memory.

Figure 5 An example of a sentence

Table 1 shows the abstraction of the noun phrases and the verb sequences for this sentence.

Table 1 Abstraction process

Chunks	Syntactic type	Head/Connective
the intel 8086 microprocessor	Noun Phrase	microprocessor
completes	Verb	completes
the routine execution cycle	Noun Phrase	cycle
and	Conjunction	and
sends	Verb	sends
the binary data	Noun Phrase	data
to	to	to
the static memory	Noun Phrase	memory
.	.	.

The final abstraction is as shown in Figure 6. A sentence is thus abstracted to a sequence containing the head words of the phrases and their connectives.

microprocessor completes cycle and sends data to memory .

Figure 6 Sentence Abstraction.

3.3. Noisy word elimination

In the next step of processing, the abstracted sentence is filtered to eliminate "noisy" tokens. Such tokens include coordinating conjunctions, abbreviations and punctuation characters. These tokens are considered "noisy", as they do not contribute heavily to the core meaning of the sentence. Figure 7 shows the abstraction after filtering the conjunction "*and*" and the punctuation character ".". Appendix B shows a selection of the noisy words file.

microprocessor completes cycle sends data to memory

Figure 7 Abstraction of the sentence after filtering noisy words.

3.4. Normalization

The verbs and nouns in the abstraction may be of any form. In the frames, the verbs are in root form and the nouns are in singular form. Hence, a normalization of the abstraction words is required. The nouns are normalized from their plural to their singular form. Likewise, the verbs are normalized to their root forms. Figure 8 shows the abstracted sentence after normalizing the verbs to their root form. Since this sentence does not have any nouns in plural form, the nouns are not mapped. Appendix C shows the nouns and their respective root forms used in this thesis. Appendix D shows the verbs and their corresponding root forms.

microprocessor complete cycle send data to memory

Figure 8 Abstraction of the sentence after normalizing.

3.5. Semantic generalization

Nouns, verbs, and most adjectives have semantic attributes called their concept types [Sowa84]. The type of a concept constrains the relations to which it may be linked. While some concept types are general, others are more specific. A type hierarchy is used to partially order the set of all types based on the subtype relation. For example, **microprocessor** is a subtype of **processor**, and **receive** is a subtype of **action**.

In this thesis, each word of the abstraction is mapped to one of a small set of general semantic types. Table 2 shows a complete listing of the concepts that are generalized and their corresponding supertypes. Any skeleton token that is a subtype concept is generalized to its corresponding supertype given in Table 2 . Such a type hierarchy is called a two-level type hierarchy. The two-level hierarchy used in this thesis can be visualized as shown in Figure 9.

Table 2 Two-level Type Hierarchy.

Subtype	Supertype
sdma	controller
dma	controller
sdc	controller
microcomputer	processor
computer	processor
microprocessor	processor
characteristic	attribute
command	value
constant	value
frequency	value
information	value
number	value
ordinal	value
time	value
transducer	device

The use of semantic generalization on the abstraction is shown in Figure 10. The noun **microprocessor** is generalized to its super type **processor**.

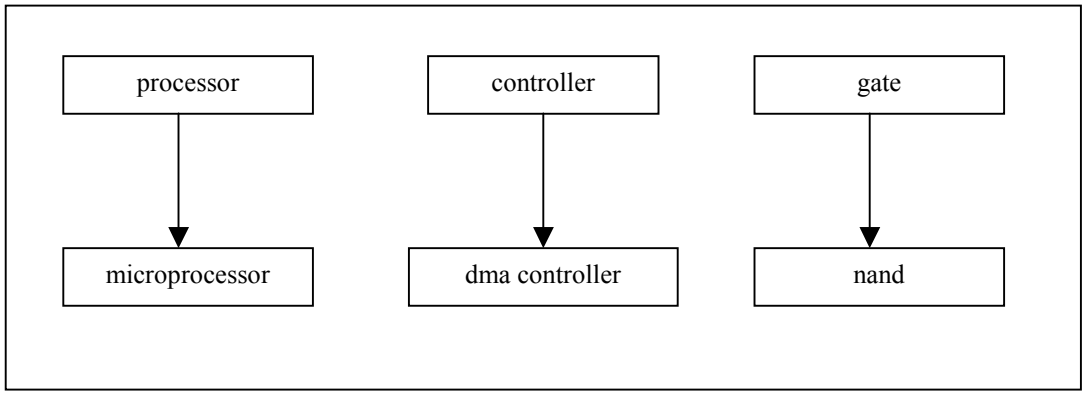

Figure 9 Two level type hierarchy.

processor complete cycle send data to memory

Figure 10 Abstraction of the sentence after semantic generalization: skeleton.

The final result of this abstraction process is a sentence skeleton. The skeleton contains the core meaning of the sentence. Hence, instead of the sentence, the skeleton of the sentence will be covered using a set of frames. The next section explains the notion of a frame.

3.6. Knowledge Representation - Frames

Grishman [Grishman86] observes that the primary challenge of text analysis is to identify and organize the requisite knowledge such that it effectively bears on the problems associated with text analysis. Recently, knowledge has been organized based on the following observation reported by Grishman: "*Human mind assimilates new information by identifying it as an instance of a pattern with which the mind is already familiar*". With this observation providing guidance for computer based text analysis, new

information can be interpreted in terms of known patterns. Minksy [Minsky75] calls such patterns as **frames** and systems using such patterns as **frame-based systems**.

In this thesis a frame is viewed as an abstract canonical sentence. A frame consists of a verb and a list of **cases**, and is of the general form *verb (subject subj, object obj, other cases)*. Each case consists of a tag (**marker**) and **filler**, and specifies some role related to the verb, such as agent, object, purpose or manner. A tag can be a preposition, subordinating conjunction or an implied marker. The primary implied markers are *subject* and *object* markers and their fillers are *subj* and *obj* respectively. Figure 11 shows some examples of frames. Note that a subject or object filler may or may not be present in a frame. Appendix A shows a larger selection of frames.

send (subject processor, object data, to memory)	(a)
abort (object cycle)	(b)
arrive (subject byte)	(c)

Figure 11 Examples of frames.

We consider two types of frames: **prototype frames** and **instance frames**. In prototype frames, the verb and fillers are semantic generalizations. Figure 12 shows a prototype frame.

action (subject device , object data , to device)

Figure 12 An example of a prototype frame.

Instance frames on the other hand, include verb and fillers that are instances of a prototype frame. A prototype frame can have any number of instance frames. Figure 13 shows two examples of instance frames for the prototype frame in figure 12.

send(subject microprocessor, object bits, to register)
transmit(subject controller, object bits, to memory)

Figure 13 Examples of instance frames.

In practice, a large number of frames are both a prototype frame and an instance frame at the same time. We call such frames as **mixed frames**. The verb and fillers of such frames

may be semantic generalizations or instances actually found in sentences. Figure 14 shows an example of a mixed frame.

send (subject device , object data , to memory)

Figure 14 An example of a mixed frame.

In this thesis two sets of frames were used. The first of these sets, "frames1.txt", contains about 1600 frames. This file was generated manually from microprocessor product data sheets for 8-bit microprocessors. A second set "frames2.txt" contains about 2400 frames. This file was generated automatically with a parser using a context free grammar. The frames were extracted from DMA controller patents. The same DMA controller patents were also used as an input sentence file for evaluating the performance of the algorithm in this thesis.

Both sets of frames consist largely of instance frames and mixed frames. Whenever a sentence abstraction contains a verb or filler of a prototype frame, the verb and fillers of the mixed frame and instance frame of the repository are generalized using a type hierarchy. Unlike the simple two level hierarchy used with sentence abstractions, a more elaborate multilevel hierarchy was used with the tokens of the frames. A simple two level hierarchy is extended using transitivity to a multilevel hierarchy. Figure 15 shows a partial multi-level type hierarchy. For instance, if **microprocessor** is a subtype of **processor** and **processor** is a subtype of **device**, then by transitivity **microprocessor** is a *subtype* of **device**. The *universal type* is defined to represent the common supertype to all concept types. Similarly, the *absurd type* is defined to represent the common subtype of every concept type. Note that while all entities are instances of the universal type, no entity is ever an instance of the absurd type. If the common subtype of two types, such as **processor** and **value**, is the absurd type, this indicates that it is logically impossible for an entity to be both a processor and a value.

In a multi-level type hierarchy, two concept types at the same distance from the universal type belong to the same level. Correspondingly, they differ in levels if they are at unequal distances from the universal type. For example the types **processor** and **memory** are at the same level but **processor** and **device** are at different levels (see Figure 15). A

concept type closer to the universal type is at a higher level than a type that is farther away. For instance, the type **device** is at a higher level than the type **processor**.

We loosely define the term semantic distance as the distance between any two concepts, where distance is measured as the number of arcs between the two concept types in the hierarchy. Suppose the skeleton had a token **processor** and a frame had a token **computer**, then the semantic distance between the two concepts is one. In order to minimize the semantic distance, the skeleton token and the frame token may need to be abstracted to a more general semantic type. The minimum semantic distance is zero, which occurs when both the skeleton token and the frame token are generalized to the same semantic type. For instance when the frame token **computer** is generalized to the **processor** semantic type, the skeleton token and the frame token have the minimum semantic distance as they are of the same semantic type.

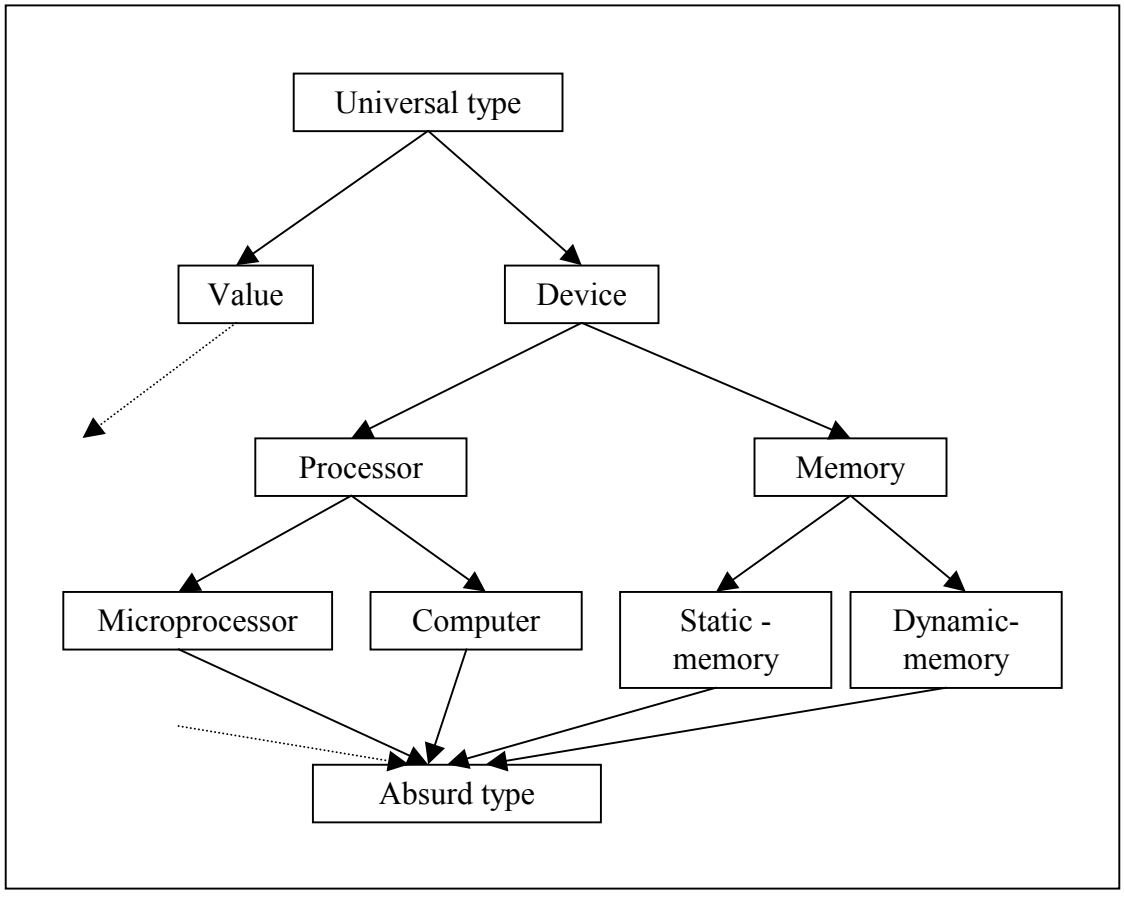

Figure 15 Multilevel Type Hierarchy.

In some cases, to achieve the minimum semantic distance, a skeleton token or a frame token may need to be generalized more than once. For instance, consider the skeleton token **microprocessor** and the frame token **static-memory**. The semantic distance between these concept types is four. When the concept type **microprocessor** is generalized to **processor** and **static-memory** is generalized to **memory**, the semantic distance is reduced to two. A semantic distance of two calls for further generalization of the concept types in order to achieve the minimum semantic distance. Thus **processor** is generalized to **device** and **memory** is also generalized to **device**. When two concept types are separated by the minimum semantic distance, one concept type can be used to cover the other.

While a multi-level type hierarchy can be useful if implemented for both the skeletons and the frames, that was beyond the scope of this thesis. In this thesis, the skeleton tokens are generalized using only a two-level hierarchy as described in section 3.5. The multi-level hierarchy is used only with the verb and fillers of instance frames and mixed frames. The algorithm used to implement the multilevel hierarchy is shown in Figure 16. The MultiHierarchy() function determines whether a skeleton token is a supertype concept. For every skeleton token that is a supertype concept, each frame token that is at a lower level than the skeleton token, is generalized to the same level as the skeleton token, provided that it does not match any of the skeleton tokens. The tokens of all the frames are generalized.

Figure 17 shows an example of a multi-level hierarchy file. In this file each line is stored in the general form: (*line_num)* <*sub_type1*> | <*sub_type2*> |.....| <*sub_typen*> → <*super_type*> *[level_num]*. The level separator lines (denoted by "******") are used to separate the adjacent levels and are provided only for the sake of readability. The concepts to the left of the arrow are subtypes of the concept to the right of the arrow. The "|" symbol is used to separate multiple subtypes of a supertype concept. The *level_num* is used to denote the level of the supertype concept. The *level_num* increases with the increase in distance between the concept type and the universal type, i.e., higher the level of a concept type lower is its *level_num.* The use of multilevel hierarchy to generalize the tokens of the frames is explained below.

```
S → Skeleton;
F → Frame;

MultiHierarchy(S)

/ skel_level  indicates the hierarchy level of a  supertype skeleton concept in the hierarchy
/ fra_level  indicates the hierarchy level of a subtype  frame concept in the hierarchy

For every skel_token in S
  skel_level = 0;                  / 0 indicates skeleton token is not a supertype concept
  For every line_num `             / determine if skeleton token is a supertype concept
    If (skel_token == super_type)
      skel_level == level_num;   / identify hierarchy level of skeleton concept.

  If (skel_level !=0)   /if skeleton token is a supertype concept.
    For every F in the repository
      For every fra_token in the frame
        If (fra_token is not found in S)      /if the frame token does not cover a skeleton token
          For each line_num                   /generalize frame token until fra_level = skel_ level
            For each sub_type in line_num
              If (fra_token == sub_type)      /identify hierarchy level of frame token
                fra_level = level_num;
                If (fra_level >= skel_level)
                    fra_token = super_type;           / update frame token to super type
                    break;                            / go to next line_num
```

Figure 16 Multilevel Hierarchy Algorithm

A frame token is compared with a skeleton token. If the skeleton token is at a higher level in the type hierarchy, then the frame token is generalized to the same level as the skeleton token. This reduces the semantic distance between the two tokens. Now a frame token may or may not cover the skeleton token depending on whether or not the frame token was a subtype of the skeleton token before generalization. For instance, consider the frame concept type **microprocessor** and the skeleton concept type **processor** (see Figure 15). Since **microprocessor** is a subtype of **processor**, after generalizing, it covers the skeleton concept type **processor**. However, a frame concept type **static memory**, which is at the same level as the type **microprocessor**, does not cover the type **processor** even after generalizing. This is because **static memory** is not a sub type of **processor**. On the other hand if the skeleton concept type was **device**, then since both **microprocessor** and **static memory** are subtypes of **device**, both concept types will cover the skeleton concept type after generalization.

Multilevel Type Hierarchy

(1) microcomputer -> computer [3]
******
(2) microprocessor | computer | cpu -> processor [2]
(3) dma | sdma | sdc | dma -> controller [2]
(4) nand -> gate [2]
******
(5) characteristic -> attribute [1]
(6) processor | memory | controller | register | gate -> device [1]
(7) frequency | constant | information | number | ordinal | time | command -> value [1]

Figure 17 Multi Level Hierarchy File

It should be noted that a frame token is only generalized if it does not already cover a skeleton token. For instance, consider the skeleton "*device send data to memory."* When a frame *send (subject processor, object data, to memory)* is considered for covering the skeleton, each token of the frame is compared with all the tokens of the skeleton. The tokens **processor** and **memory** in the frame are both subtypes of the token **device**. However, while the token **processor** is generalized to **device**, the token **memory** is not generalized as it covers the token **memory** of the skeleton. In this way the frames are generalized using type hierarchy. Once the frames have been generalized they can be used to cover the skeleton. A covering algorithm is used for this purpose. The next section describes the covering algorithm used in this thesis.

3.7. The Covering Algorithm

The goal of this thesis is to semantically decompose sentences to a set of frames that best cover the skeleton of a sentence. There are many approaches that may be adopted to solve this problem. In order to facilitate a memory efficient implementation, a covering approach using bit vectors was adopted. The algorithm is similar to Rudell and

Vincentnelli's covering algorithm. However since this approach deals with sentence covering, the skeletons and the frames had to be mapped to their respective vectors before using the algorithm. This section explains the mapping of the skeleton and the frames to a vector form. The next section explains how the branch and bound algorithm is used on these vectors to obtain a cover for the skeleton.

For a given sentence skeleton, a **frame vector** is generated for each frame in the repository. A frame vector is generated by comparing the tokens of the skeleton with the verb and cases of the frame. If the tokens of the skeleton "match" the verb or the cases of the frame they are set to one otherwise they are set to zero. The tokens are set to one only if both the marker and the filler of a case are matched. Figure 18 below shows two frames and their corresponding frame vectors for the skeleton "*processor complete cycle send data to memory"*. Note that a frame vector is generated by setting a component to "1" whenever a skeleton token matches a frame token.

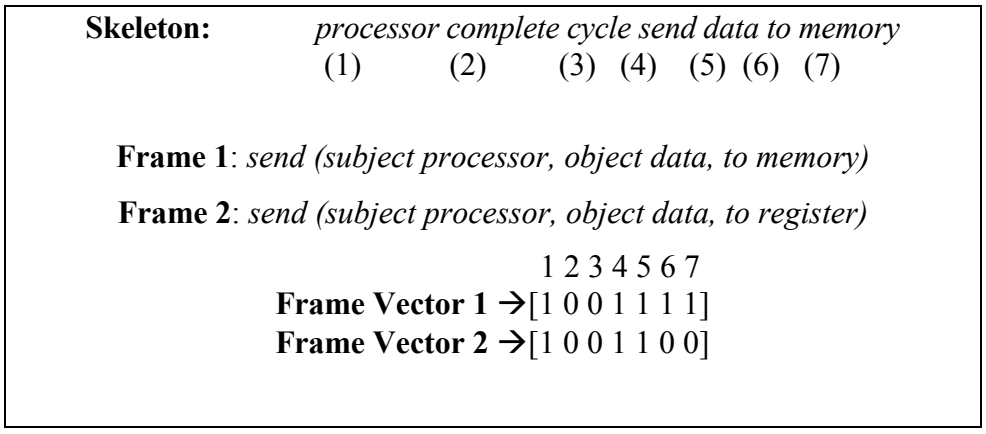

Figure 18 Frames and Frame Vectors.

For the above skeleton, given the two frames f_a: *send (subject processor, object data, to memory)* and f_b: *complete (subject processor, object cycle),* their respective frame vectors F_a and F_b, may be written as [1 0 0 1 1 1 1] and [1 1 1 0 0 0 0]. These frame vectors are used to make the rows of a matrix called the **table**. The table has as many columns as the length of the frame vectors. For the skeleton and the frames used above, the table is as shown in Figure 19 with two rows, one each corresponding to F_a and F_b.

F_a:	1 0 0 1 1 1 1
F_b:	1 1 1 0 0 0 0

Figure 19 Table.

We define a **solution vector** as a vector obtained by the sum (logical OR) of all the frame vectors included in the solution. Thus, if only the first row of the table corresponding to frame vector F_a is included in the solution, then the solution vector is [1 0 0 1 1 1 1]. Since not all the columns of the solution vector are covered, we have only a **partial cover** at this stage. Alternately, if the second row corresponding to F_b is added to the solution, then the new solution vector is the sum of the vectors of F_a and F_b i.e., the solution vector is [1 1 1 1 1 1 1]. Since no element of the solution vector is a zero, we have a cover. A **cover** of a sentence is defined as a set of frames, such that, all the tokens of the skeleton are matched in the union of the verbs and cases of these frames. It is possible that some sentences may only have a partial covering solution for a given set of frames in the repository. This happens when one or more words of the skeleton are not found in the vocabulary of the frames whose verb occurs in the skeleton.

In order that the frames forming the final solution provide meaningful covers, a restriction is imposed on the type of frames that can cover a skeleton. Frames whose verb matches at least one verb in the skeleton are considered plausible candidates for a cover. Such frames are called candidate frames. The number of candidate frames in the repository for a given skeleton defines the true size of the problem. In order to find a minimal set of frames that covers the skeleton, a branch and bound algorithm is used.

3.7.1. The Branch and Bound Algorithm

The covering algorithm is a branch and bound algorithm. Due to the general nature of the branch and bound algorithm, it is explained here in the perspective of the skeleton covering problem. The skeleton covering problem involves identifying a minimal set of frames from a repository to provide a maximal partial cover of the skeleton.

Let $\mathbf{f} = \{\mathbf{f_1}, \mathbf{f_2}, ..., \mathbf{f_n}\}$ denote n candidate frames in the repository. A systematic approach to solving the problem would be to list all possible solution subsets of **f** from among which the minimal cover is identified. If the solution is defined as a binary vector $\mathbf{D} = \{\mathbf{D_1}, \mathbf{D_2}.... \mathbf{D_n}\}$, where $\mathbf{D_i}$ takes on a binary value 1 or 0, depending on whether or not the frame $\mathbf{f_i}$ is part of the solution, then, a total of 2^n possible solutions exist. The search space can be visualized as a tree, whose nodes represent all possible solutions (see Figure 20). This is a problem of exponential complexity. In our case, the average problem size is typically 350 frames, and in some cases there were as many as 1000 frames or more. The average case with 2^{350} ($=2 * 10^{105}$) solutions cannot be explored fully. This warrants the need for a branch and bound algorithm, as it is infeasible to explore the solution space fully for most cases. The branch and bound algorithm identifies good covering solutions (not optimal) by truncating some of the search paths.

Figure 20 shows an exponential tree, which is the complete solution space for an exponential search algorithm for a given ordering of frames. The branch and bound algorithm explores this solution space partially. A **search path** is defined as a set of nodes that are connected by arcs, originating at the **root** and ending in a **leaf**. The root is defined as the node that does not have any arcs entering the node. A leaf is a node that does not have any arcs leaving the node.

Each node is labeled by a frame, $\mathbf{f_i}$, which is a member of the set of frames **f**. The solution is indicated as a vector beside each node in Figure 20. The elements of the vector are the binary decision variables, $\mathbf{D_i}$'s, that denote the inclusion or the exclusion of the frames along the path from the root in the solution. A "1" indicates the presence and a "0" indicates the absence of the corresponding frame in the solution. The search tree is explored systematically, by successively considering the search paths from left to right in the figure.

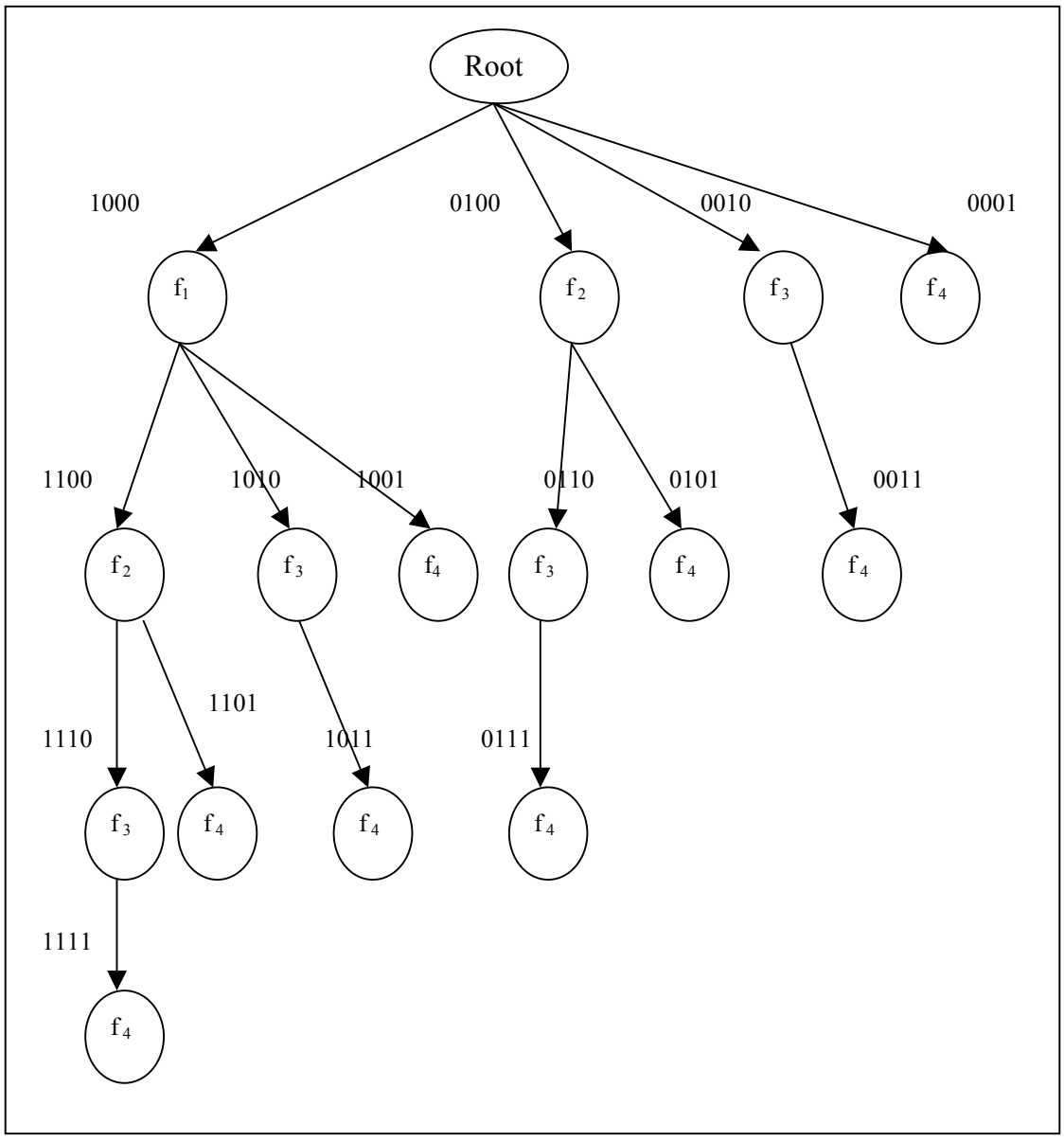

Figure 20 An exponential solution tree.

The exploration is depth first. A solution is found along a search path when the frames included in the solution, cover the skeleton. The search path is terminated when a cover is found. The shortest path along which a cover is found is stored as the best solution seen thus far. When a new cover is found that is better than the best seen thus far, it replaces the best. A search path is also terminated if it is found to be costlier than the best solution. The cost is a function of the number of frames in the solution. All search paths with the same initial selection of frames are also terminated as they are conceivably of greater cost than the best solution. Such a truncation of the search paths is called "pruning" of the tree. Pruning is very useful as it reduces the average search time [DeMichelli94].

Figure 21 shows a section of the solution tree. Assume a current best solution size of four has been found. This indicates that currently, there exists a covering solution for the skeleton that contains four frames. Next, assume that node $\mathbf{f_2}$ has been included in the solution and at this point the current solution is a function of two frame vectors, corresponding to nodes $\mathbf{f_1}$ and $\mathbf{f_2}$. The search is now for a cover with three frames or less. Thus a cover has to be found at the next node in order to replace the existing best solution containing 4 nodes. Thus the search tree will be pruned if the algorithm is unable to find a cover at the nodes b through e. The gray region indicates the sections of the tree that are pruned.

Branch selection that leads quickly to promising solutions can be used. Biasing selection of branches that have a higher number of token matches over those that have a lower number of token matches is an effective heuristic that can cause the algorithm to converge on a good solution faster. If the cost function (current solution size) attains values close to the final cover size early in the search tree, then it is likely that a large part of the search tree will be pruned. These heuristics help in reducing the execution time but do not affect the nature of the solution. It should be noted though, that the ordering of the frames does affect the nature of the solution. A different set of frames may form the final cover for a different ordering of frames.

BranchAndBound() is a recursive algorithm, invoked with two arguments: the frame vector table, $\boldsymbol{T}$; and the current solution set $\boldsymbol{C}$. Set $\boldsymbol{B}$ is the best solution seen by the algorithm thus far. The size of a solution $\boldsymbol{X}$ is denoted by $|\boldsymbol{X}|$, where size is computed as the number of frames in the solution. Variable $\boldsymbol{u}$ stores the size of $\boldsymbol{B}$, i.e., $\boldsymbol{u} = |\boldsymbol{B}|$. Variable $\boldsymbol{u}$ is assigned an initial value that is the worst case solution size. The worst case solution size is computed by assuming that one frame covers only one skeleton token. Thus a skeleton length number of frames will be required to cover all the words of the skeleton. $\boldsymbol{B}$ is global to the **BranchAndBound()** method. The solution set $\boldsymbol{C}$ is associated with a solution vector C that contains a "1" in all the columns that the frames in $\boldsymbol{C}$ cover. The solution vector C is initially a zero vector as solution $\boldsymbol{C}$ is empty. The pseudo code for the branch and bound algorithm is given in Figure 29.

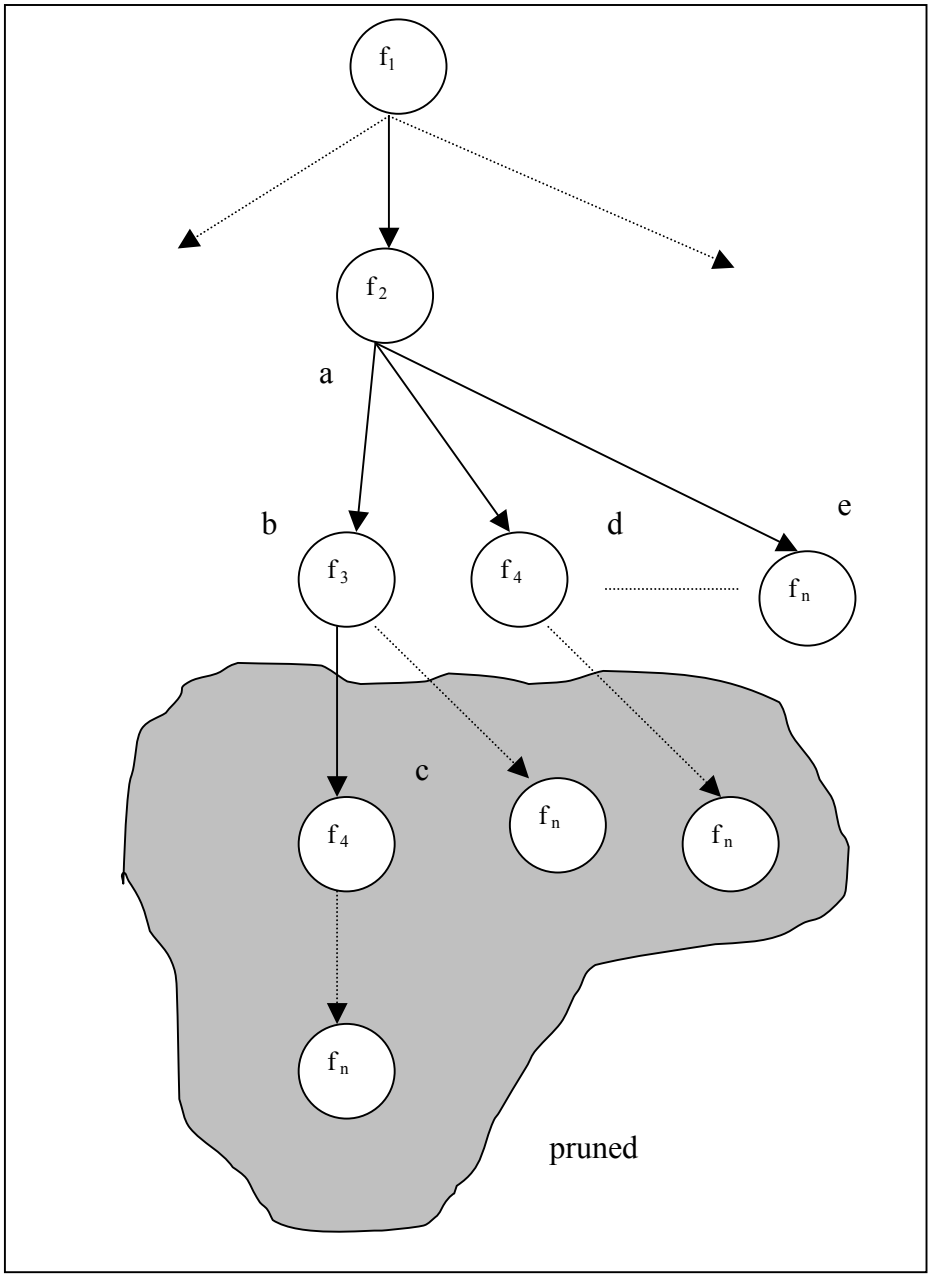

Figure 21 Example of Pruning.

The size of the table ***T*** is first reduced. A call to **ReduceTable()** algorithm is invoked to reduce the size of the table by: (1) deleting **dominated frame vectors** and (2) identifying **essential frame vectors**; The pseudo code for the **ReduceTable()** algorithm is as shown in Figure 22. The algorithm takes the table ***T*** as its argument and returns the same after reducing its size. Table reduction is performed by eliminating dominated frame vectors. A frame vector **dominates** another if the former has at least as many entries as the latter

and these entries occupy all the columns occupied by the latter. Any frame vector dominated by any other frame vector is deleted from ***T*** to form a new table ***T'***.

The table ***T''*** is obtained from ***T'*** by identifying the essential frame vectors. An **essential** frame vector has only one entry for some column. The skeleton token corresponding to such a column can only be covered by a frame corresponding to the essential frame vector. Consequently, it is imperative that such a frame be added to the solution ***C***. The corresponding columns covered by the new frame vector are updated in **C**. All the columns that are covered by the essential are also removed from the table. A column is removed by setting all its elements to zero. ***T''*** is now copied to ***T***, and the original table is thus reduced.

The process of identifying the essentials and the dominant frame vectors and thus reducing the table size is repeated until no more essential frame vectors can be found. When the call returns, the original table ***T*** is replaced by a reduced table that contains only **dominant frame vectors**.

```
ReduceTable(T){
        {       T' = T after deleting the dominated rows;
                T'' = T' after picking essentials and deleting the
                corresponding columns. The essential frame
                vectors are also deleted;
                p = number of essentials picked;
                Add frames corresponding to the essential frame
                vectors to the current solution C; update C;
                T = T'';
        }while (p!=0)
return (T);
}
```

Figure 22 The ReduceTable() method pseudo code.

As an example of this process, consider the following sentence:

"*The microprocessor sends data to the static memory and displays the value.*"

The skeleton generated for this sentence is:

"*processor send data to memory display value*".

Let the repository include the frames shown in Figure 23.

$\mathbf{f_1}$:	*send(subject processor, object data, to memory)*
$\mathbf{f_2}$:	*send(subject computer, object data, to memory)*
$\mathbf{f_3}$:	*send(subject processor, object instruction, to memory)*
$\mathbf{f_4}$:	*send(subject processor, object data, to register)*
$\mathbf{f_5}$:	*display (subject processor, object value)*
$\mathbf{f_6}$:	*display(subject processor, object data)*

Figure 23 Frames.

The frames are converted to their frame vector form as shown in Figure 24.

$\mathbf{F_1}$	[1 1 1 1 1 0 0]
$\mathbf{F_2}$	[0 1 1 1 1 0 0]
$\mathbf{F_3}$	[1 1 0 1 1 0 0]
$\mathbf{F_4}$	[1 1 1 0 0 0 0]
$\mathbf{F_5}$	[1 0 0 0 0 1 1]
$\mathbf{F_6}$	[1 0 1 0 0 1 0]

Figure 24 Frame vectors.

Table ***T*** is formed with each frame vector occupying one row of the table as shown in Figure 25.

	1	2	3	4	5	6	7
$\mathbf{F_1}$	1	1	1	1	1	0	0
$\mathbf{F_2}$	0	1	1	1	1	0	0
$\mathbf{F_3}$	1	1	0	1	1	0	0
$\mathbf{F_4}$	1	1	1	0	0	0	0
$\mathbf{F_5}$	1	0	0	0	0	1	1
$\mathbf{F_6}$	1	0	1	0	0	1	0
C	0	0	0	0	0	0	0

$\boldsymbol{u} = 7$

Figure 25 The table *T*.

Rows $\mathbf{F_1}$- $\mathbf{F_6}$ identify the frame vectors that occupy the rows of ***T***. **C** is initialized to all 0's. A worst case initial best solution size, ***u***, of seven is assumed corresponding to the number of tokens in the skeleton. When the **ReduceTable()** algorithm is invoked on this

table, the dominated rows are deleted from ***T***. F_1 dominates F_2, F_3 and F_4. Hence these rows are removed. At this time, no new frames are added to the solution. The new table ***T'*** and the corresponding current solution vector **C** are as shown in Figure 26.

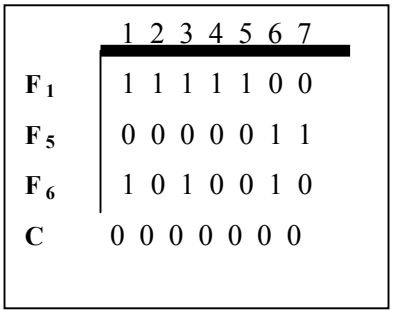

Figure 26 The reduced table *T'*.

In table ***T'***, frame vector F_1 is identified as an essential, since columns 2, 4 and 5 do not have an element 1 in any of the other frame vectors. All the columns covered by the essential frame vector F_1 are removed. Also F_5 is an essential since it is the only frame vector covering column 7. Consequently, columns 6 and 7 are also removed from table ***T'***. These columns are correspondingly set to 1 in the solution vector **C**. Frame vectors F_1 and F_5 are now deleted from ***T'*** to get a new table ***T''***. ***T''*** is an empty table as all its columns are covered. Since ***T''*** is copied to ***T***, ***T*** is also empty. Figure 27 shows the empty table ***T*** and the solution vector **C**.

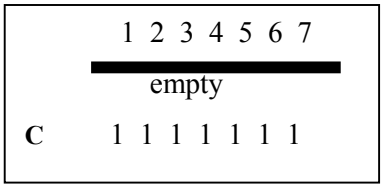

Figure 27 Empty table *T*.

Figure 28 shows the complete solution. Since u =7 is more expensive than the size of the current solution $|C|$ =2 , ***B*** is replaced by ***C***. In this example the table ***T*** is empty. Hence at line number 3 in the pseudo code the algorithm returns with the current solution ***C***.

processor send data to memory end cycle

f_1 → *send(subject processor, object data, to memory)*

f_5 → *end(subject processor, object cycle)*

Figure 28 The skeleton and its cover

In the above example, since |***C***| was less than ***u*** and since ***C*** was complete it was returned. However, if ***C*** was more expensive than ***B***, then, ***B*** would be returned. The search for the solution would continue along the current path if ***C*** was not complete and ***C*** was less expensive than ***B***.

BranchAndBound (*T*, *C*) {

T ← **ReduceTable (*T*)**. (1)

if (|***C***| > ***u***) *return* (***B***); */bound is exceeded, return* (2)

if(all columns of ***T*** are covered) *return* (***C***); */solution achieved* (3)

For every frame vector F_i in ***T*** */consider each remaining frame* (4)

Add corresponding f_i to ***C*** and update vector C; (5)

T* = ***T*** after deleting F_i and the columns incident to it; (6)

C' = **BranchAndBound(*T**, *C*)**; */branch with frame F_i in the solution* (7)

If (|***C'***|) < ***u***){ ***B*** = ***C'***;} (8)

Remove F_i from ***C***; (9)

T* = ***T*** after deleting F_i; (10)

C' = **BranchAndBound(*T**, *C*)**; */branch after deleting F_i from solution* (11)

If (|***C'***| < ***u***){ ***B*** = ***C'***;} (12)

return (***B***); (13)

}

Figure 29 Pseudo code of the covering algorithm.

A new frame vector F_i is selected from ***T*** and added to ***C***. Frame vector F_i is removed from ***T***. Any frame vector dominated by F_i is removed from ***T***. All columns covered by F_i

are also removed. ***T*** is copied to ***T**** and the **BranchAndBound()** algorithm is now invoked on ***T****. The algorithm recurses until a new cover is found or until the current search for the new solution is found to be more expensive than ***B***. Correspondingly, ***C*** or ***B*** is returned. The returned solution is copied to ***C'***, and compared with ***B***. If |***C'***| is less than ***u***, then ***C*** is copied to ***B***, otherwise it is discarded. In any event, after this step, ***B*** holds the best solution seen thus far.

Frame vector F_i is now removed from ***T*** and from ***C***. ***T*** is copied to ***T**** and the **BranchAndBound()** algorithm is re-invoked on ***T****. When the algorithm returns with the solution ***C'***, ***C'*** is compared with ***B***. The better of the two solutions is stored in ***B*** and ***B*** is returned.

Thus the algorithm finds covering solution and progressively improves it. A frame vector F_i is first included in the solution and a cover is searched. If a better cover than the best solution is found, it replaces the best solution. Otherwise the best solution is preserved. In any case, since the solution has been explored with F_i included alternate solutions can now be explored with F_i removed. The first call to the **BranchAndBound()** algorithm looks for a cover with F_i included in the current solution ***C***. In the second call the algorithm removes F_i and searches for a new cover.

3.8. Evaluation parameters

The evaluation procedure for the results of the algorithm was adopted from the second Message Understanding Conference (1989). The evaluation parameters have been adopted from Grishman's work [Grishman96]. However, for the purposes of this thesis, the parameters have been modified. The parameters are calculated by finding covers for a large set of sentences.

We define *recall* in terms of skeleton token coverage for a set of sentences as

Recall = Total number of skeleton tokens covered / Total number of tokens in all the skeletons

Equation 1. Definition of Recall

We define *precision* in terms of frame matches.

Precision = Number of correct frames selected in covers/ Total number of frames selected in covers

Equation 2. Definition of Precision.

Both recall and precision are expressed as percentages. Precision is a more useful evaluation parameter than *recall* as it more directly bears on the system performance. The higher the *precision,* the better the covering using the repository of frames. However, *recall* has been predominantly used for analyzing the performance of this algorithm for the following reasons:

(1) Precision is subject to error, as it needs human judgement.

(2) Precision is hard to compute, as it has to be computed manually.

For instance consider the sentence *"processors waits for the controller to send the acknowledgement."* Figure 30 shows the skeleton and the cover for this sentence.

Skeleton: *"processor wait for controller to send acknowledgement"*
Frame 1:*wait (subject processor)*
Frame 2:*send (subject controller, object interrupt, to controller)*
Frame 3:*send (object acknowledgement)*

Figure 30 Skeleton and frames.

The skeleton tokens that are covered are underlined. Likewise, the tokens of the frames that cover a skeleton word are also underlined. From Figure 30 we note that

The total number of tokens in the skeleton = 7

The number of skeleton tokens covered = 5

Thus recall is calculated using Equation 2 as

Recall = (5/7) *100 % = 71.4%

The calculation of precision however involves the judgement of the individual attempting to calculate the parameter. The denominator of the equation 2 can be readily computed.

The total number of frames identified = 3

However the numerator, the number of correct frame matches, is subject to judgement. For instance in Figure 30, frames 1 and 3 can be considered as correct frame matches as the verb and all the cases of these frames match the tokens of the skeleton. Moreover, these frames reflect the correct semantics of the skeleton. However, frame 2 may or may not be considered as a correct frame match. This is because, the frame correctly identifies the subject and the verb in the sentence and hence it can be considered as a correct frame. But, the object and the other cases of this frame do not cover any of the semantics of the skeleton. Hence it could be considered as a bad frame. Thus using equation 2, the parameter precision can be calculated as *Precision* = (2/3)*100 %= 66.6%, or as *Precision* = (3/3)*100 % = 100%, depending on whether we consider frame 2 as incorrect or correct. However it can be seen that *Precision* reflects the actual performance better than *Recall*.

Chapter 4. IMPLEMENTATION

4.1. Design

The various processes discussed in Chapter 3 lend themselves to be conveniently implemented as an object oriented design that uses the C++ programming language.

The classes used in this thesis are a token class (CToker), a case class (CCase), a frame class (CFrame), a class to process the abstract sentence to a skeleton (CHierarchy), a frame vector class (CFrameVector), a skeleton class (CSkeleton), a cover table class (CTable) and a solution class (CSolution).

These classes are described in detail below. Many of the implementation details are hidden, and only the important methods and the attributes of each class are shown in the class diagrams. The order in which the classes and their member functions are discussed closely follows the system overview diagram discussion from chapter 1.

4.2. Class Overview

This section describes the classes of the covering algorithm using a notation called the Unified Modeling Language (UML) [Rational96]. Only an overview of the classes is presented here. The implementation of these classes, their attributes and their methods are explained in greater detail in the next section.

In UML class diagrams, a box represents a class. The box is labeled at the top with the name of the class. Class attributes and class member functions are listed below the name. A line is used to separate the attributes from the methods in these class diagrams. The "lock" icon is used to distinguish private attributes from the public attributes.

The classes can have two types of relationships between them. These relationships are the dependency relationship and the aggregate relationship. The dependency relationship between two classes is denoted by means of an arrow. The tail attaches to the client class and the head of the arrow points to the server class. A dependency relationship is used to denote a “using” association between two classes.

The aggregate relationship is used to show a whole or part relationship between two classes. The class at the client end of the aggregate relationship is called the aggregate class. The class at the supplier end of the aggregate relationship is the part whose instances are contained or owned by the aggregate class. An aggregate relationship is a solid line with a diamond at one end. The diamond end designates the aggregate class. The multiplicity of an aggregation link is indicated beside the link in the class diagrams. A “*” indicates that *n* instances of the member class are associated with one instance of the aggregate class. Only the important dependency relationships and the aggregate relationships are shown in these class diagrams.

Figure 31 shows the relationship among the classes CToker, CCase and CFrame. The CToker class is used get the tokens of the frames from an input file containing frames. These tokens are stored in CFrame and CCase objects. The CCase class represents a case of a frame. Each CCase object points to the next case in a frame. Thus a frame includes a linked list of CCase objects. The CFrame class represents a frame.

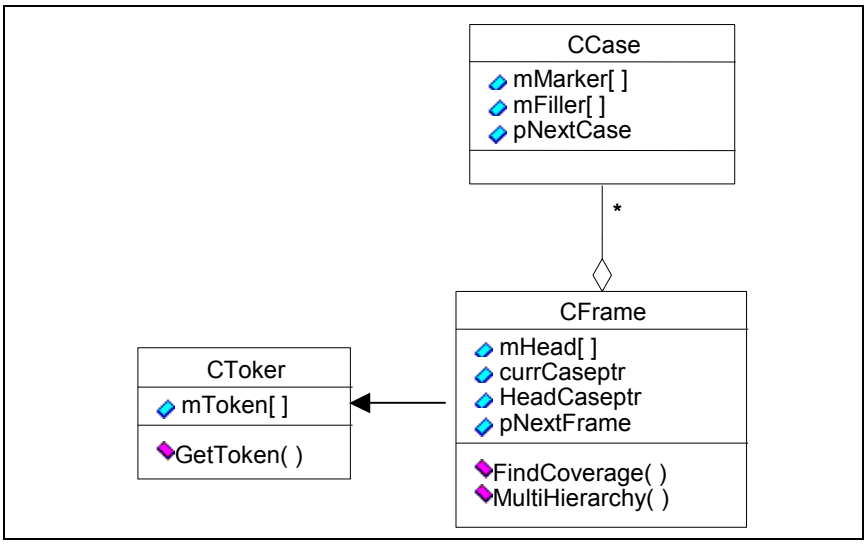

Figure 31 UML class diagram showing the relationships among the classes CToker, CCase and CFrame.

Figure 32 shows the relationships among the classes CSkeleton, CFrame and CHierarchy. The CSkeleton class represents a skeleton. Each token of the abstract sentence is stored in a CSkeleton object that has a pointer to the next CSkeleton object. In this way, the CSkeleton object is used to make a linked list of the sentence that is read from an input file. These sentences are in fact abstract sentences output by the Chunker.

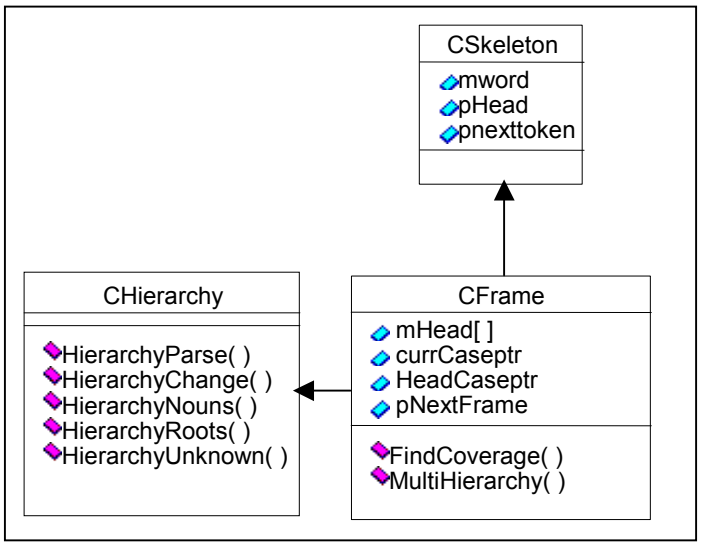

Figure 32 UML class diagram showing the relationships among the classes CSkeleton, CFrame and CHierarchy.

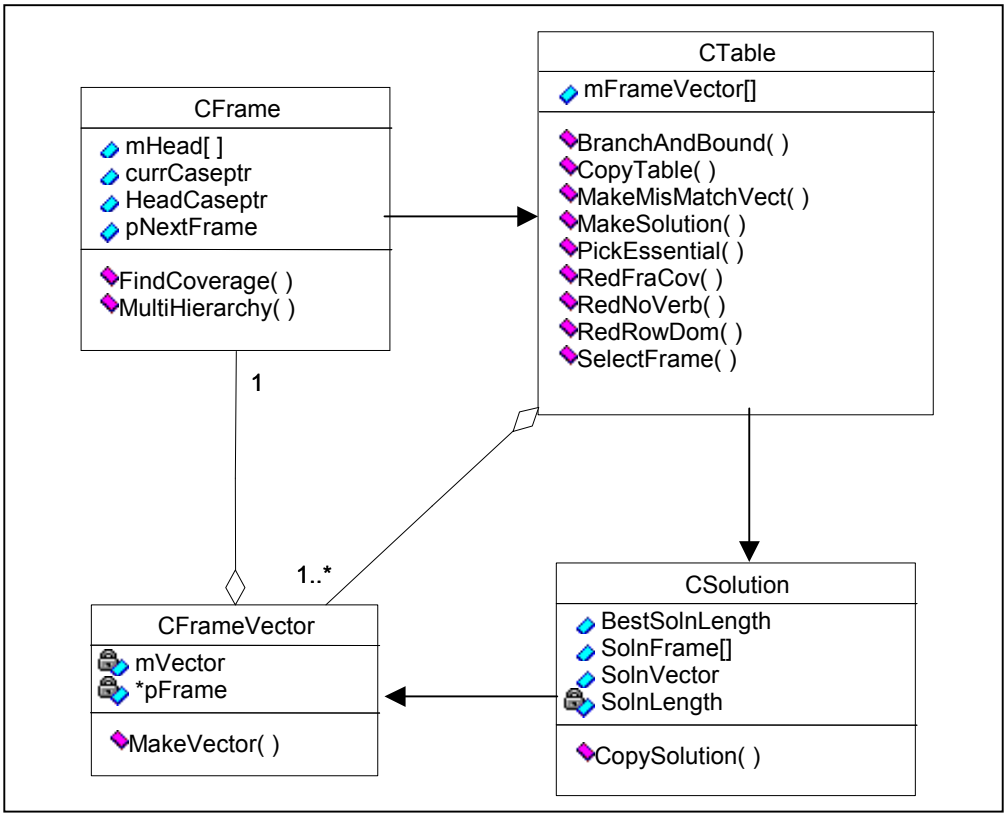

Figure 33 UML class diagram showing the relationships among the classes CFrame, CFrameVector, CTable and CSolution class.

The CFrame class uses the CHierarchy class to process the abstract sentence to generate the skeleton. The CHierarchy class provides the methods required for processing the abstract sentence.

For each frame in the repository, the skeleton is used to generate a frame vector. The CFrameVector class is used to represent a frame vector. Each CFrameVector object contains a pointer to the corresponding frame. An aggregation of frame vectors forms a covering table. A table is represented using the CTable class. The CFrame class uses the CTable class to find a covering solution for the skeleton. The CTable class uses the CSolution class to store the generated solutions. Figure 33 shows the relationships among the CFrame, CFrameVector, CTable and CSolution classes. In the next section the implementation of the covering algorithm is explained in detail.

4.3. Methods

This section explains the various methods used in the implementation of the approach discussed in chapter 3. The explanation closely follows the system overview diagram (Figure 1). The program begins with the main() routine that calls ReadSkeleton() function and terminates. The ReadSkeleton() function calls the ReadFrames() function for each abstract sentence in the input file. The ReadFrames() function reads frames from an input file containing a list of frames.

In this thesis two frame repositories have been used. These repositories store the frames in the form: *subj verb obj, list of other cases*. The first of these files, "frames1.txt", contains about 1600 frames. A second file "frames2.txt" contains about 2400 frames. This file was generated automatically with a parser using a context free grammar.

The ReadFrames() function calls the GetToken() method of the CToker class to read each token from a frame file. The GetToken() method extracts one character at a time from the input file and assembles the tokens using ".", ",", "&", "\t", "\n" and " " as the delimiting characters. But since the punctuation characters are also tokens, the method stores these tokens and returns them when the GetToken() method is called the next time.

A verb token is stored in the mHead attribute of the CFrame object. The remaining tokens are stored in the mMarker or the mFiller attributes of a CCase object depending on whether they are markers or fillers of a case. The CFrame object has a pointer to the first case in the frame. Each CCase object points to the next case in the frame. Thus a linked list of CCase objects is used to store the cases of the frame. The CFrame objects have a pointer (pNextFrame) to the next frame in the repository. Thus the repository is stored in a linked list of CFrame objects.

After loading the frames, the ReadSkeleton() function processes the abstract sentence read from the input file. Each abstract sentence is tokenized using white space as the delimiter. A CSkeleton object is used to store an extracted token. Each CSkeleton object has a pointer to the next token of the abstraction and thus forms a linked list.

For each abstract sentence in the input file, the ReadSkeleton() function invokes the FindCoverage() method with the pointer of the first frame in the repository and the pointer of the abstract sentence as arguments. FindCoverage() method is a member of the CFrame class and is used to find a covering solution for the sentence. The FindCoverage() method, generates a skeleton for the abstract sentence. This is done using the HierarchyParse() method of the CHierarchy class.

The HierarchyParse() method processes the abstract sentence in a number of stages to generate the skeleton. These stages of processing are in turn implemented as methods of the CHierarchy class. In the first stage of processing, the "noisy" tokens are eliminated. A file containing a list of noisy tokens is used to identify such tokens. This file was generated by the Chunker and contains a list of tokens that the syntactic parser failed to recognize. Such tokens include words outside the vocabulary of the parser used by the Chunker, numbers, acronyms, mis-spelt words and notational words such as "a0-a7". Some additional words were added to this list. These words include conjunctions and tokens such as ")" and "(". The elimination of noisy tokens improves covering results. The HierarchyUnknown() method is used to remove such noisy tokens.

In the second stage, the abstract sentence is processed to normalize the nouns and the verbs to their root morphology. These processes are handled by the HierarchyNouns() and HierarchyRoots() methods respectively. The nouns in the abstract sentence are normalized by replacing the plural noun forms by their equivalent singular forms. Appendix C shows a file containing a list of nouns and their root forms. This was used by the HierarchyNouns() method to normalize the nouns in the abstraction. Verb abstraction involves replacement of non-root form verbs of the sentence by their equivalent root forms. Appendix D shows a file containing a list of verbs and their root form used by the HierarchyRoots() method.

In the next stage of processing, the HierarchyChange() method is used to semantically generalize the abstraction words. Semantic generalization allows the generalization of subtype concepts to their corresponding supertypes as dictated by the type hierarchy. The HierarchyChange() method is used to implement the semantic generalization of the

abstraction words. Any subtype word is automatically generalized to its supertype if it is present in the type hierarchy file (see Table 2).

Processing the abstract sentence thus generates a skeleton. The tokens of the frames are also semantically generalized. The MultiHierarchy() method of the CFrame class is used for the same. The MultiHierarchy() method implements the generalization of the frame tokens as explained in section 3.6.

FindCoverage() method uses the MakeVector() method to form the frame vectors of the frames. The MakeVector() method is a member of the CFrameVector class. A frame vector is generated based on the token matches between the skeleton and the frame. The frame vectors are unsigned integers of fixed length. The bit representation of the vectors was preferred over integer array representation in order to save memory. Consequently, the processing speed is also reduced as binary operations consume less time compared to loop implementation that would have been required had the vectors been implemented as integer arrays. Unlike an array implementation where the bits can be directly accessed, the bit vector representation using unsigned integers does not allow direct access to the individual bits. Consequently, methods were needed to abstract the lower level details of dealing with bits.

The frame vectors are stored in the mVector attribute of a CFrameVector object. An array of CFrameVector objects, mFameVector[], forms a CTable object, mTable . Initially, mTable is formed using all the frames in the repository. However, since only verb frames are considered as plausible candidates for a cover, any frame vector whose corresponding frame is not a verb frame, is removed from the table. The size of mTable is thus reduced. The method RedNoVerb() of the CTable class is used here.

While the abstract sentence has been processed to eliminate the noisy tokens to generate the skeleton, there may still be some tokens in the skeleton that are not in the frames' vocabulary. The frames will not cover such skeleton tokens. Consequently the algorithm will not return a solution as it returns only exact solutions. In order to enable the algorithm to return the best cover that it can find, the tokens that will not be covered by the frames are pre-covered using a vector called the mismatch vector. This vector covers

all the elements of the skeleton vector that will not be covered by the frames. The mismatch vector is generated using the method MakeMisMatchVect(). This vector is added to the solution and thus all the columns covered by this vector are eliminated from the covering problem.

The FindCoverage() method invokes a call to the BranchAndBound() method of the CTable class to obtain a solution to the covering problem. The BranchAndBound() algorithm is invoked with mTable and a CSolution object, CurrentSolution (***C***), as arguments. The attribute SolnVector of the CurrentSolution object is used to store the solution vector (C). It is initialized with the MisMatchVector. The attribute SolnLength (|***C***|) stores the size of the solution in terms of number of frames in the solution. The SolnLength of CurrentSolution is initialized to zero. The CurrentSolution stores the covering solution that is being attempted at any given time.

BestSolution (***B***), which is also a CSolution object, contains the best solution seen by the algorithm thus far. The size of BestSolution is stored in a static integer, BestSolnLength (***u***). The BestSolnLength is initialized to the length of the skeleton since there cannot be a minimal covering solution for the skeleton whose size (number of frames in the solution) is greater than the number of skeleton tokens. Such a high value of BestSolnLength is chosen to ensure that as the algorithm proceeds, the BestSolution can be progressively improved. The BestSolnLength serves to bound the algorithm. The attribute SolnVector of the BestSolution object is used to store the solution vector.

The BranchAndBound() method receives mTable as Table (***T***). The algorithm starts out by reducing the size of Table. Table is reduced by invoking the methods RedRowDom() and PickEssental() in a loop. The RedRowDom() method removes the dominated rows from Table. The PickEssential() identifies the essential frame vectors in Table. These frame vectors are removed from Table and added to the solution CurrentSolution. All columns covered by these frame vectors are eliminated from Table by zeroing all the elements of the corresponding columns. The process is repeated until no new essentials are picked.

Since the essentials were added to the solution, CurrentSolution now contains a cover that may be partial or complete. The size of this cover is stored in the variable CurrentEstimate. If the CurrentEstimate is greater than or equal to BestSolnLength, then, BestSolution is returned regardless of whether CurrentSolution is partial or complete. This is because the search is only for a solution better than BestSolution. If, however, CurrentSolution is better than BestSolution, the search continues. At this point, the algorithm may or may not return depending on whether CurrentSolution is partial or complete. If all columns of Table are covered then CurrentSolution is complete and this solution is returned. However, if the cover is partial, then the search for a cover along this path continues.

A new frame vector, say $\mathbf{F_i}$, is picked. This frame vector is the first frame vector in Table. Frame vector $\mathbf{F_i}$ is added to CurrentSolution. Table is copied to a new table called TablePrime after deleting $\mathbf{F_i}$ and all the columns that it covers. The method RedFraCov() is used to delete all the columns covered by $\mathbf{F_i}$. The method CopyTable() is used to copy Table to TablePrime (T^*).

The BranchAndBound() method is invoked on this new table, TablePrime. The algorithm recurses until a cover is found or until the current search proves to be more expensive than BestSolution. When either of these conditions are met the algorithm returns with a cover, which is either BestSolution or CurrentSolution depending on which one was better. The returned solution is stored in CurrentSolutionPrime (C'), which is again a CSolution object. The size of CurrentSolutionPrime ($|C'|$) is stored in CurrentEstimate and compared with the BestSolnLength. If CurrentEstimate is smaller, then BestSolution is replaced by CurrentSolutionPrime.

In the above discussion, a cover was attempted with the frame vector $\mathbf{F_i}$ included in the solution. Now the frame vector should be deleted from the solution and covers with other frame vectors should be considered. Thus, Table is again copied to TablePrime after deleting the frame vector, $\mathbf{F_i}$. However, unlike the last time, the frame vectors dominated by $\mathbf{F_i}$ are not deleted from Table. $\mathbf{F_i}$ is also deleted from CurrentSolution.

The BranchAndBound() method is reinvoked on this new table, TablePrime. Again the size of the returned solution, CurrentEstimate is compared with BestSolnLength. BestSolution is

replaced by the newly returned solution, CurrentSolution, if its size CurrentEstimate is smaller than BestSolnLength.

In this way, the algorithm explores the search space to find a minimal solution for a given ordering of the frame vectors. Once a solution is found, the next abstract sentence is read by the ReadSkeleton() method and this process is repeated until the abstract sentence file is exhausted.

4.4. Test Files

The covering algorithm was tested using a file "details.txt" (see Appendix E) that included multiple patents involving DMA controllers. This file was a specimen of the type of patents for which this system is intended. The abstract sentences generated by the Chunker for the sentences in this file were incomplete and incomprehensible. The file required some processing before it could be used. Three digit numerals (e.g. 105 for timer, 106 for controller, 107 for line and 108 for SDMA) were used to identify the *elements* (e.g. Timer, controller, line, SDMA) in this file. The problem was that instead of the elements, the numerals were being identified as the subject or the object. This resulted in the Chunker eliminating the elements as noise and generating incomprehensible skeletons. This problem was alleviated by deleting all the words that contained numerals from the original sentence file.

The abstract sentences output by the Chunker were incomplete because the sentences in the "details.txt" file used the period as a short-form word terminator (e.g. FIG.) as well as an end-of-sentence marker. The Chunker uses period as the end-of-sentence marker. Hence, in a number of cases (about 30%) the abstractions output by the Chunker were incomplete as the Chunker could not distinguish the short-form words terminator from the end-of-sentence marker. Often the abstractions consisted of only one or two words. In order to improve the test file, all abstractions whose size were less than twenty-five characters, were deleted from the test file.

Two files containing frames were used to cover the skeletons. One file was generated manually and the other file was generated automatically. The automatically generated file, “frames2.txt” (Appendix A.2), contained more than 80% of frames that were “noisy”. Since these frames were generated from patents, they contained a lot of tokens that were alphanumeric strings rather than elements. The frames thus required cleaning. The frames were cleaned using simple C++ programs and a spreadsheet. The numerals were replaced by their corresponding elements. In some cases (<10 %) the replacements were incorrect.

Chapter 5. RESULTS

Two sets of frames were used to cover the sentences in the test file "details.txt". The first file "frames1.txt" (Appendix A.1) contained about 1600 frames that were generated manually from 8-bit microprocessor product data sheets. The second file "frames2.txt", containing about 2400 frames, was generated automatically using a number of DMA controller patents as input files.

The multi-level hierarchy file used in this thesis is shown in Appendix F. Figure 17 shows a more complete version of this file. The tests presented in this chapter were performed on the multi-level hierarchy file shown in Appendix F, and this file was subsequently completed as shown in figure 17. Since the two versions of the multi-level hierarchy file produced similar results and the change in the values of recall and precision were not appreciable, only the results using the multi-level hierarchy file shown in Appendix F are presented here.

The following sections provide some examples of covers and describe some of the tests that were conducted to evaluate the performance of the system.

5.1. Example Covers

Figure 34 shows five examples of covers. Each example contains a sentence followed by the skeleton and a set of frames that covers the skeleton. In these examples, Chunker was

not used to generate the skeletons. The skeletons were manually generated by combining two or more frames from the files “frames1.txt” and “frames2.txt”. These skeletons are used to demonstrate the performance of the algorithm on skeletons that are free from errors produced by Chunker.

Example 1
Sentence: move the content from register and load the controller with value.
Skeleton: move content from register load controller with value
Frames:
load (object controller, with value)
move (object content, from register)

Example 2
Sentence: use the instruction to test bit, if the condition is true then take action and clear bit after test .
Skeleton: use instruction to test bit if condition is true take action clear bit after test
Frames:
clear (object bit, after test)
is (subject condition, object true)
take (object action, in instruction)
use (object instruction, to test)

Example 3
Sentence: select operation to divide operand A by operand B.
Skeleton: select operation to divide operand by operand
Frames:
divide (object operand, by operand)
select (object operation)

Example 4
Sentence: add operand A to operand B and take the complement by passing the output through inverter.
Skeleton: add operand to operand take complement by pass output through inverter
Frames:
add (object operand, to operand)
pass (object output, through inverter)
take (object complement)

Example 5
Sentence: maintain the flip-flop at zero level until the processor makes a decision .
Skeleton: maintain flip-flop at level until processor make decision
Frames:
maintain (object flip-flop, at level)
make (subject processor, object decision)

Figure 34 Example covering results (I)

The sentences were generated manually by completing the skeletons. The skeleton tokens that are covered by the frame tokens are underlined. The frame tokens that cover the skeleton tokens are also underlined. It can be seen that in all the covers, the correct frames have been identified by the system, i.e., the semantic content of the frames

strongly correlates with the semantic content of the skeleton. For these sentences, the precision is calculated to be 100 % and the recall about 95 %.

Figure 35 shows the covering results obtained on sentences from multiple patents. In this figure the original sentences from the patent are shown. The abstractions however, were generated using the corresponding processed sentences from the "details.txt" file. The sentences were processed as described section 4.4. In order to illustrate how precision was calculated in the previous examples, the frames that are considered "incorrect" have been identified using a lighter font.

In example 1, since frames 1 and 2 have poor semantic correlation with the skeleton, they have been identified as incorrect. Frame 4 is an example of an automatically generated frame. Notice that the subject in this frame is "d", which is incorrect. The correct subject is "signal".

In example 2, the Chunker had deleted the subject "operator" as noise. Hence the subject in frame 2 could not cover the token, which results in a lower value of recall. The last frame in example 3, is meaningless. Such frames need to be deleted from the repository. In example 5, the subject "SDMA" was deleted as noise by Chunker. The token "cpu" is generalized using a two-level hierarchy to its supertype "processor".

The following sections describe some of the tests conducted on the system to evaluate its performance. These tests use the parameters recall and precision to evaluate system performance. The definitions of recall and precision introduced in section 3.8 are restated below.

Recall is defined as *the ratio between the total number of skeleton tokens covered and the total number of tokens in all the skeletons*, and precision is defined as *the ratio between the number of correct frames selected in covers to the total number of frames selected in covers.* Both recall and precision are calculated over a large number of skeletons. They are expressed as a percentage.

Example 1
Sentence: The signal D WRITE from the SDC control bus is passed through an inverter 808 to become the signal D WRITE A.
Abstract: signal write from the bus is passed through inverter to become signal write a .
Skeleton: signal write from bus is pass through inverter to become signal write
Frames:

become (subject bus, object free)
is (subject ack, object signal)
pass (object signal, through inverter)
write (subject d, from bus)

Example 2
Sentence: A similar function is accomplished when the operator depresses a reset key on the control panel.
Abstract: function is accomplished when depresses key on panel .
Skeleton: function is accomplish when depress key on panel
Frames:

accomplish (object function)
depress (subject operator, object key, on panel)
is (subject (IOPORTS), object applicable, to modes)

Example 3
Sentence: Flip-flop 842 is also reset when the signal BUS GRANT is terminated and the lead 832 drops to the low level.
Abstract: flip-flop is also reset bus grant is terminated lead drops to level .
Skeleton: flip-flop is also reset bus grant is terminate lead drop to level
Frames:

drop (subject lead, to level)
grant (subject processor, object bus)
is (subject but, object also)
is (subject flip-flop, object reset, at time)
terminate (object bus)

Example 4
Sentence: During the set-up sequence the CPU generates an input instruction followed by three output instructions.
Abstract: during sequence cpu generates instruction followed by three instructions .
Skeleton: during sequence processor generate instruction follow by three instruction
Frames:

follow (subject instruction, object add)
generate (subject processor, object instruction)

Example 5
Sentence: If the SDMA is busy its status is returned to the accumulator
in the CPU.
Abstract: if the is busy status is returned to accumulator in cpu .
Skeleton: if is busy status is return to accumulator in processor
Frames:

is (subject controller, object busy)
return (object status, to accumulator, in processor)

Figure 35 Example covering results (II)

5.2. Variation of Recall with Number of Sentences

Two experiments were conducted to evaluate the system performance in terms of recall. In the first case, a cover was attempted on the sentences of the patent using the "frames1.txt" file. The program was run on 500 abstractions from the test file. The recall is calculated as

Recall = (2794/6262)*100 % = 44.61 %

The frames from both files (frames1.txt and frames2.txt after cleaning as explained in section 4.4) were used to generate a file "frames3.txt" (Appendix A.3). The "frames3.txt" file had a total of 3459 frames after deleting duplicate frames. The recall, in this case was found to be

Recall = (4219/6233)*100 % = 67.68 %

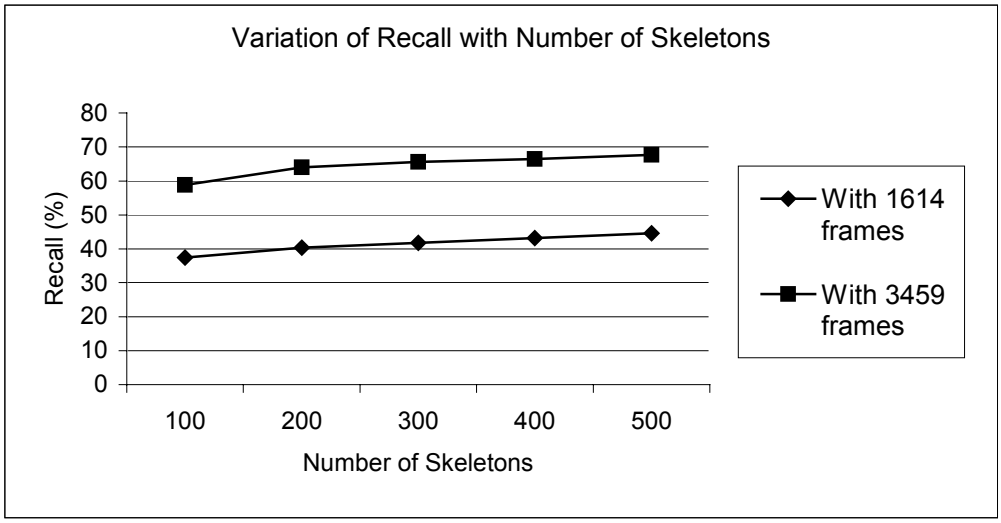

Figure 36 Variation of Recall with Number of Skeletons (First 500 skeletons)

Figure 36 shows the variation of recall with the number of skeletons. The recall was calculated for the first 100 skeletons of the test file, then the first 200 skeletons of the test file and so on. Two curves were plotted, one each corresponding to the two frames file "frames1.txt" and "frames3.txt". It can be observed that the results obtained with "frames3.txt" are about 20% better than the results obtained with "frames1.txt".

Recall linearly increases with the number of skeletons. This is because the sentences of the input file contained the sentences in the same order as they were found in the patent. Many sentences at the beginning of the patent were introductory sentences that the system was not intended to cover. This accounts for the lower initial recall value.

Another experiment was conducted to validate this observation. In this experiment, the recall was calculated for sentences towards the end of the patent. Figure 37 shows the drooping characteristic that was observed. The recall decreased because fewer sentences at the end of the file tended to be microprocessor related.

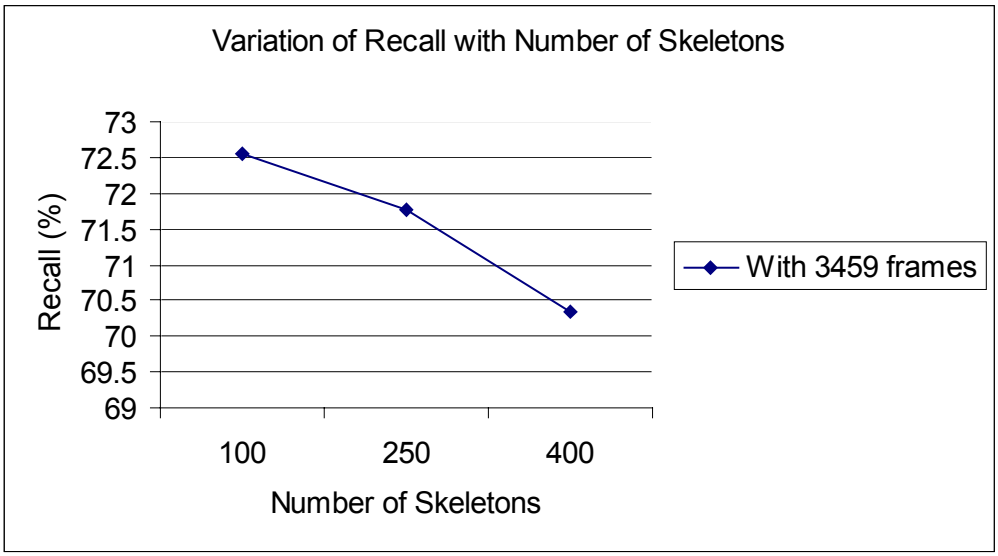

Figure 37 Variation of Recall with Number of Skeletons (Last 400 skeletons)

5.3. Variation of Recall with Number of Frames

The number of frames in the repository was increased from 432 to 3459 in steps of approximately 435 frames. In each step, the frames were selected at random from the file "Frames3.txt" while ensuring that previously selected frames were not selected again.

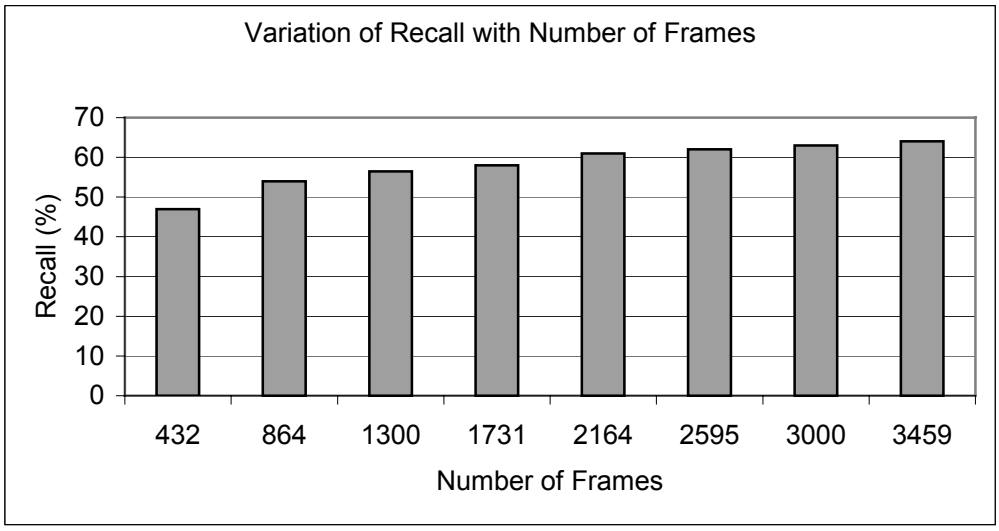

Figure 38 Percentage Recall Vs. No. of Frames.

For these differing sizes of the repository, the recall was computed and the trend was observed. Figure 38 shows a plot of percentage recall versus number of frames in the repository. As expected, the recall showed an increase with increase in the number of frames in the repository. This is because the algorithm has a better range of frames to choose from, when the number of frames is increased. Also the vocabulary of the repository increases when more frames are added, facilitating better coverage. However, this increase in vocabulary of the repository is not proportional to the increase in the number of frames in the repository. For instance, a random selection of frames that amounted to 12% (432 frames) of the full size of the repository (containing 3459 frames) contained 36% of the vocabulary of the repository. The vocabulary of the repository with 3459 frames in it is 1186 words. Hence when more frames are added to the repository, the improvement in the quality of the frames (precision) is more significant than the increase in percentage recall. This can be inferred from Figure 39.

Figure 39 shows the contribution from the verbs, subject case, object case and other cases to the total number of verbs and cases of the frames that cover the skeletons. Due to the constraint that the frames forming the cover should necessarily match the verb in the skeleton, the coverage due to verbs records the highest percentage (approximately 50%). As the number of frames in the repository was increased the contribution of the verbs fell from 56.5% to 47.15% even though the total number of skeleton tokens covered

increased from 47% to 64 % (See Figure 38). The contribution of the subject case fell by marginal 1.1% and the contributions from the object case and other cases increased from 19.5% to 26.5% and 6.2% to 9.75 % respectively. For the same set of skeletons, frames with more case matches were identified as the number of frames in the repository was increased. This indicates that the quality of frames that formed the cover improves as the number of frames in the repository increases.

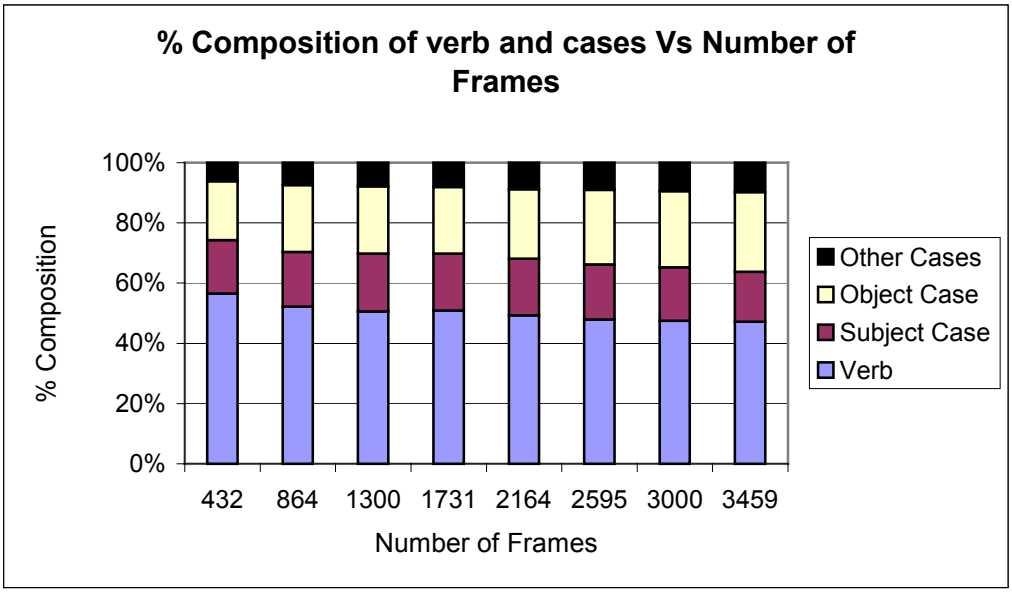

Figure 39 % Composition of verbs and cases in the total number of frame tokens covering the skeleton tokens.

5.4. Performance using hierarchy

The effects of the hierarchy were analyzed over 200 abstract sentences generated by the Chunker. First, the performance parameter recall was calculated for the skeletons without using multi-level hierarchy for the frames and without using the two-level hierarchy for the skeletons. Then, the frames were semantically generalized using the multilevel hierarchy and skeletons were generalized using a two-level hierarchy and the recall was calculated. Figure 40 shows the generated plots for increasing number of sentences and with 3459 frames in the repository. Appendix F shows the Multi-level hierarchy used in this experiment.

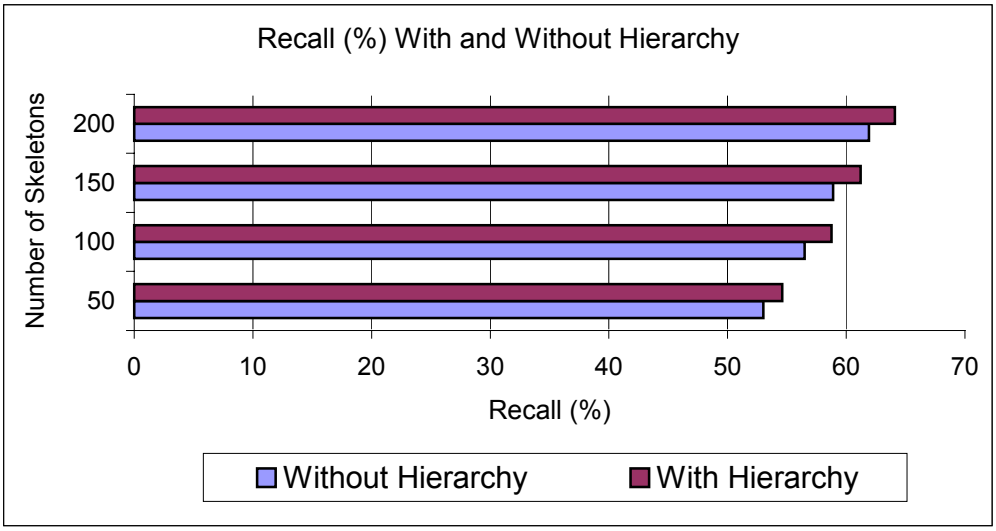

Figure 40 Performance improvement using hierarchy.

On an average, recall improved by about 3% when hierarchy was used on the tokens of the frames and the skeletons. The average was computed over increasing number of sentences as shown in Figure 40. The improvement in percentage recall was not appreciable because of the small size of the two-level hierarchy file and the multi-level hierarchy file.

5.5. Precision

In order to calculate the precision, an output file containing about 200 skeletons and their corresponding covering frames was first generated. Fifty sentences (that were the kind of sentences that this system was meant to cover) were selected and the precision was calculated on their skeletons. A frame was "adjudged" a "correct frame" if the semantic content of the frame strongly correlated with the semantic content of the skeleton that it covered. The author of this thesis judged all the covers. The total number of correct frames and the total number of frames in all the covers were summed over 50 skeletons. Precision was calculated using equation 2 as

Precision = 102/188 *100 % = 54.25%

This value of precision indicates that one in every two frames in the cover is “incorrect.” A number of reasons can be attributed to this poor precision percentage. In a number of cases only the verbs of the covering frame matched the skeleton tokens, such frames were considered incorrect frames by the judge. Often such verbs were common verbs such as “is”, “be” and “or” (the third frame in example 2 in figure 35 is an instance of such a frame selected to be part of the cover). This indicates that more frames are needed in the repository in order that the cases also cover the skeleton tokens. Precision can be improved by imposing the constraint that a verb and at least one case of a frame should match the skeleton tokens. This will eliminate arbitrary frames from being selected as a cover for the skeleton.

One other reason for a poor precision percentage is that many of the sentences were incomplete. This was due to the limitations of the Chunker software. An incomplete sentence is one in which the subject, verb or object may be missing. An attempt to cover incomplete sentences often results in incorrect frames forming the cover.

In many cases the sentences were long and complex. Such sentences may contain multiple verbs, subjects or objects. In such cases, the subjects and objects may be matched with the wrong verb. This results in the selection of an incorrect frame as part of the cover.

Sometimes in the patent, words such as “AND”, “OR” and “NOR” were used to represent the respective gates. The system identified these words as conjunctions and thus eliminated them. This resulted in incomprehensible skeletons that the system could not cope with.

It can be seen that the performance of the system significantly degrades when sentences from the patents are used. This degradation in performance is attributed to a number of reasons, some of which have been identified in the preceding sections. The next chapter identifies some additional reasons that limit the system performance. Some suggestions to improve the system performance are also recommended.

Chapter 6. CONCLUSION

This chapter discusses the advantages and the disadvantages of the covering approach to semantic decomposition of sentences. Some suggestions to improve the system performance are also recommended.

6.1. System Advantages

Conventional systems use a parser and rely heavily on the grammatical analysis of the sentences of the document for its interpretation. Grammatical analysis of sentences can fail due to a number of reasons. Incomplete sentences, bad sentence constructs, typological errors can cause the grammatical parse to fail. This covering approach that is grammar-free can be a useful back up in such cases.

The covering approach is simple and robust. While conventional approaches can sometimes fail completely due to insufficiency of grammar rules, the covering approach provides some results although they may not be accurate. The covering approach is limited only by the repository of frames for providing covers. The approach uses a standard branch and bound algorithm for sentence covering that will provide good covering results if the semantics of the sentence are contained in the repository of frames used by the system.

The covering approach can be used on the sentences of any type of document (not necessarily technical patents) if an appropriate repository of frames is available. The

repository of frames should contain frames whose semantics correlate strongly with the semantics of the sentences of the document that the system seeks to cover.

The performance of the system can be improved by increasing the number of frames in the repository. Alternately, performance can also be improved by using an expanded hierarchy file.

The covering approach to sentence decomposition is fast. Most sentences were decomposed within a second (typically ½ a second). Sentences that required semantic generalization required more processing time (typically 2-3 seconds).

6.2. System Limitation

The performance of the system is mostly constrained by the repository of frames. For sentences whose semantics are not contained in any of the frames in the repository, the covering results can be poor.

Since the covering approach is grammar-free, in the case of a long sentence that contains multiple subjects, objects and verbs, a semantically incorrect frame whose tokens match the tokens of the skeleton may be identified as a candidate for cover.

6.3. Future Work

The frames file requires updating. Duplicate frames need to be removed. Some frames are inherently dominated by other frames, such frames need to be removed from the file. One approach that can help in fine tuning the frames file is to generate the hit frequency of each frame in the file for a large number of sentences. Frames that have a low hit rate may be removed.

The number of frames in the repository needs to be increased. A direct consequence of this is that it improves results. However, more important than the number of frames added is the quality of frames added. Addition of frames that cover new concept types can be useful. An alternative to increasing the number of frames in the repository is to use a more complete generalization hierarchy. This thesis uses a primitive two level generalization hierarchy for the skeleton that can be improved to a multilevel hierarchy as outlined in section 3.6. The proposed hierarchy needs to be multi-level, expanded, and should take the semantic distance between the various concepts of the generalization hierarchy in deciding the best concept abstraction for a given concept.

The files that convert the verbs to the root form and the nouns to the singular form also need to be improved by adding more entries.

The proximity of the tokens of the frame in the skeleton can be used as a weighting factor in the selection of frames for the covers. Frames whose tokens lie in close proximity to each other are more likely to be “correct” frames than frames whose tokens are separated by other tokens in the skeleton.

References

[Baker81] Baker Edward K., "Efficient Heuristic Algorithm For the Weighted Set Covering Problem," *Comput. & Ops Res.*, Vol. 8, No.4, pp.303-310, 1981.

[Dasigi94a] Dasigi Venu, "Logical Form Generation as Abduction: Part 1. Representation Linguistic Concepts," *International Journal of Intelligent Systems*, Vol. 9, pp.571-608, 1994.

[Dasigi94b] Dasigi Venu, "Logical Form Generation as Abduction: Part 2. A Dual Route Parsimonious Covering Approach," *International Journal of Intelligent Systems*, Vol. 9, pp.609-651, 1994.

[DeMichelli94] De Michelli. Giovanni, *Synthesis and Optimization of Digital Circuits.* New York: McGraw-Hill, 1994.

[Fillmore68] Fillmore, C.J. "The Case for Case," *Universals in Linguistic Theory*. Ed. E. Bach and R.T. Harms. New York: Holt, Rinehart and Winston, 1968.

[Grishman86] Grishman Ralph, *Computational Linguistics*, Cambridge: Cambridge University Press, 1986.

[Grishman96] Grishman Ralph and Sundheim Beth, "Design of MUC-6 Evaluation," *Advances in Text Processing: Tipster Program Phase II*, Morgan Kaufmann, 1996.

[Karp72] Karp R.M., "Reducibility among combinatorial problems," *Complexity of Computer Computations*, Ed. R.E.Miller and J.W. Thatcher. New York: Plenum Press, 1972.

[Lin95] Lin Dekang, "University of Manitoba: Description of the PIE System Used for MUC-6," *Proceedings of the Sixth Conference on Message Understanding (MUC-6)*, Columbia, Maryland, 1995.

[McCluskey56] E.J. McCluskey, “Minimization of Boolean Functions,” *Bell Syst. Tech. J.*, Vol.35, pp.1417-1444, Nov.1956.

[Minsky75] Minsky Marvin, “A Framework for Representing Knowledge,” *The Psychology of Computer Vision.* Ed. Winston H. Patrick. New York: McGraw-Hill, 1975.

[Nagao96] Nagao Makoto, “Varieties of Heuristics in Sentence Parsing,” *Recent Advances in Parsing Technology*. Ed. H.Bunt and M.Tomita. Dordrecht: Kluwer Academic Publishers, 1996.

[Rational96] Rational Software Corp. Rational rose 4.0.3, http://www.rational.com/uml, 1991-1996.

[Rudell87] Rudell Richard L. and A. Sangiovanni-Vincentelli, “Multiple-Valued Minimization for PLA Optimization,” *IEEE Trans.Computer-Aided Design*, vol. CAD-6 NO.5, pp. 727-750, Sept. 1987.

[Sowa84] Sowa, John F., *Conceptual Structures: Information Processing in Mind and Machine.* Reading, Mass.: Addison-Wesley, 1984.

Appendix A. Frames

This appendix lists a set of randomly chosen frames from three different files. These frames are of the form *verb(subject subj, object obj, list of cases)*. This is the form in which the frames are stored in the linked list. The frames in all three files include both mixed frames and instance frames. One example of a mixed frame has been identified by "**bold**" font in A.1.

A.1. Frames1.txt

This file contains frames that were manually generated. The first 25 frames from the file are listed below:

1 **be (subject cycle , object indivisible , in device)**
2 abort (object instruction)
3 accept (subject bus , object data , in length)
4 accept (subject bus , object data)
5 accept (subject device , object data)
6 accept (subject operation , object size)
7 accept (subject alu , object word , from source)
8 access (object area)
9 access (subject program , object code)
10 access (object data , in order , from processor)
11 access (object data , on boundary)
12 access (subject processor , object device)
13 access (subject program , object information)
14 access (subject code , object instruction)
15 access (subject address , object memory , if be)
16 access (object memory)
17 access (object memory)
18 access (object ram , faster-than memory)
19 access (subject microprocessor , object register , using pin)
20 access (subject processor , object register)

```
21 access (  subject fetch ,  object rom ,  if hold )
22 access (    object space )
23 access (    to memory )
24 accomplish (    object arithmetic ,  using instruction )
25 accomplish (  subject loop ,  object delay )
```

A.2. Frames2.txt

The frames in this file were automatically generated. A partial listing of the frames from "Frames2.txt" file is shown below. It can be seen that more than 80% of these frames are unusable in their current form.

```
 1  generated (    object b ,  by decoder )
 2  adapted (    object d )
 3  differs (  subject error ,  object in ,  , in )
 4  is (  subject error ,  object bit )
 5  provided (    object 01uf )
 6  produces (  subject 622 ,  object signal )
 7  takes (  subject 701 ,  object place )
 8  associated (    object 702 ,  with mar )
 9  signals (  subject 1090 ,  object to )
10 shows (  subject 10a ,  object format )
11 shows (  subject 10b ,  object configuration )
12 shows (  subject 10c ,  object format )
13 shows (  subject 10d ,  object format )
14 shows (  subject 10e ,  object configuration )
15 is (  subject 1185 ,  object input )
16 are (  subject mar ,  object read )
17 provided (    object 1865 ,  from source )
18 applied (    object pulse ,  to input )
19 goes (  subject 1a ,  object to )
20 applied (    object 1a ,  to 821 )
21 activated (    object 1y ,  ,  from 1y0 )
22 are (  subject command ,  object commands ,  for devices )
23 derived (    object req ,  from sdc )
24 shown (    object 200 )
25 shown (    object configuration ,  in table ,  in fig )
```

A.3. Frames3.txt

The frames from this file include frames from both "frames1.txt" and "frames2.txt". The frames from "frames2.txt" were processed before being added to this file. This file does not contain any tokens with numeric characters in them. The first eight frames are some

examples of frames that were processed and added to this file from the “frames2.txt” file (See A.2). A partial listing of the frames from “Frames3.txt” file is shown below.

```
1  generated (    object b ,  by decoder )
2  adapt (    object d )
3  differ (  subject error ,  in function ,  in mode )
4  is (  subject error ,  object bit )
5  provide (    object having )
6  produce (  subject AND ,  object signal )
7  take (  subject circuit ,  object place )
8  associate (    object circuit ,  with mar )
9  accept (  subject alu ,  object word ,  from source )
10 accept (  subject bus ,  object data ,  in length )
11 accept (  subject bus ,  object data )
12 accept (  subject device ,  object data )
13 accept (  subject encoder ,  object signal ,  at one )
14 accept (  subject operation ,  object size )
15 accept (  subject sdc ,  object byte )
16 access (    to memory )
17 access (    object area )
18 access (    object content )
19 access (    object data ,  in order ,  from processor )
20 access (    object it ,  on line ,  by processor )
21 access (    object memory )
22 access (    object ram ,  faster-than memory )
23 access (  subject address ,  object memory ,  if be )
24 access (  subject code ,  object instruction )
25 access (  subject fetch ,  object rom ,  if hold )
```

Appendix B. Noisy Words Listing

A partial listing of the noisy words used by the covering algorithm is shown here. The *Abstraction Processor* uses the noisy words file to identify the tokens in the abstraction that are noisy. The file contains tokens that were not recognized by the semantic parser of the Chunker. In addition, some tokens were manually added. The first six tokens identify the tokens that have been manually added (identified by the "**Bold**" character font).

,
:
and
fig
(
)
--indicating
2840a-c
2845a-c
285
2850a-e
2855a-e
286
2860
2860a-e
2860a-e---inverters
2860a-p
2860f-m
2860n
2860o
2860p
2865a-e
2865f-m
2865n
2865o
2865p
2870
2870a-
2870a-e
2870a-n
2870a-p
3k
3l
3state
3u
3uf
4-
4-radro
4-rsl
4-rsl-2
40-byte
400-404
401-403
401-404
4011a
4020a
403
405
4060
4067059txt
407
4096
41
411
4137565txt
415
417
4180855txt
42
426
4271466txt
428
4283760txt
43
430
4346439txt
436
438
4394726txt
44
440
4404650txt
442
444
446

intei
intel
intend
interposably
interposively
interrups
intreq
inv
inverer
inz
io
ioc

Appendix C. Nouns and their Root Form

A complete listing of the nouns and their corresponding root forms used in this thesis is listed below. This list was used by the *Abstraction Processor* to normalize the nouns in the sentence abstraction to their root form.

activities	activity	controls	control
addresses	address	conventions	convention
advances	advance	counters	counter
amounts	amount	cycles	cycle
areas	area	designations	designation
arrangements	arrangement	details	detail
bits	bit	devices	device
blocks	block	digits	digit
buffers	buffer	diodes	diode
buses	bus	drawings	drawing
bytes	byte	drivers	driver
capabilities	capability	elements	element
channels	channel	embodiments	embodiment
characteristics	characteristic	engineers	engineer
characters	character	errors	error
chips	chip	examples	example
circuits	circuit	exceptions	exception
circumstances	circumstance	features	feature
combinations	combination	figures	figure
components	component	flip-flops	flip-flop
computations	computation	flipflops	flipflop
concepts	concept	formats	format
conditions	condition	functions	function
connections	connection	gates	gate
constants	constant	groups	group
constructions	construction	indications	indication
contents	content	inputs	Input
controllers	controller	instructions	instruction

instruments	instrument
interfaces	interface
inversions	inversion
inverters	inverter
isolators	isolator
items	item
kinds	kind
latches	latch
legs	leg
limits	limit
lines	line
means	mean
memories	memory
microfarads	microfarad
microprocessors	microprocessor
modes	mode
modifications	modification
modules	module
mulltiplexers	mulltiplexer
multiplexers	multiplexer
multivibrators	multivibrator
nanoseconds	nanosecond
negations	negation
networks	network
numbers	number
ohms	ohm
ones	one
operations	operation
orders	order
outputs	output
pairs	pair
parts	part
patents	patent
paths	path
patterns	pattern
phases	phase
pins	pin
places	place
portions	portion
ports	port
positions	position
possibilities	possibility
principles	principle
printers	printer
procedures	procedure
processes	process
processors	processor
pulses	pulse
purposes	purpose
rams	ram
readers	reader
receipts	receipt
receivers	receiver
registers	register
relationships	relationship
reports	report
resistors	resistor
results	result
roms	rom
samples	sample
sections	section
senses	sense
sensors	sensor
sides	side
signals	signal
stages	stage
states	state
steps	step
strobes	strobe
sub-modules	sub-module
substitutions	substitution
systems	system
terminals	terminal
terminations	termination
things	thing
times	time
transfers	transfer
transients	transient
transistors	transistor
transmissions	transmission
transmitters	transmitter
units	unit
users	user
values	value
variations	variation
versions	version
volts	volt
ways	way
words	word
zeros	zero
zones	zone

Appendix D. Verbs and their Root Form

A complete listing of the verbs and their corresponding root forms used in this thesis is listed below. This list was used by the *Abstraction Processor* to normalize the verbs in the sentence abstraction to their root form.

accept	accept	allowing	allow
accessed	access	allows	allow
accesses	access	altered	alter
accessing	access	analyzes	analyze
accepts	accept	analyzed	analyze
accompanied	accompany	appears	appear
accomplished	accomplish	applied	apply
accumulating	accumulate	applies	apply
achieves	achieve	applying	apply
achieved	achieve	arbitrates	arbitrate
acknowledges	acknowledge	arranged	arrange
acted	act	associated	associate
acts	act	assume	assume
activated	activate	assumes	assume
activates	activate	assumed	assume
activating	activate	assure	assure
actuates	actuate	autoinitialize	autoinitialize
adapting	adapt	autoinitialized	autoinitialize
adapted	adapt	avoided	avoid
adapts	adapt	avoiding	avoid
added	add	back	back
addressed	address	backed	back
addresses	address	based	base
addressing	address	becomes	become
adjust	adjust	begins	begin
adjusted	adjust	beginning	begin
advances	advance	begins	begin
advanced	advance	biased	bias
aligned	align	block	block
aligning	align	blocks	block
allow	allow	borrow	borrow
allowed	allow	buffer	buffer

buffers	buffer
burst	burst
called	call
capture	capture
captured	capture
carries	carry
cascaded	cascade
cascading	cascade
catch	catch
cause	cause
caused	cause
causes	cause
causing	cause
chained	chain
chaining	chain
changed	change
check	check
checking	check
checks	check
claimed	claim
claims	claim
clear	clear
cleared	clear
clears	clear
clocked	clock
codes	code
coding	coding
combine	combine
comes	come
commands	command
communicate	communicate
communicating	communicate
compared	compare
complete	complete
completed	complete
completes	complete
composed	compose
compress	compress
compressed	compressed
comprises	comprise
comprising	comprise
conducted	conduct
configure	configure
configured	configure
configuring	configure
conflict	conflict
conform	conform
connected	connect
consists	consist
constituted	constitute
construed	construe
contain	contain
containing	contain
contains	contain
continue	continue
contrast	contrast
control	control
controlled	control
controlling	control
controls	control
cope	cope
corresponding	correspond
corresponds	correspond
counted	count
counts	count
coupled	couple
coupling	couple
cut	cut
cuts	cut
decoded	decode
decodes	decode
decoding	decode
decouples	decouple
decreasing	decrease
decrement	decrement
decremented	decrement
decrementing	decrement
depending	depend
derives	derive
derived	derive
described	describe
designates	designate
designating	designate
designated	designate
designed	design
detected	detect
determine	determine
determined	determine
determines	determine
differs	differ
disable	disable
disabled	disable
disables	disable
disabling	disable
discovered	discover
disposed	dispose
disenables	disenable
distributed	distribute
divided	divide
do	do
does	do
drive	drive
driven	drive
drives	drive
drops	drop
effected	effect
effectuate	effect
eliminate	eliminate
eliminated	eliminate
employed	employ
employing	employ
enable	enable

enabled	enable
enables	enable
enabling	enable
encoding	encode
encountered	encounter
end	end
enhance	enhance
ensure	ensure
enter	enter
entered	enter
enters	enter
equal	equal
equals	equal
establish	establish
exceeds	exceed
exchanged	exchange
exchanging	exchange
execute	execute
executed	execute
executes	execute
executing	execute
exhausted	exhaust
exists	exist
exit	exit
expanded	expand
express	express
extended	extend
fed	feed
fetch	fetch
fetches	fetch
filled	fill
finding	find
finished	finish
fixed	fix
fixes	fix
flip	flip
floating	float
flop	flop
flush	flush
fly	fly
followed	follow
following	follow
force	force
formed	form
forming	form
found	find
function	function
functioning	function
functions	function
gather	gather
generate	generate
generated	generate
generates	generate
generating	generate
get	get
give	give
given	give
gives	give
gated	gate
go	go
goes	go
going	go
grant	grant
granted	grant
grants	grant
ground	ground
grouped	group
grounded	ground
guaranteed	guarantee
had	have
halt	halt
handles	hand
has	have
have	have
having	have
held	hold
hold	hold
holds	hold
identifies	identify
ignored	ignore
illustrating	illustrate
improve	improve
improved	improve
include	include
included	include
includes	include
including	include
increased	increase
incremented	increment
incrementing	increment
increments	increment
indicate	indicate
indicated	indicate
indicates	indicate
indicating	indicate
inform	inform
informs	inform
initialize	initialize
initializes	initialize
initiate	initiate
initiated	initiate
initiates	initiate
inputs	input
inputted	input
inserted	insert
inspect	inspect
insured	insure
integrated	integrate
intends	intend
interchanging	interchange
interconnected	interconnect
interfering	interfer

interlace	interlace
interposed	interpose
interpreted	interpret
interprets	interpret
interrupted	interrupt
intervening	intervene
introduced	introduce
inverted	invert
involved	involve
issue	issue
issued	issue
issues	issue
know	know
known	know
latch	latch
latched	latch
latches	latch
leads	lead
level	level
limited	limit
load	load
loaded	load
loading	load
loads	load
located	locate
look	look
looking	look
made	make
maintained	maintain
maintaining	maintain
make	make
makes	make
making	make
manipulated	manipulate
masking	mask
means	mean
meet	meet
memorizes	memory
modified	modify
modifying	modify
monitored	monitor
monopolizing	monopolize
move	move
moved	move
moves	move
multiplexes	multiplex
need	need
needed	need
needs	need
note	note
noted	note
obtain	obtain
obtained	obtain
occur	occur
occurred	occur
occurring	occur
occurs	occur
offers	offer
omitted	omit
opened	open
operate	operate
operates	operate
operating	operate
optimize	optimize
orchestrates	orchestrate
order	order
organized	organize
output	output
outputs	output
outputted	output
outputed	output
outputting	output
overwritten	overwrite
own	own
owns	own
packaging	package
pass	pass
passed	pass
passes	pass
passing	pass
perform	perform
performed	perform
performing	perform
performs	perform
permit	permit
place	place
placed	place
point	point
pointing	point
positioned	position
power	power
preserved	preserve
prevent	prevent
prevented	prevent
prevents	prevent
prioritizing	prioritize
processes	process
processing	process
proceeds	proceed
produced	produce
produces	produce
programmed	program
programming	program
propagate	propagate
provide	provide
provided	provide
provides	provide
providing	provide
pulling	pull
re-established	re-establish
reached	reach
read	read

reading	read
reads	read
receive	receive
received	receive
receives	receive
receiving	receive
recognized	recognize
recommended	recommend
reconfigure	reconfigure
recovered	recover
reduce	reduce
referring	refer
regains	regain
regarding	regard
reinitialized	reinitialize
relates	relate
relating	relate
release	release
released	release
releases	release
relinquished	relinquish
reload	reload
reloaded	reload
remain	remain
remaining	remain
remains	remain
removed	remove
removing	remove
renders	render
repair	repair
repeated	repeat
replaced	replace
reports	report
represent	represent
reprogram	reprogram
reprogrammed	reprogram
request	request
requested	request
requesting	request
requests	request
require	require
required	require
requires	require
resemble	resemble
reset	reset
resets	reset
resolves	resolve
responds	respond
responding	respond
restarted	restart
restore	restore
restored	restore
resume	resume
retain	retain
retrieve	retrieve
retrieved	retrieve
retrieves	retrieve
retrieving	retrieve
returned	return
returning	return
rises	rise
rolls	roll
rotating	rotate
routed	route
run	run
said	said
sample	sample
satisfy	satisfy
save	save
saved	save
scatter	scatter
see	see
seen	see
select	select
selected	select
selects	select
send	send
sending	send
sends	send
sent	send
separated	separate
service	service
services	service
servicing	service
serves	serve
set	set
sets	set
setting	set
settle	settle
shifts	shift
showing	show
shown	show
shows	show
signal	signal
signals	signal
simplified	simplify
specified	specify
specifies	specify
speeded	speed
spends	spend
starting	start
starts	start
stays	stay
steer	steer
steering	steer
stops	stop
store	store
stored	store
stores	store
storing	store
strobed	strobe
subject	subject

suggested	suggest
supporting	support
supplied	supply
switch	switch
synchronizes	synchronize
take	take
takes	take
taken	take
tells	tell
terminates	terminate
terminated	terminate
terminating	terminate
testing	test
tests	test
tied	tie
time	time
timing	time
track	track
tracked	track
trade	trade
transfer	transfer
transferred	transfer
transferring	transfer
transfers	transfer
transformed	transform
transmit	transmit
transmits	transmit
transmitted	transmit
transpire	transpire
treated	treat
unmasked	unmask
updated	update
updating	update
upgraded	upgrade
use	use
used	use
uses	use
utilize	utilize
utilizes	utilize
utilizing	utilize
varies	vary
vary	vary
verify	verify
violates	violate
wait	wait
waiting	wait
were	be
wins	win
working	work
write	write
writes	write
writing	write
written	write

Appendix E. Details.txt

A section of the file “details.txt” is shown below. Note that alphanumeric strings and three digit numerals have been widely used to denote elements (bus, gate, multiplexer and other components and devices.)

These latter signals are applied to the memory controls over the system control bus 204 to cause the memory to perform a read or a write operation. FIG. 4 shows the address recognition circuits and the path followed by an address in passing through the SDMA from the system address bus 202 to the SDC address bus 208. Address bits A1-A4 are passed through a set of inverters 401-404 having their outputs connected to the A inputs of a multiplexer 406. The strobe input of MUX 406 is tied to ground and the select input receives the signal INT SEQ EN FF. If the signal INT SEQ EN FF is at the low level the address bits A1-A4 are gated through inverters 401-404 and the MUX 406 to the SDC address bus 208. An AND 405 receives the output of inverter 400 and the signal INT SEQ EN FF hence when A1-A4 are gated through MUX 406, A0 is gated through AND 405 and NOR 407 to the SDC address bus. A NAND 408 is provided for recognizing the address of the SDMA when that address appears on the system address bus 202. Since it is assumed that the present SDMA is assigned address 5, address bits A7 and A5S are applied directly to NAND 408 while address bit A6S is passed through an inverter 410 before being applied to NAND 408. When the system address bus bits A7-A5S have the value 101, NAND 408 produces a high level output signal that enables one input of NAND's 412, 414 and 416. In actual practice, the address recognition circuits of all SDMA's may be identical and the SDMA card position and back plane wiring utilized to determine exactly which address will be recognized by the SDMA. The CPU places a signal CPU SYNC on the system control bus at about the time that an address is placed on the system address bus. The signal CPU SYNC is passed through an inverter 418 and applied to a second input of NAND 412. The signal I/O RD or WR is at a high level any time an input or an output instruction is on the system control bus. The signal I/O RD or WR is applied to a further input of NAND 412 and is also applied to one input of NAND 414, a NAND 420 and the reset input of a D-type flip-flop 422. NAND 412 also receives the clock pulse . phi. 1A. Therefore, if an input or an output instruction is present on the system control bus and the address on the system address bus is that of the SDMA, NAND 412 produces a low level output signal that is applied to the set input of a GO FF 424. This sets the flip-flop so that the signal GO FF on output lead 426 rises to the high level.

Appendix F. Multi-level Hierarchy File

The multi-level hierarchy file used in this thesis is shown below. Note that this file is not complete. In this file, the supertype of the concepts such as "gate" in line 4 and "attribute" in line 5 are not defined. This file needs to be completed and extended to include more concept types.

Multilevel Type Hierarchy

(1) microcomputer -> computer [3]
******
(2) microprocessor | computer | cpu -> processor [2]
(3) dma | sdma | sdc | dma -> controller [2]
(4) nand -> gate [2]
(5) characteristic -> attribute [2]
******
(6) processor | memory | controller | register -> device [1]
(7) frequency | constant | information | number | ordinal | time | command -> value [1]