SOFTWARE---PRACTICE AND EXPERIENCE, VOL. 25(8), 831--862(August 1995)

Implementation Issues in the Development of the PARSEC Parser

MARY P. HARPER*, RANDALL A. HELZERMAN†, CARLA B. ZOLTOWSKI, BOON-LOCK YEO‡, YIN CHAN§, TODD STEWART¶, and BRYAN L. PELLOM‖

School of Electrical Engineering, 1285 Electrical Engineering Building, Purdue University, West Lafayette, IN 47907-1285

SUMMARY

This paper describes the implementation of a constraint-based parser, PARSEC, which has the required flexibility for a user to easily construct a custom grammar and test it. Once the user designs grammar parameters, constraints, and a lexicon, our system checks them for consistency and creates a parser for the grammar. The parser has an X-windows interface that allows a user to view the state of a parse of a sentence, test new constraints, and dump the constraint network to a file. The parser has an option to perform the computationally expensive constraint propagation steps on the MasPar MP-1. Stream and socket communication was used to interface the MasPar constraint parser with a standard X-windows interface on our Sun Sparcstation.**

The design of our heterogeneous parser has benefited from the use of object-oriented techniques. Without these techniques, it would have been more difficult to combine the processing power of the MasPar with a Sun Sparcstation. Also, these techniques allowed the parser to gracefully evolve from a system that operated on single sentences, to one capable of processing word graphs containing multiple sentences, consistent with speech processing. This system should provide an important component of a real-time speech understanding system.

KEY WORDS Constraint Parsing C++ Implementation Parallel Parsing Heterogeneous System

INTRODUCTION

This paper describes the implementation of a constraint-based parser that is used to parse either single sentences or word graphs containing multiple sentence hypotheses produced by a speech recognizer. PARSEC is able to parse sentences in grammars that are beyond context-free grammars (CFGs), but the parser does not handle general context-sensitive grammars.

* Email address is: harper@ecn.purdue.edu.

† Email address is: helz@ecn.purdue.edu.

‡ Current address is: Computer Engineering, Electrical Engineering Department, Princeton University, Princeton, NJ 08544. Email address is: yeo@ee.Princeton.edu.

§ Current address is: Computer Engineering, Electrical Engineering Department, Princeton University, Princeton, NJ 08544. Email address is: alien@ee.Princeton.edu.

¶ Current address is: DuPont Engineering - N11526, 1007 Market Street, Wilmington, DE 19898. Email address is: tjrs@strauss.udel.edu.

‖ Current address is: Duke University, Department of Electrical Engineering, Durham, NC 27708. Email address is: bp@ee.duke.edu.

** PARSEC stands for **P***arallel* **AR***chitecture* **SE***ntence* **C***onstrainer*.

0038--0644/95/080831--32$21.00

Received 23 June 1994
Revised 24 January 1995

Because constraints are a uniform knowledge representation, they are able to represent a variety of information sources in a uniform way, which is an added advantage. The control over which constraints to apply is extremely flexible in this parsing approach. Universally applicable constraints could be applied first, followed by constraints that are applicable for the context at hand. An additional advantage of our parser is that it is amenable to effective parallel implementation. PARSEC provides an extremely fast and flexible framework for parsing natural language, whether text-based or spoken. However, constraint grammar parsers represent a new approach for natural language processing researchers. Hence, we have designed a parser that helps a grammar designer to create constraint grammars and test them for correctness.

PARSEC is a heterogeneous software system in two different senses. First, the system was constructed using three programming languages and a variety of software tools. Second, the system can execute either serially or in parallel. Without using object-oriented programming techniques, this system would have been very difficult to create and debug. With these techniques, we were able to gradually add capability to the system to make it a more effective research tool in constraint-based parsing of speech.

THE THEORETICAL BASIS OF THE PARSEC PARSER

Our parser uses Constraint Dependency Grammar (CDG), originally defined by Maruyama[1,2,3], to process sentences. We have expanded the scope of CDG to allow for the analysis of sentences containing lexically ambiguous words, to allow feature analysis in constraints, and to efficiently process multiple sentence candidates that are likely to arise in spoken language processing[4]. In this paper, we summarize the basic framework of the parser, and refer the reader to our technical report[4] for justification of our choices.

Constraint-based Parsing

Maruyama[1,2,3] has shown that parsing can be cast as a constraint-satisfaction problem (CSP) with a finite domain[5,6,7,8,9,10,11,12,13], so constraints can be used to rule out ungrammatical sentences. Figure 1 depicts the process of parsing a sentence with CDG constraints.

Initially, each of the roles of a word are assigned a set of role values which encode the parse relations that are allowed by the grammar constraints. In our examples, we use a single role, the governor role (denoted as **G** in figures), which indicates the function a word fills in a sentence when it modifies its head word. Role values consist of a label (accessed using the lab function) and a modifiee (accessed using the mod function) and have access to its role (accessed using the rid function), its word (accessed using the word function), the word's position (accessed using the pos function), and the word's category (accessed using the cat function).

Unary constraints contain a single variable, and test whether role values are legal. A role value is incompatible with a unary constraint iff it satisfies the antecedent, but not the consequent. Notice in topmost constraint network of Figure 1 that all of the role values for the governor role of *the* satisfy the antecedent of the first unary constraint, but only det-nil, det-2, and det-3 satisfy the consequent and so are the only role values that are supported by that constraint. When a role value violates a unary constraint, node consistency eliminates those role values from their role because they can never participate in a parse for the sentence. Note that eq tests its two arguments for equality and lt tests whether its first argument is less than its second. After all of the unary constraints are applied, the constraint network is in the

state shown in the second constraint network in Figure 1.

Next, binary constraints are propagated. Binary constraints determine which pairs of role values can legally coexist. To keep track of pairs of role values, *arcs* are constructed connecting each role to all other roles in the network, and each arc has an associated *arc matrix*, whose row and column indices are the role values associated with the two roles it connects. The entries in an arc matrix can either be a **1** (indicating that the two role values indexing the entry are compatible) or a **0** (indicating that the role values cannot simultaneously exist). Initially, all entries in each matrix are set to **1**. The binary constraint shown in Figure 1 is applied to the pairs of role values indexing the entries in the matrices. In this case when x = det-3 for *the* and y = root-nil for *eats*, the consequent of the binary constraint fails; hence, the role values are incompatible. This is indicated by replacing an entry of 1 with 0.

Finally, arc consistency eliminates role values that are inconsistent with all the role values associated with at least one other role. Notice that at the end of this process the sentence has a single parse because there is only one value per role in the sentence. A parse for the sentence consists of an assignment of role values to roles such that the unary and binary constraints are satisfied for the assignment. Note that the assignment for our sentence is:

```
(and (= (governor (pos 1)) det-2)
     (= (governor (pos 2)) subj-3)
     (= (governor (pos 3)) root-nil))
```

There can be more than one parse for a sentence; hence, there can be more than one assignment of values to the roles of the sentence.

The running time of the algorithm can be determined by considering each of the steps. Note that n is the number of words in the sentence, k_{u} is the number of unary constraints, and k_{b} is the number of binary constraints.

1. Constraint network construction prior to unary constraint propagation: O(n^2)
2. Unary constraint propagation and node consistency: O($k_{\mathrm{u}} * n^2$)
3. Constraint network construction prior to binary constraint propagation: O(n^4)
4. Binary constraint propagation: O($k_{\mathrm{b}} * n^4$)
5. Arc consistency: O(n^4)

Notice that the time required to propagate binary constraints is the slowest part of the algorithm. For more information on constraint-based parsing see these papers[1,4].

Our Enhancements to CDG Text-Based Parsing

Many words in the English language have more than a single part of speech. For example, the word *dog* can be either a noun or a verb. Maruyama's algorithm requires that a word have a single part of speech, which is determined by dictionary lookup prior to the application of the parsing algorithm. Since parsing can be used to lexically disambiguate a sentence, ideally, a parsing algorithm should not require that the part of speech be known prior to parsing. In addition, lexical ambiguity, if not handled in a reasonable manner, can cause correctness and/or efficiency problems for a parser[4,14]. To handle lexically ambiguous words, we allow role values within the same node to have their own parts of speech as shown in Figure 2[††]. When the constraint network (CN) is constructed, the parts of speech for each word are determined by looking the word up in the dictionary. If a word is lexically ambiguous then

[††] Rather than show each role value with its corresponding part of speech in the figure, we show the set of all of the role values with a particular part of speech to save space.

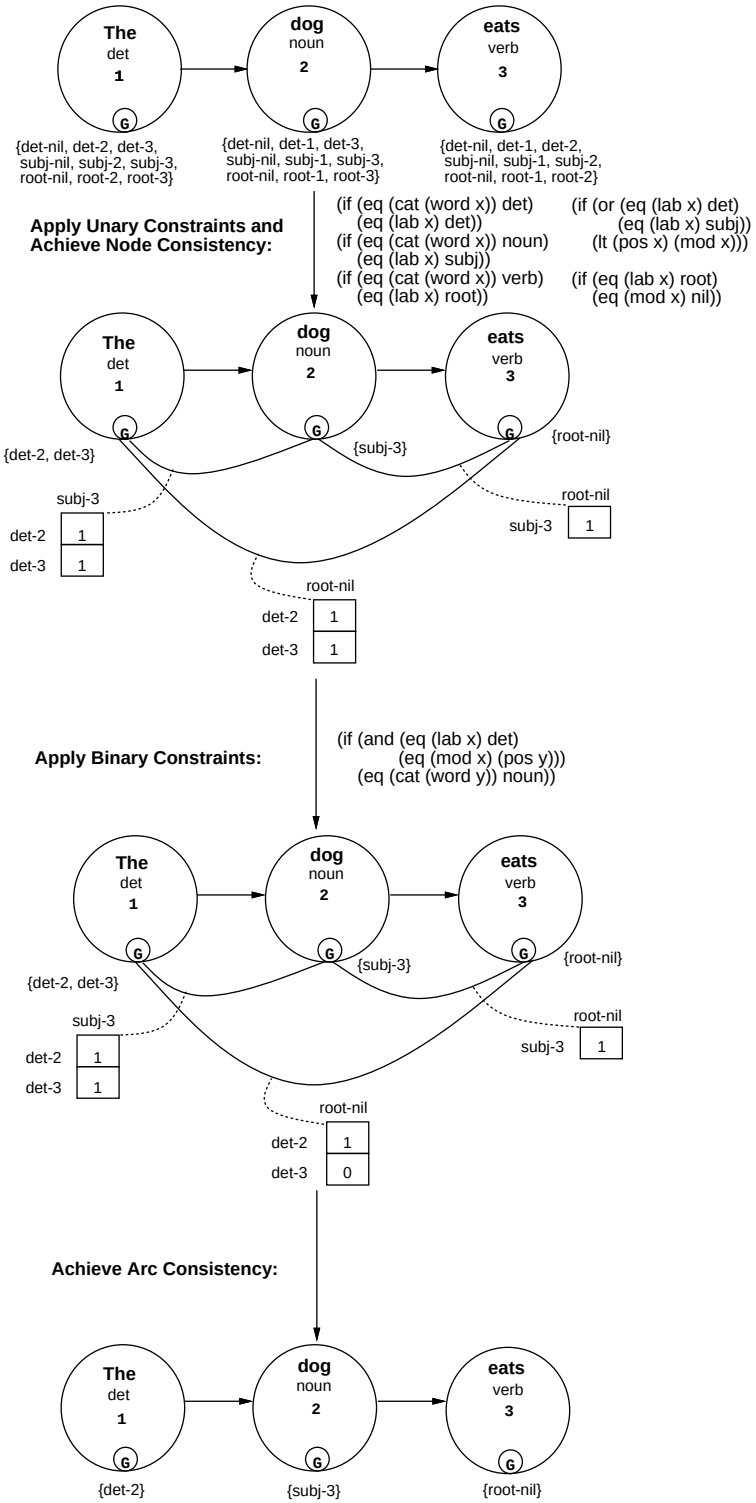

Figure 1. Using constraints to parse the sentence: The dog eats.

for each part of speech, a set of role values in the domain is created and assigned that part of speech. Because each role value has its own part of speech, the constraints are written so that the access function for syntactic category, cat, now operates on a role value rather than on a word node addressed by its position. Note also that the domains of the roles are restricted to a smaller set a role values than in Figure 1 by using a mechanism we describe next.

The set of role values assigned to each role for each part of speech for a word could be initialized to an exhaustive set of values as before; however, it is a simple matter to restrict the role values assigned to a role's domain by using a *label_table* which indicates the possible labels for each role and each part of speech when the CN is constructed. Though the table is not a necessary aspect of the grammar, it does make the analysis of a sentence more efficient because the roles are initialized to smaller domains, and many of the unary constraints (i.e., those which restrict labels of role values to lexically appropriate values) can be omitted. The label_table for our example in Figure 2 is shown in Figure 3. In practice, using a label_table reduces the number of role values in the initial CN by a factor of five to seven, and eliminates the need to propagate some unary constraints. Hence, it improves the actual running time of the CDG algorithm, but not the asymptotic running time.

Lexical features, like number, person, or case, are used in many natural language parsers to enforce subject-verb agreement, determiner-head noun agreement, and case requirements for pronouns. This information can be very useful for disambiguating parses for sentences or for eliminating impossible sentence hypotheses, hence our CDG parser was extended to support them. To support feature constraints, the parser must know about the supported features. This requires adding information about the types of features that can be tested with constraints, a list of legal values for each feature type, and some way of indicating which categories of words can have a certain feature type and what feature values they can take for that feature. This information is quite important for verifying that constraints are well-formed and that dictionary entries make sense. Hence, we have added a feature list to the grammar to keep track of the feature types and their legal feature values, and a feature table indicating those features and their legal values for each part of speech and a default value if no value is indicated in the dictionary. See Figure 4 for the feature table for our grammar given that the feature list includes the feature number/person with the possible values of 1s, 2s, 3s, 1p, 2p, and 3p.

In order to process number/person feature tests while parsing sentences, it is also necessary to store the feature information in the lexicon, as shown in Figure 5. The feature information stored with the grammar is very useful for checking the dictionary for correctness.

The algorithm for constructing the initial CN must be modified to look up number/person information for each word in the dictionary and store the information with the role values. In addition, agreement constraints must be created to ensure that the number of the determiner agrees with the number of the head noun and that the number of the subj agrees with the number of the root. These constraints require the addition of one access function number and a predicate agree. The function (number x) returns the number/person information associated with the role value, and the predicate (agree (number x) (number y)) returns true only if its two number/person arguments agree.

To properly test a feature constraint in a CN, it is necessary to store one number/person feature value per role value. In this case, the agree predicate becomes equivalent to an equality test. However, if there are two feature types (say number/person and case) to be used in constraints for a grammar, then the role values will have to be duplicated and assigned feature values from the cross product of the features' values. This could easily lead to a combinatorial explosion of role values. Fortunately, there is an excellent strategy for limiting

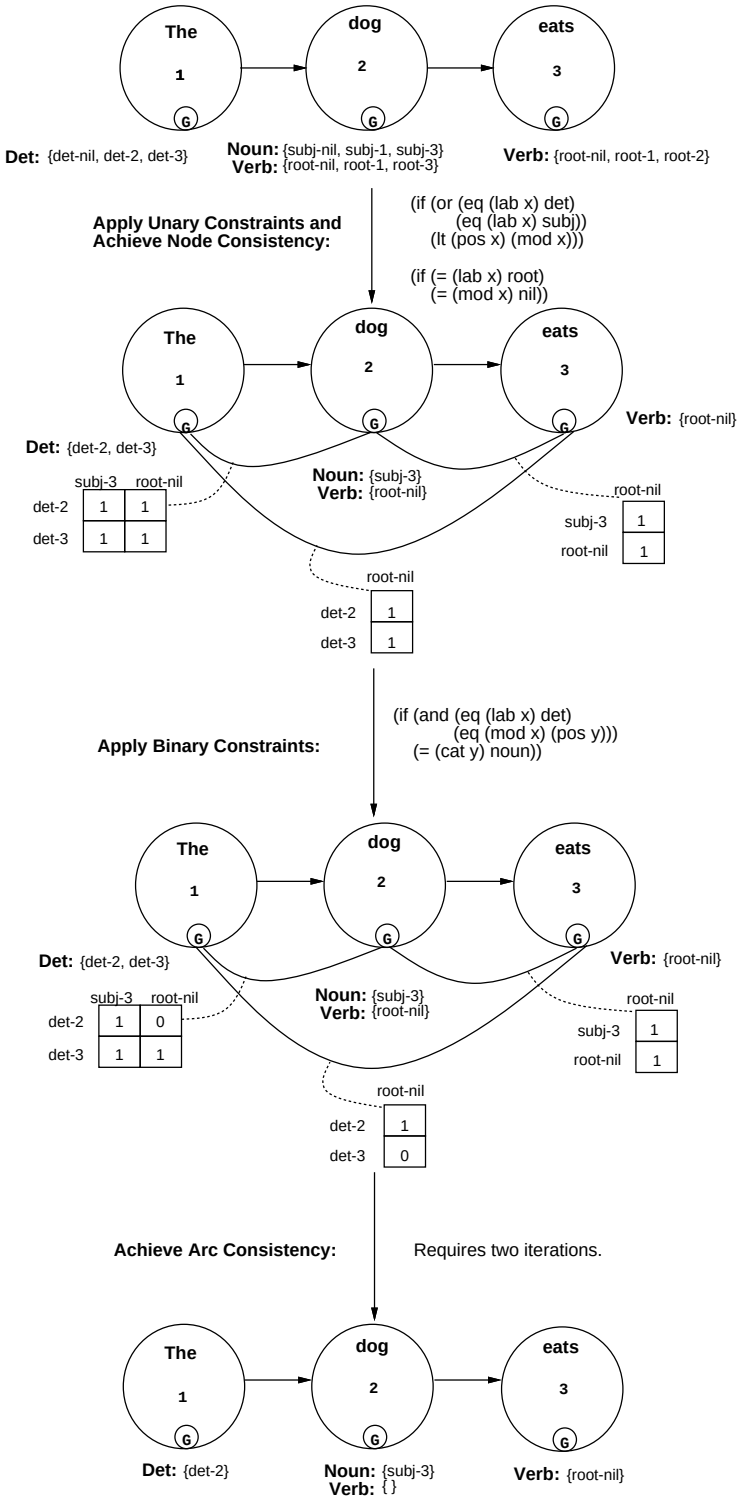

Figure 2. Using constraints to parse the lexically ambiguous sentence: The dog eats.

		Labels:		
		subj	root	det
Categories:	noun	1	0	0
	verb	0	1	0
	det	0	0	1

Figure 3. A table of legal labels for word categories in the governor role for our grammar.

		Number	Number_Default
	det	{3s, 3p}	3s
Categories:	noun	{3s, 3p}	3s
	verb	{1s, 2s, 3s, 1p, 2p, 3p}	3s

Figure 4. A table of legal feature values for word categories and defaults for our grammar.

```
((dog (category noun (number 3s))
      (category verb (number 1s 2s 1p 2p 3p)))
 (the (category det (number 3s 3p)))
 (eats (category verb (number 3s)))
```

Figure 5. A simple dictionary for The dog eats.

the number of role values. The basic idea is to store the sets of feature values with a single role value and to duplicate the role values only on demand, when a particular feature type is being tested by a constraint. A grammar writer can then order constraints in a constraint file in such a way that role values are reduced by pure phrase structure constraints prior to the feature constraints. Also, feature constraints can be ordered to minimize useless role value duplication. When the parser is preparing to propagate a constraint with a particular feature test, each of the role values having multiple values for that feature is duplicated and assigned one of the feature values. The corresponding feature constraints should then eliminate many of the duplicated role values before other types of feature constraints are propagated.

Parsing Spoken Language with Constraints

The output of a hidden-Markov-model-based speech recognizer is often a list of the most likely sentence hypotheses (i.e., an N-best list) where parsing can be used to rule out the impossible sentence hypotheses. CDG constraints can be used to parse single sentences in a CN; however, individually processing each sentence hypothesis provided by a speech recognizer is inefficient since many sentence hypotheses are generated with a high degree of similarity. An alternative representation for a list of similar sentence hypotheses is a word graph which contains information on the approximate beginning and end points of each word. A word graph represents a disjunction of all possible sentence candidates that a speech recognizer provides.

Word graphs are typically more compact and more expressive than N-best sentence lists. We[15] have constructed word graphs from three different lists of sentence hypotheses. The word graphs provided an 83% reduction in storage, and in all cases, they encoded more possible sentence hypotheses than were in the original list of hypotheses. In one case, 20 sentence hypotheses were converted into a word graph representing 432 sentence hypotheses. This increase in the number of sentences represented is often useful. The N-Best list can omit the correct sentence hypothesis which the word graph constructed from that N-Best list contains.

We view the problem of parsing spoken language as a graph processing problem. The constraints used to parse individual sentences can then be applied to eliminate impossible hypotheses. Hence, we have adapted the CDG constraint network to handle the multiple sentence hypotheses stored in a word graph, calling it a Spoken Language Constraint Network (SLCN). An example of an SLCN is shown in Figure 6. It was derived from a word graph constructed for the sentence hypotheses: **A fish eat* and **Offices eats*. By representing these hypotheses in a word graph, we are able to process additional sentences (i.e., *A fish eats* and *Offices eat*) not present in the list of hypotheses, one of which might represent the intended utterance.

Each word node in the SLCN contains information on word beginning and end points, represented as an integer tuple (b, e), with b $<$ e. The tuple is more expressive than the single integer used for CNs and requires modification of some of the access functions and predicates. Notice that word nodes contain a list of all word candidates with the same beginning and end points, and that nodes that can be adjacent to one another are joined by directed *edges*. A sentence hypothesis must include a word candidate from one word node at the beginning of the utterance, a word candidate from a word node at the end of the utterance, and a word candidate from each of the nodes along a path of edges connecting the two nodes. The number of sentence hypotheses represented by a graph of n nodes can be exponential in n. The goal of our system is to utilize constraints to eliminate as many impossible sentence

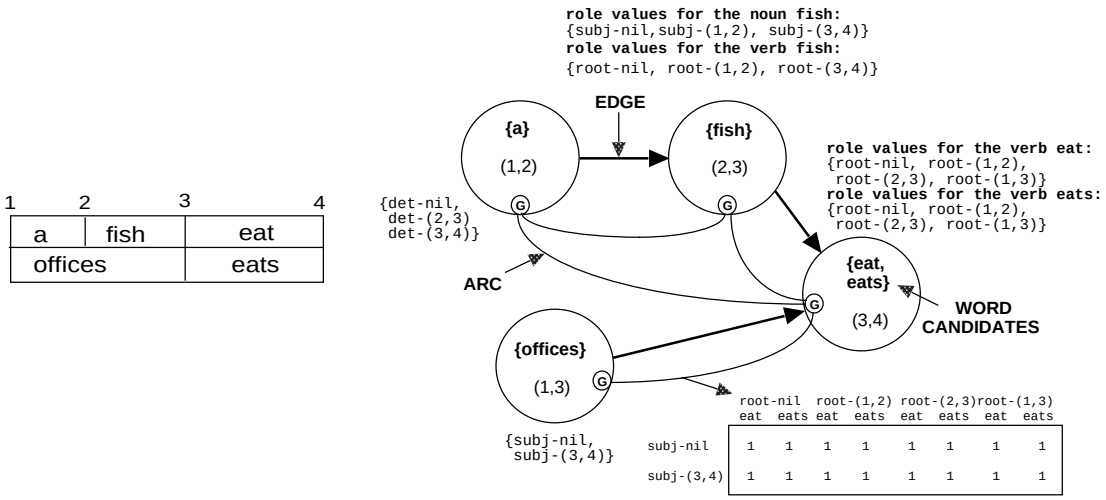

Figure 6. Example of a spoken language constraint network constructed from a word graph.

hypotheses as possible, and then to select the best remaining sentence hypothesis (given the word probabilities given by the recognizer).

Some of the constraint access functions and predicates must be adapted for SLCN parsing. For example, the access functions for position, (pos x), and modifiee, (mod x), now return a tuple (b, e) which describes the position of the word associated with the role value x. The equality predicate is extended to test for equality of intervals (e.g., (eq (1,2) (1,2)) should return true). Also, the less-than predicate, (lt (b1, e1) (b2, e2)), returns true if e1 $<$ b2, and the greater-than predicate, (gt (b1, e1) (b2, e2)), returns true if b1 $>$ e2.

To parse an SLCN, each word candidate contained in a word node is assigned a set of role values for each role, requiring O(n^2) time, where n is the number of word candidates in the graph. Unary constraints are applied to each of the role values in the network, and like CNs, require O($k_u * n^2$) time.

The preparation of the SLCN for the propagation of binary constraints is similar to that for a CN. All roles within the same word node are joined with an arc as in a CN; however, roles in different word nodes are joined with an arc if and only if they can be members of at least one common sentence hypothesis (i.e., they are connected by a path of directed edges). To construct the arcs and arc matrices for an SLCN, it suffices to traverse the graph from beginning to end and string arcs from each of the current word node's roles to each of the preceding word node's roles (where a node precedes a node if and only if there is a directed edge from the preceding to the current node) and to each of the roles that the preceding word nodes' roles have arcs to. For example, there should be an arc between the roles for *a* and *fish* in Figure 6 because they are located on a path from the beginning to the end of the sentence *a fish* {*eats, eat*}. However, there should not be an arc between the roles for *a* and *offices* since they are not found in any of the same sentence hypotheses. After the arcs for the SLCN are constructed, the arc matrices are constructed in the same manner as for a CN. The time required to construct the SLCN network in preparation for binary constraint propagation is O(n^4) because there may be up O(n^2) arcs constructed, each requiring the creation of a matrix with O(n^2) elements. Once the SLCN is constructed, binary constraints are applied to pairs of

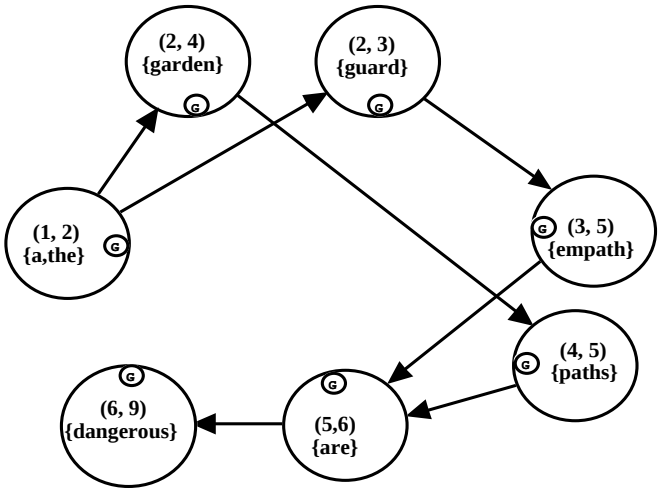

Figure 7. A simple SLCN.

role values associated with arc matrix entries (in the same manner as for the CN), requiring O($k_b * n^4$) time, where n is the number of word candidates.

Arc consistency in an SLCN is complicated by the fact that the limitation of one word's function in one sentence hypothesis should not necessarily limit that word's function in another sentence hypothesis. For example, consider the SLCN depicted in Figure 7. Even though all the role values for *are* would be disallowed by the singular subject *empath*, those role values cannot be eliminated since they are supported by *paths*, the subject in a different hypothesis. The SLCN arc consistency algorithm cannot disallow role values that are allowed by at least one sentence in the network, in contrast to the CN arc consistency algorithm. Hence, we have modified the CN algorithm to accommodate word graphs. We have developed an algorithm, which operates in O(n^4) time, to achieve arc consistency in an SLCN by using the properties of the directed acyclic graph representing the word network to eliminate role values that can never appear in any parse[4,16]. This algorithm operates correctly with single sentences as well as word graphs. Since an SLCN can represent one sentence as well as multiple sentences, we chose the SLCN representation in the implementation of our constraint-based parser.

Benefits of a Constraint-Based Approach

There are many benefits to using a constraint based parser, with the primary one being flexibility. When a traditional context-free grammar (CFG) parser generates a set of ambiguous parses for a sentence, it cannot invoke additional production rules to further prune the analyses. In CDG parsing, on the other hand, the presence of ambiguity can trigger the propagation of additional constraints to further refine the parse. A core set of constraints that hold universally can be propagated first, and then if ambiguity remains, additional, possibly context-dependent, constraints can be applied. We have developed semantic constraints that are used to eliminate parses with semantically anomalous readings from the set represented in the constraint network[17]. Additional knowledge sources are quite easy to add given the uniform framework provided by constraints.

Tight coupling of prosodic[18] or semantic rules with CFG grammar rules typically increases the size and complexity of a grammar and reduces its understandability. Semantic grammars have been effective for limited domains, but they often do not scale up well to larger systems[19]. The most successful modules for semantics are more loosely coupled with the syntactic module. The constraint-based approach represents a loosely-coupled approach for combining a variety of knowledge sources. It differs from a blackboard approach in that all constraints are applied using the uniform mechanism of constraint propagation[20]. Hence, the designer does not need to create a set of functionally different modules and worry about their interface with other modules. Constraint propagation is a uniform method that allows the designer to focus on the best way to order the sources of information impacting comprehension.

The set of languages accepted by a CDG grammar is a superset of the set of languages that can be accepted by CFGs. In fact, Maruyama[1,2] is able to construct a CDG grammar with two roles (degree = 2) and up to two variables in a constraint (arity = 2) that accepts the same language as an arbitrary CFG converted to Griebach Normal form. We have also devised an algorithm to map a set of CFG production rules into a CDG grammar. This algorithm does not assume that the rules are in normal form, and the number of constraints created is $O(G)$, where G is the size of the CFG. In addition, CDG can accept languages that CFGs cannot (e.g., $a^n b^n c^n$ and ww, where w is some string of terminal symbols). There has been considerable interest in the development of parsers for grammars that are more expressive than the class of context-free grammars, but less expressive than context-sensitive grammars[21,22,23]. The running time of the CDG parser compares quite favorably to the running times of parsers for languages that are beyond context-free. For example, the parser for tree adjoining grammars (TAG) has a running time of O(n^6).

CFG parsing has been parallelized by several researchers. For example, Kosaraju's method[24] using cellular automata can parse CFGs in O(n) time using O(n^2) processors. However, achieving CFG parsing times of less than O(n) has required more powerful and less implementable models of parallel computation than used by Kosaraju[24], as well as significantly more processors. Ruzzo's method[25] has a running time of O($log^2(n)$) using a CREW P-RAM model (Concurrent Read, Exclusive Write, Parallel Random Access Machine), but requires O(n^6) processors. In contrast, we have devised a parallelization for the single sentence CDG parser[26,27] which uses O(n^4) processors to parse in O(k) time for a CRCW P-RAM model (Concurrent Read, Concurrent Write, Parallel Random Access Machine), where n is the number of words in the sentence, and k, the number of constraints, is a grammatical constant. Furthermore, this algorithm has been simulated on the MasPar MP-1, a massively parallel SIMD computer. The MP-1 supports up to 16K 4-bit processing elements, each with 16KB of local memory. The CDG algorithm on the MP-1 achieves an O($k + log(n)$) running time by using O(n^4) processors. By comparison, the TAG parsing algorithm has also been parallelized, and operates in linear time with O(n^5) processors[28].

To parse a free-order language like Latin, CFGs require that additional rules containing the permutations of the right-hand side of a production be explicitly included in the grammar[29]. Unordered CFGs do not have this combinatorial explosion of rules, but the recognition problem for this class of grammars is NP-complete. A free-order language can easily be handled by a CDG parser because order between constituents is not a requirement of the grammatical formalism. Furthermore, CDG is capable of efficiently analyzing free-order languages because it is does not have to test for all possible word orders.

In summary, CDG supports a framework that is more expressive and flexible than CFGs, making it an attractive alternative to traditional parsers. It is able to utilize a variety of different

knowledge sources in a uniform framework to incrementally disambiguate a sentence's parse. The algorithm also has the advantage that it is efficiently parallelizeable.

PARSER DEVELOPMENT

In this section, we describe our constraint-based parser which operates on a Sun Sparcstation. For this parser to be as useful as possible for grammar designers, they must be able to specify the grammar parameters, write and test constraints consistent with the grammar parameters, and design a grammar appropriate lexicon. We have provided tools that check the grammar parameters for consistency, and then, using the grammar parameters, check the constraints and dictionary for compatibility with those parameters. In addition, there are several ways that the user can input a sentence to the parser, either as a text file (one or more sentences) or as a word graph. The user is also able to perform the computationally expensive operations of constraint propagation and arc consistency on the MasPar MP-1.

This section is divided into two subsections. The first describes the software tools we used in the implementation. The second discusses the functional organization of the parser.

Software Tools Used

The parser we developed utilized C++, C, AMPL (Ansi MasPar Language)[30], PCCTS (Purdue Compiler Construction Tool Set)[31], and the Athena toolkit. The Athena toolkit (which has the advantage that it is universally available), together with C routines, were used to create the X-windows interface to the parser. AMPL, a data-parallel version of C, was used to create the fast constraint propagation modules to run on the MasPar. Finally, C++ and the PCCTS tool kit were used for everything else. There are 30,030 lines of code in the sequential implementation of the algorithm and another 11,550 lines of code for the MasPar.

C++ was an ideal language for this project for a number of reasons. First, the parser had to be flexible because it was being used for research in the area of constraint-based parsing. Hence, if there were one attribute that characterized the design and implementation of PARSEC, it would be ''Revision.'' For example, initially the parser only parsed single sentences. Later we modified it to handle word graphs. The network construction routines changed several times. C++ helped us by preventing changes in one module from interfering with another. Also, this was a community effort of seven people. By using RCS to facilitate mutual exclusion and the encapsulation that the C++ language itself enforces, we were able to parallelize the effort of writing code effectively.

Second, C++ was perfect for creating and parsing the constraint network, which is inherently a hierarchical structure. Organizing this complex data structure and operating on it procedurally was simplified because of the use of data objects. The constraint network of our parser provided the basis for the software decomposition into underlying data objects, where these objects mirror the user interface to the network. The design of our object-oriented system was closely tied to the C++ implementation, allowing for rapid prototyping.

Third, the constraint-based parser is a highly parallelizable algorithm because of the parser's locality of computation[26,27,32]. This locality can be encapsulated in objects, hence, there is a good fit between the object-oriented approach and our desire to parallelize the algorithm. Furthermore, we wished to implement the parser on the MasPar MP-1, and C++ is similar to AMPL, the MasPar language. We were able to reuse much of the C++ code for the MasPar version. The C++ code also interfaces easily with CASE tools such as PCCTS and X-windows code.

C++, despite its appeal, also has some negative points. First, the C++ compilers are currently in flux, and it has not been uncommon for a new release of a C++ compiler to break our code. Second, C++ does not currently produce the most efficient code when compared to C. We traded off speed for structure by using C++ in our implementation of the PARSEC parser.

The PCCTS tool kit[31] has also proven to be quite valuable for this project. PCCTS is an integrated lexical analyzer generator and parser generator that generates LL(k) parsers. We chose PCCTS over several other competing compiler compilers for several reasons. First, because the parsers generated by PCCTS are LL parsers, they provide precise and meaningful error messages when an error is found. Entering thousands of words into a lexicon is a very tedious and error-prone process, and PCCTS provides an excellent mechanism for checking the files input by users for well-formedness, for cross checking the various grammar inputs provided by the user, and for converting the information to a parser internal form. PCCTS provided the perfect tool for checking the grammar parameters, dictionary, unary and binary constraints, and input sentences for good form. Second, for this project, a particularly valuable feature of PCCTS is that it can switch on demand between different grammars and lexers. For example, the word *if* is a keyword in our constraint interpreter; however, in our lexicon, *if* is treated like any other word. Instead of having to handle *if* as a special case, we simply use a different lexical analyzer when reading the lexicon. Finally, PCCTS effectively supports the C++ language by allowing grammars and lexers to be encapsulated as C++ classes. Hence, each parser's output fits very well into our overall object-oriented framework allowing its output to be easily interfaced to the rest of the system.

The Functional Organization of PARSEC

The data flow diagram for the PARSEC parser is depicted in Figure 8. The square boxes with depth shading indicate the files that must be provided by the user. These files include the grammar table, the unary and binary constraints, the lexicon, and the sentence file.

The Grammar Table

The user is required to provide a grammar table for the parser. From a grammar designer's point of view, this should be the first file created. The information provided in this file is helpful for checking the dictionary and constraints for consistency and for constructing the constraint network. The file contains the following information:

- A list of all legal parts of speech for words that can appear in a sentence: used to check the dictionary and constraints;
- A list of all role names required for the grammar: used to check the constraints and construct the constraint network;
- A list of all labels used in the grammar: used to check the constraints and construct the constraint network;
- A label table indicating the legal labels for each part of speech given the role: used to construct the constraint network;
- A list of all legal feature types along with lists of legal feature values for those types: used to check the dictionary and constraints;
- A feature table indicating the legal feature types for each part of speech along with a list of legal feature values and a default value: used to check the dictionary and construct the constraint network;

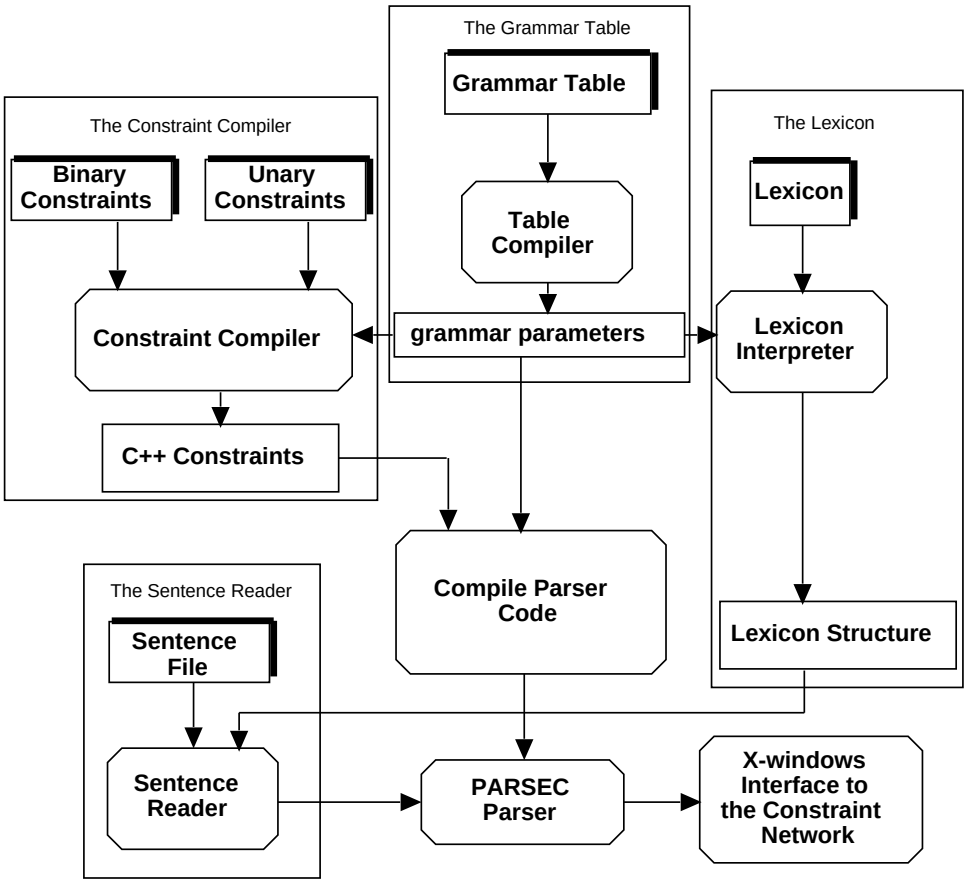

Figure 8. The Data Flow Digram for the PARSEC parser. The square boxes with depth shading indicate the files that must be provided by the user.

```
; A list of the legal parts of speech
(categories det noun verb)

; A list of role names for the grammar
(roles governor)

; A list of labels used in the grammar
(labels det subj root)

; A label table for restricting the domains given a part of speech and role name
(label_table (governor (noun subj)
                       (verb root)
                       (det det)))

; A list of legal feature types and their possible values
(grammar_features (number 1s 2s 3s 1p 2p 3p))

; A feature table indicating the possible values for each feature type associated
; with a part of speech and its default value if none is specified in the lexicon
(feature_table (det (number 3s 3p [3s]))
               (noun (number 3s 3p [3s]))
               (verb (number 1s 2s 3s 1p 2p 3p [3s])))

; A list of labels that take only a nil modifiee
(multiply_modifiees_not blank)
```

Figure 9. The parameter file for a sample grammar.

- A list of labels that take only a nil modifiee when a role value is created: used to construct the constraint network.

A simple example of a parameter file for our example grammar is shown in Figure 9. This grammar parameter file is very simple because of the simplicity of the grammar it represents. The file for a more general grammar of English would contain ten to twenty parts for speech, two or more role names, 50 to 100 labels, five to ten feature types with a varying number of legal feature values per type. Our program supports feature tests for seven different feature types.

The format for an input parameter file is described next. The core parameters of the grammar are the legal parts of speech, role names, and labels. These values must be provided first in the parameter file and in that order. These values are represented as lists of words with the key words *categories*, *roles*, and *labels* at the head of the associated list.

Next, the user specifies the label table. The label table is identified by the keyword *label_table* at the head of the list. The remaining elements in this list are lists headed by the legal role names in the grammar. These lists contain lists of the legal labels for that role

associated with each part of speech, where the part of speech is indicated at the head of the associated list of allowable labels.

Then the user specifies a list containing the allowable feature types and possible feature values for those types. This list is identified by the key word *grammar_features*, and it contains lists of allowable feature values for feature types which appear at the head of each list. For example, in Figure 9, *number* is a legal feature type, and it has the legal values of 1s, 2s, 3s, 1p, 2p, and 3p. The user then specifies which feature types are associated with each part of speech, and the associated allowable values and defaults. This feature table is identified by the keyword *feature_table*, and it contains lists of legal feature types for each part of speech, their allowable feature values, and a possible default value. In the above example, the number feature is allowed for categories of type *noun*, *det*, and *verb*. The default value for that feature is 3s for all of those parts of speech. These default values allow users to create compact dictionary entries by omitting the feature value for that feature type when it corresponds to the default.

We have written a routine using PCCTS that checks a user's grammar table for correct format and consistency. This routine creates data structures that are used by other modules in the parser such as the constraint compiler and the dictionary reader. Once the grammar table is available, the user can set up the lexicon and design constraints. First we will describe the setup of the lexicon file, and then describe how the constraint files are formatted.

The Lexicon

In this section, we describe the lexicon, detailing the design rationale and its most interesting characteristics. We also discuss how it interfaces with the rest of the parser.

The lexicon must specify all of the words that can appear in a sentence. It is represented as a list of word entries, where each word entry is a list headed by the associated word along with other important information. A word should appear in the lexicon only once with all forms of the word listed with the word entry. Each word entry can potentially have more than a single part of speech. The text for the word must come first in the list, and it is followed by a single root specifier (i.e., (root_word $<word>$)) if there is one. The remaining items in the entry are either a single syllable specifier (i.e., (syllable $<number>$)), which is optional, and one or more category specifiers. These items can occur in any order. The syllable information is used when PARSEC is processing a list of sentence hypotheses and must convert them to a word graph before parsing. A category specifier is a list consisting of the word *category*, a valid category given the grammar table, an optional root specifier, and one or more feature specifiers. The feature specifier is a list containing a feature type and one or more feature values. The dictionary interpreter checks each word entry for correctness of form and for consistency with the grammar table. In particular, the lexical categories and feature types used in the dictionary must be defined in the grammar table. Furthermore, each feature type used in a lexical entry must be compatible with the part of speech for the word, and the feature values used in an entry must be compatible given the part of speech and feature type. Two example word entries are shown in Figure 10. Notice that *frog* has no root word, but *frogs* has *frog* as its root word.

The grammar designer can specify default feature values for each valid feature associated with a lexical category in the grammar table. These defaults allow the lexicon designer to eliminate a feature specifier under certain circumstances. For example, given the grammar table from the previous section, the lexicon entry for *frog* can be simplified as shown in Figure 11. Another source of reduction stems from the fact that in many cases, morphological

```
((frog (category noun (number 3s))
       (syllable 1))
 (frogs (root_word frog)
        (category noun (number 3p))
        (syllable 1)))
```

Figure 10. A simple lexicon containing the words frog *and* frogs.

variants of a word share some of the same feature information. For example, all verb forms of the word *run* share the same possible subcategorization (i.e., subcat) feature values. By mentioning the root word in a word's entry, the user can also sometimes abbreviate the entry for that word. For example, because *runs* is a the present form of *run*, it can inherit the subcat feature values from that root word. However, the entry must explicitly specify feature values for those features it cannot inherit from the root word, like number. If a root word is not mentioned, it is assumed that the current word is the root word for itself, and so all feature information must be explicitly provided in the entry, as in the case of the entry for *frog*, unless a default value is assumed.

Root word specifiers can appear not only in a word entry, but also in a category specifier for a word entry. To illustrate why this is necessary, we have selected *saw* for discussion. The word entry for *saw* should contain information about both its noun and verb forms. The lexicon must contain three category specifiers in this case, to distinguish *saw* as the past tense form of *see* from *saw* as a singular noun and *saw* as the infinitive and present forms of the verb meaning *to cut with a saw*, as shown in Figure 12. Notice that the first verb entry for *saw* has an explicitly mentioned root word, *see*. Because in this case, *saw* is a the past form of *see*, it can inherit the subcat feature values from that root word. If a root word is not mentioned in the word entry or in the category specifier, then it is assumed that the root word for the category is the word itself.

In our system, we have defined methods for handling the following features: subcat, number, type, semantic type, case, gender, and behavior (which is a subcat feature for determiners). Each of the feature types has a print method, a setting method, and ways of testing them for equality. For the semantic type, it is useful to define a hierarchy of types that allows one to test whether something is +living if it is a subtype of something that is alive. This is handled in the code written for semantic feature tests. If the user wishes to add additional features,

```
((frog (category noun) (syllable 1))
 (frogs (root_word frog)
        (category noun (number 3p))
        (syllable 1)))
```

Figure 11. A simple lexicon that uses the default for number.

```
(saw (category verb (root_word see)
                    (type past))
     (category verb (type present infinitive)
                    (number 1s 2s 1p 2p 3p)
                    (subcat none obj))
     (category noun (type count)
                    (number 3s)))
```

Figure 12. A simple lexicon entry for saw.

they must modify the feature table, but they must also add C++ code to the system to support their newly defined feature tests.

In addition to basic feature support, we have also encoded knowledge of which feature values are incompatible with each other. To understand why, consider the second verb form of *saw*. One might assume that this word has 20 different feature forms for each role value, corresponding to the cross product of the the three feature sets associated with the word entry. However, the only forms that can legally take a number feature are the tensed forms of the verb, and so this entry would generate 12 different feature forms. We allow users to use the compact entry for *saw* because the code supporting features is able to separate the second entry into two incompatible verb entries.

We have written a routine using PCCTS that checks a user's lexicon for correct format and for consistency with the grammar parameters, and it also makes certain that all required feature information is provided for a word. If a legal feature type for a word's category specifier is not included in the lexicon entry then the default value for that category's feature type is used. If a required feature is missing from a word entry and there is no default for the entry's part of speech and feature type, then an error is signaled. As the lexicon is being checked for correctness, the routine also creates a data structure of word entries that will be used by the parser. Once the dictionary is read in and checked, the data structure is available to the sentence reader, which constructs the initial representation for the constraint network prior to the propagation of unary constraints.

The Constraint Compiler

In this section, we describe how our constraint compiler functions. We will also discuss briefly the constraint interpreter and constraint optimizer.

After the grammar designer creates the grammar table and dictionary, he or she should generate the constraints for the grammar. Our parser supports only unary and binary constraints, which can be distinguished because unary constraints contain a single variable (i.e., x) and binary constraints contain two variables (i.e., x and y). The constraints are divided into two files, unary constraint and binary constraint files. The unary constraints are applied before the constraint network is expanded to include arcs and arc matrices, and so the unary constraint file must contain only unary constraints. However, the binary constraint file can contain both unary and binary constraints.

Each file of constraints must be read by the constraint parser and checked for correctness

given the grammar parameters defined for the grammar. A constraint consists of an open parenthesis, the word if, an antecedent boolean expression, a consequent boolean expression, and a close parenthesis. A boolean expression can be a predicate (i.e., eq (i.e., =), gt (i.e., >), lt (i.e., <), elt, leq (i.e., ≤), geq (i.e., ≥), agree) or predicates joined by a logical connective (i.e., and, or, not). The logical connectives and and or can operate over one or more boolean expressions; whereas, not operates on only one boolean expression.

The primitive boolean expressions lt, gt, lte, or gte can only compare two ordinal arguments (such as position tuples). The equality predicate is more complex because the types of its two arguments must be compatible for the predicate to make sense. For example, it would be quite odd to use the predicate (eq (lab x) noun) if noun is not a label defined in the grammar. If such an expression appears in a user's constraints, our constraint compiler would catch the error and report it. The membership boolean expression elt is somewhat like the equality operator, except that its first argument identifies a single type and the second argument is a set.

Access functions like (pos x) appear quite regularly in constraints, each with a well defined syntax and type structure. The constants in the grammar can also be used in constraints, and each falls into a well-defined type hierarchy. This organization is ideal for allowing constraints to be checked for syntactic well-formedness.

We have written a routine using PCCTS that checks a user's constraint files for correct format and for consistency with the grammar parameters, and then converts them to C++ code which is linked in with our parser. The advantage of representing the constraints this way is that the parser is able to operate much more efficiently than if it had to interpret the constraints and apply them to the constraint network. The C++ constraints can also be optimized to make the constraint propagation phase even faster by combining constraints together so that access functions and tests shared across constraints are carried out only once, much like a RETE network[33]. The constraint optimizer can be used in place of the constraint compiler to create a file containing more efficient constraints in C++ code.

This is not to say that there is no use for a module that interprets and applies a file of constraints; we have also developed a constraint interpreter that does precisely this. The major purpose of the interpreter is to allow a grammar designer to add new constraints to the grammar without recompiling the parser. Debugged constraints should always be compiled; whereas, newly developed constraints can be tested more easily using a constraint interpreter. The constraint interpreter can be invoked after the compiled constraints have been applied to a sentence. Interpreted constraints, like compiled constraints, are checked for well formedness and then are applied to the constraint network.

Even though the constraints are checked for well-formedness, they may not provide the behavior the user desires. To check constraints for correctness requires that they be propagated and the constraint network checked to validate that each constraint worked as expected. This requires that we provide a user interface that allows the user to examine the impact of a constraint.

The Sentence Reader

Once a grammar is written and constraints are checked for well-formedness, the parser is ready to parse a sentence. The parser can parse sentences in either of two formats. The first format is a word graph representation which consists of a list of nodes and their edge connections as depicted in Figure 13. Notice that the nodes are numbered sequentially from one to the total number of nodes. Each word node contains information about its beginning

```
(node 1
      (begin 1)
      (end 2)
      (word what when)
      (previous)
      (next (node 2 (pi 1))))

(node 2
      (begin 2)
      (end 3)
      (word kind)
      (previous (node 1))
      (next (node 3 (pi 1))))

(node 3
      (begin 3)
      (end 4)
      (word of)
      (previous (node 2))
      (next (node 4 (pi 1))))

(node 4
      (begin 4)
      (end 7)
      (word aircraft)
      (previous (node 3))
      (next (node 5 (pi 1))))

(node 5
      (begin 7)
      (end 8)
      (word is these do use eight)
      (previous (node 4))
      (next (node 6 (pi 1))))

(node 6
      (begin 8)
      (end 9)
      (word the these)
      (previous (node 5))
      (next (node 7 (pi 1))))

(node 7
      (begin 9)
      (end 10)
      (word eight tx date two ex)
      (previous (node 5) (node 6))
      (next))
```

Figure 13. A word graph input file for the parser.

and end point, the word candidates corresponding to that time interval, and pointers to nodes that precede or follow it in the word graph given directed edges. Note that all word candidates that occur over the same time interval are placed in the same node. We have written a routine using PCCTS that checks the word graph for correct format and then invokes the routines to construct the SLCN from the graph.

Our parser also supports a text file containing one or more sentences if the user uses the appropriate command line option. In that case, the list of the sentences is converted to a word graph in which the duration of the node is approximated by using the syllable count for the words in the utterance. This option allowed us to conduct experiments comparing N-best lists of sentence hypotheses to word graphs constructed from them[15].

The Constraint Network Data Structure

The sentence reader uses the grammar parameters, the lexicon, and the input sentence structure to create a constraint network for parsing. Figure 14 depicts the data structures used to define the constraint network, which is an object containing node, edge, and arc objects.

A node consists of a position and a list of word candidate objects, each of which contains a word string and a list of feature objects. A feature object has a single part of speech and lists of compatible allowed feature values for the defined features. For the word entry *saw* shown Figure 12, there would be initially four feature objects, one for the noun, one for the past

tense of *see*, one for the infinitive form of *saw*, and one for the present form of *saw*. When a feature object is created, the sets of feature values are stored for each allowable feature if they are specified in the lexicon, otherwise a default value is used.

Each feature object has an associated list of role objects, one for each role in the grammar. These role objects keep track of the role values for its feature object. To determine the legal roles, we use the label table. For each possible modifiee (i.e., nil and all positions other than the position of this node) and for each label allowed given the part of speech for the feature class and the current role, the program creates a set of role values for the roles. Role values are objects that keep track of which role they assigned to in this structure. They also have access to their lexical category, feature values, and position.

It is necessary to keep track of whether a role receives the support of one ore more role values, both at the feature class level and at the level of the node. If a feature class ever contains a role with no role values, that feature class would be eliminated from the set of possible parses that the word graph represents. If a node contains a role with no role values, then the node would be eliminated from the word graph.

The edges in the network indicate which nodes are able to follow one another in the word graph. The word graph must be a directed acyclic graph. The edge information is important for the arc consistency step of the algorithm because we utilize the properties of the DAG to filter as thoroughly as possible given the composite structure of the word graph. The arcs are created by joining all roles that can occur in at least one common sentence hypothesis. Associated with each arc object is a pointer to the role values corresponding to the x indices and the y indices for constraint propagation. The values max_x and max_y correspond to the dimension of the arc matrix, which is another piece of information stored in the arc object. x_index and y_index are indices for the matrix and they interrelate the matrix index and its corresponding role value. The step_matrix is used to keep track of which arc elements were just eliminated by the last constraint stepped through. This is useful for debugging.

Parsing and the User Interface

To execute the parser, the user utilizes the command parsec<*input-file*, which causes the parser to read *input-file* as a word graph input file, construct the network, and parse it using the compiled unary and binary constraints. In order to facilitate the testing of various modes of operation, the parser also supports various command line options that modify its behavior at run time. These include:

1. **-X : Run X-windows Interface.** This flag invokes the X-windows interface after unary and binary constraints are propagated. When omitted, the program simply prints out some diagnostic information and exits. By omitting the flag, we are able to run PARSEC in batch mode.
2. **-M : Use MasPar.** This flag causes a slave process to be created on the MasPar to execute unary and binary constraint propagation and arc consistency.
3. **-N : Read N-best Sentence List.** When this flag is used, the input to the parser is assumed to be a list of sentence hypotheses such as might be output by an N-best speech recognizer. This flag indicates that the list of sentences should be converted to a word graph before parsing. We used this option to demonstrate both the compactness of word graphs and the effectiveness of our SLCN parser for eliminating ungrammatical sentence hypotheses[15].
4. **-C : Simulate chip.** The PARSEC research group is implementing the parsing algorithm in a custom VLSI chip[34]. This command line flag allows us to use the parser as a simulator for this chip to collect statistics about timing and functional unit utilization.
5. **-I : Interleave Arc Consistency and Constraint Application.** When this flag is used, the arc consistency and the constraint application algorithms are dovetailed and executed simultaneously.

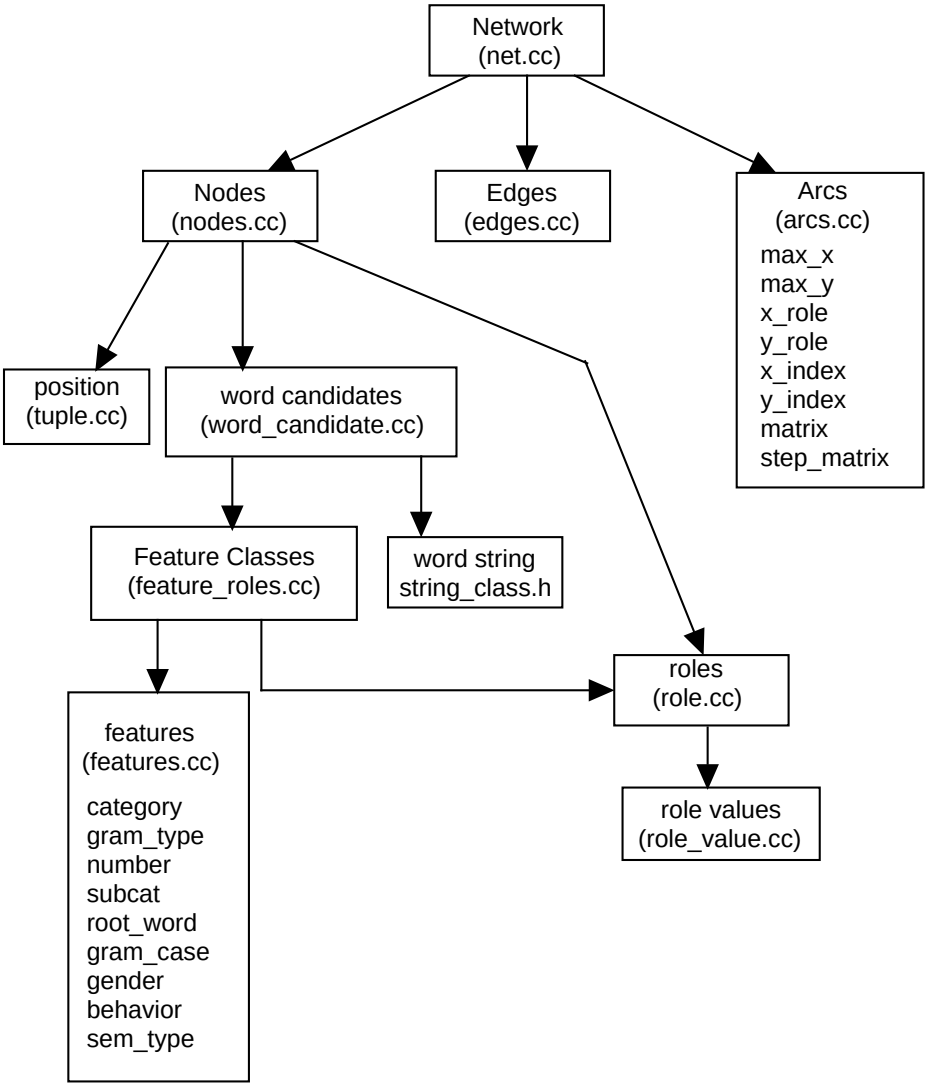

Figure 14. The classes used to define an SLCN.

6. **-D : Display MasPar and Sparcstation Parse Information Simultaneously**. When this flag is used, the parser is executed on both the MasPar and the Sun Sparcstation, and the results of both are available for comparison via the X-windows interface. This option was invaluable for debugging the algorithm on the MasPar.

Most of these command line options are orthogonal to each other and can be used simultaneously. For example, parsec -X -N< *input-file* causes *input-file* to be interpreted as a list of sentence hypothesis with the results of parsing displayed using X-windows.

When parsing a sentence, our parser first applies the core set of unary constraints to eliminate as many role values from the roles as possible. By reducing the role domain size before arc construction, the dimensions of the arc matrices created for binary constraint propagation are reduced. Unary constraints are first applied to the network, which in turn applies them to the nodes, which then applies them to the role values associated with each of its roles. Next, the network constructs the arcs for the constraint network. Once the arcs are created, binary constraints are applied to the pairs of role values which index the entries of the matrices associated with each arc. Each set of constraints is applied twice to each arc, once where the x_index and y_index corresponds to x and y respectively in the constraints, and once where the x_index and y_index corresponds to y and x.

At this point, an X-windows interface for the parser appears on the screen. Figure 15 depicts the interface to an SLCN. Once the window is present, the user can choose from several menu options. Among the options available are: perform a single step of arc consistency, achieve arc consistency, view the arcs joining the roles of two word nodes, apply constraints from a file, print node information to a file, and print arc information to a file. The user can also view the state of the parse. This interface allows the remaining role values for each word candidate's roles to be viewed by clicking on the word in the word node. For example, the role values for the three roles for the word *windows* over the interval (3,5) in Figure 15 are viewed by clicking on that word. This interface also allows viewing of the matrices stored on each of the arcs in the network. The arcs joining the roles of two words are displayed when a user selects the appropriate menu item along with two word nodes. Figure 16 shows the arcs joining the roles for the words *of* and *the*. The matrix associated with each of the arcs can then be viewed by clicking on its arc. Figure 17 shows the matrix for the arc joining the governor roles of the two words.

THE MASPAR INTERFACE

Since our parser applies the same constraint to every role value or every pair of role values, the algorithm lends itself to implementation on a SIMD machine. Also, since inter-processing element (PE) communication demands are small, it can be efficiently implemented on a distributed-memory system. The MasPar MP-1 is such a system: a massively parallel SIMD computer, which supports up to 16K 4-bit PEs, each with 16KB of local memory. Because typical English sentences contain on the order of tens of words, the MP-1 has a sufficient number of processors for this algorithm. The MasPar also has a powerful communication network which implements primitives that allow logarithmic-time ANDing and ORing of data values stored in the PEs[35].

The language in which our algorithm is implemented is AMPL (Ansi MasPar Language)[30], an extension of C that supports the SIMD parallelism of the MasPar. The data visualization capabilities and the well integrated and extensive debugging support of the MasPar made the job of implementing the parsing algorithm much simpler.

Integrating the MasPar with our sequential implementation was facilitated by the object-oriented structure of the parser. The routines and data structures necessary to manage the propagation of constraints are encapsulated in a C++ object called network (defined in net.cc). The network object is depicted in Figure 14. The network object, together with the top-level flow of control code, corresponds to the ''PARSEC parser'' process depicted in Figure 8.

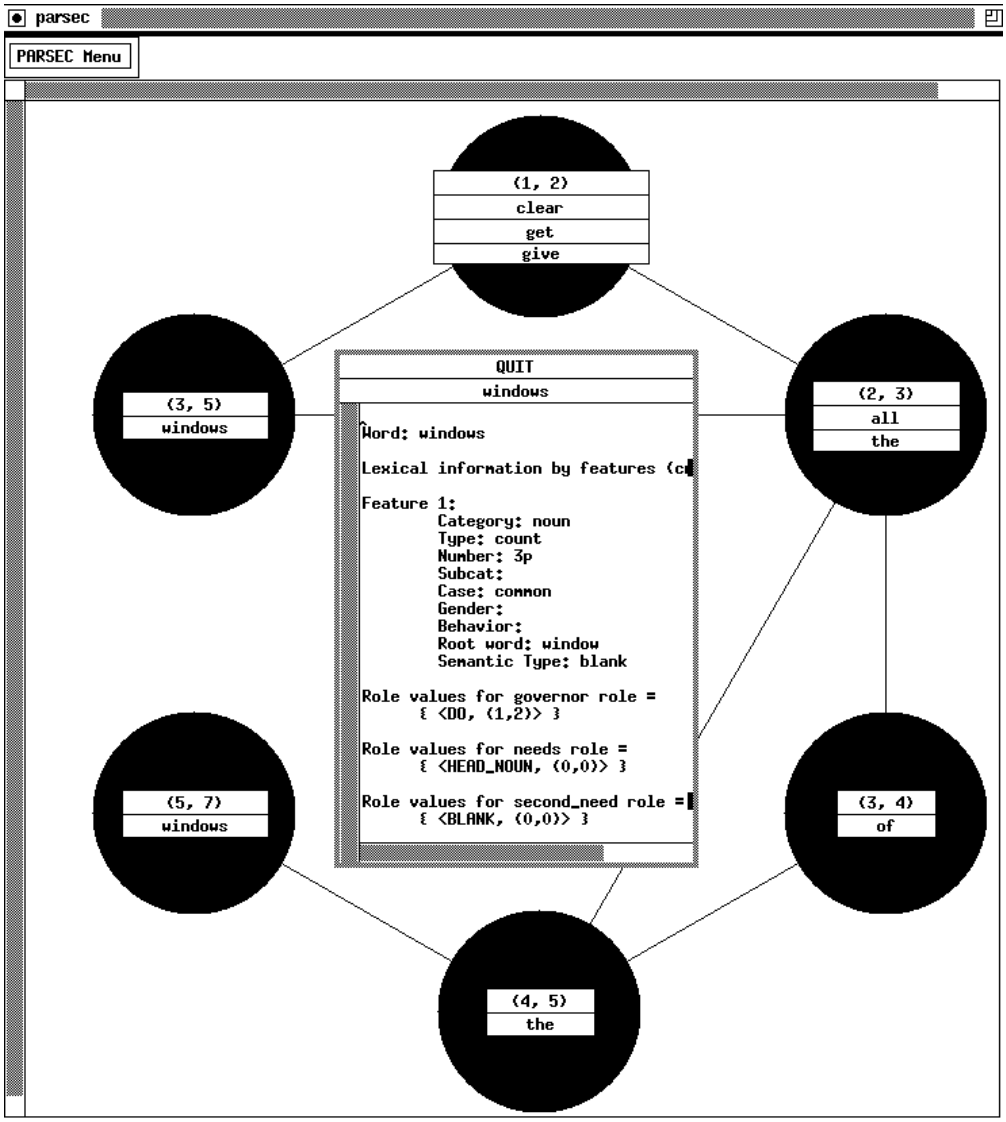

Figure 15. The X-windows interface to an SLCN.

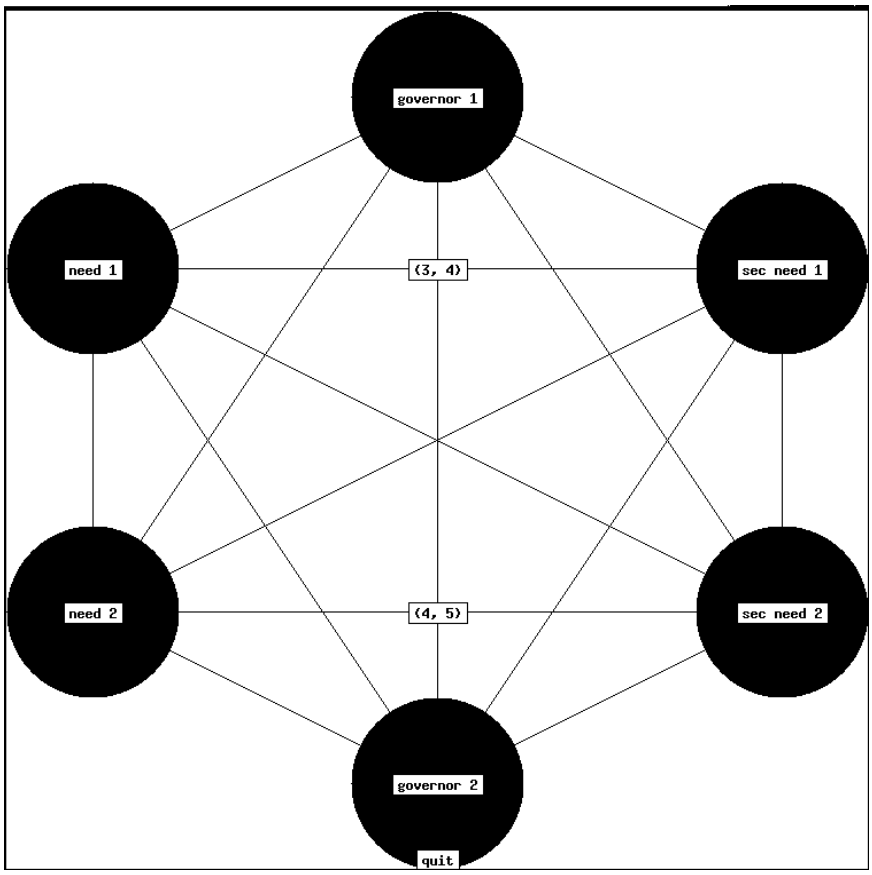

Figure 16. The X-windows interface to the arcs connecting the roles for of *and* the.

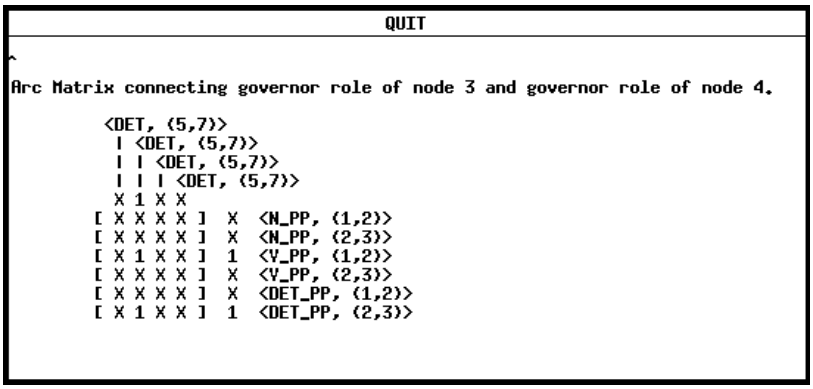

Figure 17. The X-windows interface to the arc matrix for the governor roles for of *and* the.

In order for the system to utilize the MasPar, it was only necessary to create a new class, maspar_network, which inherits from network the knowledge necessary to communicate with the other modules (such as the sentence reader and the X-windows interface). The maspar_network object translates the normal messages that objects use to communicate with each other into the TCP/IP socket communications that connect the Sparcstation to the MasPar.

The MasPar version of our parser was modified so that the network could be laid out on the PEs and so that the operations of constraint propagation and arc consistency could be carried out in the parallel architecture. In addition, the constraint compiler was modified to generate AMPL code rather than C++ code. The MasPar parser can be thought of as a slave to the sequential parser. When the parser is invoked using the -M flag, the MasPar parser does the parsing, but the information associated with the parse is displayed on the sequential machine. The user of the parser must have a compiled version of the parser on the MasPar and a compiled version of the parser on the Sun Sparcstation. When the user invokes the parser using the -M flag, a stream and socket communication is set up with the MasPar MP-1 and the sentence is passed to the MasPar for Parsing. Once the MasPar is finished parsing, the X-windows interface is displayed on the sequential machine, and the user can view the state of the parse on the MasPar by using the interface as before. Speedup from O(n^4) to O($log(n)$) has been observed[26,27].

The parallel parser is somewhat simpler than the serial version because many of the techniques needed to optimize performance in the serial implementation (such as interleaved arc consistency) are unnecessary. In our massively parallel implementation, each arc matrix element can often be associated with its own processor (we use processor virtualization when this is not possible). This entails a radical departure from the network construction and constraint propagation modules, but the word candidate generator, lexicon, and other modules are conceptually the same. Many of the modules developed for the serial algorithm were reused for the MasPar (close to 90% of our code did not require modification). We consider the fact that we could reuse so much of our code, even across machines as radically different from one another as a single-user workstation and a massively parallel SIMD machine, to be a striking confirmation of the validity of the object-oriented approach.

In a medium scale parallel processing implementation, (10-100 processors) the nodes and arcs would all be on separate processors. We have written the serial code such that messages passed between these objects could be replaced by IPC calls between loosely coupled CPU's and run in parallel. Because local computation dominates the communication overhead, we believe that almost linear speedup in parallel processing time could be achieved on a network of workstations communicating over ethernet.

Establishing A Socket Connection with the MasPar

One limitation of the AMPL programming environment is that it does not include libraries that allow its programs to communicate with other networked machines with sockets. In order to establish a socket connection between our Sun front end and the MasPar, we had to use an indirect technique which might be of use to other MasPar programmers who wish to tap power of the MasPar in programs that run on other machines.

Even though AMPL itself has no knowledge of what sockets are or how they work, the C compiler for the DEC front end can fork off AMPL-based child processes. Because the child process inherits the file descriptor table from the parent process, we can first establish a socket connection in the parent program and then give it to the forked child. The child can then use the socket using standard read and write commands as if the socket were a normal file.

The code fragments that do this are shown in Figures 18 and 19. These fragments are part of a complete skeleton demonstration code which is available via ftp from:

```
int sock, length;
struct sockaddr_in server;
int msgsock;
char buf[1024];
int rval;
int kidpid;
char parg[5];

/* create socket */

sock=socket(AF_INET, SOCK_STREAM, 0);
if(sock<0){
   perror("opening stream socket");
   exit(1);
}

/* name socket using wildcards. */

server.sin_family=AF_INET;
server.sin_addr.s_addr=INADDR_ANY;
server.sin_port=2123;
if(bind(sock, (struct sockaddr *)&server, sizeof server)<0){
   perror("getting socket name");
   exit(1);
}

printf("%d\n", ntohs(server.sin_port));
fflush(stdout);

/* start accepting connections. */

listen(sock, 5);

msgsock=accept(sock, (struct sockaddr *)0, (int *)0);
if(msgsock==-1){
   perror("accept\n");
   exit(1);
}

if(kidpid=fork()){     /* I'm the parent */
   exit(1);
}
else{                  /* I'm the child */
   sprintf(parg, "%d\0", msgsock);
   execl("maspar_back_end", "maspar_back_end", parg, (char *)0);
}
```

Figure 18. Code fragment that runs on the DEC front end. It is spawned by the Sun, which establishes a socket connection with it, and then it forks a MasPar process.

ftp://en.ecn.purdue.edu/parsec/sockets.tar.Z.

Network Construction and Arc Layout on the MasPar

After the Sun establishes a socket connection with the MasPar, it sends the MasPar the information needed to parse the sentence, including the word node, lexical category, and feature information for the sentence. The MasPar reads in this input, constructs the parallel

```
int initialize_maspar_connection(){
   /* This routine takes care of spawning the server and
       setting up a socket connection to it. It returns the
       number of the socket, or -1 upon any sort of error. */

   int sock;
   struct sockaddr_in server;
   struct hostent *hp, *gethostbyname();
   char buf[1024];
   FILE *pp;
   int remote_socket;
   int command;

   /* spawn the server */

   system("rsh -n marvin stream_accept &");
   sleep(4);

   /* create socket. */

   sock=socket(AF_INET, SOCK_STREAM, 0);
   if(sock<0){
      perror("opening stream socket");
      return -1;
   }

   server.sin_family=AF_INET;
   hp=gethostbyname("marvin");
   if(hp==0){
      fprintf(stderr, "%s: unknown host", "marvin");
      return -1;
   }

   bcopy((char *)hp->h_addr, (char *)&server.sin_addr, hp->h_length);
   server.sin_port=htons(19208);

   if(connect(sock, (struct sockaddr *)&server, sizeof server)<0){
      perror("connecting stream socket");
      return -1;
   }

   return sock;
}
```

Figure 19. Function called on the Sun to establish a connection with the MasPar.

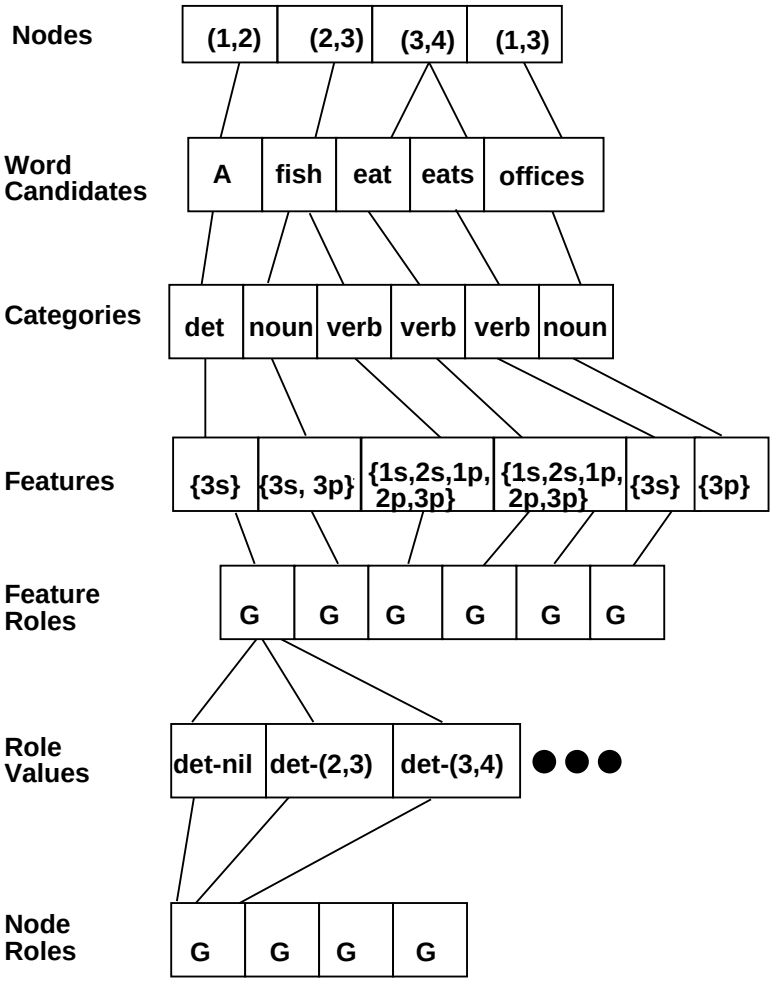

Figure 20. The PE layout of the SLCN in Figure 6 on the MasPar.

constraint network, and then propagates the constraints in parallel. It informs the user when the parse is complete, and the user, using the X-Windows interface described previously, can query the MasPar about the state of the parse.

The parallel constraint network is conceptually similar to the serial constraint network. Instead of C++ objects, PEs and AMPL structures are used to represent nodes, roles, lexical categories, features, feature roles, and role values. In place of message passing between objects, communication between the PEs is accomplished through the use of the flexible *sendwith* family of communication primitives, which includes sendwithADD, sendwithAND, and sendwithOR. Calling sendwithAND($destination$,$data$) sends *data* to the *destination* PE which receives the logical AND of all of the data sent to it.

The n nodes of the constraint network depicted in Figure 20 are mapped onto the first n PEs of the MasPar. Each PE maintains the following information about the node it represents: the position of the node in the sentence and which PEs represent its word candidates, lexical categories and roles. The program determines the number of roles per word node in the network, *numb-roles*, and the roles for the nodes are mapped onto the next *numb-roles* * n PEs. After *numb-roles* is broadcast to each of these PEs, it can calculate the word node it belongs to from its processor number, as well as its role-id.

Each of the word candidates can have an arbitrary number of parts of speech. As the program reads in each word, it looks up the possible parts of speech for each word and assigns a PE to represent that category. The PE has a structure that contains a valid flag to indicate at the end of parsing whether the part of speech is sanctioned by at least one consistent role value. To facilitate communication with its parent node, the PE also stores an integer that represents the processor number of the node to which it belongs. It uses this to inform its parent node if it becomes unsanctioned.

For each lexical category of a word, there may be many possible values for each of the features: type, number, subcategory, behavior, case, gender, and semantic type. The features for each part of speech are sent to the MasPar by the front end Sun after dictionary lookup. The PE for each lexical category delegates the responsibility of keeping track of legal feature values to another PE, which stores the PE number of the parent lexical category. This PE will not require the duplication of role values for features with more than a single value until a feature constraint is applied to its role values. If a constraint tests a feature with more than a single value, then the PE will duplicate the role values associated with it and assign them each a single feature value. If a feature value looses support, this PE records that information. If all feature values loose support, then the PE must report this to its parent PE.

For each feature PE, there are *numb-roles* roles to keep track of, which we call feature roles. This is a convenient way to keep track of the role values associated with the many possible lexical categories and feature values associated with words in a sentence. Feature roles are constructed similarly to the node roles. After a global broadcast tells each PE the value of *numb-roles*, it calculates the PE number of its parent feature, the PE number of its node role, and its role-id. If a feature role no longer contains any legal role values, it passes that information to the PE associated with its features and the PE associated with the role of its word node.

After feature role construction, it is a simple matter to create the role values by appropriate local computation. In this implementation, we restrict the domains of the feature roles to those role values that are appropriate using the table of legal labels for a given category and role id. After the role values for the feature roles are created, unary constraints are propagated over the role values. If a role value violates a unary constraint, it informs its feature role and node role that it has been eliminated from the network.

Next, the arcs are constructed. Figure 21 illustrates the arc layout. The x-role values and y-role values are simply all of the role values remaining after unary constraints have been propagated. If it is possible to assign each element to a PE, one entry is assigned to a single PE; otherwise, we use processor virtualization. Each of the arc element PEs must then propagate the binary constraints, which it receives by global broadcast, for the x-y pair of role values.

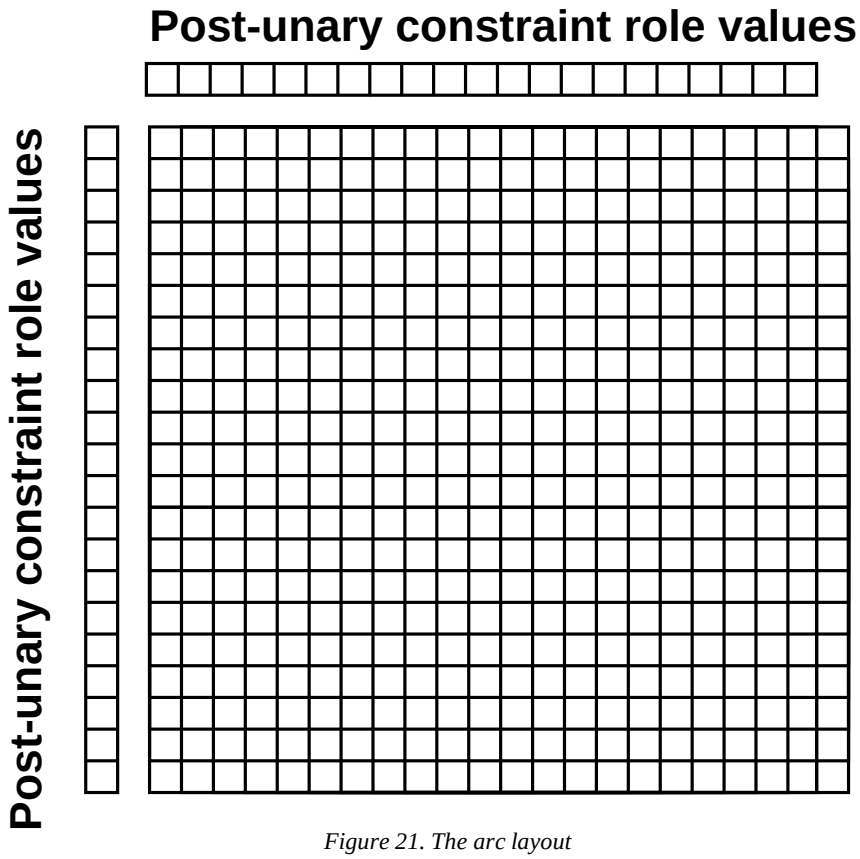

Figure 21. The arc layout

PARSER EXPERIMENTS

In order to demonstrate the effectiveness of our SLCN parser, we have developed two grammars. The first grammar was designed to parse sets of sentences in the Resource Management database[36] The second grammar covered sentences in the ATIS database[37,38] (Air Travel Information System).

The Resource Management database grammar contains 3 roles, 11 categories, 70 labels, 100 unary constraints, and 200 binary constraints, capable of parsing statements, yes-no questions, commands, and wh-questions[15]. The constraints consist of phrase structure rules and features tests. The feature tests include subject-verb agreement, determiner-head-noun agreement, and case restrictions on pronouns. Additionally, the subcategorization feature is used to make certain that a verb has the appropriate set of objects and complements to be complete. The lexicon used along with this grammar contains many lexically ambiguous words, and its word entries contain the necessary feature information to support our feature value constraints.

To demonstrate the effectiveness of CDG parsing for eliminating sentence hypotheses from a word graph, we converted the word graphs described into SLCNs and parsed them using the first grammar. More grammatical sentences were parsed in the SLCN than were available in the original sets of sentences, however, all of the additional parses had similar meanings to one of the original grammatical N-best sentences.

Syntactic constraints are effective at pruning a word graph of many ungrammatical sentence hypotheses and limiting the possible parses for the remaining sentences. However, it is often the case that syntactic information alone is insufficient for selecting a single sentence hypothesis from a word graph. Effective use of multiple knowledge sources plays a key role in human spoken language understanding. It is, therefore, likely that advances in spoken language understanding will require effective utilization of higher level knowledge.

We conducted a second experiment to determine the effectiveness of syntactic and semantic constraints for reducing the ambiguity of word networks constructed from sets of BBN's N-best sentence hypotheses[39] from the ATIS database (Air Travel Information System). For this experiment, we selected twenty sets of 10 N-best sentence hypotheses for three different types of utterances: a command, a yes-no question, and a wh-question. The lists of the N-best sentences were converted to word graphs in which the duration of each node was determined by maintaining a syllable count through the utterance. Syntactic constraints were added to the Resource Management grammar to demonstrate the ease of adding additional grammar constraints to a previously developed grammar. Then semantic constraints were constructed to further limit ambiguity[17]. Semantic constraints were relatively easy to create and incorporate into our parser. In fact, they were added to the grammar without modifying a single syntactic rule. Preliminary work also suggests that prosodic constraints should be as simple to add to our grammar[40,41].

Syntactic and semantic constraints were very useful for pruning syntactically or semantically anomalous word nodes from an SLCN. On the average, 3.11 word nodes per SLCN were eliminated by syntactic constraints and .66 by semantic constraints. However, they do not, in many cases, sufficiently constrain the SLCN to a single sentence hypothesis with a single parse. Contextual information represents an additional knowledge source that can be exploited to reduce the ambiguity in an SLCN.

CONCLUSION

We have described the implementation of PARSEC, a constraint-based parser that provides the flexibility a user needs to design and test constraint grammars. The constraints and lexicon are checked for consistency with the grammar, simplifying the process of creating the grammar. Constraints can be either compiled for efficiency or they can be interpreted so that the impact of a new constraint can be tested without recompiling the constraints. The

X-windows interface allows the user to view the state of a parse of a sentence, test new constraints, and dump the constraint network to a file. The user also has the capability to perform the computationally expensive constraint propagation steps on the MasPar MP-1. We utilize stream and socket communication to interface the MasPar constraint parser with a standard X-windows interface to the state of the parse on our Sun Sparcstation.

PARSEC is a heterogeneous software system because of the variety of languages and tools it utilizes and because it is capable of operating both serially and in parallel. By using object-oriented techniques and PCCTS, our system evolved into an integrated tool for researchers to explore the power and flexibility of a constraint-based approach to disambiguating spoken language.

One issue that remains open is how to enable a user of our system to create a new feature type without modifying the C++ code. Creating a new feature type presents a problem if the user wants the feature type's values to be inherited from other word entries in the lexicon. Also, it is difficult to determine what types of feature tests a user might wish to use. Our system supports seven features, for which users must use our dictionary conventions and feature tests, unless they wish to modify the appropriate C++ code segments.

ACKNOWLEDGEMENTS

This work was supported in part by Purdue Research Foundation, NSF grant number IRI-9011179, and NSF Parallel Infrastructure Grant CDA-9015696.

REFERENCES

1. H. Maruyama, 'Constraint dependency grammar', Technical Report #RT0044, IBM, Tokyo, Japan, (1990).
2. H. Maruyama, 'Constraint dependency grammar and its weak generative capacity', *Computer Software*, (1990).
3. H. Maruyama, 'Structural disambiguation with constraint propagation', *The Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 1990, pp. 31--38.
4. M. P. Harper and R. A. Helzerman, 'PARSEC: A constraint-based parser for spoken language parsing', Technical Report EE-93-28, Purdue University, School of Electrical Engineering, West Lafayette, IN, (1993).
5. A. L. Davis and A. Rosenfeld, 'Cooperating processes for low-level vision: A survey', *Artificial Intelligence*, **17**, 245--263, (1981).
6. R. Dechter, I. Meiri, and J. Pearl, 'Temporal constraint networks', *Artificial Intelligence*, **34**, 1--38, (1988).
7. R. Dechter and J. Pearl, 'Network-based heuristics for constraint-satisfaction problems', *Artificial Intelligence*, **34**, 1--38, (1988).
8. E. Freuder, 'Partial constraint satisfaction', *Proceedings of the International Joint Conference on Artificial Intelligence*, 1989, pp. 278--283.
9. E. Freuder, 'Complexity of K-tree-structured constraint-satisfaction problems', *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, pp. 4--9.
10. V. Kumar, 'Algorithms for constraint-satisfaction problems: A survey', *AI Magazine*, **13**, (1), 32--44, (1992).
11. A. K. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, (1), 99--118, (1977).
12. A. K. Mackworth and E. Freuder, 'The complexity of some polynomial network-consistency algorithms for constraint-satisfaction problems', *Artificial Intelligence*, **25**, 65--74, (1985).
13. M. Villain and H. Kautz, 'Constraint-propagation algorithms for temporal reasoning', *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, pp. 377--382.
14. P. Dey and B. R. Bryant, 'Lexical ambiguity in tree adjoining grammars', *Information Processing Letters*, **34**, 65--69, (1990).
15. C. B. Zoltowski, M. P. Harper, L. H. Jamieson, and R. A. Helzerman, 'PARSEC: A constraint-based framework for spoken language understanding', *Proceedings of the International Conference on Spoken Language Processing*, October 1992, pp. 249--252.
16. R. A. Helzerman and M. P. Harper, 'An approach to multiply segmented constraint satisfaction problems', *Proceedings of the National Conference on Artificial Intelligence*, July 1994.
17. M. P. Harper, L. H. Jamieson, C. B. Zoltowski, and R. A. Helzerman, 'Semantics and constraint parsing of word graphs', *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume II, April 1992, pp. 63--66.
18. J. Bear and P. Price, 'Prosody, syntax, and parsing', *Proceedings of the 28th Annual Meeting of Association for Computational Linguistics*, 1990, pp. 17--22.

19. J. Allen, *Natural Language Understanding*, The Benjamin Cummings Publishing Company, Inc., Menlo Park, CA, 1987.
20. M. P. Harper, L. H. Jamieson, C. D. Mitchell, G. Ying, S. Potisuk, P. N. Srinivasan, R. Chen, C. B. Zoltowski, L. L. McPheters, B. Pellom, and R. A. Helzerman, 'Integrating language models with speech recognition', *Proceedings of the AAAI-94 Workshop on the Integration of Natural Language and Speech Processing*, 1994.
21. A. K. Joshi, L. S. Levy, and M. Takahashi, 'Tree adjunct grammars', *Journal of Computer and System Sciences*, **10**, 136--163, (1975).
22. K. Vijay-Shanker and A. K. Joshi, 'Some computational properties of tree adjoining grammars', *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, 1985, pp. 82--93.
23. K. Vijay-Shanker, D. J. Weir, and A. K. Joshi, 'Characterizing structural descriptions produced by various grammatical formalisms', *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1987, pp. 104--111.
24. S. R. Kosaraju, 'Speed of recognition of context-free languages by array automata', *SIAM Journal of Computing*, **4**, (3), 331--340, (September 1975).
25. W. Ruzzo, 'Tree-size bounded alternation', *Journal of Computers and System Sciences*, **21**, 218--235, (1980).
26. R. A. Helzerman, *PARSEC: A Framework for Parallel Natural Language Understanding*, Master's thesis, Purdue University, School of Electrical Engineering, West Lafayette, IN, 1993.
27. R. A. Helzerman and M. P. Harper, 'Log time parsing on the MasPar MP-1', *Proceedings of the Sixth International Conference on Parallel Processing*, August 1992, pp. 209--217.
28. M. A. Palis, S. Shende, and D. S. L. Wei, 'An optimal linear-time parallel parser for tree adjoining languages', *SIAM Journal of Computing*, **19**, 1--31, (1990).
29. M. D. Moshier and W. C. Rounds, 'On the succinctness properties of unordered context-free grammars', *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, 1987, pp. 112--116.
30. MasPar Corp., *MasPar Parallel Application Language MPL Reference Manual*, March, 1991. PN 9302-0000.
31. T. J. Parr, H. Dietz, and W. E. Cohen, *PCCTS reference manual*, Purdue University, August 1991. Version 1.00.
32. R. A. Helzerman, M. P. Harper, and C. B. Zoltowski, 'Parallel parsing of spoken language', *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, October 1992, pp. 528--530.
33. C. L. Forgy, 'RETE: A fast algorithm for the many pattern/many object match problem', *Artificial Intelligence*, **19**, 17--37, (1982).
34. M. J. Rowland, B. Perazich, R. A. Helzerman, M. P. Harper, J. P. Robertson, G. D. Rogers, J. R. Johoski, E. Toepke, and H. Rosario, 'Parsing with the PARSEC vector processing chip', *Proceedings of the Sixth IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems*, October 1994.
35. MasPar Corp., *MasPar System Overview*, July, 1990. PN 9300-0100-2790.
36. P. J. Price, W. Fischer, J. Bernstein, and D. Pallett, 'A database for continuous speech recognition in a 1000-word domain', *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1988.
37. C. T. Hemphill, J. J. Godfrey, and G. R. Doddington, 'The ATIS spoken language systems pilot corpus', Technical Report NTIS PB91-505354, (1990). NIST Speech Disc 5-1.1.
38. P. J. Price, 'Evaluation of spoken language systems: The ATIS domain', *Proceedings of the DARPA Workshop on Speech and Natural Language*, 1990, pp. 91--95.
39. R. Schwartz and Y-L. Chow, 'The N-best algorithm: An efficient and exact procedure for finding the N most likely sentence hypotheses', *IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1990, pp. 81--84.
40. M. P. Harper, R. A. Helzerman, and C. B. Zoltowski, 'Constraint parsing: A powerful framework for text-based and spoken language processing', Technical Report EE-91-34, Purdue University, School of Electrical Engineering, West Lafayette, IN, (1991).
41. C. B. Zoltowski, 'Current research in the development of a spoken language understanding system using PARSEC', *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, 1991, pp. 353--354.