Proceedings of the 1988 Open House

César A. Quiroz, Editor

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 209

May 1988
Table of Contents

Introduction .................................................................................................................. i

Language Instance Generation and Test Instance Construction
for NP-hard Problems
Laura A. Sanchis ........................................................................................................ 1

Attacking the I/O Bottleneck with Parallelism
Peter C. Dibble ........................................................................................................ 13

PENGUIN: A Language for Specifying User Interface Dialogues
Sue-Ken Yap ............................................................................................................... 19

Direct Inferences in a Connectionist Knowledge Structure
S. C. Hollbach ............................................................................................................ 29

List Productions in a Parallel Incremental Parser
Neal M. Gafier .......................................................................................................... 37

Object Recognition from a Large Database
Michael Swain .......................................................................................................... 47

Plans, Speech Acts, and Conversational Implicature
Elizabeth A. Hinkelman ............................................................................................ 59

A Model for Parallel Programming
Lawrence A. Crowl .................................................................................................... 71
Introduction

This document records the Proceedings of our 1988 Open House, the second one organized by our Department.

The idea of our Open Houses is to give the graduate students of the Department an opportunity to show the progress of their research. The objective is to reach the rest of the Department (not everyone can manage to keep track of what everyone else is doing, so this is a chance to catch up), the rest of the University, the Rochester community interested in Computer Science, and our colleagues in nearby cities.

The format of these events is still subject to evolution: this year it consists of eight talks spread in four sessions, sprinkled with lab tours and demos of work in progress. The general intention is to maintain a conference/workshop structure, but trying to keep all the activity to one day.

The idea of these events was born in 1985, and must be credited to Jim Geller of SUNY Buffalo. He noticed both the need to establish these contacts, and the rich opportunities that having so many fine universities around this area permits. He then organized a meeting of about this same format in Buffalo, and suggested to Richard (Nemo) Newman-Wolfe to organize a similar one in Rochester. Nemo adopted the idea enthusiastically. His efforts (and of those who worked with him) produced our first Open House, in March 1986.

The idea at the time was to continue these Open Houses annually, and to extend them eventually into regional events. The first goal originated in the realization that these meetings provide valuable visibility to the work of the Department, and so should become an activity held regularly. The second of these goals consisted in the desire to infect with this idea other schools, as well as in the intention to invite graduate students of those schools to present their work in our meetings; the motivation was to reap the benefits of the closeness of our research institutions, by establishing a network of reciprocating "small conferences".

Due to various reasons, we haven't succeeded in reaching those goals yet. For one thing, we didn't have an Open House in 1987, so we are still working in gaining the momentum that will make these events occur annually. For another, we haven't managed to persuade more schools to organize their own Open Houses. Our neighbors in Buffalo have been more faithful to the plan, as they have already had three of these events (that they renamed "Graduate Conferences"), and for the latest one they invited speakers from three other schools (Waterloo, Toronto and Rochester).

We are still going through the growing pains of our Open Houses, and so this year we decided to keep the event local. I hope that in due time we will be able to host larger meetings, much in the direction pioneered by our colleagues in Buffalo. In the meantime, the experience gained this year should be valuable towards establishing the Open Houses as a regular part of our Department's calendar.

As usual, undertakings of this magnitude are the work of many people. I hope to mention here at least some of those involved in making this Open House happen, with my sincerest apologies to the many I may be unintentionally forgetting. (Some of them go unmentioned here just because,
as of this writing, I still don’t know who will do some of the work in the actual day of the Open House.)

First of all, I have the pleasure of having worked with a splendid committee: Patricia Armstrong, John Mellor-Crummey, Rob Fowler, Stuart Friedberg, and Jay Weber. Some of them have been working since early 1987 in making this Open House possible. Peggy Frantz, Peggy Meeker and Rose Peet provided (as always) their invaluable help both in setting administrative details straight and in getting these Proceedings printed. Our Department Chairman, James Allen, supported our efforts. My warmest thanks to all of them.

A special word of thanks to Lawrence Crowl, who designed the layout and wrote the \TeX code needed for these papers. He and Ken Yap contributed their \TeX expertise to steer unrepentant TROFF users (the Editor included) through the writing of this document.

I also want to thank the authors. They shared in the vision that this activity is worthwhile doing and repeating. They also put up cheerfully with my deadlines and random editorial demands, in whose spite they managed to produce the outstanding reports that form this document.

And, finally, I cannot forget thanking Jim Geller and Nemo, whose enthusiasm and perseverance got all of this moving in the first place.

César A. Quiroz  
Chairman, Open House Committee

\footnotesize
\begin{itemize}
\item \textsuperscript{1} And approved the funding!
\item \textsuperscript{2} OK, almost.
\end{itemize}
Language Instance Generation
and Test Instance Construction for NP-hard Problems

Laura A. Sanchis

Abstract

The performance of heuristic approximation algorithms for NP-hard problems can often only be determined by experimentation. This paper explores some of the issues involved in the efficient generation of useful test sets for such problems; i.e., test sets consisting of instances of the problem for which the answer is known and having other properties useful for testing the performance of approximation algorithms. Some theoretical results are given concerning what kinds of test sets can and cannot be generated; these are derived by examining the complexity of length-restricted instance generation for languages in NP and co-NP. Also examples of test set generation procedures are presented.

1 Introduction

As a consequence of the apparent intractability of NP-complete problems, many of which are of great practical importance, much attention has been devoted to designing and evaluating approximation algorithms for NP-complete and NP-hard problems. One common type of approximation consists of an algorithm which runs in polynomial time but which is not guaranteed to provide the correct answer for all instances of the problem. These algorithms are usually based on heuristics and intuitions about the problem. Sometimes some facts can be proved about the worst case or expected performance of an approximation algorithm which certify that the approximation is "good" in a certain sense. Many times, however, empirical testing is the only means of determining the effectiveness of the algorithm. In order to do such testing, it is useful to have test cases for which the correct answer is known. However, such test cases are not easy to obtain, since given an arbitrary instance of an NP-hard problem there is no known efficient procedure for determining the answer. This paper deals with some of the issues involved in the efficient generation of useful test sets for NP-hard problems.

Various methods are commonly used to generate test cases for hard problems. Some of these consist of generating random instances of the problem and then doing some informed guessing to approximate what the correct answer should be for each generated instance. In this paper we will instead concentrate on the generation of instances of the problem for which the correct answer is known a priori (see for instance [Krishnamurthy 1985]). We will consider whether it is possible to efficiently construct instances of problems having certain desired characteristics, such as a given size and answer. We will also be concerned with the generation of sets of instances which are hard; i.e., sets of instances for which no polynomial time approximation algorithm can give the right answer.

Section 2 deals with the notions of construction and generation of instances of a language. The principal results concern efficient length-restricted instance generation for languages in NP and co-NP; this concept will be shown to have direct bearing on the test set construction problem for NP-hard problems. The fact that a language can be recognized in deterministic or nondeterministic

1 AT&T Bell Labs Scholar
polynomial time does not immediately imply that it or its complement can be nondeterministically generated in polynomial time. By generation of a language we mean that there exists a nondeterministic machine which when given as input a positive integer, nondeterministically outputs some string in the language having the input size; moreover for every string in the language there is some computation of the machine which will output the string (i.e., the string has nonzero probability of being output). We investigate the question of whether or not all languages in NP can be efficiently generated. This question is still open but some of the consequences of deciding this question are shown. It is also shown that complements of NP-complete languages cannot be efficiently generated unless NP=co-NP. The notion of length-restricted generation can also be generalized to the generation of instances having other specific polynomial time checkable parameter values.

Section 3 relates the results of section 2 to the problem of test case construction for NP-hard problems, and gives some results concerning the interaction between efficient test case generation and the generation of hard instances. It is shown that for NP-complete problems and for certain kinds of NP-hard optimization problems it is not possible to have an efficient test set construction method which is capable of generating all possible instances of the problem with their respective answers, unless NP=co-NP. Given the fact that we probably cannot achieve generation of all instances, there are some requirements which we want to impose on useful test set construction methods. One of these is the ability to obtain instances having various desired characteristics; i.e., we want test sets having a variety of different kinds of instances. This problem is related to the parameter based generation discussed at the end of section 2. Another requirement is that there be no polynomial time algorithm which can solve the problem for all instances generated by the construction method. In other words, we do not want to generate sets of easy instances. We call test sets which cannot be polynomially approximated as just described, with respect to the problem for which they are designed, non-approximable. Section 3.2 presents some definitions and observations on non-approximability as it relates to efficiently generated sets of instances.

Finally, section 4 presents an example of a test case construction method for the minimum vertex cover problem. The generation procedure described produces test sets which can be shown to be non-approximable. The procedure also has the property that instances with any given valid combination of answer, number of vertices, and number of edges, can be constructed.

## 2 Language Instance Construction and Generation

### 2.1 Generation for P and NP languages

**Definition 2.1**

1. A polynomial time tallier (PTT) for a language $L$ is a polynomial time deterministic machine which on input $1^n$ outputs 1 if there exist strings of length $n$ in $L$ and outputs 0 otherwise.

2. A polynomial time constructor (PTC) for a language $L$ is a polynomial time deterministic machine which on input $1^n$ outputs a string in $L$ of length $n$, if such a string exists, and outputs $\Lambda$ otherwise.

3. A polynomial time generator (PTG) for a language $L$ is a polynomial time nondeterministic machine which on input $1^n$ outputs a string in $L$ of length $n$, if such a string exists, and outputs $\Lambda$ otherwise. Moreover, for every string $x$ in $L$ of length $n$ there exists some computation of the generator on input $1^n$ which outputs $x$.

It is interesting to ask which languages have PTT's, PTC's, and/or PTG's. It is clear that any language that has a PTG has a PTC, and any language that has a PTC has a PTT. In addition any language that has a PTG must be in NP, while any language in NP which has a PTC actually
has a PTG. If a language \( L \) in NP has a PTC \( C_L \), we can define a PTG \( G_L \) for \( L \) as follows. On input \( 1^n \), \( G_L \) randomly constructs a string \( x \) of length \( n \) and runs a polynomial time NDTM for \( L \) on \( x \). If the machine accepts, \( G_L \) outputs \( x \), otherwise it runs \( C_L \) on \( 1^n \) and outputs its output.

**Proposition 2.1** The following are equivalent:

(a) All languages in NP have PTT's.
(b) All languages in P have PTT's.
(c) There are no tally languages in \( NP \).  

**Proof:** (sketch) It is not hard to see that (a) \( \Rightarrow \) (b) \( \Rightarrow \) (c) \( \Rightarrow \) (a). For (b) \( \Rightarrow \) (c), let \( L \) be any tally language in NP with polynomial time NDTM \( M \) which accepts \( L \) and which runs in time \( p(n) \), where \( M \) makes exactly \( p(n) \) nondeterministic branches on input \( 1^n \) and \( p(n) \) is a strictly increasing polynomial. Define the language \( S \) to consist of all strings \( x \) of length \( p(n) \) such that \( M \) accepts on input \( 1^n \) if it follows the nondeterministic branches coded in \( x \). Clearly \( S \) is in P, so by assumption \( S \) has a PTT. Also \( 1^n \) is in \( L \) if and only if \( S \) has a string of length \( p(n) \). Hence \( L \) is in P. 

Turning now to PTC's and PTG's, we want to investigate which languages in NP have PTG's (or equivalently PTC's). It appears possible to construct PTG's for most known natural languages in NP. As an example, consider the following PTG for the 3-SAT problem. Given inputs \( 1^n, 1^m \) denoting respectively the number of variables \( x_1, x_2, \ldots, x_n \), and the number of clauses, the generator will randomly assign a truth value \( T \) or \( F \) to each of \( x_1, x_2, \ldots, x_n \). Let \( u_i = x_i \) if \( x_i \) was assigned \( T \), \( u_i = \overline{x_i} \) otherwise. To form each of the \( m \) clauses the generator will first randomly choose some \( u_i \), thus assuring that the clause will be true, and then randomly choose any 2 more variables for the clause from among \( x_1, x_2, \ldots, x_n, \overline{x_1}, \overline{x_2}, \ldots, \overline{x_n} \). Clearly each satisfiable formula with \( n \) variables and \( m \) clauses is generated in this manner. Moreover the number of times a particular satisfiable formula is generated is proportional to the number of different assignments that satisfy it. Notice that what is generated in each run is really a pair: an assignment for the variables accompanied by a formula which is satisfied with this assignment. (The input \( 1^n, 1^m \) to the above procedure is not actually the length of the string that will be generated, although it is polynomially related to it. It is however not hard to construct from this informally described procedure a formal PTG for some encoding of the 3-SAT language).

Now let \( S \) be the 3-SAT language in some suitable encoding in the alphabet \( \{0, 1\}^* \), and consider the language \( S' \) consisting of all strings \( x \# y \) such that \( x \) is the encoding of a truth assignment for \( n \) variables, and \( y \) is the encoding for a 3-SAT formula on \( n \) variables which is satisfied by \( x \). Clearly \( S' \) is in P. In fact, the way one usually shows that \( S \) is in NP is by noting that \( S' \) or some variant of it is in P. Moreover \( S' \) has another property, which is defined next.

**Definition 2.2** The prefix closure of a language \( L \) consists of the set of strings \( \{x \# y^m | m \geq 0, \exists y \text{ with } |y| = m \text{ and } xy \in L\} \).

**Definition 2.3** Prefix-P consists of the class of languages \( L \) such that the prefix closure of \( L \) is in P.

In other words, \( L \) is in Prefix-P if there exists a polynomial time algorithm for determining whether a string \( x \) is a prefix of a string in \( L \) of size \( n \). Clearly Prefix-P \( \subseteq \) P. The inclusion is proper iff and only if \( P \neq NP \). Also it is not hard to see that \( S' \) as defined above is in Prefix-P. Moreover if a language \( L \) is in Prefix-P it has a PTG which constructs a string of length \( n \) in \( L \) by successively adding a bit and checking whether the string constructed so far is a valid prefix for a string in \( L \) of length \( n \). We therefore have the following.
Proposition 2.2 If $P=NP$ then all languages in $NP$ have PTG's.

Even if $P \neq NP$, the class of languages that have PTG's is not restricted to those in Prefix-$P$. For instance, as shown above, $S$, the language encoding 3-SAT, has a PTG which generates all satisfiable formulas, even though unless $NP = P$, $S$ is not in $P$. The languages having such PTG's can be characterized as the images of languages in Prefix-$P$ under polynomial mappings of a certain type.

Definition 2.4 A language $L$ is prefixable if there exists a language $L'$ in Prefix-$P$, a polynomial $f$ which is one-to-one, and a polynomially computable onto function $g : L' \rightarrow L$ such that $|x| = f(|g(x)|)$ for all $x \in L'$.

The following proposition says that the prefixable languages are exactly those that have PTG's. The proof may be found in [Sanchis 1987].

Proposition 2.3 A language $L$ has a PTG iff it is prefixable.

Most natural languages in NP examined so far have PTG's. It is not known whether all languages in NP must have PTG's. The following corollary shows that to decide this question it is sufficient to resolve it for languages in $P$.

Corollary 2.4 All languages in NP have PTG's iff all languages in $P$ have PTG's.

Proof: (idea) This follows from the fact that any NP language can be prefixed by the language consisting of the accepting computations for the strings in the language. □

From Proposition 2.2 we know that if there is a language in $P$ (or $NP$) which is not prefixable, then $P \neq NP$. On the other hand, if all languages in NP are prefixable, we have the following consequence.

Proposition 2.5 If all languages in NP are prefixable then there are no sparse languages in NP-P.

Proof: If all NP languages are prefixable then they all have PTT's, and hence by Lemma 2.1 there are no tally languages in NP-P. [Book 1974] shows that there are no tally languages in NP-P iff $E=NE$, while in [Hartmanis 1983] it is shown that there are no sparse languages in NP-P iff $E=NE$. □

It is an open question whether the converse of the last statement is true. However, we can show that if there are no sparse languages in $D^P-P$ then all NP languages have PTC's. See [Sanchis and Fulk 1988] for the proof of this statement. (The class $D^P$ [Papadimitriou and Wolfe 1985] consists of all languages formed from the difference of two NP languages; i.e., $D^P = \{ L_1 - L_2 | L_1, L_2 \in NP \}$.) One may also ask whether $P \neq NP$ implies that not all NP languages have PTG's. The answer to this question appears very hard to resolve since we can construct a relativization for which $P \neq NP$ and all NP languages have PTG's, and another relativization for which $P \neq NP$ and yet there is a language in $P$ which does not have a PTG. These relativizations may be found in [Sanchis and Fulk 1988].
2.2 Parameter Based Generation

Up to now we have considered construction and generation of strings required to have a specified length. In practical situations it is usually desired to generate a string having certain parameter values related to length. For example, it may be required to obtain a CNF satisfiable formula having \( n \) variables and \( m \) clauses, or a graph possessing a certain property having a specified number of vertices and edges. This section presents a connection between this kind of parameterized generation and the length-restricted generation discussed in the previous section.

Definition 2.5 Let \( l_1, \ldots, l_k \) be polynomial time computable functions, \( l_j : \sum^* \rightarrow N \) for \( 1 \leq j \leq k \), and \( q, r \) be polynomials such that \( l_j(x) \leq q(|x|) \) for \( 1 \leq j \leq k \) and \( |x| \leq r(l_1(x), \ldots, l_k(x)) \) for all \( x \). An \((l_1, \ldots, l_k)\)-PTG for a language \( L \) is a polynomial time NDTM \( M \) which on input \( 1^{v_1} \# 1^{v_2} \# \ldots \# 1^{v_k} \), either outputs a string \( z \) in \( L \) such that for \( 1 \leq j \leq k \), \( l_j(z) = v_j \), or outputs the symbol \( \Lambda \) indicating that no such string exists. Furthermore for every string \( z \) in \( L \) such that \( l_j(z) = v_j \) for \( 1 \leq j \leq k \) there exists some computation of \( M \) on input \( 1^{v_1} \# 1^{v_2} \# \ldots \# 1^{v_k} \) which outputs \( z \).

Note that, as is the case for (length-restricted) PTG's, any language that has an \((l_1, \ldots, l_k)\)-PTG must be in NP.

Theorem 2.6 Let \( (l_1, \ldots, l_k) \), \( q, r \) be as in the above definition. Then if all languages in NP have (length-restricted) PTG's, then all languages in NP have \((l_1, \ldots, l_k)\)-PTG's.

Proof: Let \( L \) be a language in NP and suppose all NP languages have length-restricted PTG's. Define \( g(n, m) = (n + m - 1)(n + m - 2)/2 + n \). It is not hard to see that \( g : N^+ \times N^+ \rightarrow N^+ \) is injective and that \( g(n, m) \geq nm \) for all \( n, m \in N^+ \). By repeated compositions of \( g \) one can define a polynomial function \( g_{k+1} : (N^+)^{k+1} \rightarrow N^+ \) on \( k + 1 \) variables which is injective and for which \( g_{k+1}(n_1, \ldots, n_{k+1}) \geq n_1 \cdots n_{k+1} \) for all \( n_1, \ldots, n_{k+1} \in N^+ \). We will define a language \( S \) as follows. For \( x \in L \), let \( f(x) = g_{k+1}(l_1(x) + 1, \ldots, l_k(x) + 1, r(l_1(x), \ldots, l_k(x)) + 1) \) and let \( s(x) \) be the string of length \( f(x) \) consisting of the concatenation of \( x \) and \( f(x) - |x| \) * symbols (where * is not in \( \sum \)). For each \( x \in L \), put the corresponding \( s(x) \) in \( S \). \( S \) is clearly in NP, so if all languages in NP have (length-restricted) PTG's, \( S \) has a PTG \( G_S \). We can then define an \((l_1, \ldots, l_k)\)-PTG \( G_L \) for \( L \) as follows. On input \( 1^{v_1} \# \ldots \# 1^{v_k} \), \( G_L \) computes \( m = g_{k+1}(v_1 + 1, \ldots, v_k + 1, r(v_1, \ldots, v_k) + 1) \) and runs \( G_S \) on input \( m \). \( G_S \) will then output a string \( y \) of length \( m \) in \( S \) (if such a string exists); by construction of \( S \), \( y \) will consist of a string \( z \in L \) with parameter values \( v_1, \ldots, v_k \) followed by *'s. \( G_L \) should then output \( x \).

3 Instance Generation for NP-hard Problems

3.1 Efficient Generation

In this section we use the results of the preceding section to examine the kinds of test sets that can be efficiently generated for NP-hard problems. First we need to define what we mean by a test set generator. A test instance for a problem is a pair consisting of an instance of the problem together with its answer. A test instance construction method (TICM) for a problem \( \Pi \) is a machine that given some parameter values, usually including a size, and an answer \( A \), outputs in some nondeterministic (preferably random) fashion and in an efficient manner an instance of the problem with the given parameter values and having answer \( A \). If \( \Pi \) is a decision problem, \( A \) will be either yes or no, while for an optimization problem \( A \) will be one of the possible answers for the problem.
Definition 3.1 Let \( \Pi \) be a problem and \( l_1, \ldots, l_k \) be polynomial time computable functions whose domains consist of the set of instances for \( \Pi \) and whose ranges are the set of natural numbers and suppose \( l_j(I) \leq q(|I|) \) for all \( I, j \) for some polynomial \( q \). A test instance construction method (TICM) for \( \Pi \) (with respect to \( (l_1, \ldots, l_k) \)) is a polynomial time nondeterministic Turing machine \( C \) that given as input \( I \# \cdots \# I \# A \), outputs either an instance \( I \) of the problem having answer \( A \) such that for \( 1 \leq j \leq k \), \( l_j(I) = v_j \), or the special symbol \( A \), denoting that it cannot output any such instance. Denote by \( \text{Gen}(C, v_1, \ldots, v_k, A) \) the set of all instances that may be generated by \( C \) on input \( I \# \cdots \# I \# A \); by \( \text{Gen}(C, A) \) the set of all instances generated by \( C \) with answer \( A \); and by \( \text{Gen}(C) \) all instances generated by \( C \).

In most cases we expect \( l_1, l_2, \ldots, l_k \) to be some measure of the size of the desired instance. For example, for a satisfiability instance, \( l_1 \) could be the number of variables and \( l_2 \) the number of clauses. We have defined a TICM in the most general way, not putting any restrictions on how many instances it must be able to generate, or on the probabilities with which instances are generated. It must however run in polynomial time. Clearly some TICM's are more useful than others, based on the above considerations. We will look at what kinds of TICM's we can expect to obtain, based on the results of the preceding section. The main observation which links PTG's and TICM's is the following.

**Lemma 3.1** For each possible answer \( A \), a TICM \( C \) acts like a \( (l_1, \ldots, l_k) \)-PTG for the language \( \text{Gen}(C, A) \).

From this fact it follows that \( \text{Gen}(C, A) \) must be in NP for any TICM. We then have the following result.

**Lemma 3.2** If \( \Pi \) is an NP-complete decision problem, and if there exists a TICM \( C \) for \( \Pi \) such that \( \text{Gen}(C, \text{no}) \) consists of all instances of the problem for which the answer is no, then \( \text{NP=} \text{co-NP} \).

**Proof:** If \( \text{Gen}(C, \text{no}) \) consists of all instances of the problem for which the answer is no, then since \( \text{Gen}(C, \text{no}) \) is in NP, we have that the complement of an NP-complete language is in NP. Theorem 7.2 of [Garey and Johnson 1979] states that if the complement of an NP-complete language is in NP, then \( \text{NP=} \text{co-NP} \).

**Corollary 3.3** If \( \Pi \) is an NP-complete decision problem and there exists a TICM generating all instances of \( \Pi \), then \( \text{NP=} \text{co-NP} \).

Consider now NP-hard optimization problems. An NP-hard optimization problem \( \Pi' \) consists of instances \( I \) for which answers \( \text{OPT}(I) \) are being sought. Consider the language consisting of all tuples \( (I, \text{OPT}(I)) \) where \( I \) is an instance of \( \Pi' \). If it can be shown that this language is not in NP unless \( \text{NP=} \text{co-NP} \), then an argument similar to that used in the above proof shows that \( \Pi' \) cannot have a TICM generating all instances unless \( \text{NP=} \text{co-NP} \).

**Corollary 3.4** Let \( \Pi' \) be an NP-hard optimization problem consisting of instances \( I \) with corresponding answers \( \text{OPT}(I) \), and suppose that the language consisting of the tuples \( (I, \text{OPT}(I)) \) is not in NP unless \( \text{NP=} \text{co-NP} \). Then if there exists a TICM generating all instances of \( \Pi' \), \( \text{NP=} \text{co-NP} \).

It is shown in [Leggett and Moore 1981] that a variety of NP-hard optimization problems, including but not restricted to those derived from strongly NP-complete problems, have the property that the set of tuples \( (I, \text{OPT}(I)) \) is not in NP unless \( \text{NP=} \text{co-NP} \). By Corollary 3.4 these problems cannot have TICM's generating all instances unless \( \text{NP=} \text{co-NP} \).
3.2 Generation of Hard Instances

We now want to further investigate the existence and structure of test sets which are not only efficiently generable but also "hard" with respect to the NP-hard problem for which they are designed.

First one should note that different definitions of polynomial approximation are used in the literature. Some sources ([Meyer and Paterson 1979], [Lynch 1975]) refer to algorithms which recognize NP-complete languages exactly (and are therefore superpolynomial time algorithms) and investigate on what portion of the inputs the algorithms terminate within time bounded by some polynomial. Others [Orponen et al. 1986] consider polynomial time algorithms which answer 1, 0, or ? (yes, no, or don't know), always answering correctly in the first two cases. Clearly the first type of algorithm can be transformed into one of the second type by always stopping after a polynomial amount of time and answering ? if the computation could not be completed within the allotted amount of time. An algorithm of the second type is transformed into one of the first type if instead of answering ? it switches to an exponential time algorithm and solves the problem exactly.

We can also relate these theoretical views of polynomial approximation to that used in practical approximation algorithms for NP-hard optimization problems, as described for instance in [Garey and Johnson 1979]. Such an algorithm A works on an instance I of a maximization problem and outputs a result A(I) which is no greater than the optimal answer OPT(I). This can be converted to a polynomial approximation for the related decision problem as follows. The approximation algorithm for the decision problem takes input I, k and attempts to determine whether OPT(I) ≥ k by running A on I; if A(I) ≥ k, it answers 1, since OPT(I) ≥ A(I). If A(I) < k, it answers ?. In addition, if some performance guarantee of the form A(I)/OPT(I) ≥ R (R is a constant) is known for algorithm A, it can answer 0 if A(I)/ R < k.

For the rest of this section we will refer to NP-complete decision problems and their approximations. An approximation algorithm for such a problem will be defined as in [Orponen et al. 1986], namely a polynomial time algorithm which answers yes, no, or ?. Consider an NP-complete language L. Suppose a TICM C generates some instances of the problem, namely Gen(C, yes) and Gen(C, no) consisting of strings in L and not in L, respectively. The set of strings Gen(C) = Gen(C, no) ∪ Gen(C, yes) is in NP. We do not want any polynomial time algorithm to be capable of distinguishing between Gen(C, no) and Gen(C, yes).

Definition 3.2 Let L be an NP-complete language. Let B be a subset of ∑*. B is non-approximable with respect to L iff no polynomial time approximation algorithm for L gives the correct answer for all strings in B, unless NP=P.

For a useful TICM C for an NP-complete language L we want Gen(C) to be non-approximable. Note that an approximation algorithm for L will not necessarily be able to determine in polynomial time whether or not its input is in Gen(C). We saw in the last section that the set of all instances of the problem, (which is of course non-approximable), cannot be efficiently generated unless NP=co-NP. A further question one may ask is whether it might still be possible to have a non-approximable efficiently generated test set which includes all of the hard instances of the problem. This is also highly unlikely.

Lemma 3.5 If L is an NP-complete language, C a TICM for L, then unless NP=co-NP, no approximation algorithm for L correctly decides for all strings not in Gen(C).

Proof: Suppose such an approximation algorithm A existed. So A answers correctly for all strings not in Gen(C). Let F denote the complement of Gen(C) and let K be the set of strings for which A answers no. We have K = (F ∩ L) ∪ (F ∩ L) = (F ∩ L) ∪ K. K is in P. K is in NP. Hence L is in NP. Since L is NP-complete this implies NP=co-NP. ■
Some results in the area of structural complexity attempt to characterize the intractability of NP-complete sets in terms of certain sets of “hard” instances of the problem which cannot be solved by any polynomial time algorithm. One of these results concerns the existence of complexity cores [Lynch 1975]. A complexity core for a language $L$ is defined to be an infinite recursive set $X$ such that for all polynomials $p$ and all algorithms $A$ which recognize $L$, $A$ takes more than $p(|x|)$ time on all but a finite number of strings $x$ in $X$. In other words, a complexity core is an infinite subset on which all approximations for $L$ will do poorly. It has in fact been shown that any language $L$ not in $P$ has a proper complexity core; i.e., a complexity core $X$ such that $X \subseteq L$. A simple modification of the proof in [Lynch 1975] of the existence of complexity cores for languages not in $P$ yields the following.

**Lemma 3.6** If $B$ is non-approximable with respect to $L$, and $L$ is not in $P$, then there exists an infinite set $X \subseteq B$ such that for all polynomials $p$, any algorithm which recognizes $L$ takes more than $p(|x|)$ time on all except a finite number of $x \in X$.

## 4 Example: the Minimum Vertex Cover Problem

### 4.1 Generation of Instances

In this section we define a test case construction method to generate test instances for the minimum vertex cover problem. An instance of the vertex cover decision problem consists of a graph $G = (V, E)$ and an integer $k$. The question is whether there exists a cover for $G$ of size at most $k$, i.e., a subset of vertices $V' \subseteq V$ of size less than or equal to $k$ such that every edge of $G$ is incident on at least one vertex of $V'$. An instance for the optimization version of the problem consists of a graph $G$, and the answer is the size of the smallest possible vertex cover for $G$. The generation process to be described will produce test instances for the optimization version of the problem, i.e., graphs with known minimum vertex covers. Clearly from each of these instances one can obtain test instances for the decision problem, having both yes and no answers.

The basic idea of the construction method is simple. It consists of throwing together a set of special graphs for each of which a minimum vertex cover set is known, and then adding extra edges connecting these graphs while making sure not to increase the minimum vertex cover size. More formally, let $S$ be a set of pairs $(G, V')$ where $G = (V, E)$ is a graph and $V' \subseteq V$ is a minimum vertex cover set for $G$. Then other graphs with known minimum covers can be constructed as follows. Select any $(G_1, V'_1), (G_2, V'_2), \ldots, (G_t, V'_t)$ from $S$ and take their disjoint union; then add zero or more edges in such a way that each added edge is incident on some vertex from $\bigcup_{i=1}^{t} V'_i$. The generated graph will then have a minimum vertex cover consisting of $\bigcup_{i=1}^{t} V'_i$.

### 4.2 Characteristics of the Generated Graphs

Denote by $\text{gen}(S)$ the set of graphs that can be generated from a set $S$ as described above. In this section we show that if $S$ contains all cliques, then it is possible to obtain from $\text{gen}(S)$ graphs with any given number of vertices, edges, and minimum vertex cover size (if such a graph exists).

Denote by $\text{ver}(G)$ the number of vertices of $G$, and by $\text{vc}(G)$ the size of a minimum vertex cover for $G$. Note that there exists a graph with $n$ vertices and minimum vertex cover size $c$ if and only if $0 \leq c \leq n - 1$. Suppose we want to generate a graph with $n$ vertices and minimum vertex cover size $c$. Then we should choose $k$ graphs $G_1, G_2, \ldots, G_k$ from $S$ such that $\sum_{i=1}^{k} \text{ver}(G_i) = n$ and $\sum_{i=1}^{k} \text{vc}(G_i) = c$. If $S$ contains all cliques, then one can do this by putting $k = n - c$, randomly choosing positive integers $n_1, n_2, \ldots, n_k$ such that $\sum_{i=1}^{k} n_i = n$, and letting $G_i$ be a clique of $n_i$. 

8
vertices, for each i. The desired graph is formed by taking the disjoint union of the $G_i$ and adding extra edges as desired.

One can also control the number of edges in the generated graph by choosing appropriate cliques $G_i$. In order to produce a graph having $m$ edges, one should choose the cliques such that the total number of edges in their disjoint union is less than or equal to $m$ (since extra edges can be added later). In order to do this we need to find out for given $n, c$, what the minimum number of edges is that can be obtained from a graph formed from the union of cliques as described above. We also want to find out for which combinations of $n, m,$ and $c$ there exists a graph with $n$ vertices, $m$ edges, and minimum vertex cover size $c$. Propositions 4.1 and 4.2 provide answers to these questions.

**Proposition 4.1** Let $n \geq 1, c \geq 0, m \geq 0$ be integers. There exists a graph with $n$ vertices, $m$ edges, and minimum vertex cover size $c$ if $0 \leq c \leq n - 1$ and $h_1(n, c) \leq m \leq h_2(n, c)$ where $h_1, h_2$ are defined as follows. Given $n, c$, let $q = n - c$, and let $b, r$ be the quotient and remainder when $n$ is divided by $q$; i.e., $n = qb + r$ where $b \geq 0, 0 \leq r < q$. Then

$$h_1(n, c) = r\left(\frac{b+1}{2}\right) + (q-r)\left(\frac{b}{2}\right)$$

$$h_2(n, c) = \left(\frac{c}{2}\right) + qc$$

**Proof:** As remarked above, we must have $0 \leq c \leq n - 1$, and for any such $n, c$ there exists a graph with $n$ vertices and minimum vertex cover size $c$.

Consider a graph with $n$ vertices and minimum vertex cover size $c$. Every edge must be incident on at least one of the vertex cover vertices, hence all of the possible edges of the graph must be among the $\left(\frac{c}{2}\right)$ edges joining vertices in the cover, and the $c(n-c)$ edges joining a vertex in the cover to one not in the cover. So the graph can have at most $h_2(n, c)$ edges. Moreover a graph with this maximum number of edges does exist: take any graph with $n$ vertices and minimum vertex cover size $c$ and add all possible extra edges incident on at least one cover vertex.

The above argument also shows that for any $m_1 \leq m_2 \leq h_2(n, c)$, if there exists a graph with $n$ vertices, minimum vertex cover size $c$, and $m_1$ edges, then there exists a graph with $n$ vertices, minimum vertex cover size $c$, and $m_2$ edges. It only remains to show that the minimum number of edges a graph with $n$ vertices and minimum vertex cover size $c$ can have is $h_1(n, c)$. This follows from Turan's theorem ([Tur41]), which can be stated to say that the only graph with $n$ vertices, minimum vertex cover size $c$, and $h_1(n, c)$ edges, is the graph consisting of $r$ cliques of size $b+1$ and $(q-r)$ cliques of size $b$ ($q, b, r$ are defined in the statement of the proposition), while any graph of $n$ vertices and minimum vertex cover size $c$ must have at least this many edges. ([Tur54] expresses this in terms of maximum cliques, while a statement for maximum independent sets can be found in [B85]).

The following proposition now follows from Proposition 4.1 and the fact that the lower bound on the number of edges given in that proposition can always be achieved by a disjoint union of cliques.

**Proposition 4.2** If $S$ contains all cliques, then for all $n, m, c$ such that there exists some graph with $n$ vertices, $m$ edges, and minimal vertex cover size $c$, gen($S$) contains at least one such graph.
and \( \sum_{i=1}^{k} n_i^2 \leq 2m + n \). This can be done by first selecting \( n_1 \) such that \( 1 \leq n_1 \leq n - k + 1 \) and \( n_i^2 \leq 2m + n - k + 1 \), then choosing \( n_2 \) such that \( 1 \leq n_2 \leq n - n_1 - k + 2 \) and \( n_i^2 \leq 2m + n - n_1^2 - k + 2 \), etc. Take the disjoint union of \( k \) cliques, where the \( i \)th clique has order \( n_i \); this will provide a graph with \( n \) vertices, minimum vertex cover size \( c \), and \( m' \) edges, where \( m' = \frac{1}{2}(\sum_{i=1}^{k} n_i^2 - n_i) \leq m \). Finally, randomly add \( m - m' \) edges, each incident on at least one of the clique vertex cover vertices.

### 4.3 Non-approximability

We now want to show that no polynomial time approximation algorithm for the minimum vertex cover problem can provide the right answer for all instances in \( gen(S) \), for appropriate \( S \). We assume an approximation algorithm for this problem will return a vertex cover size which is greater than or equal to the optimum (minimum) vertex cover size for the input graph.

**Proposition 4.3** Suppose \( S \) contains a clique of size 2 and a clique of size 3 (together with possibly other graphs). Then, unless \( P=NP \), no polynomial time approximation algorithm for the minimum vertex cover problem gives the right answer for all graphs in \( gen(S) \).

**Proof:** To prove \( gen(S) \) is non-approximable we will use the reduction from 3-SAT that was used in [Garey and Johnson 1979] to prove that vertex cover is NP-complete. Let variables \( U = \{u_1, u_2, \ldots, u_n\} \) and clauses \( C = \{c_1, c_2, \ldots, c_m\} \) be an instance of 3-SAT. As in [Garey and Johnson 1979], for each such formula we can define a graph \( G = (V, E) \) as follows. For each variable \( u_i \), we define the vertices \( V_i = \{u_i, \bar{u}_i\} \) and edge \( E_i = \{(u_i, \bar{u}_i)\} \), to form \( n \) 2-cliques. In addition, for each clause \( c_j \) we define the vertices \( V_j' = \{a_1[j], a_2[j], a_3[j]\} \) and the edges \( E_j' = \{\{a_1[j], a_2[j]\}, \{a_2[j], a_3[j]\}, \{a_3[j], a_1[j]\}\} \) forming \( m \) 3-cliques. To establish the dependence between literals and clauses we add an edge connecting each literal to each clause in which it appears. If the three literals in \( c_j \) are denoted by \( x_j, y_j, \) and \( z_j, E_j'' = \{\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}\}. \) So \( V = (\bigcup_{i=1}^{n} V_i) \bigcup (\bigcup_{j=1}^{m} V_j') \bigcup (\bigcup_{j=1}^{m} E_j'') \bigcup (\bigcup_{j=1}^{m} E_j''') \). The reduction just described can be performed in polynomial time.

Each 2-clique must contain at least 1 vertex in any minimum vertex cover for \( G \), and each 3-clique must contain at least 2 vertices. Therefore a minimum vertex cover for \( G \) must contain at least \( n + 2m \) vertices. As shown in [Garey and Johnson 1979], \( G \) has a minimum vertex cover of \( n + 2m \) vertices if the 3-SAT formula from which \( G \) is derived is satisfiable. But if \( G \) has a minimum vertex cover of \( 2n + m \) vertices, \( G \) is in \( gen(S) \), since \( G \) consists of the disjoint union of 2-cliques, 3-cliques, and extra edges (the ones in \( E_j'' \) connected to minimum vertex cover vertices for the cliques. If we had an algorithm which found the minimum vertex cover size for all instances in \( gen(S) \) then we could recognize 3-SAT in polynomial time. So \( gen(S) \) is non-approximable.

As a remark, each 3-clique in the above construction may be replaced by a \( q \)-clique for any \( q \geq 3 \). In this case we would partition the \( q \) vertices of the clique into 3 nonempty sets and associate each set with one of the literals from the clause the clique represents; all vertices in each set would then be connected to the same vertex in the 2-clique representing the literal. This shows that the hardness of the generated instances is not restricted to those generated using only 2-cliques and 3-cliques.

It can be shown that unless \( P=NP \) no polynomial time approximation algorithm \( A \) for the minimum vertex cover problem can satisfy \( |A(I) - OPT(I)| \leq k \) for all instances \( I \) and any fixed \( k \). (See similar proof for the maximum independent set problem in [Garey and Johnson 1979], Thm. 6.7.) It is not hard to see that a similar restriction applies to \( gen(S) \) whenever \( gen(S) \) is non-approximable. In other words, if \( gen(S) \) is non-approximable, no polynomial time approximation algorithm can satisfy \( |A(I) - OPT(I)| \leq k \) for all \( I \in gen(S) \), for any fixed \( k \).

Because of the correspondence between the minimum vertex cover, maximum independent set, and maximum clique problems [Garey and Johnson 1979], the procedure described in this section can
also be used to generate test sets for the latter two problems which have similar non-approximability properties. A test graph for the minimum vertex cover problem having \( n \) vertices and minimum vertex cover size \( c \) is also a test graph for the maximum independent set problem having \( n \) vertices and maximum independent set size \( n - c \). By taking the complement of the graph we obtain a test graph for the maximum clique problem having \( n \) vertices and maximum clique size \( n - c \).

5 Conclusions

We have investigated some issues having to do with the generation of hard instances of NP-hard problems, for which the answers are known. It was shown that unless \( \text{NP}=\text{co-NP} \), it is not possible to have efficient test instance generators which can generate all instances of an NP-complete problem or of certain types of NP-hard optimization problems.

However, it was also shown by the example in the last section that it is possible at least in some cases to have test instance constructors which generate instances of NP-hard problems having certain desired parameter values and answers, and such that the set of instances generated is "hard".

We have also looked at and proved some results about the complexity of (nonuniform) generation for languages in \( \text{P} \), \( \text{NP} \), and \( \text{co-NP} \). In particular, the class of languages which can be generated in polynomial time was characterized and it was shown that all languages in \( \text{NP} \) are efficiently generable iff all languages in \( \text{P} \) are. The question of whether all languages in \( \text{NP} \) can be efficiently generated is still open. We do know, however, that if \( \text{P}=\text{NP} \) all languages in \( \text{NP} \) are efficiently generable, while if all languages in \( \text{P} \) (and hence in \( \text{NP} \)) are efficiently generable, then there are no sparse languages in \( \text{NP-P} \).

Acknowledgments

The author is grateful to Mark Fulk for his advice and support, and for useful discussions, to Mandayam Srinivas for discussions while these ideas were germinating, and to Joel Seiferas for pointing out the equivalence between construction and nonuniform generation for languages in \( \text{P} \), and the possible connections to sparse languages.

References


Attacking the I/O Bottleneck with Parallelism

Peter C. Dibble

Abstract

Every program that performs I/O can be constrained by an I/O bottleneck. This is a particularly serious problem for parallel computers. These computers have an aggregate performance (and I/O requirement) that is many times the performance (and I/O throughput) of a single processor. This imbalance can be addressed by using a parallel file system.

Most programs spend some portion of their run time waiting for I/O. If the speed of the file system is fixed while the speed of the rest of the system is increased, I/O will become an important performance issue even if it was initially trivial. On a parallel computer this can be an instance of Amdahl's law. When a program is parallelized without parallelizing the file access, the I/O will become the performance bottleneck.

On a uniprocessor the I/O bandwidth can be matched to the processor's speed by selecting I/O devices (disk drives) with suitable performance specifications, and by using multiple I/O channels. Since the file system runs on the same hardware that executes application programs, file system performance is bound to the performance of the programs. The relationship remains constant across processor upgrades.

The transfer rate of disk drives need never be the cause of an I/O bottleneck. Disk drives can be combined into systems with practically unlimited data rates. If a single drive delivers \( x \) bytes per second, \( n \) bytes can be transferred in \( n/x \) seconds from one drive. If the \( n \) bytes are spread across \( y \) drives and I/O on all the drives proceeds in parallel, \( n \) bytes will be transferred in \( n/(yz) \) seconds.

When files are interleaved across multiple drives under the supervision of a file system the technique is called striping [Salem and Garcia-Molina 1984]. When the interleaving is done in the I/O device so multiple data streams are combined before they reach the file system, the device is called a parallel-transfer disk [CDC 1986], or a storage array [Manuel and Barney 1986]. In either case I/O bandwidth delivered to the file system is proportional to the breadth of the interleaving.

A high I/O transfer rate is important to uniprocessor super computers and special-purpose systems, but it cannot remove the I/O bottleneck from a parallel computer. A single processor on a parallel computer supplies only a fraction of the processing power of the entire system. This is a central premise of parallel computing. Parallel applications run faster as processors are added. Since an ordinary file system is not a parallel program, it will be limited to the speed of a single processor. On a machine like the a 128-processor Butterfly multiprocessor, an ordinary file system is restricted to 1/128th of the total system processing power. This imbalance between parallel applications and a sequential file system creates an I/O bottleneck in the file system software.

After the file system has been optimized by careful coding and algorithmic improvements, improvements in I/O bandwidth depend on increased processor performance. Processor power can be added in three ways.

- The processor running the file system can be upgraded until it is in the same performance class as all the other processors combined.
• Separate file systems can run on separate processors. If enough file systems are used the aggregate performance can be brought to any value.

• A single parallel file system can run on enough processors to reach the required performance.

A I/O uniprocessor with performance that matches the aggregate performance of the rest of a parallel system becomes large and expensive very quickly. If each node in a multiprocessor can process 100 kilobytes of data per second, a 128-processor parallel computer can use a 12 megabyte per second input stream. A single high-performance disk drive [CDC 1986] can supply 12 Meg/s for short bursts, but sustaining that input rate through a file system would require careful programming on a mainframe. This is a workable technique for widening an I/O bottleneck, but it is subject to all the factors that limit uniprocessor performance.

A sufficient number of separate file systems can provide any desired aggregate I/O bandwidth. If one processor can support 200 kilobytes per second, five processors will supply a megabyte per second. This approach requires no innovations in the operating system or I/O hardware. Its hardware requirements are several processors from the multiprocessor system and disks that match the performance of the processors. It does, however, require that all application programs understand the partitioning of data among several file systems.

A parallel file system has the advantages of a group of separate file systems, but it can hide parallel I/O and parallel file processing algorithms under an ordinary file system interface. Applications need not know the parallel structure of the files.

1 The CPU Component of the I/O Bottleneck

Intuitively a file system seems likely to be I/O intensive. This intuition is likely to be incorrect for file systems on parallel computers. Optimized disk access will place the I/O bottleneck in the file system software. If the file system only transferred I/O requests from application programs to the I/O hardware and let the data move by DMA between the application and the device, the file system overhead could be counted in tens of CPU instructions per I/O operation. The result of this straightforward approach to file access would be remarkably slow I/O. Even the fastest disk drives have seek times on the order of 10 milliseconds [CDC 1986] and average rotational latency of about 8 milliseconds. A high transfer rate is almost useless when each 4K block of data requires an average of 20 milliseconds of delay before half a millisecond of data transfer. A good file system uses scheduling, caching, and other techniques to increase the throughput of the entire I/O system.

Consider, for example, the EFS file system [Thomas and Toner 1984]. EFS was designed as a robust distributed file system. It is not especially fast. A typical read operation involves at least three disk blocks, and our early implementations of EFS on the Butterfly multiprocessor required over 80 milliseconds to read a record from a simulated disk drive with a 30 millisecond average access time. Some simple algorithm changes reduced this to about 45 milliseconds per read of which 12 milliseconds per read was processor time. At that time the file system was reading approximately one record for each record it delivered to the application. The other records were in the file system's cache. Later we improved the file system to the point where reads average about 10 milliseconds per read with a 15 millisecond average access time. Less than half the time for an average read is spent waiting for input.

It is practical to invest CPU time in file system software to optimize disk access until the optimization uses more time than disk access it avoids. Using this rule, the CPU component of I/O time can easily exceed the time spent waiting for the I/O device.

1 A file system can be left intentionally I/O intensive. In a multitasking environment it may make better sense to devote memory and processor time to other tasks than to tie it up in the file system.
Table 1: Bridge Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>$20 \cdot \text{filesize}/p \text{ ms}$</td>
</tr>
<tr>
<td>Create</td>
<td>$145 + 17.5p \text{ ms}$</td>
</tr>
<tr>
<td>Open</td>
<td>$80 \text{ ms}$</td>
</tr>
<tr>
<td>Read</td>
<td>$9.0 + 500p/\text{filesize} \text{ ms}$</td>
</tr>
<tr>
<td>Write</td>
<td>$31 \text{ ms}$</td>
</tr>
</tbody>
</table>

2 The Bridge File System

We have implemented a parallel file system named Bridge on the Butterfly multiprocessor. Bridge has shown good performance with up to 32 processors. We have not yet experimented with broader parallelism.\(^2\)

Bridge incorporates \(p\) instances of a local file system (LFS) which manages the component of a Bridge file that resides on a single processor. The local file systems are combined into a unified parallel file system by the Bridge server.

An application that communicates with the Bridge server can elect to see a parallel file as a single conventional file. Bridge will hide all the parallel aspects of the file and present an ordinary file system interface. The application may see a performance improvement due to parallel caching, but its data will be transferred in a single stream.

Several processes can form a group and perform parallel I/O through the Bridge Server. A lead process initiates file operations by sending commands to the Bridge Server. Bridge will direct simultaneous data transfers between the processes and the local file systems to the extent of Bridge's parallelism. If the application requests more parallelism than Bridge supports, Bridge will offer virtual parallelism by transferring \(p\) blocks at a time until the request is satisfied. This technique provides applications with true parallel I/O while it hides the details of the file system's parallel structure.

A program requests parallel transfers and lists the members of the process group with a special parallel open operation. This type of parallel access is therefore a characteristic of an I/O path, not a file.

The fastest access to Bridge files is through Bridge tools. These are programs that integrate themselves with the file system. A typical Bridge tool will locate a process near each LFS, if possible on the same processor. These processes are components of the Bridge tool that will communicate with their local LFS and with one another. A tool need not route its I/O through the Bridge server. The I/O requests can move directly between the tool's processes and the local file systems. This saves the cost of routing requests through the Bridge Server, and it makes all I/O requests local, saving the cost of remote communications.

3 Interleaving

Bridge files are interleaved across the local file systems. If the I/O processors are numbered 0 through \(p - 1\), the \(n\)th record is stored in record \((n \div p)\) in a file on LFS \((n \mod p)\). The Bridge directory records the LFS file names that make up a bridge file, and the Bridge server dispatches I/O requests to the local file systems after converting the file name and block number from Bridge terms to LFS terms.

\(^2\)We hope to reach 64 processors, but we are constrained by memory limitations.
Table 2: Copy Tool Performance (10 Mbyte file)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Copy Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>311.6 sec</td>
</tr>
<tr>
<td>4</td>
<td>156.0 sec</td>
</tr>
<tr>
<td>8</td>
<td>79.3 sec</td>
</tr>
<tr>
<td>16</td>
<td>41.0 sec</td>
</tr>
<tr>
<td>32</td>
<td>21.6 sec</td>
</tr>
</tbody>
</table>

Table 3: Merge Sort Tool Performance (10 Mbyte file)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Local Sort</th>
<th>Merge</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>350 min</td>
<td>17 min</td>
<td>367 min</td>
</tr>
<tr>
<td>4</td>
<td>98 min</td>
<td>16 min</td>
<td>111 min</td>
</tr>
<tr>
<td>8</td>
<td>24 min</td>
<td>11 min</td>
<td>35 min</td>
</tr>
<tr>
<td>16</td>
<td>6 min</td>
<td>7 min</td>
<td>13 min</td>
</tr>
<tr>
<td>32</td>
<td>0.67 min</td>
<td>4.45 min</td>
<td>5.12 min</td>
</tr>
</tbody>
</table>

This simple interleaving is the best strategy for sequential I/O. Interleaving guarantees that \( p \) sequential records will be stored on \( p \) different processors. Only interleaving and permutations of interleaving guarantee this much parallelism. Published studies of Unix file I/O [Floyd 1986, Ousterhout et al. 1985] and our own informal study of I/O on the University of Rochester's IBM mainframe show that there is a strong tendency for applications to read or write files sequentially.

Although the Bridge file system is optimized for sequential access, it does not rule out random access. The Gamma database project [DeWitt et al. 1986] has built a special-purpose file system for database operations. Databases can choose from several record distribution strategies, including interleaving. Bridge does not directly support any distribution other than interleaving, but Bridge applications and tools can superimpose any order on a Bridge file. We believe that Gamma could be implemented as a Bridge tool without major changes to Bridge or Gamma.

4 Tools

We have implemented two Bridge Tools. One tool copies a Bridge file. The other tool sorts a Bridge file. Both tools exhibit roughly linear speedup to 32 processors.

The sort tool and the copy tool are fairly simple algorithms. Copy proceeds by copying the portion of the file on each LFS in parallel. Except for startup and termination, copy runs entirely in parallel. The sort tool spends most of its time sorting the part of the file on the local LFS. When all the local components are sorted they are merged into a single bridge file. The local sorts run without communicating with one another. The merge phase, however, requires extensive communication between the processors.

Many file operations are similar to copying; for instance:

- One-to-one translation
- Counting objects that don't span record boundaries
- Summarizing numeric fields
- Searching for a record that matches a template
All these problems, and others that involve sequential processing of one record at a time, are good candidates for Bridge Tools. They run in time $O(n)$ on a sequential file system and time $O(\log(p) + n/p)$ when they are implemented as Bridge tools.

Our merge algorithm requires $O(n)$ messages on a sequential critical path. This sequential time is covered by the time it takes for merge to read and write the file in parallel. The sequential part of the algorithm is so small compared to the parallel part that we expect speedup well beyond 32 processors.

Merge, sort, and other operations that consider groups of records together are not such obvious tools. They involve interprocess communication that may be $O(n)$ or worse. Even when a tool's algorithm lets the interprocess communication run in parallel the communication medium may serialize the messages, but this is not the problem it appears to be. The sort tool shows that parallel I/O time can cover a large sequential component.

We have done a detailed analysis of the sort tool algorithm. The result is:

$$
t_{sort} \leq O(N/p \log(N/p)) + \log(p)(k_{init} + \max(T_1 n + T_2 n/2, T_p n/p) + k_{pass})$$

where $n$ is the size of the file and $p$ is the number of processors used for the sort. The $k$ and $T$ values are constants.

5 Summary

A parallel computer will have difficulty maintaining a conventional I/O system that is well balanced with the performance of the rest of the computer. A parallel file system is an appropriate solution to the problem.

We have implemented a parallel interleaved file system on the Butterfly multiprocessor with these results:

- When the file system's parallelism is not used performance does not degrade appreciably as processors are added.
- Two common file operations have been integrated with the file system as tools. They both exhibit linear speedup.

Acknowledgments

This research was supported by NSF CER grant DCR8320136, DARPA/ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

References


3 Any connection other than a completely connected graph may introduce some serialization. A shared bus will serialize communication completely.


PENGUIN: A Language for Specifying User Interface Dialogues

Sue-Ken Yap

Abstract

PENGUIN (Programming Environment for Graphical User Interfaces) is a language for programming advanced human-computer interfaces. The structure of a conversation is described using augmented context-free attributed grammars. PENGUIN supports multiple threads of interaction, which are essential for interactive graphical interfaces.

1 The problem

Computers are tools that amplify human power. Unfortunately much of this power is inaccessible to the unsophisticated user because of the arcane incantations that must be learnt before a computer can be gainfully used. Human-computer interaction is that branch of computer science that seeks to understand how people use computers and how software can be designed so that people can use computer programs more effectively.

Part of the problem lies with the primitive models of human-computer interaction built into software. The terminal window in which I am typing this text is not much more than an emulation of a CRT terminal, which again is not so far removed from the teletype. Jargon such as tty line betrays this ancestry. Much existing software has been designed to run on such hardware.

This workstation I am using is also capable of graphical interaction. Graphical interaction offers richer modes of communication between human and machine. Instead of typing in words that name objects or actions, users can directly manipulate graphical representations of the objects [Shneiderman 1983]. This reduces the cognitive burden for new users.

Creating a high-quality graphical interface is difficult. There are more degrees of freedom, more parameters available with graphics. Current techniques use ad-hoc programming, relying on hand-coded calls to libraries of graphical primitives. Automated code generation can eliminate much tedious and error-prone coding. A user interface compiler must support the construction of dialogues with concurrent conversations. This concurrency is seen in windowing systems, for example. Another source of concurrency arises from the management of input from multiple devices [Buxton and Myers 1986].

2 Model

In the traditional model of interactive input/output in a program, I/O is delegated to blocking subroutines. This model is inadequate for graphical interfaces.

We use a more general model called the Seeheim Model [Thomas 1983] in which a program comprises three components: application, dialogue and presentation. The application does the “real work” of the system, whether number-crunching or searching a database. The presentation
component handles device dependencies. It translates physical I/O devices into logical I/O devices. Scrollbars and buttons are logical I/O devices, shared by and operated with a single physical I/O device, a mouse. The dialogue component captures the notion that certain aspects of a conversation are invariant across different machine setups. A login sequence may be presented in different ways, depending on the I/O device. The dialogue determines the set of legal sequences of interactions between human and machine. Another service provided by the dialogue is ensuring consistency by collecting data and invoking actions at specific times.

![Diagram](image)

Figure 1: The Seeheim model of a UIMS.

- Data is passed between the components as tokens. The information conveyed by a token can be as little as a keystroke or as much as a complete picture. We assume reliable delivery semantics are guaranteed by the operating system and network.

3 Notations

A dialogue consists of the exchange of information between pairs of participants. Let an interaction be a triple \((X, Y, \text{data})\), where data is the information sent from \(X\) to \(Y\). \(X\) and \(Y\) are normally user and machine. A conversation is a sequence of interactions. The structure of a conversation is the set of all legal orderings of interactions.

The meaning of data is here less literal than in common usage. For the purposes of this formalism, the fact that user passwords are usually unique is irrelevant.

Graphical dialogue structure differs from conventional dialogue structure in the larger sets that comprise the structure of a conversation. This arises from the non-linear nature of graphics that allows many different interleavings of interactions. A conventional interface allows less freedom in the order in which data is entered; there is a single cursor on a "teletype" terminal but on a graphical workstation the user can choose between several concurrent windows with the mouse.

Several notations have been proposed for dialogues. In transition networks, the dialogue is specified as a directed graph. Nodes represent states of the dialogue and arcs represent events. Event handlers are another notation. These are collections of modules that can accept events, change their internal state in response, and possibly invoke program actions. In grammar notations the set of legal inputs is specified with a context-free grammar. Program actions are invoked at appropriate times by action symbols "embedded" in the grammar.

Newman, Edmonds, Guest, Jacob and Wasserman [Edmonds 1981, Guest 1982, Jacob 1983, Newman 1968, Wasserman 1985] have used transition networks. Nodes in the network correspond to states in the program and arcs to actions that cause a change in state. Actions are triggered by user input. Nested interactions require recursive transition networks. Transition diagrams provide an excellent means of presenting information visually; Harel [Harel 1987] uses them as the basis of a visual programming notation called statecharts. The major drawback with transition diagrams is the verbosity of the representation. The major cause is the need to specify intermediate states in the dialogue that would normally be represented by juxtaposition in a more compact notation. A typical textual representation of a transition diagram lists for each state: the input tokens, the
successor states resulting from acceptance of tokens, and actions, if any. Guest's paper contains a 4-page example of a dialogue that prompts for names and addresses from the user.

Green [Green 1985] proposed event handlers. Event handlers have been used in the U. of Alberta UIMS [Green 1985] and ALGAE [Flecchia and Bergeron 1987]. The dialogue component is divided into event handlers. Each handler contains internal state that may be altered by the execution of actions on receipt of events from outside. The source of events may be the application, the presentation, or another handler. Event handlers may choose the types of event they are willing to accept. As the dialogue proceeds, handlers are created and destroyed. The entire collection of handlers resembles an object-oriented system such as Smalltalk-80, except that inheritance is not generally required. Event handlers have the drawback that the input/output language cannot readily be determined without inspecting handler code. Clarity and efficiency suffer because an event is allowed to activate multiple handlers.

Hanau and Lenorovitz [Hanau and Lenorovitz 1980], and Olsen [Olsen 1983, Olsen and Dempsey 1983] have used grammars. In Olsen's SYNGRAPH a BNF specification is processed and interpreted by an Interactive Pushdown Automata (IPDA) to parse the input stream. Escape and reentry states are provided for canceling an action or suspending an interaction to obtain help, but no multi-thread capability was provided.

The Input Tools notation of Van den Bos [van den Bos 1979] can in some sense be considered a cross between event handlers and CFG notations. In this framework, a dialogue is a collection of input tools. An input tool comprises a specification of the input language accepted by the tool and some code. An input tool accepts tokens and optionally does some computation before returning a value. This value becomes input to some higher level tool. This allows the composition of input tools. At the beginning of execution of a dialogue, all the input tools are activated but only the tools that accept primitive tokens can proceed.

Of these major notations, we consider grammars to be the most promising. The notation is concise. The separation of the specification of input/output language from the program actions leads to a program that is easier to comprehend. This aspect is important if the notation is to be useful for communicating the designer's intent to future maintainers of a program.

Our work borrows the fork operators from Input Tools. To our knowledge, no previous work has approached the problem of providing multi-threading and token dispatch in CFGs by extending the notation and hence the class of languages accepted. Unlike Input Tools, our method does not have the prohibitive run-time overhead resulting from searching for the input tool(s) that will accept a token [Matthys 1985].

Standard context-free grammars are inadequate for our purposes. The foremost deficiency is that a standard grammar has only one thread of control, which is inadequate to represent the interleaved nature of multiple, simultaneous conversations. To remedy this, we augment grammars with fork operators and contexts. We use an extended Backus-Naur notation in what follows.

### 3.1 Fork operators

The definitions of $\&\&$ and $|\mid$ are:

Two productions joined by $\&\&$ are a production. The combined production succeeds when both of the sub-productions succeed. More formally, the combined production generates all interleavings of strings generated by the sub-productions.

Two productions joined by $|\mid$ are a production. The combined production succeeds when either of the sub-productions succeeds. More formally, the combined production generates all interleavings of a string generated by one sub-production with a prefix of a string generated by the other sub-production.
Here are examples of their use:

\[ \text{form} \rightarrow A \&\& B \]

Both \( A \) and \( B \) must appear in the input for \( \text{form} \) to succeed. The tokens that \( A \) and \( B \) derive may appear in any order in the input.

\[ \text{getnumber} \rightarrow \text{interrupt} \| \text{read.digits} \]

The completion of either \( \text{interrupt} \) or \( \text{read.digits} \) will cause \( \text{getnumber} \) to succeed. Different levels of interrupts can be used to back the parser out to arbitrary pre-arranged positions.

Together, the parallel \( \text{and} \) and \( \text{or} \) operators allow us to construct a hierarchy of sub-parsers to manage a conversation with multiple levels of aborts, nested parallel conversations (e.g. for interactive help), and other useful structure.\(^1\)

### 3.2 Contexts

With multiple threads, there now is the problem of where to dispatch input. If more than one thread exists, there is usually some way to distinguish between the sources of the inputs. For example, threads may be associated with windows. We rightly expect that input to one window has no bearing on the behavior of a logically unrelated window. This is formalized in the notion of \( \text{context} \), which is a special attribute attached to each token identifying its source. Each thread inherits one or more contexts. Every terminal symbol in a thread has a context that an incoming token must match.

Several rules ensure that there is no more than one recipient for every incoming token. First, contexts are unique. Every source of tokens is labelled with a unique context throughout the lifetime of the program. Secondly, the branches of a fork must have distinct contexts, with the one exception that if branches share a context, then their input alphabets must be disjoint.

Distinct contexts for branches are used if each thread should run independently. Shared contexts are used if the sharing threads deal with the same window but with disjoint sets of events. Both the distinct and shared modes of passing context are needed. An example of the first mode is a set of independent subwindows. A example of the second mode is a window that has to highlight itself when the cursor enters its boundaries.

It is worth noting that user input are not the only tokens that are received by the dialogue component. For example, events indicating the cursor entering the window or the window being unobscured are sent by the presentation component to the dialogue component. The user will in general have some conversation in progress inside the window, such as an incomplete string, so a separate thread is necessary. This thread should act appropriately, e.g. highlight the window, independently of the thread receiving user input.

Besides user input arriving via the presentation, the dialogue may also receive tokens from the application. The application may need to notify the dialogue of some exceptional event, for example, the arrival of a system-wide broadcast message. Again the use of separate threads relieves the programmer of having to watch explicitly for these events in the “main” thread of interaction.

\( L \)-attributed copy rules [Knuth 1968] specify the context attributes of symbols in the grammar. Here is an example:

\[ A \rightarrow B \&\& C \]
\[ B \rightarrow a\ B\ c \mid c \]
\[ C \rightarrow b\ C\ d \mid c \]

is one example; its intersection with the regular set \( a^*b^*c^*d^* \) is \( a^*b^*c^*d^* \). This proof is by Dave Sher.

---

\(^1\) It is worth noting that our augmented grammars are not in general context free. The language generated by
4 Parsing

There are at least two algorithms for parsing these augmented grammars. The first is a modification of a standard LL(1) parser [Aho et al. 1986], while the second uses a grammar transformation.

Before we explain the parsing, we briefly explain how the fork operators may be handled at runtime. Since sub-parsers working in parallel may complete in any order, a plain stack is insufficient to represent the tree of sub-parsers. We need a cactus stack. When a fork operator is encountered, the stack is split. First a special frame that indicates the type of fork operator is pushed on the stack, then the branches are attached to this frame. The frame contains a count of the number of unsatisfied branches remaining. For | forks, the count is initially 1, for && forks, the count is initially the number of branches.

```
form &&
```

Figure 2: An && node in the cactus stack.

4.1 Dispatch by context, then value

Our rules on the use of context attributes ensure that at most one parser will accept a given token. When a token arrives, the parser uses an associative table (e.g. hash table) to find the interested sub-parser. For an alphabet split, there may be more than one sub-parser. In this event, the value is used to resolve the ambiguity. This associative table must be updated whenever contexts are created or destroyed; i.e., when a fork operator is encountered.

With the sub-parser in hand, we can now attempt a match. The case of a terminal on the top of the parse stack is trivial. If a non-terminal is on the top of the stack, there are two sub-cases.
In the first case, the non-terminal derives a non-empty string. The non-terminal and input token are used to obtain a list of productions to predict, distinguished by context. Note that although the actual values of the contexts to match are unknown at compile time, the locations are known. These locations can be computed at compile time by tracing the path in the attribute graph from the predicting token to the closest ancestor that will be on the parse stack when a prediction is required. Furthermore, the inheritance rules guarantee that the contexts will be unique, hence no ambiguity arises. If the context of the input token matches one in the list of candidates, then we push the associated production on the stack.

In the second case, the non-terminal at the top of the stack derives the empty string. Here we match ε, and pop the non-terminal. This is repeated until one other case obtains.

If the non-terminal can derive either the non-empty string or the empty string, the former match is attempted first before the latter.

The LL(1) prediction sets are constructed with a modification of the standard algorithm. Each entry in the parse table is a list of context and production pairs instead of a single production.

4.2 Grammar transformation

Since input to a sub-dialogue is limited to a fixed number of contexts, it becomes possible to transform an extended grammar with context attributes into a conventional context-free grammar without context attributes. Any standard parsing algorithm can then be employed. The idea is to replace each symbol X in the extended grammar with a new symbol for every possible set of values of X’s context attributes. The dialogue compiler can assign a name to each of the contexts of the sub-dialogue even though the actual context values will not be known until run time. A table created when the sub-parser begins execution can be used to translate from <token value, context value> pairs to token values in the alphabet of the new, conventional grammar.

If a sub-dialogue inherits c contexts, a token in the original extended grammar may be replaced by up to c tokens in the new grammar. A non-terminal A with t inherited context attributes may be replaced by up to ct! non-terminals in the new grammar. Each of the productions for which A forms the LHS will be replicated up to ct! times.

This technique allows any standard parsing algorithm (including LR algorithms) to be used but has the potential to increase parse table sizes dramatically. Since LR techniques are probably of limited use for interactive dialogues, because of their bottom up nature, it is not clear that this advantage is useful.

5 Example

Here is an example of an existing tool rewritten to use a dialogue grammar. We describe the skeleton of the dbxtool source debugger for Sun Workstations [Adams and Muchnick 1986]. Dbxtool has a command window, a button window, a status window and a source window. Assume that the command window parses its input stream and sends only complete tokens such as print or continue to the dialogue. The button window provides many of the commands of the command window as radio buttons for user convenience.

The initial production starts up all windows:

\[
S \rightarrow \#create \ (C+ || status+ || source+)
\]

\[
[C.cmd = \#create.cmd, C.but = \#create.but, status.ctx = \#create.st, source.ctx = \#create.src]
\]
The action #create starts up all four windows. Sub-productions of C accept input from both the command and button windows. For each radio button, C derives two productions, one starting with the command typed in and the other with the button clicked on. In the productions below, the capitalized tokens are associated with and derive context from the button window. The actions triggered have been elided. The triggered actions are different in print and PRINT as the latter uses the currently highlighted entity.

\[
\begin{align*}
C &\rightarrow \text{print entity.name ...} \\
C &\rightarrow \text{PRINT #check.selection ...} \\
C &\rightarrow \text{stop location ...} \\
C &\rightarrow \text{STOP #check.selection ...} \\
C &\rightarrow \text{next | NEXT | step | STEP}
\end{align*}
\]

6 Modules

In the grammar fragments above, the dialogues for the sub-windows have been grouped together. A better grouping is to place productions close to related data structures and program actions. Such a group is called a module.

Formally, a module comprises four parts: an input grammar, output ports, data and code. The input grammar determines the sequence of input tokens. Tokens for other components are sent through output ports. The data and code provide for retention of state and control over who is allowed to alter internal state. An instance of a module comes into existence when the start symbol of its input grammar is predicted.

A module is an encapsulation of an interaction technique. Designers can create reusable interaction modules to be deposited into a library. This collection of standard modules should reduce the effort needed to build a working interface, as compared to coding from scratch.

The constrained rectangle is an example of a module. This is a rubberband rectangle with lines joining the midpoints of opposite sides dividing the rectangle into four congruent rectangles. This module may be used for the interactive sizing of windows. As the user drags the mouse, the module echoes the tentative size of the window. The module must calculate the positions of the midpoints to draw those lines. This is done in the code section. Output to the presentation consists of tokens that redraw the rectangle as the mouse moves. Only the final size of the window is returned to the parent, after the user is happy with the new size. In general note that the code section can enforce any constraint that can be expressed programatically.

Modules may be composed to form larger groups. The dialogue component is a collection of modules. We can generalize the Seeheim model from three components to \( n \) components. Consider a component called a splitter. This component simply copies all input to two output ports. For example, we can transparently interpose a splitter between the dialogue and application. This allows us to obtain a log of the entire session. An instructor could interpose a splitter between the dialogue and the presentation to obtain an exact replica of a student’s screen.

7 Current work

We are currently engaged in completing the definition of PENGUIN. We will soon begin the implementation of the PENGUIN compiler and tools.
Acknowledgments

This work was supported in part by ONR/DARPA research contract N00014-82-K-0193, ONR research contract N00014-84-K-0655 and an IBM Faculty Development Award.

References


Direct Inferences in a Connectionist Knowledge Structure

S. C. Hollbach

Abstract

A model of human cognition is proposed in which all concept properties are context dependent. Concepts are comprised of multiple facets, each motivated by a different functional property. A connectionist implementation is presented in which conceptual modification yields the 'direct inferences' implicit in the structure of a knowledge base.

Introduction

When a noun is modified by a descriptive adjective, the result is often a significant modification of the original concept denoted by the noun. For example, while a peach is soft, juicy and tangy, a green peach is hard, dry and bitter. Clearly, the adjective 'green' when applied to a peach conveys more than merely the colour.

This paper advances a computational model of conceptual modification that captures the 'direct inferences' arising from property correlations. The domain of this investigation is concrete nouns and their attendant descriptive adjectives. Each noun denotes a concept, where a concept is represented by a structured collection of properties and values, indexed by function. For example, an apple viewed as food brings different properties to mind than an apple viewed as a projectile. Ultimately it is the goals and plans of the agent that determine how an object is thought of: a hungry agent thinks of apples differently than an angry one. Thus the functional properties of an object provide the context for interpretation, in that they select only the currently relevant facet of the complete description.

We have built a connectionist implementation of the functional context-sensitive model of category representation. The system runs on a Sun Workstation as an application of the Rochester Connectionist Simulator [Goddard 1987], the results of which are visible in iconic form thanks to the Graphics Interface [Lynne 1987]. The system uses an extensive knowledge base of categories and their interrelations to draw direct inferences about modified categories, answer queries about object properties, and model property dominance effects [Whitney 1986, Tabossi 1986].

The Structure of Knowledge

This investigation focusses on the mental representation of physical objects. The building block of these mental representations are categories, classifications of physical objects sharing one or more common properties. A property is a set of descriptors applicable to a physical object, where the elements of the set are property values. We classify properties into three groups: perceptual, constitutive and functional. Perceptual properties pertain to the five senses, functional properties relate to an object's usefulness by humans, and constitutive properties are in some sense the definitional properties of a category, often expressed in terms of genetics, compositional makeup and so on. Functional properties play a special role in category representation, supplying as they do the various perspectives from which the category can be viewed. For example, the is edible property provides the focus of relevance for the tangy and has seeds properties of apples.
Figure 1: Circles represent categories, diamonds property values and triangles binder nodes associating categories with their attendant values. Lines do not represent direct links, but rather indirect connections mediated by subnet structures of varying complexity.

Categories, in addition to having a complex internal structure, are related to one another in a hierarchical subsumption taxonomy. (A familiar example of such structuring is ontological knowledge, the ordering of natural kinds according to common biological characteristics.) The links in the taxonomy represent subsumption relations between categories. Thus to some degree a lower level category participates in the higher level category. The form this participation takes differs depending on whether one is looking up or down the taxonomy. All the properties and values possessed by the higher level category are also possessed by the lower level one. And for each property or value possessed by a lower level category it is true of the higher level category that there exists an element of that set having that property or value. For example, since all things have colour as a property, all apples must also have a colour. Furthermore there exists a red (or green or yellow) apple, by virtue of the colour values associated with the various apple varieties.

The connectionist implementation of this cognitive model follows the ‘localist’ paradigm of Feldman and Ballard [1982]. Each category is represented by a single exemplar object. Each object, property and value is represented by a distinct (named) network node. All relations between these nodes are captured in separate subnets of regular structure, allowing the network to be compiled from a series of high level input language statements. Concepts are represented as patterns of activity over all the nodes in the network. ‘Thoughts’ are formed in the network by keying in activation on a noun and (optionally) adjectives, and allowing the simulation to run a few steps to permit these activations to propagate fully. Activation flows out from the noun denoting the category to all relevant properties and values. Relevance is determined by context, or more specifically, by the currently active functional property of the category. Each category has associated with it a default context or facet; for example, the default view of ‘apple’ is edible. So when the noun is activated in isolation, the system responds by selectively activating its defining properties and values with respect to its default context. For example, Figure 1 depicts the graphics display of certain key elements of the network after keying in activation on the ‘apple’ node and allowing the simulation to run a few steps to stability. The selective effects of context mean that only a subset of all possible property values of the concept are active at one time, although a given property value can participate in any number of facets.

Direct Inferences

A major feature of the connectionist knowledge base is its dynamic nature. Rather than having the knowledge encoded in a purely passive (declarative) format requiring a distinct reasoning component to interpret and apply it, the knowledge encoded in a connectionist network incorporates several simple forms of inferencing directly in the structure of the network. These direct inferences, that is, inferences not requiring an interpreter but contained entirely within the terminological component,
can be either mediated or immediate. Immediate inferences are drawn about object properties at the
level of the object itself, while mediated inferences involve property inheritance. Mediated inferences
can be drawn either from more general knowledge, or, if the information is not available at a higher
level, a weaker answer can be derived from more specific knowledge.

Both forms of direct inference, mediated and immediate, arise from this fundamental mode of
operation of the network, as demonstrated in Figure 1. One of the many immediate inferences drawn
about apples is the fact that they are crunchy; one of the mediated inferences is the fact that they
are edible. A further inference, as to the existence of red apples, is obtained by ranging down the
hierarchy, rather than up as is customary.

Property Queries

Property queries take the general form “does (modified) category x have property value y?”. Phrased
more naturally, this becomes “are y’s x?” or “do y’s have x’s?”. Of course, given that each category
is represented by a single exemplar, a more accurate portrayal of the query forms would be to say
“is a y z?” or “does a y have z?”, for example, “is a black bird large?” or “does a red apple have
seeds?”. Each property value is represented by two nodes, one corresponding to the adjective as a
category modifier, the other corresponding to a query on that property value. So to pose a query
to the system, the user activates the adjective and noun forming the target category, thus invoking
the fundamental mode of operation of the network, namely, the drawing of direct inferences. The
queried property value is then keyed in on the query-specific twin of the property value node.

There are five possible responses to a query: a ‘yes’ or ‘no’ in context, a ‘yes’ or ‘no’ out of
context, and ‘category error’. An ‘yes’ in context occurs when the property value is an element
of the set characteristic of the current facet. A ‘no’ in context occurs when a modifier negates the
queried value, as in “is a green apple red?”. A ‘yes’ out of context is reported when a shift of context
is required to answer in the affirmative, as in “are sweet apples easily thrown?”. An answer of ‘no’
out of context results when the property value is not associated with the category in any context,
although other values of that property are, as in the query “are small apples purple?”. Answers
out of context take a little longer than answers in context, since context shifting takes time. A
category error occurs when the property associated with the value is not a property of the category
(“are ideas purple?”). Category errors are interesting not only for their possible role in cognitive
development [Kei 1979], but also for the fact that when they occur in conversation it is generally to
signal a metaphor. We are currently in the process of extending the model to account for metaphoric
interpretation of such category errors.

Once the query has been keyed in, the simulation is run to a point where either an answer of
‘yes’ or ‘no’ in context or a report of a category error would be detected by the system. If neither
condition exists, alternate facets are explored in parallel until either an answer of ‘yes’ out of context
is reported or the possibilities are exhausted, resulting in a ‘no’ (out of context).

Property Dominance Effects

In addition to participating in a subsumption taxonomy, the mental representation of an object has
a complex internal structure. The design of this internal structure is based on the premise that all
object properties are context dependent. This idea arose from the debate between Whitney [1986]
and Tabossi [1986] over the problem of lexical access modelling, or the question of whether the
meaning of an ambiguous word is selected at the lexical access stage or interpreted later. Whitney
presents related results concerning the semantic access of unambiguous words in support of the
multiple access model (akin to the delayed interpretation model for ambiguous words). In Whitney’s
work, all the properties germane to a concept (as denoted by a concrete noun) are accessed or primed
in parallel, by mention of the noun, regardless of any bias built into the sentential context. The effect of the bias, to promote some properties to prominence and inhibit others, is only visible several hundred milliseconds after initial mention of the nouns, and must thus be occurring at a later (post-lexical) processing stage. Tabossi, on the other hand, contends that the stimuli used in Whitney's work are too neutral with respect to the target concept to induce any significant bias, and presents results to support the competing notion of selective access, or lexical level biasing and inhibition of properties.

The question of whether or not lexical access is context sensitive is still an open one. There is agreement in the literature, however, on the fact that certain concept properties are correlated, both positively and negatively [Malt and Smith 1984], and that context is used to decide which of a number of competing property associations or coalitions should be permitted to dominate [Cohen and Murphy 1984]. As Tabossi points out, ice is both hard and cold, yet a sentence like "The bartender served the drinks with ice" that primes the property value 'cold' also inhibits the value 'hard' and vice versa, while a neutral sentence neither primes nor inhibits either property.

These results are consistent with the model advanced in this work, in that competing contexts are mutually exclusive, neutral contexts are unrelated, and reinforcing contexts are mutually excitatory. Dominance effects are modelled by asymmetric link weights between the two competing facets, permitting a high dominance property (e.g., ice temperature) to exhibit stronger effects than a low dominance one (e.g., ice hardness).

A fundamental assumption underlying this work is that categories, as mental constructs of active agents, are inseparably linked with the agent's planning goals. These goals or situational contexts are so influential on the mental structure of categories that a category is meaningless when out of context. Since categories (indeed, all ideas) are by definition meaningful, they must carry with them a default context to supply meaning in the absence of other information. Very often, particularly for physical objects, this default context is simply visual recognition. When the word 'apple' is spoken, a mental image is conjured up of the visual appearance of an apple. If the agent is hungry at the time, the apple's taste might spring to mind. If the agent is angry, its properties as a handy projectile might leap into significance. And so on. Thus while a category appears stable to the agent, since the same basic set of properties and values are being drawn on at all times, the structure is actually dynamic, shaped by context.

Contexts interact amongst themselves in different ways than do categories. Where categories can combine with each other in an arbitrarily complex fashion, contexts afford less latitude. A context is nothing more than a particular way of looking at a category. Two perspectives on a category are either the same or different. So contexts can be either mutually supportive or mutually inhibitory, depending on whether they are compatible or not. For example, if the agent's goal is to eat an apple, he must first locate one visually. Thus the edible and visual-id contexts are compatible. On the other hand, if the agent decides to throw the apple at a passing car, any thought of eating it will be suppressed, as throwing and eating are mutually inhibitory contexts.¹

The general characteristics of the model derive partly from the psychological model described above and partly from properties of connectionist models. Categories are represented by the various patterns of activity over the set of properties and values associated with that category. Each distinctive pattern is characteristic of a different context or goal. There is a default context associated with each category. If in the process of specifying the structure of the knowledge base the user fails to name a default context for a category, the 'visual-identification' context is used, since this is generally appropriate for the chosen domain of physical objects.

Property values can be either neutral or biasing. A neutral value displays no strong correlation with one context over any other, while a biasing value is characteristic of only one context. For example, the modifier 'red' is neutral in the phrase 'red pillow' but biasing in the phrase 'red

¹This version of events is admittedly simplistic, but it suffices for the problem at hand.
rose', raising as it does visions of romance and long-stemmed floral offerings. Thus context can be established implicitly by mentioning a biasing property value; it can also be established explicitly, by turning on the context node by hand. Not all property values are biasing, or, more accurately, not many property values are sufficiently biasing to override the currently active or the default context in favor of another. The values biased toward the context can be guessed at with greater confidence than the more neutral ones, although there is a slight bias built into neutral values. In fact, the neutral/biasing distinction is not a very good one, as it represents the attempt to quantify a gradual change. A more accurate characterization would be to speak of strong, moderate, weak and negligible biases.

Results of running the simulation are shown in Figures 2, 3 and 4. Shown are the iconic representations of individual network nodes. Activation was keyed on the phrase 'expensive diamond' and the simulation allowed to run to stability. Figure 2 shows this initial state of the network. Figure 3 shows the network in an intermediate state shortly after changing contexts from the adornment aspect of diamonds to the industrial aspect, achieved by shifting the external activation of 'expensive' over to 'hard'. That is, the system is being forced to consider the phrase 'hard diamond' after being primed with the phrase 'expensive diamond'. As Tabossi's property dominance studies predict, there is a significant latency period between presentation of the stimulus and recognition of its appropriateness, as shown in Figure 4. The details of the implementation, including a description of a connectionist interpreter that translates statements in a high level language into a structured network, are given in [Hollbach 1988].
Summary and Conclusions

This research advances a context sensitive model for conceptual modification and uses it to capture not only property dominance effects but also direct inferences, both immediate and mediated. Concepts are denoted by concrete nouns, optionally modified by one or more descriptive adjectives. The agent’s current plans and goals supply the relevance criteria for focusing on a coherent subset of a given object’s disparate properties and values. Goals are represented simply by a concept’s functional properties. A characteristic use of an object will dominate uncommon ones, as will all perceptual and constitutive properties associated with that functional property, leading to property dominance effects. The interproperty associations yield immediate inferences, and property inheritance gives rise to mediated inferences. The connectionist implementation of this model operates as a question-answering system, permitting the user to pose queries about the various properties of a concept.

Acknowledgments

This work was supported by U.S. Army contract no. DAAB10-87-K-022. Thanks to Jerry Feldman, and to Gary Dell and Josh Tenenberg for invaluable comments.

References


34


List Productions in a Parallel Incremental Parser

Neal M. Gafter

Abstract

An incremental SLR(1) parser designed to run on a multiprocessor is described. The parser is based not on integration with a language-directed editor, but on a standard text editor that has been modified to output a log of user editing actions. Only the parts of the program that have changed are reparsed, and this processing is highly parallel. The parser makes use of special list productions to improve parsing efficiency.

1 Introduction

Most work on incremental compilers takes the point of view that the parser should be integrated into the editor, and that the user should not be allowed to enter a syntactically incorrect program. The language-oriented editor approach has both advantages and disadvantages compared to the text editing model. Unfortunately there has been little systematic study of the impact of this paradigm on programmer productivity. Waters [Waters 1982] gives an argument in favor of the retention of text oriented commands in a program editor. Whichever form of editing is supported, the compiler can be run as a background process during editing.

There have been studies of the use of parallelism in compilation, but we are the first to consider using parallelism in an incremental parser. The ultimate goal is to reduce the response time of the environment by making use of the processing power available in a multiprocessor. We add special list productions in the parser to improve the asymptotic performance for source programs that contain lists. Many large programs, for instance, appear as long lists of declarations.

Reducing compilation time will improve programmer productivity and make it practical to reconsider language and compiler features that were previously considered too time consuming. This is a first step in the development of high-performance compilers and programming environments for multiprocessor systems.

2 Related Work

Morris and Schwartz describe a structured editor based on incremental parsing [Morris and Schwartz 1981]. A “split” of the parse tree is maintained at the current position of the cursor. Although the algorithm supports editing using the textual viewpoint, it requires the use of a special editor, and it must consider the user’s changes sequentially as they are entered.

Ghezzi and Mandrioli described a sequential incremental parsing algorithm for a single change to the text file [Ghezzi and Mandrioli 1979]. Parsing begins just to the left of a single change, and to avoid parsing the rest of the program, a similar reparse from right to left begins just to the right of the change.

Jalili and Gallier describe a general algorithm for sequential incremental parsing based on a number of changes to the text file [Jalili and Gallier 1982]. The algorithm assumes the existence of
an easily computable function from parse nodes to source positions and the reverse; their implementation provides this by integrating the parser (and the parse tree data structure) into the editor. Stromberg [Stromberg 1982] separates the implementations of text editor and parser by having the editor output a log of the users editing actions. Attributes in the parse tree record the length of the text derived from each nonterminal, allowing easy location of affected nodes. These attributes are unaffected by nonlocal changes.

Poe [Fischer et al. 1984] uses an ambiguous grammar for list productions to make it easy for the user to enter items at any position in a list. Since Poe is based on a structured editor, this ambiguous grammar is not used in a parser. Syned [Horgan and Moore 1984] uses the text editing model for program parts, and completely reparses the syntactic construct that is being edited. Syned includes special processing for list productions to flatten the abstract tree representing recursively parsed lists, but this processing does not improve parse time.

Fischer’s dissertation [Fischer 1975] is a seminal work in parallel parsing that describes a synchronous parallel parsing algorithm for a vector machine. Cohen and Kolonder provide upper bounds for its speedup [Cohen et al. 1982].

Schell [Schell 1979] considers the problem of scanning and parsing on an asynchronous multiprocessor, and develops a parallel generalization of standard bottom-up techniques. Ligett, McCluskey, and McKeeman describe a similar parallel parser [Ligett et al. 1982]. The performance of these and related algorithms has been analyzed using modeling [Cohen and Kolonder 1985] and simulation [Sarkar and Deo 1986, Ligett et al. 1982, Schell 1979].

There has been little work to date on the use of parallelism in incremental compilation. Kaplan and Kaiser consider the problem of multiple programmers working on different parts of a common program, using workstations sharing a common high-speed network [Kaplan and Kaiser 1986]. However, individual source files are compiled sequentially. Our previous paper [Gafter 1987] develops the framework on which this paper builds.

3 Model and Assumptions

The model parallel computer used in this work shares memory among a set of homogeneous processors. Because our algorithms use applicative data structures, the processors share memory according to the CREW (concurrent read, exclusive write) model. The parser inputs are the text file for a single compilation unit, a log of editing actions recorded by the editor, and the parse tree for the program prior to the editing changes. The inputs are assumed to be in memory when processing begins. Similarly the parser output, a parse tree, is left in memory when processing completes.

It is assumed that a standard text editor has been modified to produce a log of user editing actions. The actions recorded in the log are simple insertions and deletions; other actions are translated into a sequence of insertions and deletions before inclusion in the action log.

The scanning question is not considered in any detail in this paper. The processing of the log of edit actions, and the algorithm for constructing lists of tokens for changed program parts is described in detail in the earlier paper [Gafter 1987].
4 Overview

The following figure gives an overview of the parser, from [Gafter 1987]:

The task of the parser is to translate a textual representation of the program into a parse tree. Because the parser is incremental, it constructs the parse tree by modifying the parse tree of the previous version of the program. The log of editing actions produced by the editor is translated by the fragment list construction into a concise description of the difference between the old and new source files. The fragment list and the old parse tree are used during symbol list update to construct an updated symbol list for the program. The symbol list describes the source file as a sequence of tokens from the source file and special nonterminal symbols that represent unchanged syntactic constructs; each contains the root of the old parse tree for the construct. Finally, the new parse tree is constructed using a parallelization of incremental LR techniques.

The strategy for symbol list updating is to disassemble nodes of the old parse tree where the user has changed the text. A nonterminal symbol can be disassembled by replacing it with the nodes directly below it in its parse tree. Breaking up the parse tree in this way yields a list of symbols. From the list, symbols are deleted where the user deleted text, and symbols are inserted where the user inserted text. Finally, the symbol list is parsed to repeatedly reduce sets of symbols into a nonterminal symbol, until only the start symbol of the grammar remains. It is this final parsing that is the emphasis of this paper.

4.1 The Symbol List Data Structure

The symbol list data structure represents a list of tokens. There are two types of symbols: a terminal symbol represents a token appearing directly in the source program, and a nonterminal symbol represents a syntactic construct that has already been recognized. Each nonterminal stores the root of a parse tree for its syntactic construct, and replaces its yield in the symbol list. An invariant of the symbol list data structure is that the concatenation of the yields of the symbols in the list is exactly the source program. In this sense, the symbol list is similar to a sentential form. As will be seen later, it may not be a true sentential form because it may contain nonterminals from the old parse tree that are not correct in the modified program; such symbols will be disassembled during parsing. Our earlier paper [Gafter 1987] describes how the symbol list is built from the edit log and the previous parse tree.
The symbol list is maintained using an 
\textbf{applicative} variant of an AVL balanced binary tree [Myers 1984]. This data structure supports the division of a list of \( n \) elements at a given position in \( O(\log n) \) time, and allows two lists of length \( n \) and \( m \) to be concatenated in \( O(\log(n + m)) \) time.

5 Parsing Algorithm

Having constructed the symbol list for the new program, the final problem is to turn it into a parse tree for the entire program. As with all bottom-up parsing, a list of symbols representing a single syntactic construct is replaced by a single nonterminal. This reduction replaces the group of symbols with the corresponding enclosing syntactic construct, constructs a tree node for it, and inserts the representative nonterminal into the symbol list. Reductions are applied repeatedly until only the symbol for the start symbol remains.

Conventional LR techniques perform only the leftmost reduction at every step, reducing only constructs that appear on the top of the parsing stack. These techniques are inherently sequential. Schell [Schell 1979] describes a non-canonical generalization of LR parsing in which a number of processors simultaneously work on different parts of the input. In Schell's algorithm, the input string is divided into a number of sections, and each section is parsed by a separate processor. Each processor independently finds locally leftmost phrases to reduce as in conventional LR parsers. When it is unable to continue because of an error, inadequate state, or insufficient stack depth for a reduction, a parser passes the terminal and nonterminal symbols in its stack to the processor working on the segment to its left to be treated as a continuation of its input. The LR parsing techniques are easily generalized to allow the resulting nonterminal symbols.

The parser completes when the leftmost processor completes parsing the entire program. Because much of the program is "pre-reduced" by other parsing modules, the leftmost parser potentially sees many fewer symbols than a sequential parser would.

In the parallel parser, finding a phrase is difficult because the non-leftmost parser may not have sufficient context to determine the leftmost phrase. A further complication arises when the processor does not have the prefix of a phrase on the parse stack when a reduction is attempted. These are the two fundamental problems that Schell solved by modifying the state construction process and the parsing actions.

Since only the leftmost processor knows its context in the parse of the entire string, the non-leftmost processors start in a \textit{super-initial} state, which is constructed by including all non-initial items. This state reflects the fact that these parsers start without the benefit of left context normally provided by the contents of the parse stack—it may be starting out at an arbitrary position in any syntactic construct. The closure and parser actions are constructed as usual.

During the construction of the states, some of these additional states may display shift-reduce or reduce-reduce conflicts on certain inputs. A parser cannot distinguish the phrase to be reduced, and will pass the nonterminals on the parse stack to its neighbor to the left, which presumably has more context. In this case, Ligett et. al. [Ligett et al. 1982] restarts the parser in the super-initial state on its remaining input. The more refined continuation states constructed by Schell reflect knowledge gathered on the stack prior to the conflict [Schell 1979], and is therefore used for the construction of our parse tables.

The problem of insufficient stack depth to perform a reduction is handled by appending the symbols on the parse stack of the current processor to the input for the processor on the left, and restarting the current processor in the state accessed from the super-initial state by shifting the left hand side of the production. This simplification of the algorithm described by Schell has little effect on the running time.
The parsing algorithm of Schell will not make a reduction that is an incorrect derivation for the input string, so each nonterminal seen is part of the correct derivation. However, our incremental parser may start with nonterminals from the old parse tree in its input. These nonterminals do not necessarily appear in the correct derivation of the new program, and might therefore have to be disassembled during the parse. Even worse, these incorrect nonterminals may be shifted by a parser, allowing the parser to make incorrect assumptions about the syntactic structure of the input. These problems and their solution are the main differences between the parsing algorithm of Schell and that described here.

For example, consider the following grammar:

\[
S ::= "PASCAL" SP | "ALGOL" SA \\
SP ::= ...(a pascal grammar)...
SA ::= ...(an algol grammar)...
\]

In this example, the interpretation of the entire string rests on the first token. But if only the first token is changed, the nonterminals for the rest of the program will not be broken up by the symbol list update phase of the parser. This language is still SLR(1), for instance, if \( SP \) and \( SA \) are SLR(1).

A parser working on a segment in the middle of this program will enter a subset of the states that include only Pascal items as soon as it shifts a Pascal-only symbol. Unfortunately, the program may well be an Algol program. While this does not affect the correctness of our algorithms, it does affect their efficiency. This language displays a kind of non-locality that makes parallel parsing difficult; languages that display locality are easier to parse, and are easier for people to read because there are frequent hints to the reader about the context.

Under the above scenario, an incorrect nonterminal symbol will appear eventually to a parser as an error condition. It is not a true error because it does not necessarily represent an incorrect program.

In order to efficiently distinguish between the various causes of an error, each parse tree node records the leftmost token in its yield. The grammar is constrained to allow no \( \epsilon \)-deriving nonterminals. Now, when an error occurs, the parser can distinguish these three cases:

- if the input symbol is a nonterminal, and there is a valid action in the current state for the first token of its yield, then the reduction represented by the input nonterminal is not one the current parser would have chosen. The reduction was made in the old parse tree or by a parser to the right. The input symbol is therefore disassembled and prepended to the current parser's input.

- if the input token is a terminal \( t \), or a nonterminal with \( t \) as its leftmost terminal symbol, and the top of the stack is a nonterminal \( n \), and \( t \notin FOLLOW(n) \) then the top of the stack is not a reduction that would have been chosen by the local SLR(1) parser. Therefore the top of the stack is popped, disassembled, and prepended to the input.

- in any other case, an invalid nonterminal symbol has been shifted earlier (or possibly the program is in error). The symbols on the stack are passed to the parser to the left, and the current parser begins in the super-initial state on the remaining input. The idea is to pass the problem on to a parser which is likely to have more context with which to solve it. If this is the leftmost parser, then a genuine error has been recognized, and normal error recovery is invoked.

The correctness of the parsing algorithm follows from the correctness of the parallel parsing algorithm in [Schell 1979]. The only difference is the existence of incorrect nonterminals in the input symbol stream. The proof assumes the grammar is SLR(1) and the input program is in the language.
Since the leftmost parser breaks up old nonterminals from the top of the stack when it detects an error, the parse will be correct if the leftmost parser never lets an erroneous nonterminal get past the top of its stack. By contradiction, it can be shown this will never happen. Assume the leftmost parser shifts some symbol s onto the stack above the erroneous symbol t while parsing a correct program. Let w be the symbols corresponding to the stack contents below t. Then wts is the prefix of a sentential form (the prefix of some valid program in the language). The language is unambiguous because the grammar is SLR(1), and there are no \( \epsilon \)-deriving nonterminals, so a sequential parser would construct the same derivation. This contradicts the assumption that the symbol t is incorrect.

5.1 Complexity Analysis

Our parallel complexity analyses assume a pool of \( O(n) \) processors is available for computation. The results here do not take into account latency or contention.

The analysis appearing in [Gafter 1987] shows that a new symbol list can be constructed from a log of editing actions in time \( O(d + \log^2 n) \), with \( n \) processors, where \( n \) is the length of the updated symbol list and \( d \) is the depth of the parse tree. The nesting structure of typical programs can be expected to be bounded by \( \log(n) \), however the parse tree can be much deeper due to the use of recursion to describe lists in the grammar.

Upper bounds for speedup in parallel parsing based on modelling [Cohen and Kolonder 1985] and simulation [Sarkar and Deo 1986, Ligett et al. 1982, Schell 1979] would appear to be directly applicable to our algorithm, but a number of practical differences makes it difficult to estimate the actual speedup which will be observed. Specifically, the locality of a language discussed in section 5, may have a large impact on the actual efficiency. Also, the extent to which constructs from the previous parse tree can be used without modification will affect the parsing speed, and we do not have a measure of this effect on parsing efficiency.

One rough estimate for a lower bound on the parsing time is the depth of the newly constructed portions of the parse tree. While the nesting structure of typical programs is probably small, the parse tree can be much deeper due to syntactic lists. The straightforward method for parsing lists induces a linear depth parse tree, and this will result in a linear parsing time for a program consisting entirely of a single list.

5.2 List Productions

The standard way to represent lists of items in an LR-based grammar is using a left-recursive rule:

\[
\begin{align*}
\text{<list>} & : = \text{<item>} \\
\text{<list>} & : = \text{<list>} \text{<item>}
\end{align*}
\]

This causes the parser to construct a syntax tree that has linear depth in the length of the list. This is unfortunate for our parallel parser, because its execution time has a component that is linear in the depth of the newly constructed portions of the parse tree. We expect that syntactic lists are a common cause of deep parse trees in typical computer programs. To improve parsing efficiency for long lists of syntactic items, we allow special list productions. All right and left-recursive lists may be replaced in the grammar by an application of this special rule form and a few "helper" productions. For example, the productions

\[
\begin{align*}
\text{<stmt>} & : = \text{BEGIN} \text{END} \\
\text{<stmt>} & : = \text{BEGIN} \text{<stmt_list>} \text{END}
\end{align*}
\]
may be replaced by

```
<stmt_list> ::= <stmt>
<stmt_list> ::= <stmt_list> ; ; <stmt>
```

In general, list productions are specified as follows:

```
<list> ::= <item> +
```

This describes a syntactic list that derives one or more of the syntactic item on the right hand side. We do not allow zero items because our parsing algorithm assumes that the grammar is constrained to allow no $\epsilon$-deriving nonterminals. While more complicated forms of list productions might be allowed, this restricted form allows a fairly simple correctness proof and analysis.

A list production such as the above is translated by the parser builder into the set of productions

```
<list> ::= <item>
<list> ::= <list> <list>
```

Although this subgrammar is ambiguous, it has the desirable property that it describes all binary syntax trees with the items at the leaves. The strategy for efficient parsing will be to maintain these trees internally as balanced binary trees [Myers 1984], as if the parse had constructed a fully balanced derivation. To a large extent, the increase in efficiency will be due to the ability of the parser to actually construct a more balanced derivation than before.

Of course, we must deal with the ambiguities that are introduced by the set of productions that are generated from list productions. These new productions are handled during state construction exactly as other productions, with the following exceptions:

- The item
  
  `[ <list> ::= <list> • <list> ]`

  is not included in the super-initial state. This allows the local processor to fully process all adjacent items appearing in its input.

- If a state contains the item
  
  `[ <list> ::= <list> <list> • ]`

  then the item
  
  `[ <list> ::= • <item> ]`

  is removed from the state. This resolves a potential shift-reduce conflict and allows the local processor to fully reduce its items as would a normal left-recursive parser.

- During parsing, a reduction by the rule
  
  `[ <list> ::= <list> <list> ]`

  causes the concatenation of two balanced structures, and results in the creation of a single balanced tree structure. Note that the resulting abstract parse tree does not necessarily represent the actual derivation constructed by the parser because of the rebalancing that may occur.
These rules allow a local processor to fully reduce all adjacent items appearing in its input into a single list nonterminal, and resolve all ambiguities introduced by the list productions. By an extension of the proof of correctness in previous sections, it can be shown that the resulting parser correctly parses only strings in the language.

5.3 List Productions and Parsing Efficiency

Because each processor is able to fully reduce its subset of items in a list, and is able to fully process all lists that are passed from other processors, the parsing speed is limited only by the depth of the actual derivation and the additional processing required for rebalancing. The depth of the actual derivation depends strongly on when processors are allowed to drop out of the computation [Sarkar and Deo 1986], but it no better than $O(\log n)$. The rebalancing time is also $O(\log n)$, so the time to parse a list of length $n$ is $O(\log^2 n)$.

Because list productions produce more balanced parse trees, their use can improve the performance of the symbol list construction from [Gafter 1987]. The complexity of symbol list construction is $O(d + \log^2 n)$, so under reasonable assumptions we can expect the compiler front end to execute in $O(\log^2 n)$ time.

6 Summary

Research on advanced programming environments has conventionally used a syntax-directed editor as the front end to the compiler, in part because this was the only way seen to get immediate feedback to the user. Our inclusion of special list productions helps to improve the performance of the parser in some common situations. We attempt to combine both parallel and incremental techniques by providing algorithms for parallel incremental parsing, a first step in the development of a full parallel incremental compiler. Exploiting large-scale parallelism in a compiler would appear to be one way to significantly reduce compilation time and generate quick feedback from the compiler.

7 Future Work

We are planning an implementation of the parser described here to measure its performance relative to sequential techniques, and to measure the effect of list productions and various language and grammar features on the parsing efficiency. There is some overhead associated with dividing programs for parallel parsing; at what point and to what extent do the techniques speed up the compilation process? When global control structures improve the efficiency of parsing? What language features make these techniques more or less effective? Can these algorithms be extended to LALR(1) or LR(1) formalisms?

This is only a first step in the development of algorithms for a complete, high-performance programming environment for multiprocessors. We are currently designing semantic analysis algorithms based on an attributed grammar formalism.

Acknowledgments

This work was supported by the ONR/DARPA research contract N00D14-82-K-0193, the U.S. Army Engineering Topographic Laboratories research contract DACA76-85-C-0001, and by the NSF Coordinated Experimental Research grant DCR-8320136.
References


Object Recognition from a Large Database

Michael Swain

Abstract

Decision trees have been used for many years in pattern recognition systems. However, their use in visual recognition has been restricted to the pattern recognition domain, where features are required to be measurements which form the axes of a vector space, and which are often assumed independent. Computer vision researchers have studied the use of more powerful recognition techniques that consider object topology and viewpoint consistency, among other things. How to structure a large database for efficient recognition when using these techniques has not been well studied. Here I claim that the decision tree data structure can be employed as successfully outside of the pattern recognition domain as it has been within. To substantiate my claim I describe a system for recognition from a large database that recognizes polyhedra in crowded scenes and uses object topology, comparisons of length and angle, and viewpoint consistency as tests.

Given the a priori probability of polyhedra to appear in the image, viewpoints from which they are seen, and the errors which occur in detecting their edges, the decision tree is automatically constructed from the database using the criterion of minimizing the expected recognition time. Since the optimal solution of this problem is intractable, a greedy method is used based on minimizing the expected entropy of the a posteriori distribution of possible matches. The motivation for the entropy measure is derived.

Since object topology is a major source of information for the recognition system, the space of viewpoints is divided according to principal view regions. The large number of regions that this results in does not reduce the run-time efficiency of the system, since any number of regions can be considered at one time by the recognition system, whose state is determined by a position in the decision tree. Other measurements are also used, for example angle measurements and comparisons of length, where they will provide more information than the topological data. The viewpoint consistency constraint is used to provide an exact viewpoint determination as soon as is economical from an information-theoretical point of view. By considering matches directly to the model at run-time instead of principal views the redundancy of the principal view representation is avoided and the viewpoint consistency constraint can be efficiently applied.

1 Introduction

The human visual system can recognize objects from an enormous database in real time. Many vision systems operate under real-time constraints, and a vision system operating in a fairly unconstrained environment may have to recognize objects from a database that is an appreciable fraction of the size of the human database. Large databases have been considered by Pattern Recognition researchers, who have achieved success at indexing into them using decision trees [Wang and Suen 1984]. Under the Pattern Recognition paradigm the features form the axes of a vector space and an object is represented by a point in this feature space [Fu 1968]. The features are often assumed independent.

The recognition techniques considered here are object topology, local quantitative measurements and the viewpoint consistency constraint. They do not fit naturally into the Pattern Recognition paradigm, but this does not prevent a vision system that uses them to structure its database in the form of a decision tree.
Given a probability distribution on the frequency of objects, the viewpoints from which they are seen, and the errors which occur in the low-level system, a decision tree can be built that optimizes the expected cost of recognition. Thus, a decision tree data structure differentiates among more important and less important features for distinguishing among objects. This difference in importance of features cannot be determined a priori but is instead a function of the database. Turney et al. [1985] refer to this concept as saliency.

The aim of this work is to show that a decision tree should be considered as a data structure to store a database for fast visual recognition outside of the field of pattern recognition. Here a recognition system is described, called the Tree system, that uses a decision tree. For simplicity, the objects are restricted to be polyhedra, but it is expected that the overall design of the system will extend to less restricted worlds. This is because the tests used do not have to be restricted to the ones used in the Tree system, but tests for closure, termination, inside-outside, color and texture could be used as well, for example.

Because the Tree system uses object topology as a major route to recognition, the concept of principal views [Koenderink and van Doorn 1976] [Freeman and Chakravarty 1980] is used in the construction of the decision tree. Each principal view defines a family of possible projections, all members of which are topologically identical and differ only by a continuous transformation. The Tree system uses principal views in the preprocessing stage to calculate the possible topological appearance of the objects in the database. The decision tree itself, however, is almost completely independent of them, and instead deals directly in matches to the object model. This was done to avoid the problems associated with the naive application of principal views, which are

1. The correspondence among the same edges in different principal views is not represented.
2. There are a large number of principal views. To solve for the viewpoint it is often not necessary to discover the exact principal view that occurs in the image.

Because the decision tree deals directly in matches to the model, states in the decision tree correspond to matches to object edges, and so problem 1) does not exist. Problem 2) is eliminated by allowing the possibility of solving for the viewpoint before the principal view in the image has been uniquely determined. Instead it is done when its utility rises above the utility of testing for more topological constraints or for other viewpoint invariant features. In the Tree system principal views provide only the lowest grain to which the possible viewpoints are partitioned; at any point in the tree the possible range of viewpoints consists of a collection of principal views of one or more models.

2 Previous Work

The theory of the use of decision trees in sequential pattern recognition is covered in [Fu 1968]. An application to a large database of two-dimensional objects can be seen in [Wang and Suen 1984].

Burns and Kitchen [1987] have designed and implemented a system with a similar approach to the one described here. Rather than constructing a decision tree as the Tree system does, they construct what they call a prediction hierarchy. They have a set of rules for building the tree whose aim is to minimize the size of the hierarchy, rather than to minimize run-time. At run-time the hierarchy is searched from the bottom up by combining features found in the image whenever the hierarchy allows it. They do not consider the run-time complexity, nor does the design explicitly aim to achieve the minimum complexity, but since the search can benefit from parallel processing it could be competitive.

Cooper and Hollbach [1987] have suggested using a series of filters arranged in a hierarchy, using local computationally inexpensive tests in parallel over the image and principal view database
at the beginning and more computationally expensive tests at later stages when the number of possible models has been substantially reduced. The optimal decision tree approach, by contrast, can eliminate models without explicitly testing against each of them, and is constructed to use the features that are most salient, as dictated by the database. The decision tree approach cannot benefit from massive parallelism, but does the best possible without it and does not suffer from its costs.

Robert Bolles has designed systems for fast recognition for robotics that use preprocessing to design an optimal strategy at runtime [Bolles and Cain 1982] [Bolles and Horaud 1986]. His approaches only match against one model at a time, and therefore rely on the uniqueness of the features being detected to achieve fast runtime. Chris Goad [1987] has also investigated using preprocessing to minimize recognition time. His approach can only consider one possible assignment of image features to model features at a time, and so will have complexity linear in the size of the database. The decision tree has the potential of logarithmic time complexity.

3 Decision Tree Construction

The standard way of building an optimal decision tree is to build it back-to-front using a dynamic programming approach [Fu 1968]. The tree is built back-to-front because the optimal decision at a particular node depends on knowing all future optimal decisions. Building an optimal decision tree is intractable for the domain the Tree system is designed for because the space of possible states is large.

Instead of constructing an optimal decision tree, one can use a greedy method to construct the tree in a forward fashion. An estimate is made of the expected cost of recognition in the subtrees that would result from all the possible tests at a given leaf of the partial decision tree. Based on this estimate, the test is chosen. In the appendix it is shown that in many cases the estimate takes the form of an information entropy function (see [Shannon 1948] or a text in information theory [Jones 1979]). The examples in Section 5 assume an entropic cost function. To improve the estimate, lookahead can be used, in a similar fashion to game tree search [Slagle and Lee 1971]. The ‘opponent’ in this case is nature, or chance.

4 Dealing With Errors

It is possible to account for errors in the input to the recognition system either in the construction of the decision tree or at runtime. If errors are considered in the construction of the decision tree, the nodes in the decision tree represent states which include the possibility that an error has been made. The same object becomes a possible match on more than one branch of the tree, a condition termed overlap in the Pattern Recognition literature [Wang and Suen 1984].

If the runtime approach is used, as is used in the Tree system, the states do not include the possibility of error. Instead multiple processes explore the decision tree concurrently. If doubt in a decision rises above a threshold, processes simultaneously investigate the branches of the decision tree that descend from each plausible outcome. The number of processes is kept to within the number of processors available by ranking them according to confidence. The confidence associated with the initial thread of control, \( C_1 \), is initially set to 1. Whenever decision \( i \) is made, the confidence of the thread of control is multiplied by the probability, \( P_i \), that the decision made was the correct one. So the confidence of the \( i \)'th thread of control after its \( k \)'th decision is

\[
C_i(k) = \prod_{j=1}^{k} P_j
\]
With $C$ defined as described, it corresponds to the probability that the thread of control is the correct one, under the assumption that the probabilities of each decision being correct are independent.

Processes are terminated either when they fall below a confidence threshold or when they arrive at a leaf node in the decision tree. The confidence threshold can be made dynamic at the cost of some communication among the processors. A dynamic threshold is desired because not all objects are necessarily of the same probability, and low-probability errors may occur, pushing the confidence of all the processes downward. A list of the most promising terminated processes can be maintained to be reactivated if the confidence of running processes drop.

5 Examples

This section describes two examples that demonstrate various aspects of the recognition system.

The first example builds a complete decision tree for some very simple objects. An interesting feature of this example is that a greedy technique would not guarantee an optimal tree, despite its simplicity.

The second example builds part of a decision tree for one object that is significantly more complex than those in the first example.

5.1 Example 1

The objects are a triangle, a square, a pentagon and a heptagon. They are shown in Figure 1. All edges are the same length in each model. The models are all assumed equiprobable, so each may appear with probability 0.25. The range of possible viewpoints is such that (scaled) orthographic projection is a good approximation, and so parallel lines appear parallel and lines of equal length in the models appear equal length in the image.

Figure 2 shows all possible search trees. Since the objects are so simple there is only one point at which a choice can be made. The choice is whether to (1) expand the fourth edge or (2) test the first and third line segments for parallelism. As is evident, the choice to expand the fourth edge is the better one. Let us calculate the expected entropy of both choices.

We are at state $i$, which is reached with probability 0.75. For the choice of expanding the fourth edge, we find two possible outcomes. In the case when the object is a square the cycle of edges closes (state $ii$). Both for the case of the pentagon and the heptagon the cycle is not closed (state $iii$). The probabilities of each state, given that we have reached state $i$, are

$$P(i|ii) = \frac{.25}{.75} = 0.333$$
e = expand edge
p = test for parallel

Figure 2: Two Possible Search Trees (Example 1)
The entropy of state $iii$ is zero, since the match is known. The probabilities each match once state $iii$ has been reached are $P$(pentagon) = 0.5 and $P$(heptagon) = 0.5, and so

$$H(iii) = - \sum_j P_j \log P_j$$

$$= -2(0.5 \log 0.5)$$

$$= 0.693$$

The expected entropy is

$$E(H_1) = (0.333)(0) + (0.667)(0.693)$$

$$= 0.462$$

A similar calculation shows that the expected entropy of choice 2 is equal to the expected entropy of choice 1 and so the greedy method of minimizing entropy does not guarantee choosing the optimal tree in this case. Searching ahead one ply would enable the entropy method to choose the correct tree, however.

5.2 Example 2

Figure 3 shows the object whose pose the decision tree will be designed to recognize (the 'house' figure from [Burns and Kitchen 1987]). Figure 4 shows the principal views chosen a priori to have greater than zero probability of occurrence. I will ignore testing for parallelism, length of lines or solving for the viewpoint and only consider the choice of edge to expand. This will be done using the greedy entropy measure. Figure 5 shows possible search trees starting from a vertex of degree 4 and assuming the first edge expanded finds that the vertex of degree 4 is adjacent to another vertex of degree 4.

Because of the symmetries of the 4–4 pair of vertices there are only two different choices of edge to expand which I will call end and side (left and right branches respectively in Figure 5). Tables 1 and 2 summarize the relevant information for the end and side choices respectively. In these tables the views columns give the number of times the graph or match occur in each principal view. The frequency $f$ can be calculated from the information in the views column. For a graph or match $G$ it is just

$$f(G) = \sum_{v \in V} n_v(G)P(v)$$
where \( V \) is the set of principal views, \( n_v(G) \) is the number of occurrences of \( G \) in principal view \( v \), and \( P(v) \) is the probability of the principal view \( v \). Note that \( n_v(G) \) may be greater than one for a match \( G \) because of symmetry. The probabilities are obtained from the frequencies are by normalization. The entropy is

\[
H(G) = - \sum_{i \in S} p_i \log p_i
\]

where \( S \) is the sample space under consideration. In our case the sample space is the set of possible matches. The expected entropy is

\[
E(H) = \sum_G P(G)H(G)
\]

where the sum is over all possible graphs that are obtained by expanding the edge being considered.

The tables show that the expected entropy is lower if one chooses to expand the end edge \( (E(H) = 0.231) \) over the side edge \( (E(H) = 0.722) \). Therefore, using the minimum entropy method without lookahead to construct the tree one would choose to expand the end edge.

6 Conclusion

The Tree recognition system described in this paper is being implemented to test the growth in size of the decision tree with the size of the database, its robustness to errors in input and to inaccurate priors, and its runtime complexity. We are also looking into a real-time incremental decision tree construction algorithm, which would allow the system to collect its priors from experience and
adjust its recognition strategy on-line. The off-line procedure described here requires statistics to be collected and the decision tree constructed prior to runtime.

**Acknowledgements**

I would like to thank my advisor Dana Ballard for his guidance and suggestions, Paul Cooper and Nancy Watts for influential discussions, and Chris Brown for his encouragement and support.

This work was supported in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, in part by NSF Coordinated Experimental Research grant no. DCR-8320136, in part by ONR/DARPA research contract no. N00014-82-K-0193 and in part by a Natural Sciences and Engineering Research Council of Canada Postgraduate Scholarship.

### A Entropy as a Measure of Expected Cost

Since the optimality criterion being used for decision tree construction is minimum expected search depth, the best evaluation function to use for constructing the tree is one that estimates the expected search depth from that node. Here we show that under many conditions that function takes the form of an appropriately scaled entropy measure.

**Theorem 1** Suppose every test has \( m \) equiprobable outcomes and that each of the \( n \) possible events falls into exactly one of the outcomes of each test. If every test has cost \( c \) then the expected cost \( C \)
of the search is

\[ E(C) = -c \sum_{i=1}^{n} p_i \log_m p_i \]

with the understanding that

\[ p_i \log_m p_i = 0 \]

if \( p_i = 0 \) or 1.

**Proof.** The proof is by recursion.

\( E(C) = 0 \) at a leaf node. Since at a leaf node \( p_j = 1 \) for some \( j \) and \( p_i = 0 \) for all \( i \neq j \) then

\[ \sum_{i=1}^{n} p_i \log_m p_i = 0 \]

by definition and the theorem holds.

For a node that is not a leaf assume the theorem holds for all its \( k \) subtrees. Then the expected cost of the search is

\[ E(C) = c + \frac{1}{m} \sum_{j=1}^{k} \left( -c \sum_{i=1}^{n'} p'_i \log_m p'_i \right) \]

Now since

\[ n = \sum_{j=1}^{k} n' \]

and

\[ p_i = \frac{p'_i}{m} \]

we have

\[ E(C) = c - \frac{c}{m} \sum_{i=1}^{n} mp_i \log_m (mp_i) \]

\[ = c \left( 1 - \sum_{i=1}^{n} p_i \log_m (mp_i) \right) \]

\[ = c \left( 1 - \sum_{i=1}^{n} p_i \log_m m - \sum_{i=1}^{n} p_i \log_m p_i \right) \]

\[ = -c \sum_{i=1}^{n} p_i \log_m p_i \]

and the theorem is proved. **Q. E. D.**

**B Unequal Test Outcomes**

Tests normally do not have equally likely outcomes. What can we say about tests whose outcomes are not equally likely? It is possible to analyze this problem analytically if we assume

1. the distribution of probabilities of the \( m \) outcomes of each test is constant,
2. the probabilities of each outcome are all powers of $\frac{1}{k}$, and
3. all $k^n$ events are equally likely.

Let the outcomes of the test be $o_i$, $i = 1, 2, \ldots, m$ and denote their probabilities by $P(o_i)$. Without loss of generality assume that $o_1$ is the most likely outcome. Say its probability is $\frac{1}{k}$ for some $k$. Then the expected depth of the search tree is

$$E(k^n) = c + \sum_{i=1}^{m} P(o_i)E(k^nP(o_i))$$

If we rewrite this equation it can be put in a form in which it is readily recognizable as a difference equation. Let

$$\Delta_i = -\log_k P(o_i).$$

and

$$a_n = E(k^n).$$

Then

$$a_n = c + \sum_{i=1}^{m} P(o_i)a_{n-\Delta_i},$$

or

$$a_n - \sum_{i=1}^{m} P(o_i)a_{n-\Delta_i} = c. \quad (1)$$

Ignoring initial conditions for now, we can try to find a solution to the difference equation by trying a general form and solving for the undetermined coefficients. A good rule of thumb is to try an expression that looks like the right hand side when the equation is written in the form of Equation 1. Trying a constant solution $b_0$ does not work because then the left side sums to zero. If we try a linear solution $b_1n$ we get by substituting into Equation 1

$$b_1 = \frac{c}{\sum_{i=1}^{m} P(o_i)\Delta_i},$$

and so we have found a particular solution to the difference equation. The difference between any two solutions to a nonhomogeneous equation with constant coefficients is a solution to the homogeneous equation

$$a_n - \sum_{i=1}^{m} P(o_i)a_{n-\Delta_i} = 0.$$

Since $a_n$ is a just a weighted average of previous solutions it is easy to see that the solutions to the homogeneous equation do not grow with $n$. Therefore any solution to the nonhomogeneous equation is the particular solution we found above, to within a factor that does not grow with $n$.

Replacing $a_n$ by the original notation we have

$$E(k^n) = b_1n$$

or letting $n_i = k^n$ and substituting for $b_1$

$$E(n_i) = \frac{c\log_k n_i}{\sum_{i=1}^{m} P(o_i)\log_k P(o_i)}.$$

Here again an entropy-type factor is introduced.
C Different Tests

What if different tests have different distributions of outcomes? Suppose there are $T$ different tests $t_i$, each of which has $k_i$ equally likely outcomes, for $i = 1, 2, \ldots, T$. Suppose each test is used with probability $P(t_i)$ and has cost $c$. Then an idea of how to obtain a combined estimate of expected cost can be obtained by imagining the successive application of the tests in proportion to their relative probabilities. Consider it as a combined test. The probability of each outcome will be

$$P(o) = \prod_{i=1}^{T} \left( \frac{1}{k_i} \right)^{\beta P(t_i)}$$

where

$$\beta = \text{lcm}(\frac{1}{P(t_i)})$$

assuming the $P(t_i)$'s are rational. This is just as if one test with

$$\prod_{i=1}^{T} (k_i)^{P(t_i)}$$

outcomes was repeated $\beta$ times. So we see that the geometric mean of the numbers of outcomes of each test is the effective number of outcomes.

References


Plans, Speech Acts, and Conversational Implicature

Elizabeth A. Hinkelman

Abstract

This paper explicates the role of plan recognition in conversational implicature. It argues that knowledge of goals and plans is necessary for the computation of some implicatures, and very useful for others. Our model is based on a set of inference rules about STRIPS-style plans for implicature computation. It also incorporates a computational model of indirect speech acts, based on linguistic features. Particular attention is given to the plan recognition mechanisms that link these two components.

1 Background

Sam and Bob are old school pals from Backwoods, Montana. Sam tells Bob that he is leaving on a shopping trip to Great Falls. The conversation continues:

(1) Sam: By the way, do you have a radar detector?
Bob: Sure, it's out in my truck. When you getting back?

Bob has interpreted Sam's question as a request to borrow his radar detector. He pursues the details of the loan. Sam now assumes that Bob's radar detector is in working order, will allow him to drive to Great Falls at high speeds despite the authorities, and all the usual facts about radar detectors as applied to this detector and this trip. These conclusions are not traditional logical entailments of these utterances, nor truth conditions. Rather, they follow from the sentence in context and general knowledge of human communication and behavior. This type of conclusions was called conversational implicature by the philosopher H. Paul Grice [1975], and this paper examines the many roles of plans and plan reasoning in computing such conclusions.

1.1 Plan-based Natural Language Processing

Plan formalisms and reasoning play several roles in computing conversational implicatures. The general plan-based approach to natural language processing relies on plan formalisms to represent the subject matter of discourse [Grooz and Sidner 1986]. Plan formalisms can also be used to represent the communication process itself [Litman 1985]. Speech act theory [Austin 1962] suggests that utterances at the sentence level are like any other actions, with a role in achieving agents' goals. [Hinkelman and Allen 1988] extends the plan-based approach to better integrate surface properties of utterances, which are necessary adjuncts to plan reasoning in identifying agents' discourse actions. The present work uses several aspects of plan reasoning to derive conversational implicatures from discourse actions.

In STRIPS-like formalisms [Fikes and Nilsson 1971, Sacerdoti 1974] actions are operators which map between database states to correspond with changes produced by putative real-world events. They are represented in several parts: preconditions, constraints, body, arguments or variables, and effects. Preconditions are propositions that must be true in order for an action to succeed, such
as an agent’s having the authority to issue a command. Effects are propositions that describe the
normal results of the action, such as the hearer’s actually performing the command.

The body is some partially ordered set of component actions, empty for so-called basic actions
which are primitive. Variables are the agents and objects that participate in the action. Constraints
are propositions similar to preconditions, but representing conditions the agent cannot plan to
achieve. This entire action description amounts to a syntactic rule determining what states may be
linked by a particular action instance. Other plan formalisms may have different components, but
must still contain the same general information.

The structured action description may also be regarded as a set of syntactic rules linking the
parts. An occurrence of a plan is no less than the union of its preconditions, constraints, effects and
body, with the proper variable bindings, including temporal ones. (For a desired effect we may infer
the actions that will achieve it; mentioning a variable may allow inference of an action and hence its
goal and so on. Specifically, we have bidirectional rules mapping between plan parts and their type
or instance, under the appropriate assumption about their relationship:

1. Act(A) ⇔ P, Precond(P, A)
2. Act(A) ⇔ C, Constr(C, A)
3. Act(A) ⇔ E, Effect(E, A)
4. Act(A) ⇔ S, Step(S, A)
5. Act(A) ⇔ Using(V), Value(V, Role, A)

Note that this allows reasoning connecting several plans, as a proposition P may be a precondition
of some action A which must be achieved, and also the effect of the action B which thus enables
A. The only peculiar rule is the last one, because it asserts only that we can find some value that
will work for each variable (simultaneously), not that the roles are actually bound. A completed
instance, with all times in the past, must have a value for each role. The processing described in
this paper will be based on several kinds of propositional attitudes toward the links in our plan
representation:

• Being a plan or occurrence. (The above rules.)

• Constructing a plan instance. Novel sequences of actions can be constructed to achieve a goal
from some state; typically reasoning is backward from goal to action to precondition, then
treating the precondition as goal.

• Having a plan. To have a plan is to intend the plan’s body, effects, preconditions, and to bind
its variables, and to believe that the constraints hold. (Also the causality and sufficiency part.
See [Hinkelman 1987]). The actions may be performed entirely by other agents, but this will
normally require an assumption or evidence that they also have the plan. We will use the
predicate I for intend: I(A Act(A)) ⇔ I(P), Precond(P, A) and so on. Believe is B.

• Executing a plan. Establish its preconditions and constraints (find out about them, plan for
untrue preconditions), and do the body according to any time constraints.

• Attributing a plan. This is to believe that another agent has a plan; i.e., that they have the
appropriate beliefs and intentions. B(Agent1, I(Agent2, A Act A)) ⇔ B(Agent1, I(Agent2, P)),
Precond(P, A).
• Inferring a plan. This is to attribute a plan based on some evidence which is likely some plan part or related to one. An inferred plan can provide the basis for resolving references, understanding indirect speech acts, disambiguating among speech act interpretations, and other linguistic and nonlinguistic processes.

• Conversational implicature. If an agent gives evidence for a plan linguistically, the agent has conversationally implicated having it, or endorsing it to the hearer if the goal is the hearer's. This is really a variation on plan inference.

• Correcting a plan. We mention this only as another application. It involves believing that an agent has a plan, but that some link is faulty although the agent doesn't know it, for instance a plan's not really having the desired effect. See [Pollack 1986].

Each attitude has some additional inference rules and mechanism. It is important to note that the basic structure in alternative representations could be used in the same manner.

1.2 Plans as Conversational Implicatures

The central claim of this paper is that an analysis of intention is essential to computing some conversational implicatures, and a great help in finding many others. Not only must the system know about plans and be capable of reasoning about them, but it must also be able to identify and make use of the specific plans and goals active in this situation, since implicatures are based on them as well as on the utterance. Suppose the conversation between Sam and Bob had been different:

(2) Sam: By the way, do you have a radar detector?
    Bob: Sure, it's a Passport. They're real tiny, but big bucks.

Bob recognized that Sam implicated planning to buy one, and wanting information in order to select a model. Bob implicates that the Passport model is generally adequate, and available as far as he knows (albeit at a cost). Instead of offering his as a loan, he merely provides information about it. Sam's question, the same in both cases, is effectively a request for a loan in the first case, and soliciting information here. By the same token, a disjunctive utterance like "It's either in the basement or the attic" normally implicates that the speaker doesn't know which; during a treasure hunt or said by a teacher, the opposite is true. Implicatures depend on plan context.

2 Speech Act Interpretation

To calculate conversational implicatures, we need to identify the content and speech act interpretation of the utterance, and any plans. For the purposes of this paper we will adopt a simplified model of the early stages of utterance understanding. Assume that incoming utterances are parsed into a representation having the form (category <slot filler>*), where fillers may be words, feature values, or nested category structures. Semantic interpretation derives the literal meaning of the sentence by composing word meanings into an action/event description corresponding to the appropriate verb sense. This is filled out by reference processing, which interprets anaphora and definite descriptions, and yields a representation of the sentence content in a knowledge representation for general reasoning. The speech act interpretation of the utterance can then proceed using pragmatic interpretation rules as described in [Hinkelman and Allen 1988].
2.1 Pragmatic Interpretation Rules

Consider the speech act interpretation of “Can you go to the store?” The speech act representation that we use for now is the speech act category with its roles and values in the general knowledge representation. We develop pragmatic interpretation rules, whose left hand sides match against slot-and-filler sentence representations, and whose right hand sides are speech act descriptions. Here is a rule:

\[
(S \text{ MOOD YES-NO-Q}) \iff (\text{ASK-ACT ACTOR } !s
\text{ Addressed } !h
\text{ PROP V(REF))})
\]

The mood matches that in the representation of “Can you go to the store?” to yield this query of the hearer’s ability to go to the store.

\[\text{although you can REQUEST an INFORM-ACT.}\]

62
We now have both the request and yes/no interpretation.

2.2 Speech Act Disambiguation

Pragmatic interpretation rules combine to constrain and detail the possible interpretations of speech acts. They may yield a unique interpretation of the utterance, several, or none. (These in turn may be consistent or inconsistent, complete or incomplete.) The next bottom-up phase in interpretation is based on the principle that only plausible plans should be recognized (if there are any). If we attribute a plan to a speaker we must attribute all the appropriate beliefs and intentions that go into having a plan, as well. Each of the possible speech act interpretations is subjected to a verification process, in which the set of beliefs is constructed and checked for plausibility (of course it's even better if there is evidence for it).

1. \( B(Agent_1, I(Agent_2, A \text{ Act } A)) \Rightarrow B(Agent_1, I(Agent_2, P)) \), assuming \( \text{Precond}(P, A) \) and \( B(Agent_1, B(Agent_2, \text{Precond}(P, A)) \)

2. \( B(Agent_1, I(Agent_2, A \text{ Act } A)) \Rightarrow B(Agent_1, I(Agent_2, C)) \), assuming \( \text{Constr}(C, A) \) and \( B(Agent_1, B(Agent_2, \text{Constr}(C, A)) \)

3. \( B(Agent_1, I(Agent_2, A \text{ Act } A)) \Rightarrow B(Agent_1, I(Agent_2, E)) \), assuming \( \text{Effect}(E, A) \) and \( B(Agent_1, B(Agent_2, \text{Effect}(E, A)) \)

4. \( B(Agent_1, I(Agent_2, A \text{ Act } A)) \Rightarrow B(Agent_1, I(Agent_2, S)) \), assuming \( \text{Step}(S, A) \) and \( B(Agent_1, B(Agent_2, \text{Step}(S, A)) \)

5. \( B(Agent_1, I(Agent_2, A \text{ Act } A)) \Rightarrow B(Agent_1, I(Agent_2, V)) \), assuming \( \text{Value}(V, \text{Role}, A) \) and \( B(Agent_1, B(Agent_2, \text{Value}(V, \text{Role}, A)) \) register

It should be remarked here that there are some aspects of plans which can be treated as preconditions standard to most plans. They arise directly out of the planning process itself, and must be checked along with any preconditions explicit in the plan:

- the agent has some goal which the plan will achieve as an effect
- it isn’t obvious that the goal would obtain otherwise
- the plan is a reasonably direct and efficient method of achieving the goal and has a real causal relation to it
- agents are capable of performing the planned actions, especially agents other than the planner
- other agents are cooperative in small matters, capable both of planning and of understanding and recognizing intentional behavior.

The class of speech act plans has standard preconditions too, some of which we may prefer to assume rather than check:

- speaker and hearer have the same language
• the speaker can speak, the hearer can hear
• the utterance is a reasonably direct and efficient method of achieving the goal and its propositional content relates to it
• the speaker sincerely has the marked (intonation, sarcasm, explicit skepticism...) attitude toward any propositional content of the utterance
• the hearer is awake and paying attention
• speaker and hearer have a reasonable auditory channel; i.e., they are in the same room without lots of background noise, or have a telephone connection, or suchlike.

All of these things will count as conversational implicatures if the speech act interpretation is accepted, whether anyone thought about them explicitly in advance or no. If any of these conversational implicatures are implausible, a possible erroneous plan can be recognized. If the corresponding beliefs are not attributable to the speaker, the interpretation is discarded. This reduces ambiguity in cases like "Can you pass the salt?" If the speaker knows the answer to the literal question, the effect of the yes/no question already holds, and the standard precondition that you only plan for unobtained goals fails, eliminating the yes/no interpretation. The remaining interpretation is the request. If in fact the hearer couldn't pass the salt, this would rule out the request (since the speaker would presuppose that the hearer could) and answer the yes/no question instead. It is precisely this ambiguity that allows some indirect speech acts to function; the hearer is given room to maneuver. If more than one interpretation remains, the hearer has either to ask the speaker or to do more work, along the lines of the plan inference models like [Perrault and Allen 1980].

3 Domain Plan Inferences

Our utterances have been subjected to syntactic, semantic, and reference processing; also pragmatic interpretation and some disambiguation. Some ambiguity among speech acts may remain. Now we must recognize any plans that the speech acts refer to, or are part of, making use of any expectations provided by the context. Finally, we must also compute conversational implicatures.

3.1 Recognizing Embedded Plans

Besides recognizing the actions agents perform in speaking, we wish to infer what we can of agents' goals and plans which are served by discourse. A most important source is the content of their utterances, of course. When they are most explicit, action descriptions are constructed during semantic and pragmatic interpretation. So for instance an explicit request will contain a description of the requested action, as will a conventional indirect request, after pragmatic interpretation. These explicit action descriptions are processed in the same way as speech acts: the attitudes about their preconditions, constraints, effects, and so on are checked to see if they make sense. If so, the action description stands as a possible plan. If not, it may be eliminated or identified as an erroneous plan.

The situation is more complicated when the processed utterance does not contain a complete action description. If it is about a plan, however, it probably relates to one of the plan's parts or roles, and we can take advantage of this. We use the same original set of rules for plan reasoning as a basis for plan inference. Definite descriptions can trigger the role rule to index plans containing the appropriately typed variable, for instance. Propositions may unify with plan preconditions, constraints, or effects to infer a plan. An alternative mechanism for doing this is a classifier such as KL-ONE [Brachman and Schmolze 1985]. The plans that are found can be fed to a Kautz-style plan recognizer to identify the end actions that underlie them or to reduce disjunctions. The output plans must then be processed for plausibility just as the speech act interpretations were.
3.2 The Role of Expectations

Often considerable help is available to plan recognition in the form of expectations. In fact, without expectations we may be able to recognize very little [McCafferty 1986]. Expectations are of two kinds: plans already believed to be in progress, and plans that are likely in this context. The question/answer pair is a discourse plan with very strong predictive power, and if a question has been asked, the only appropriate responses are an answer (the information or performance directly requested), additional information or actions that are salient to the answer, information or actions instead of the answer, and additional information provided or actions performed that are salient to this substitute for an answer [Webber 1987]. So a strong attempt should be made to match a response to the question asked, before considering the alternative interpretations. Likewise, consider any specific domain role like being a library clerk. If someone comes up to you, you have a priori a set of maybe half a dozen plans that may apply: they may need help finding a book or article, or information about one; they may be returning books or overdue ones, checking them out, borrowing reserve books, or asking where some section is. There may be others, to be sure, but the vast majority will be one of these or related to one of these.

Incoming plan descriptions are therefore matched against any plans in progress or their parts, then against any likely plans and their parts (subject to temporal constraints). For instance, imagine arriving at the trailhead with some hiking friends, and while people are lacing their boots, one asks you,

(3) Do you have a watch on?

You answer the question, concluding that the person thinks she may later want to time some cooking, and is trying to decide whether to bring her own watch. Most of the time hiking you don't need to know the time. But suppose you were sitting in a boring lecture when someone pencilled the same question on your pad. In this situation she likely does want to know the time then, so you tell her. Or consider a library with copy machines by the desk. A student thinking about buying some candy appears.

(4) i: Can I have change? (proffers $20.)
ii: Not for a twenty.

The student leaves and manages to get two tens, but on his return is told that change is available only for the photocopy machines. The librarian had assumed he wanted to make copies, and licensed the implicatures that the student could have change for smaller bills, and that the student's plan would work. The student had computed the implicatures that he could have change for a smaller bill and that his plan would work, knowing that the plan was to buy candy. Under both interpretations the literal content of the librarian's statement is true, but the implicatures are different. Had the librarian correctly recognized the student's plan, he would have been obliged to state that he was unable to change any denomination.

In the first example, plans in progress determined speech act recognition. In the second, a likely plan was (incorrectly) recognized. If simple matching fails, we can fall back on techniques like those of [Perrault and Allen 1980, Kautz and Allen 1986].

3.3 Domain Plan Implicatures

Domain plan implicatures are very similar to speech act implicatures, inasmuch as they refer again to the rules for attributing plans. Any domain plan that we recognize, whether via discourse or not,
must still undergo the verification process that we described when discussing speech acts. And if the plan interpretation is accepted, the implications will be conversational implicatures if the recognition was through discourse.

Here are the implicatures for Grice's gas station example, in which the dialogue is as follows:

- (5) A: I'm out of petrol.
  B: There's a garage round the corner.

In this case B is suggesting a plan for A, rather than A's implicating that he has the plan, so the propositional attitude is that B believes these points:

- There are a seller, a location, a time and some gas, each with reasonable value. (The variables in the plan have reasonable bindings.)
- The agent has some money or can plan how to get some. (The plan preconditions hold or can be achieved by plans.)
- The seller owns the gas, and both are at the gas station location at the time. (Plan constraints will hold at the time of plan execution.)
- The agent need only go there, hand over the money, and receive the gas. (Plan decomposition is appropriate and workable.)
- Then the seller will own the money and the agent, the gas: (The effects of the plan will hold after its execution.)
- There isn't anything likely to interfere with this plan. (The effects of the plan will hold after its execution.)

Now that we know how our utterance processing works, and how we will recognize plans, we have the input that we need to compute such implicatures.

3.4 Examples

Recall our original example: Sam and Bob are old school pals from Backwoods, Montana. Sam tells Bob that he is leaving on a shopping trip to Great Falls. The conversation continues:

(6) Sam: By the way, do you have a radar detector?
    Bob: Sure, it's out in my truck. When you getting back?
Bob has interpreted Sam's question as a request to borrow his radar detector. He pursues the details of the loan. Sam now assumes that Bob's radar detector is in working order, will allow him to drive to Great Falls at high speeds despite the authorities, and all the usual facts about radar detectors as applied to this detector and this trip.

Initially, Sam is in the middle of his conversation with Bob, with the shopping plan hovering in the background. This state is shown in Fig. 1. Sam selects a Request plan, in order to get the use of a radar detector. His utterance is realized using the pragmatic rules.

When Bob completes linguistic processing of Sam's utterance, he has two interpretations of it: a yes/no question and the loan request. But he is able to find no explanation for the yes/no question. The UseRadar plan is one of a very small number of plans with a radar-detector variable, and is thus found using the variable-action rule. Since Sam's shopping trip is a good slot filler for a UseRadar plan, the variable binding integrates the two plans and provides a coherent interpretation of the speech act, which satisfies a precondition on borrowing it, which itself satisfies a precondition on using it. We will say that all these inferred plans are conversationally implicated by Sam, so long as he intended them to be inferred.

When Sam interprets Bob's response, he makes use of the plan structures that Bob recognized. The speech acts he takes as follows. "Sure" counts as consent to: the borrowing action, which already licenses our plan-based implicatures. Giving the location of the detector is an Inform-Act of a variable in the Borrow-Act, and thus furthers the Borrow-Act. And the question is a wh-question for establishing the endpoint of the Borrow-Act.

Let's consider what a system like this would do with some of our other plan-based implicature examples. In the library story, for example, the system models the librarian as having a set of likely plans to attribute to patrons. The student has a plan to buy candy, which occasions a plan to get change, which leads to his Request. The response is an Inform carrying the scalar implicature that he could have change, for alternate values of the bill. The librarian (erroneously) recognizes his plan as getting change for the copy machine. The implicatures that the student computes from the librarian's response are that his change and candy plans will work, for smaller denominations. However, the librarian has licensed the implicatures with the copy plan in mind. When the student

---

3See Hirschberg for the computational account of scalar implicature.
\begin{align*}
S1 & = (\text{Shopping-Act Agent Sam } \\
& \quad \text{Stores } [\text{S|Loc(S, GreatFallsMT)}]) \\
C1 & = (\text{Converse-Act Agent [Sam Bob]}) \\
A1 & = (\text{Ask-Act Agent Sam } \\
& \quad \text{Responder Bob}) \\
B1 & = (\text{Borrow-Act Agent Sam } \\
& \quad \text{Owner Bob } \\
& \quad \text{Object detector1}) \\
U1 & = (\text{UseRadar-Act Agent Sam } \\
& \quad \text{Trip S1}) \\
R2 & = (\text{Assent-Act Agent Bob } \\
& \quad \text{Hearer Sam } \\
& \quad \text{Action } [\text{B1 U1}]) \\
I1 & = (\text{Inform-Act Agent Bob } \\
& \quad \text{Hearer Sam } \\
& \quad \text{Prop Loc(detector1, car8899)}) \\
A2 & = (\text{Ask-Act Agent Bob } \\
& \quad \text{Responder Sam}) \\
Q1 & = (\text{Query-Act Agent Bob } \\
& \quad \text{Responder Sam } \\
& \quad \text{Wh-var S1.End-time } (\leftarrow \text{U1.End-time } \leftarrow \text{B1.End-time}) \\
\end{align*}

\textit{Figure 3: The Response with Plans Inferred}
returns, the error is discovered. The system thus provides a framework in which problems of plan-
derpendent implicatures can be discussed.

4 Conclusion

We have shown that the definition of a plan can be the basis for several important processes in
understanding utterances. They include having a plan, attributing a plan, recognizing both speech
acts and domain plans, and computing conversational implicatures. These processes are integrated
into a scheme for computing the plan-based implicatures in typewritten dialogues.

Acknowledgments

This work was supported in part by NSF research grants no. DCR-8502481, IST-8502746, and
DCR-8320136.

References

[Hinkelman 1987] Elizabeth Hinkelman, “Thesis Proposal: A Plan-Based Approach to Conversa-
tional Implicature”, Technical Report 202, Dept. Computer Science, University of Rochester,
June 1987.
[Hinkelman and Allen 1988] Elizabeth A. Hinkelman and James F. Allen, How to do things with
proach for Understanding Dialogues”, Technical Report 170, Department of Computer Science,
University of Rochester, 1985.
Indirect Speech Acts”, American Journal of Computational Linguistics, 6(3–4):167–82, July-
December 1980.

69


A Model for Parallel Programming

Lawrence A. Crowl

Abstract

Current parallel programming languages support only a narrow range of programming styles; force programmers to bind the specification of parallelism too early in program development; and limit the class of architectures for which they are effective. This paper presents a model for parallel programming based on uniform encapsulation and control, concurrent operations on objects, synchronous invocation with early reply, references to code in an environment, and a copy model of parameters and variables. This simple, coherent model enables programmers to use different styles of programming and to bind parallelism late in program development. In addition, the model may be efficiently implemented on a wide range of multiprocessor architectures.

1 Introduction

Current imperative languages for parallel programming tend to provide separate facilities for parallel programming on top of a base sequential model of programming. These facilities typically include a distinct notion of process, monitor constructs, and messages. The dichotomy between the constructs for sequential and parallel programming limits the effectiveness of programmers in exploiting multiprocessor hardware because parallelism is not available at all levels within a program based on the dual model. In addition, the choice of mechanisms tends to limit the style of parallel programming.

The remainder of this introduction distinguishes between parallel and distributed programming and describes three problems with current parallel programming systems. Section Two presents and justifies a model of programming that unifies the sequential and parallel aspects and is both simple and powerful. Section Three shows how the model addresses the problems with current systems. Section Four summarizes the paper.

1.1 Parallel Versus Distributed Programming

Distributed programs assume an environment in which the parallelism available does not exceed the number of relatively independent functions to be performed, e.g., file managers, transaction managers, screen managers, etc. In such an environment, programs can achieve sufficient parallelism by decomposition along functional lines. Thus, the main task of a distributed program language is managing concurrency, not exploiting it. On the other hand, parallel programs tend to assume an environment in which there is a small collection of highly coordinated functions to be performed on a large collection of data. In this environment, decomposition along functional lines does not provide enough parallelism. Parallel programming languages must be able to readily exploit significantly more of the parallelism available within a problem. In particular, they must be able to parallelize over the data available (data parallelism), in addition to possibly parallelizing along functional lines. This paper addresses parallel programming, as opposed to distributed programming. No aspect of the proposed model excludes its further development in support of distributed programming. Indeed, the model very naturally extends into distributed programming. However, this paper will not address the aspects of distribution.
1.2 Limited Programming Styles

Many parallel programming systems are available, supporting programming styles ranging from very loosely coupled messages (PLITS [Feldman 1979]) to tightly-coupled shared memory (the Uniform System [BBN 1985]). Most of these systems support only a single programming style. Different programming problems are often more naturally expressed using very different programming styles. These different problems often appear within a single application. With the available parallel programming systems, programmers must either use a single style even when awkward for the task, or use multiple programming systems.

The use of a single style unsuited to the problem at hand makes the program difficult to produce. For instance, the programmer may be forced to implement a stream filtering mechanism in a shared-memory model when it is most naturally expressed in a message-passing model. The multiple programming systems approach means that programmers must deal with several programming models, all of which have different mechanisms and constraints. This represents a considerable mental load because programmers must learn and remember the subtle interactions between different models. More importantly, managing the interface between different mechanisms in an application represents a considerable portion of the coding effort [Atkinson et al. 1983].

An ideal language model for imperative parallel programming should support a wide range of programming styles from shared memory to message passing, and a granularity of sharing ranging from integers to large data structures, while providing a single, simple conceptual framework.

1.3 Early Binding of Parallelism

Many current programming systems provide separate mechanisms for the parallel and sequential portions of a parallel program. This presents programmers with two qualitatively different models of interaction between program components forcing them to bind the location and granularity of parallelism early in program development.

In a distributed system, communication between different processors costs typically two orders of magnitude more than communication within a processor. Many distributed programming languages, such as PLITS [Feldman 1979], make distributed objects visible within the language under the assumption that programmers will manage visible costs more effectively than invisible costs. Programmers tend to make distributed objects large so as to minimize interactions between them. However, in multiprocessors, the difference in local and remote communication costs is much lower, which encourages more frequent communication within programs. The proper distribution and level of implementation for objects is not obvious before doing performance experiments on completed code. Choosing the granularity and location of parallelism should be more a part of the optimization effort rather than the algorithmic effort.

When parallel and sequential mechanisms are distinct, programmers must decide upon whether a given object will be implemented in a parallel manner or sequential manner early in program development. However, programmers may not know ahead of time which manner is appropriate for a given object. Under this situation, programmers may choose the higher-cost, more general mechanism [Greif et al. 1986]. If a choice of a mechanism is incorrect, the programmer must recode the object and reintegrate it into the environment. General purpose abstractions must often be provided in both mechanisms in order to provide programmers with the incentive to use them. In addition, programmers using an object often must know in which mechanism the object is implemented. Because the use of an object may be distributed throughout the program, a change in mechanism could involve rewriting much of the program. Programmers will only perform such changes under extreme circumstances.

A programming model that enables the programmer to delay the binding of actual parallelism must be based on abstraction mechanisms that apply to both sequential and parallel execution.
Abstraction mechanisms are more important in parallel programming because abstraction affects performance.

1.4 Limited Architectures

One of the consequences of programming systems that bind parallelism with special language constructs is that many parallel programming systems work well on only one architecture because the underlying model assumes a specific, limited hardware configuration. The appropriate architecture is embedded in the level at which the language presents parallelism. In other words, these languages represent a means for programming a machine, not a description of a parallel solution.

A parallel programming language that provides for high portability of programs must enable description of parallelism at all granularities within a program. In order to use efficiently a wide variety of parallel architectures, programmers must be able to tune a program to an architecture without affecting the semantics of the source.

2 Model Description

This section describes the proposed model of parallel programming. This is a description of a model, not a description of a language. The model has a wide range of possible instantiations as programming languages. There are still many issues in language design that this model leaves unspecified, such as static or dynamic typing.

2.1 Uniform Encapsulation and Operation

The proposed programming model provides mechanisms for encapsulation and operation that apply uniformly to both parallel and sequential portions of a program.

Objects Provide Encapsulation

Many parallel programming systems have two mechanisms for encapsulation, one for parallel and distributed objects and one for local and sequential objects. This dual mechanism approach splits the programming environment into two qualitatively different models of interaction, introducing an artificial granularity in the program's parallelism.

Parallel programming systems need encapsulation mechanisms which apply uniformly to both parallel and sequential objects. The proposed programming model addresses these problems by providing a single encapsulation mechanism, the object. The object model provides natural encapsulation of abstract data types and scales effectively from simple integers to distributed databases.

Objects provide natural units for distribution. Distributed systems and non-uniform-memory-access multiprocessors have substantial performance differences depending on whether or not communication occurs within a processor or between different processors. To use such hardware effectively, program systems must limit the amount of communication between processors. Objects provide a natural destination for communication, and hence aid the programming system in reducing communication.

Synchronous Operation Invocation

Not only must the encapsulation mechanism apply uniformly to parallel and sequential objects, the means for communicating with them must also apply uniformly. The originating object must be able
to initiate communication without knowing how the receiving object will handle it, and vice-versa.

Several programming systems, e.g. Starmod [Cook 1980], allow both synchronous communication (procedures) and asynchronous communication (messages), but usually require both sender and receiver to agree on the form. These systems have non-uniform communication.

Most computations initiate communication with the intent of receiving a reply. Message-based communication requires the programmer to filter incoming messages in search of the reply value. To support this, message-based programming languages, e.g. PLITS [Feldman 1979], usually provide complex filtering mechanisms. In addition, message-based systems usually provide synchronous invocation for primitive operations, such as waiting on a message. Thus, message-based systems tend to be non-uniform. The complexity and non-uniformity of most message-based languages has lead to a greater concentration on synchronous, procedure-based communication. The evolution of the asynchronous, message-based ECLU [Liskov 1979] into the synchronous, atomic-transaction-based Argus [Liskov and Scheifler 1983] provides evidence for this trend.

Given these considerations, the proposed model provides synchronous communication in the form of operation invocation on objects as the sole communication mechanism. The invoker of an operation waits upon receipt of the reply value. Reply values are implicitly returned to the caller and do not require explicit addressing. The model does not provide asynchronous communication directly because it has a straight-forward implementation with other concepts in the model.

2.2 Copy Model of Variables and Parameters

There are two major models of variables and parameters in imperative languages. In the conventional value (or copy) model, variables contain values. Fortran, Algol, and its derivatives use this model. In the reference model, variables refer to objects. Smalltalk [Goldberg and Robson 1983] and CLU [Liskov et al. 1977] use this model. The desire for a uniform encapsulation mechanism implies that a parallel language must choose one model and stick with it. This includes variables for parallel abstractions, parameters, and variables local to an abstraction. Most imperative parallel languages provide a reference model for concurrent abstractions, but a value model for parameters and local variables.

The reference model has some attractive properties. Unfortunately, it has some serious disadvantages in the context of parallel and distributed programming.

aliasing: The reference model naturally provides for extensive aliasing among objects. It is not generally possible to determine a priori when two variables will refer to the same object. The result is that the programmer and language system must assume that the variables may refer to the same object. This inhibits parallelism. In contrast, the copy model ensures that each variable refers to a different object. Thus the programmer and the implementation are free to operate on two different variables concurrently.

contention: In multiprocessor systems, mutable objects may be a source of contention. Many programs will require access to the current states of objects, but not to their potential future state. To reduce contention for these objects in the reference model, the programmer has two options, to make the objects immutable or to explicitly make copies of the objects. Both cases increase the amount of dynamic allocation in comparison to the copy model. More importantly, these options represent an explicit coding of the copy model, but with higher run-time and programmer costs. An implicit copy model often requires less run-time support, and hence costs less.

remote arguments: The reference model implies heavy communication between machines as operations traverse back and forth across machine boundaries to reach objects referenced by the
objects passed as parameters. This potential inefficiency on distributed systems is such that Argus, which has an internal reference model based on CLU, uses an external value model. On the other hand, the Emerald system [Black et al. 1986] uses the reference model over a local area network. Emerald achieves some measure of efficiency by moving parameter objects, and the objects they refer to, from the calling machine to the called machine. This approach fails under contention for the argument object. In contrast, the copy model moves all necessary information at the point of call. Communication occurs exactly twice, once sending the parameters, and once sending the results.

**Parameter optimizations:** Parameters passed "by value" are more amenable to optimization than parameters passed "by reference". This flexibility is more important on multiprocessors because they tend to provide a much larger set of tradeoffs between passing pointers and doing physical copying. The reference model discourages "by value" parameters, and hence provides more limitations on parameter passing mechanisms. These limitations become important when porting programs among different multiprocessors.

**Garbage collection:** Under the reference model, references to an object may spread freely. Because the existence of a reference to an object implies the existence of the object, it is generally not possible to determine the lifetime of an object statically. The undeterminable lifetime of objects implies dynamic heap allocation and system-wide garbage collection. To perform garbage collection, the system must examine the entire set of references of the system to ensure that no references to an object exist before deleting the object. This is possible on multiprocessors, but on very large or widely distributed systems, the number of possible locations for a reference is too large to be effectively searched. This issue is important because institutions invest in parallel programming for speed. They are willing to purchase performance with engineering effort. The reference model inhibits performance. On the other hand, the variables under the copy model contain objects. This enables static determination of the lifetime of an object, because the lifetime of the variable containing it is known, which in turn allows more efficient storage management.

The proposed programming model adopts the copy model of variables and parameters. Variables contain objects, they do not refer to objects. This means that operation arguments are objects, not object references. Because a strict copy model of computation can only represent hierarchical structures, the proposed model provides an explicit object reference capability. The type of a reference depends on the type of the referent.

Real programs must deal with values as well as references, so the choice of the copy model in a language represents a bias, and not an absolute choice. The bias of the copy model is more appropriate for parallel programming.

### 2.3 Uniform Control

Parallel programming distinguishes itself from distributed programming by utilizing data parallelism as well as functional parallelism. To support this, a model for parallel programming must provide mechanisms to control the partitioning of work. This means that the programmer must be able to reference code in addition to data. The desire for late binding of parallelism coupled with the need to reference work, implies that control constructs apply uniformly to parallel control as well as to sequential control. Since the appropriate parallelization of control is generally dependent on user data structures, a programmer must be able to define new control structures as well as use primitive control structures. The issue of programmable control abstractions is not unique to parallel systems, it is common to programming in general.
Single Argument and Result

Control abstractions will often need to delay the actual invocation of an operation, or apply an operation to a number of objects. In addition, control abstractions may need to combine the results of several computations. In order to describe general-purpose abstractions to perform such tasks, an intermediary must be able to handle arguments and results of operations as single units. Message based systems naturally provide this capability by referring to messages as a whole. RPC based systems do not provide a mechanism to refer to the set of arguments. (Suitable changes would enable RPC systems to refer to the set of arguments.)

The proposed model provides a handle for arguments and results by allowing exactly one parameter and one result for each operation. The effect of multiple parameters is achieved by passing a record as the single argument. If a language based on the proposed model provides record constructors, this approach can be as notationally concise as a list of parameters. Those methods that do not need an argument, or have no useful result, accept or return an empty record. Control abstractions need only handle the case of one argument and one result.

Object Reference and Operation Pair

One of the advantages of message passing systems is that they may be connected into rich networks where the nature of intermediaries is not necessarily known in advance. Passing references to neighboring objects allows effective construction of networks. However, if the operations must be directly named, each sender in a network must know the name of the operation of the recipient. This requires the sender and receiver to agree on an operation name a priori. This implies that programmers may have to place in the communication path many intermediary objects whose sole purpose is to translate operation names. Similar problems arise with static typing of message recipients. Network construction under these circumstances will be an ad hoc task.

The proposed model provides for object type and operation name independence with ports. A port is a first-class entity binding an operation to an object reference. An operation is invoked when an argument object is applied to a port, in direct analogy to sending a message. The user of a port may need to know the type of the argument and the type of the result, but does not need to know the operation name or the type of the port's object.

Blocks

In the implementation of a control abstraction, the abstraction must be able to accept a unit of work to perform. This work is usually defined in terms of the context of a procedure which called the control abstraction. For instance, the body of a for-loop is the unit of work passed to the for-loop abstraction. The local variables of the procedure in which the loop is embedded provide the context for the loop body. The most effective support of work within a context is represented by Smalltalk's blocks [Goldberg and Robson 1983], or Lisp's closures [Steele 1984]. Blocks are portions of code that have access to an operation's local variables, but may be executed by another object. This is similar to passing nested procedures in Pascal, but with the added notational convenience of being defined in-line. Blocks provide Smalltalk with substantial power. For instance, in Smalltalk the programmer may define an operation on trees that accepts a block and then executes the block for each element in a tree.

The need for blocks is even greater for parallel languages because of the need to provide many abstractions for generating parallelism. If there is no such control abstraction, each programmer must build ad hoc mechanisms for creating parallelism. This increases the cost of developing highly parallel programs and markedly increases the cost of changing the method of parallelism. This in turn inhibits the specification of extensive parallelism and especially data parallelism. For instance,
without a handle for work within a context, users cannot define general purpose control abstractions, such as a parallel for-loop, without explicitly collecting references to the relevant portion of the environment.

The proposed model provides for work within a context with blocks. The contexts of executing operations are *activation objects* (or activation records). Blocks are anonymous operations on activation objects. As such, blocks are implicitly an object reference and operation pair, so the specification of a block provides a port. This port is exactly the same mechanism introduced earlier, and may be passed outside the enclosing scope to be executed by other objects. Blocks maintain access to the variables defined within the scope of the block. (The existence of the environment is identical to the existence of the object representing the environment. Any mechanisms for determining the existence of a normal data object also apply to environments.) The ability to pass blocks as ports provides the basis for control flow within the language. For example, boolean objects have an ‘if’ operation which accepts a port (usually a block) to invoke if the boolean is true.

**Operations as Data**

A model that does not provide an indirect handle for operations (one that has no mechanism for non-literal reference to operations) prevents the programmer from building general-purpose control abstractions that can control the timing and ordering of operations. For instance, consider an abstraction to apply an operation to all the elements in an array. If operations must be directly named at the point of call, it is not possible to build a general-purpose abstraction for distributing an operation over an array. *In such abstractions the operation to be performed becomes data.* Without mechanisms to support operations as data, programmers must construct such abstractions on an *ad hoc* basis, which in turn discourages the use of such constructs. Such constructs are essential to enable effective exploitation of data parallelism.

There are two approaches to providing operations as data. The first approach is to provide variables whose values are operation names. This approach forces run-time type checking, so is only appropriate for dynamically typed languages. For statically typed languages, one may bind an operation and an object type into an *action*. An action is similar to a port, but requires an object upon which to operate in addition to an argument object. The choice between these approaches depends upon the nature of the language in which the model is implemented.

2.4 Concurrency Management

Parallel programming necessitates maximum exploitation of the available concurrency. Because the appropriate boundaries between parallel and sequential implementations may not be known until late in program development, programmers must have the ability to specify concurrency at all points within their program, and to change the implementation of an abstraction to increase or decrease the utilization of available concurrency while not affecting other abstractions. Programmers can change the implementation of general-purpose abstractions by selecting a different implementation from a library.

**Concurrent Operations**

In a system with uniform encapsulation, many abstractions will necessarily be large. The programming language must enable extensive concurrency within a single abstraction and between an abstraction and its callers. Any limitation on the potential concurrency within the encapsulation mechanism will magnify itself at each level of abstraction.
Many parallel programming systems, e.g. Distributed Processes [Brinch Hansen 1978], make processes the unit of distributed encapsulation. Since the encapsulation provides only one active thread of control, these systems will force programmers to abandon encapsulation in the higher levels of their systems in order to maintain an acceptable level of concurrency. The lack of dynamic process creation can inhibit effective programming of client/server relationships in those languages with synchronous communication mechanisms [Liskov et al. 1984]. In addition, these systems cannot represent true concurrent data structures. The ability to support concurrent data structures is an important feature of shared-memory multiprocessors. Object-based systems, e.g. Concurrent Smalltalk [Yokote and Tokoro 1986], often avoid the pitfall of making processes the unit of encapsulation, but sometimes implicitly enforce mutual exclusion between operations on an object. Such systems will not take full advantage of the capabilities of shared-memory multiprocessors.

Several programming models, e.g. Eden [Almes et al. 1985], allow concurrent operations within an object, but only after explicit dispatching. In explicit dispatching, the receiver explicitly indicates when the processing of a request begins. This explicit dispatching introduces a weak form of serialization in that the resources devoted to dispatching processes are limited. This serialization for dispatching limits the amount of potential parallelism, increases the latency of operations, and introduces unnecessary costs in many cases.

The proposed model provides concurrent execution of operations on objects. Each object invocation provides a new, concurrent thread of control within the object. The operation invocation itself implicitly dispatches the thread of control for execution. Thus, while invocations are synchronous with respect to their callers, they are asynchronous with respect to other invocations on the object. The model itself does not supply synchronization. If the programmer needs to synchronize invocations, the programmer must explicitly program such synchronization using the language or implementation defined object types that provide synchronization.

Asynchronous Processing

Many abstractions can solve the information needs of their clients long before all of the associated computations are complete. A mechanism for processing an operation asynchronously to the caller reduces the non-essential synchronization between processes. Allowing the programmer of an object to minimize synchronization with the external world increases the potential concurrency within a program.

The proposed programming model provides for asynchronous processing with an early reply. After the reply, the callee may continue processing for an arbitrary time. The use of asynchronous processing is transparent to the caller, allowing the implementation of an object to change according to the system's need for concurrency. Since the invoker may continue after receiving the reply, this early reply provides a source of parallelism. This is the sole source of parallelism provided by the model. Early reply maintains the local state of the operation after reply, but without the cost of forced concurrent access to activation variables. Early reply also implies an independence between the invoker and the operation. The termination of the invoker does not imply the termination of the invokee.

Asynchronous Invocation

Not only must a parallel programming model handle requests in parallel, it must be able to make parallel requests and receive the result at a later time in order to fully exploit available parallelism. This capability is known as asynchronous invocation. The caller's use of asynchronous invocation must be transparent to the callee. Otherwise, the programmer must provide and name two different forms for each operation.
The programming model could introduce additional mechanisms for asynchronous invocation, but early reply, coupled with generic definitions, is sufficient to enable straight-forward definition of asynchronous invocation. We demonstrate this by using the proposed model to outline (in pseudo-code) the implementation of a future, which is an object that asynchronously executes an operation and allows the result to be retrieved later. A future is a generic definition, requiring the type of the argument and the type of the result.

begin object 'future(argument,result)'
  'hold' is a variable of type result
  'ready' is a binary semaphore initially locked

begin operation 'do', parameter 'request' is of type
  record 'action' is of type port(argument,result)
  'data' is of type argument
  end record
  reply with an empty record -- this allows caller to continue
  assign result of 'request.data' applied to 'request.action'
  to the variable 'hold'
  signal 'ready' -- 'hold' now contains the result
end operation

begin operation 'wait', parameter is of type empty record
  wait for 'ready'
  reply with value of 'hold' -- return result of request
end operation
end object

The use of a future is straightforward.

'async' is a variable of type future(integer,integer)
apply record action is '3.add'
  data is '4'
  end record to 'async.do'
... do other work ...
apply empty record to 'async.wait' and use the integer result

Often, many invocations can start simultaneously. For example, each object in an array may be initialized in parallel. This capability is implied by asynchronous invocation because programmers may use an array of futures and loop through the array starting invocations.

While programmers need asynchronous invocation, a sufficiently powerful programming model need not support asynchronous invocation as a primitive operation. The future abstraction, which provides for asynchronous invocation, has a simple implementation within the proposed model. In addition, such an approach encourages the implementation of different types of futures, each tuned to the needs at hand. For instance, possible future types are single asynchronous reply, first of several asynchronous replies (including timeouts), and priority ordering of replies as they arrive.

3 Problem Solutions

This section shows how the proposed model addresses the three difficulties with current parallel programming systems discussed in the introduction.
3.1 Multiple Programming Styles

We provide evidence for multiple programming styles by showing the implementation of both shared-memory and message-passing styles within the proposed model.

Shared memory systems are characterized by processes calling procedures concurrently on shared objects. This style is easily programmed under the proposed model by delaying the reply of an operation until the end of the operation.

begin object 'shared-memory-abstraction'
  ... object's variables ...
  begin operation 'procedure', parameter 'whatever'
    ... operation's local variables ...
    ... do work ...
    reply with return value
  end operation
end object

The implementation of message passing styles is slightly less convenient, but still easy. For this example, messages are processed serially. Each object replies immediately upon receipt of an activation, synchronizes the activation with other activations, and completes the computation.

begin object 'sample'
  ready is a binary semaphore which is initially unlocked
  begin operation 'send', parameter 'message'
    reply with empty record -- this lets the caller continue
    wait for 'ready' -- this ensures mutual exclusion
    ... do work ...
    signal 'ready' -- allow next invocation (message)
  end operation
end object

Invoking the send operation with a message argument on an instance of sample, is equivalent to sending a message.

3.2 Late Binding of Parallelism

One of the goals of the proposed model is to allow the binding of parallelism late in the program development cycle. The model supports late binding through mechanisms that allow the disciplined use of data and control abstractions with multiple implementations. Each implementation exploits a different level of concurrency, at the appropriate cost. The multiple implementations strategy applies to both user-defined and primitive abstractions.

For example, consider the following implementation of matrix addition.

for i in 1..n do
  for j in 1..n do
    a[i,j] := b[i,j] + c[i,j]
There are no dependencies between distinct iterations of any of these loops. In choosing an abstraction which provides no guarantees on the relative execution of iterations, programmers may delay the choice of sequential or parallel implementations of the loops. On uniprocessors, the most efficient approach is to use no parallelism.

```plaintext
for i in 1..n .sequential do
    for j in 1 .. n .sequential do
        a[i,j] := b[i,j] + c[i,j]
```

On a uniprocessor with vector hardware, the most efficient approach may be to parallelize the inner loop, but maintain sequentiality in the outer loop.

```plaintext
for i in 1..n .sequential do
    for j in 1 .. n .parallel do
        a[i,j] := b[i,j] + c[i,j]
```

On a shared-memory multiprocessor, the most efficient approach may be just the opposite: to parallelize the outer loop, but maintain sequentiality in the inner loop.

```plaintext
for i in 1 .. n .parallel do
    for j in 1 .. n .sequential do
        a[i,j] := b[i,j] + c[i,j]
```

Late binding requires that the programmers use abstractions that meet their sequentiality needs, but that do not greatly exceed their needs. This is because the possible implementations of an abstraction are constrained by the sequentiality it guarantees.

### 3.3 Multiple Architectures

The proposed model supports multiple architectures by associating communication with abstraction. Since programmers will use layers of abstractions, the implementation can communicate across processor boundaries at any layer of abstraction. Because the binding of actual processor-to-processor communication to object invocation can occur very late in program development, the model allows the programmer to tune a program to an architecture without affecting the integrity of the algorithm.

For example, consider executing a program on both a distributed, message-based system and on a shared-memory multiprocessor. On the distributed system, object invocation at higher levels of the programs abstractions would be implemented by messages across the communication network, and invocation at lower levels would be implemented by procedure calls. Expensive message traffic is reduced by using messages only at the highest levels of program. On the shared-memory multiprocessor, object invocation at all levels of abstraction would be implemented by procedure calls, except at the lowest level were the processor reads and writes individual machine words. Processor communication occurs with reads and writes, and avoids the expense of constructing messages.

Note that not all programs that execute efficiently on a shared-memory multiprocessor will execute efficiently on a distributed system. In particular, if a program has a great deal of small, communication intensive objects with no intervening layers of abstraction, the communication graph may be too fine-grained for efficient implementation on a distributed system.

### 4 Summary

A single program may have several programming problems, each suited to a different programming style. Current imperative programming languages tend to support only one programming style.
Given a task with problems appropriate to different styles, a programmer must choose inappropriate styles for some problems, or attempt to mix programming systems. Neither is an attractive solution. This paper presents a programming model based on concurrent operations on objects with an early reply, which allows programmers to use multiple styles of programming within a single computational framework.

Abstraction mechanisms are more important in parallel programming because abstraction and parallelism interact. Abstraction affects performance, so the abstraction mechanisms must not inhibit parallelism. Programming languages that provide parallelism with separate and distinct mechanisms for encapsulation force the programmer to bind parallelism too early, which inhibits parallelism. The proposed model is based on uniform encapsulation, which provides abstraction independently of parallelism. Likewise, separate mechanisms for sequential and parallel control flow also force early binding of parallelism. The model has uniform control, in which control constructs apply uniformly to both sequential and parallel control flow, and which enables powerful user-defined control abstractions. Uniform encapsulation and control enables programmers to delay the specification of actual parallelism within a program.

The proposed model provides communication via operation invocation. This, coupled with a copy model of variables and parameters, enables communication to scale with abstraction. Uniform encapsulation and scalable communication enables the programmer and language processor to exploit parallelism at many levels within a program. Because parallelism may be exploited at many levels, some programs may execute efficiently on many different architectures by rebinding the actual parallelism to different levels within the program.

Acknowledgments

Special thanks to John M. Mellor-Crummey and César A. Quiroz for their many helpful comments in the development of this paper.

This work was supported by United States Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001 and National Science Foundation Coordinated Experimental Research grant number DCR-8320136.

References


