MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1965
AXIOMATIC SPECIFICATION
OF
SYNTAX-DIRECTED TRANSLATION
by
Sharon Sickel
W.M. McKeeman
Technical Report No. 78-8-002
**Title:** Axiomatic Specification of Syntax-Directed Translation

**Authors:** Sharon Sickel and W.M. McKeeman

**Performing Organization:**
Information Sciences
University of California
Santa Cruz, California 95064

**Controlling Office:**
Office of Naval Research
Arlington, Virginia 22217

**Abstract:**
Predicat logic is applied to the specification of syntax-directed translation. Context-free grammars are shown to be representable by logic programs, and a translation from grammars to logic programs is presented as a logic program. Coupled grammars are introduced, shown to be interpretable as translators, shown to be representable as logic programs, and translation from coupled grammars to logic programs is presented via logic programs. Mixed grammars, a concisely representable special case of coupled grammars, are given in terms of coupled grammars.
AXIOMATIC SPECIFICATION
OF
SYNTAX-DIRECTED TRANSLATION

by

Sharon Sickel
W.M. McKeeman

Information Sciences and Crown College
University of California
Santa Cruz, Ca. 95064

This work was supported by Office of Naval Research Contract # 76-C-0682
ABSTRACT

Predicate logic is applied to the specification of syntax-directed translation. Context-free grammars are shown to be representable by logic programs, and a translation from grammars to logic programs is presented as a logic program. Coupled grammars are introduced, shown to be interpretable as translators, shown to be representable as logic programs, and translation from coupled grammars to logic programs is presented via logic programs. Mixed grammars, a concisely representable special case of coupled grammars, are given in terms of coupled grammars.

1. Introduction

We define grammars to be coupled if they are context-free and there is a 1-1 correspondence between their productions. Two different interpretations are given for them. The first is a generalization of the syntax-directed transductions of Lewis and Stearns [2]. Their productions take the form

\[ A \rightarrow \alpha, \beta \]

where the parts of the form

\[ A \rightarrow \alpha \]

constitute the underlying input grammar, and the string \( \beta \) specifies the output. The nonterminals of each \( \beta \) must be a permutation of those in the corresponding \( \alpha \). As \( A \rightarrow \alpha \) is applied during the parse, the value of \( A \) is given by \( \beta \) where each nonterminal is replaced by its own value. Our paired productions take the form

\[ A \rightarrow \alpha \quad \text{and} \quad A \rightarrow \beta \]

but we relax the requirement of a 1-1 correspondence between nonterminals in \( \alpha \) and \( \beta \). A missing nonterminal in \( \beta \) is ignored, its value being discarded. An extra nonterminal in \( \beta \) is kept literally, standing in the output for any string derivable from it by the productions in the second grammar.

The second interpretation of coupled grammars links input to output via the pars
sequence. Aho and Ullman [1, Vol. II] present a similar scheme based on the production form

\[ A \rightarrow \alpha, \beta \]

where again the nonterminals in \( \alpha \) and \( \beta \) must correspond. Our productions take the form

\[ A \rightarrow \alpha \quad \text{and} \quad B \rightarrow \beta \]

where there are no restrictions on the form or correspondence of symbols within the productions.

The logic program corresponding to a grammar provides a static statement of the meaning of the grammar; that is, an axiomatic specification of it. Each predicate in the logic program specifies a relation among its arguments.

Mixed grammars are a concise notation for the simple syntax-directed translation schemata presented by Aho and Ullman [1, Vol. I]. There is a nonterminal vocabulary \( V_N \) and two terminal vocabularies, \( V_I \) for the input and \( V_O \) for output. Mixed grammars are a directly executable notation on a simple pushdown stack machine. They have a concise representation in logic. They are, furthermore, self-describing, and self-translating.

2. Logic Representations of Grammars

A context-free grammar \( G \) is a four-tuple \([V_N, V_T, P, S]\) as usual. \( P \) is a set of productions of the form

\[ A \rightarrow \alpha \]

where we have used the symbol \( \rightarrow \) instead of \( \rightarrow \) to avoid confusion with \( \rightarrow \) for implication.

\[ A \in V_N \quad \text{and} \quad \alpha \in (V_N \cup V_T)^*. \]

The empty string is denoted by \( \epsilon \). We assume a function, \( \text{start}(G) = S \). We further
follow the conventions that nonterminals are denoted by capital letters and terminals by any other symbols except Greek letters which we sometimes use to denote strings. Because of these conventions and the existence of the function, start, we can completely specify a grammar as a sequence of productions, and we frequently do.

A logic program [4] is a set of WF s in the form

\[ L \rightarrow R \]

and one WF, the call, of the form

\[ \vdash R \]

where \( L \) is a predicate and \( R \) is a conjunction of predicates. All variables are implicitly universally quantified.

Table 2.1 is an example of a context-free grammar and a corresponding logic program. Terminal symbols in the grammar are delimited by double quotes, and the parameters to the predicates are strings of symbols represented by the juxtaposition of variable names and constants delimited by double quotes. We may drop the quotes when the meaning is obvious.

\[
\begin{align*}
G &= RG \\
G &= R \\
R &= L"="F \\
F &= PF \\
E &= P \\
P &= """"U"""" \\
P &= L \\
P &= U \\
L &= "G" \\
L &= "R" \\
L &= "F" \\
L &= "P" \\
L &= "L" \\
L &= "U" \\
U &= L \\
U &= "=" \\
U &= """"
\end{align*}
\]

Table 2.1: Grammar GG and its logic program representation LPGG.
The interpretation of the grammar $GG$ is obvious [3]. It is in fact self-describing. The correspondence between each production of grammar and Horn clause is straightforward. For example,

$$G(x_1 x_2) \rightarrow R(x_1) \land G(x_2)$$

states that if $R$ is true on string $x_1$ and $G$ is true on string $x_2$, then $G$ is necessarily true on the concatenation of $x_1$ and $x_2$, which is also the meaning of

$$G = RG$$

in the grammar.

$$L("G") \rightarrow$$

states that $L$ is always true for the letter $G$, and so on.$^\S$

3. Translation from Context-free Grammars to Their Logic Interpretations

Predicate $TGL$ formally defines the relationship between context-free grammars and their interpreting logic programs, and also the translation process between the two, i.e., $TGL(G, LP)$ is true iff $LP$ is the logic program representation of grammar $G$. In the example of Table 2.1, $TGL(GG, LPGG)$.

$$TGL(e, c) \rightarrow$$

$$TGL((A = \beta) \cdot G, \text{New_rule} \cdot LP) \rightarrow$$

$$TGL(G, LP) \land TPL(\beta, 1, \text{Ipam}, \text{Conj})$$

$$\land \text{New\_Rule} = (A "(" \text{Ipam } ") " \text{Conj})^\dagger$$

$^\S$ In examples we occasionally optimize logical formulas to remove the value true where it is formally present but contributes no meaning. In particular $A \land \text{true} \rightarrow A$ and $(A \leftarrow \text{true}) \rightarrow (A \leftarrow)$.

$^\dagger$ The symbol "." appears frequently in parameters of logic programs. $u \cdot v$ denotes a parameter that is subdivided into components $u$ and $v$. For strings we denote such a decomposition simply by juxtaposition of symbols, e.g. $uv$. The use of "." denotes decomposition of more complicated structures, such as grammars. The first use here, $(A = \beta) \cdot G$, denotes a grammar that is subdivided into its first production $A = \beta$ and the remaining productions, grammar $G$. 
Given \( p \), the r.h.s. of a production, \( \text{TPL}(\beta, I, \text{Ip}, \text{Conj}) \) yields both a parameter list, \( \text{Ip} \), and a conjunction of predicates, \( \text{Conj} \). More specifically

\[
\text{TPL}(\beta, I, \text{Ip}, \text{Conj}) \text{ is true iff } \beta = \beta_0 B_{n+1} ... B_k \beta_{k+1} \in (V_N \cup V_T)^* \text{ where each } \beta_i \in V_T^* \text{ and } B_i \in V_N \text{ and } \text{Ip} = \beta_0 x_n \beta_{n+1} ... x_k \beta_{k+1}, \text{ and } \text{Conj} = B_n(x_n) \land B_{n+1}(x_{n+1}) \land ... B_k(x_k).
\]

\[
\text{TPL}(\varepsilon, I, \varepsilon, \text{true}) \leftarrow
\]

\[
\text{TPL}(s, \beta, I, \text{Conj}) \leftarrow T(s)
\]

\[\land \text{TPL}(\beta, I, \text{Conj})\]

\[
\text{TPL}(S, \beta, X(n), \text{Ip}, S \ "X(n) " \land " \text{Conj} \) \leftarrow N(S)
\]

\[\land \text{TPL}(\beta, n+1, \text{Ip}, \text{Conj})\]

\( T(s) \) is true iff \( s \) is a terminal symbol.

\( N(S) \) is true iff \( S \) is a nonterminal symbol.

\( X(n) \) is the letter \( x \) subscripted with the value of \( n \), and similarly for \( Y(n) \) which will be used later.

4. Functional Interpretation of Coupled Grammars

Coupled grammars have an input grammar and an output grammar. As each production of the input grammar is used, an output is associated with it as specified by the corresponding production of the output grammar. This relation can be specified by a logic program as shown in Table 4.1 for coupled grammars B and U translating binary to unary notation.

| A = A 1 | A = AA 1 | A(x_1, y_1, y_1) \leftarrow A(x_1, y_1) |
| A = A 0 | A = AA | A(x_1, 0, y_1, y_1) \leftarrow A(x_1, y_1) |
| A = 1  | A = 1 | A(1, 1) \leftarrow |
| A = 0  | A = \varepsilon | A(0, \varepsilon) \leftarrow |

grammar B grammar U logic program LPBU

Figure 4.1: Coupled grammars B and U and their logic program representation LPBU.
It is obvious that the input grammar describes all binary strings, and the output grammar (ambiguously) describes all unary strings. The translation doubles the value of the output string for each shift in positional notation and adds a further "1" if the number is odd.

The logic program predicates have two parameters, one for input and one for output. For example, the interpretation of the clause

\[ A(x_1, y_1) = A(x_1, y_1) \]

says that if \( x_1 \) is an acceptable string producing output \( y_1 \), then \( x_1 \) is an acceptable string producing \( y_1 \) as output.

5. Translation of Coupled Grammars to Their Functional Interpretation

Given coupled grammars \( G_1 \) and \( G_2 \), one can construct a program that will perform a functional interpretation on a string, using \( G_1 \) as the input grammar and \( G_2 \) as the output grammar. \( \text{FI}(G_1, G_2, LP) \) is true iff \( LP \), a logic program, performs that interpretation. For example, from Figure 4.1, \( \text{FI}(B, U, LPBU) \).

\[
\begin{align*}
\text{FI}(e, e, e) & \leftarrow \\
\text{FI}((A = \beta) \cdot G_1, (A = \gamma) \cdot G_2, z \cdot LP) & \leftarrow \\
\text{FI}(G_1, G_2, LP) & \\
& \land \text{FIP}(A, \beta, \gamma, z)
\end{align*}
\]

\( \text{FIP}(A, \beta, \gamma, C) \) means that the functional interpretation of coupled productions \( A = \beta \) and \( A = \gamma \) is the logic procedure \( C \) where \( A \) is a single nonterminal and \( \beta \) and \( \gamma \) are strings in \( V^* \). In the program below, \( \text{Iparam} \) and \( \text{Oparam} \) are each strings of variables and terminals and represent the input parameter and output parameter, respectively. \( R \) is a conjunction of predicates.
FIP(A,β,γ,A "(" Ipam,"" Oparam ") + " Conj) +
  IN(β,1,Ipam,Conj)
  \land OUT(Conj,γ,Oparam)

IN(β,n,Ipam,Conj) serves a similar function to TPL(β,n,Ipam,Conj) except that we
are producing two parameters instead of one to the predicates of
Conj, i.e. whenever P(x₁) appears in Conj of TPL, then P(x₁,y₁)
appears in Conj of IN.

IN(ε,n,ε,true) +
IN(s β,n,s Ipam,Conj) + T(s)
  \land IN(β,n,Ipam,Conj)
IN(S β,n,X(n) Ipam,S "(" X(n) ")", " Y(n) ") + " Conj + N(S)
  \land IN(β,n+1,Ipam,Conj)

OUT(Conj,γ,Oparam): Oparam is the output parameter and if γ = γ₁γ₂...γₙ then Oparam
denotes
s₁s₂...sₙ where
sᵢ = γᵢ if t(γᵢ)
  = yⱼ if N(γᵢ) and j is the least integer such that
  γᵢ(xⱼ,yⱼ) is a literal in Conj
  = sᵢ if N(γᵢ) and γᵢ(xⱼ,yⱼ) is not in Conj

OUT(Conj,ε,ε) +
OUT(Conj,s γ,s Oparam) + T(s)
  \land OUT(Conj,γ,Oparam)
OUT(Conj,S γ,Y Oparam) + N(S)
  \land FIND(S,Conj,Y)
  \land OUT(Conj,γ,Oparam)
FIND(S, Conj, z): For nonterminal S, and conjunct Conj,
z = y for leftmost predicate in Conj that is of the form S(x, y)
= z if no such predicate exists in Conj

FIND(S, true, S) →
FIND(S, S "(" x "," y ") " Conj, y) →
FIND(S, R "(" x "," y ") " Conj, z) → (S#R)
∧ FIND(S, Conj, z)

For example, if A, B, C and D are nonterminals and # and ! are terminals, then
the following are true:

FIP(A, #BCD, CB!, A(#x_1 x_2 x_3, y_2 y_1 !) → B(x_1, y_1) ∧ C(x_2, y_2) ∧ D(x_3, y_3))
IN(#BCD, 1, #x_1 x_2 x_3, B(x_1, y_1) ∧ C(x_2, y_2) ∧ D(x_3, y_3))
OUT((B(x_1, y_1) ∧ C(x_2, y_2) ∧ D(x_3, y_3), CB!, y_2 y_1 !)

6. Parse Sequence Interpretation of Coupled Grammars

Given a grammar G, with each string in L(G) is associated one (or more) parse
sequences. A parse sequence is a sequence of integers corresponding to the production
numbers as they are applied in a left-to-right parse.

Suppose we have two arbitrary coupled grammars, and each is used to parse a string
in its language. The strings are defined to be equivalent if they have a parse sequence
in common, as shown in Figure 6.1.

Figure 6.1: Parse sequence interpretation with the parse sequences not used
intermediately.
Alternatively, take the same two coupled grammars, where the input grammar is used to generate a parse sequence and then that parse sequence is used with the output grammar to generate output as shown in Figure 6.2. We have then defined a means of translation.

\[
\begin{align*}
S & \downarrow \\
\text{G}_1 & \\
\downarrow & \\
\text{PS} & \\
\downarrow & \\
\text{G}_2 & \\
\downarrow & \\
t & 
\end{align*}
\]

**Figure 6.2:** Parse sequence interpretation with the parse sequence used as an intermediate form.

\[
\begin{align*}
E &= E + T & P &= P P + \\
E &= T & P &= P \\
T &= T \ast D & P &= P P \ast \\
T &= D & P &= P \\
D &= (E) & P &= P \\
D &= 0 & P &= 0 \\
D &= 1 & P &= 1 \\
D &= 2 & P &= 2 \\
D &= 3 & P &= 3 \\
\vdots & \vdots & \vdots & \\
D &= 9 & P &= 9 \\
\_ & \_ & \_ & \\
\text{input} & \text{output} & \text{grammar} & \text{grammar}
\end{align*}
\]

**Table 6.1:** Parse sequence coupled grammars.
The coupled grammars in Table 6.1 relate infix expressions and Polish expressions. The input grammar is unambiguous; every parse sequence from it is a parse sequence for the output grammar. Thus, there is a Polish form for every infix expression. Because of rules $P = P$ in the output grammar, it is ambiguous. For every Polish string there are infinitely many parses which are also parses for the input grammar. Each defines a correct translation. There are also parses from the output grammar which are meaningless relative to the input grammar.

7. Translation of Coupled Grammars to Their Parse Sequence Interpretation

$\text{PSI}(G_1,G_2,s,t)$ is true iff grammar $G_1$ parsing string $s$ and grammar $G_2$ parsing string $t$ give the same parse sequence. We give two different definitions for $\text{PSI}$. The first generates an explicit parse sequence from an input string and its relevant grammar, then uses that sequence to generate a corresponding string in the language of the other grammar.

$$\text{PSI}(G_1,G_2,s,t) \leftarrow$$
$$\text{PARSE}(G_1,s,\text{PS})$$
$$\land \text{GENERATE}(G_2,\text{PS},t)$$

$\text{PARSE}(G_1,s,\text{PS})$ is true iff $\text{PS}$ is a parse sequence of string $s$ in grammar $G_1$. The parse is accomplished by creating an associated grammar $G_S$, then using logic program $\text{FI}$ to translate $G_1$ and $G_S$ into a logic program that carries out a functional interpretation between the grammars, and, finally, to execute that logic program with string $s$ as input to produce parse sequence $\text{PS}$.

$$\text{PARSE}(G_1,s,\text{PS}) \leftarrow$$
$$\text{SEQ}(G_1,1,G_S)$$
$$\land \text{FI}(G_1,G_S,\text{LP})$$
$$\land \text{EXEC}('"-" \text{Start}(G_1) "(" s "," \text{PS }")" )$$
$$\cdot \text{LP})$$
SEQ(G₁,n,GS) is true iff G₁ and GS are context-free grammars, n is a positive integer, and by considering the productions of G₁ sequenced starting at n, GS consists of corresponding productions in which each r.h.s. is the non-terminals, in order, of its corresponding production in G₁, followed by the sequence number of that production.

Formally, SEQ is defined:

\[
\begin{align*}
\text{SEQ}(\epsilon,n,\epsilon) & \leftarrow \\
\text{SEQ}((A = \alpha) \cdot G, n, (A = \beta n) \cdot GS) & \leftarrow \\
& \text{SEQ}(G, n + 1, GS) \\
& \land \text{STRIP}(\alpha, \beta)
\end{align*}
\]

STRIP(α, β) is true iff α is a string in V*, and β is α with terminals stripped away.

\[
\begin{align*}
\text{STRIP}(\epsilon, \epsilon) & \leftarrow \\
\text{STRIP}(s_{\alpha, \beta}) & \leftarrow T(s) \\
& \land \text{STRIP}(\alpha, \beta) \\
\text{STRIP}(S_{\alpha, S_{\beta}}) & \leftarrow N(S) \\
& \land \text{STRIP}(\alpha, \beta)
\end{align*}
\]

EXEC(P) is a meta-procedure that executes logic program P. For an example of PARSE, consider

\[
\begin{align*}
G₁: \\
E & = E + T \\
E & = T \\
T & = T * a \\
T & = a
\end{align*}
\]

s:

\[
\begin{align*}
a * a + a * a
\end{align*}
\]

the calls and computed values are:
SEQ(G_1,1,GS) in which GS becomes:

\[ E = ET1 \]
\[ E = T2 \]
\[ T = T3 \]
\[ T = 4 \]

FI(G_1,GS,LP) in which LP becomes:

\[ E(x_1, x_2, y_1, y_2) \leftarrow E(x_1, y_1) \land T(x_2, y_2) \]
\[ E(x_1, y_2) \leftarrow T(x_1, y_1) \]
\[ T(x_1, y_1, y_2) \leftarrow T(x_1, y_1) \]
\[ T(a, 4) \leftarrow \]

\text{EXEC}(\text{E}(\text{a}*a+a*a", PS) \cdot \text{LP}) \text{ in which PS becomes:}

432431

Now this completes the first half of the definition of PSI. Given the parse sequence constructed in the above process we can use it to drive a right-most derivation in G_2, to create string t.

We now define the predicate GENERATE. We know intuitively that parsing and generation of strings are inverse operations. That would say that we could define GENERATE in terms of PARSE, thus:

\[ \text{GENERATE}(G_2, PS, t) \leftarrow \text{PARSE}(G_2, t, PS) \]

The way we have used PARSE (and think of parsing) is that the string in the language is given and we generate the parse sequence as a side-effect of the recognition process. Suppose the parse sequence and the grammar are given. Can we use PARSE to create the input? A useful property of logic programs is that they describe truth about relationships, and while they can drive computations, the direction of the computation is usually arbitrary. Let's follow the computation to see if the string t can be appropriately computed.
Continuing the example above, let $G_2$ be:

\[
\begin{align*}
P &= P \ P + \\
P &= P \\
&= P \ P \ * \\
P &= a \\
\end{align*}
\]

Then $\text{PARSE}(G_2,t,432431)$ calls $\text{SEQ}(G_2,1,GS')$ which produces $GS'$:

\[
\begin{align*}
P &= P \ P \ 1 \\
P &= P \ 2 \\
P &= P \ P \ 3 \\
P &= 4 \\
\end{align*}
\]

Then $\text{FI}(G_2,GS',LP')$ computes $LP'$:

\[
\begin{align*}
P(x_1x_2 +,y_1y_2 \ 1) &\leftarrow P(x_1,y_1) \land P(x_2,y_2) \\
P(x_1,y_1 \ 2) &\leftarrow P(x_1,y_1) \\
P(x_1x_2 *,y_1y_2 \ 3) &\leftarrow P(x_1,y_1) \land P(x_2,y_2) \\
P(a,4) &\leftarrow \\
\end{align*}
\]

Executing $LP'$ with call $P(t,432431)$, we compute $t = aa*aa*$, the desired answer.

Therefore, we could have defined $\text{PSI}$ as:

\[
\begin{align*}
\text{PSI}(G_1,G_2,s,t) &\leftarrow \\
&\text{PARSE}(G_1,s,PS) \land \text{PARSE}(G_2,t,PS) \\
\end{align*}
\]

And, we see that our original claim of the equivalence of the two definitions is reflected in their having a single formal specification.

8. Mixed Grammars

Suppose we have a set of nonterminals $V_N$ and two disjoint sets of terminals
$V_1$ and $V_0$, and a context-free grammar

$$[V_N, V_I \cup V_O, P, S].$$

The productions in $P$ are of the form

$$A = a$$

where

$$a \in (V_N \cup V_I \cup V_O)^*.$$  

They are equivalent to the coupled grammars

$$[V_N, V_I, P_I, S] \quad \text{and} \quad [V_N, V_O, P_O, S]$$

where all elements of $V_O$ are deleted from the productions of $P$ to give $P_I$ and vice-versa for $P_O$.

The grammars can be interpreted either as functionally coupled or parse sequence coupled. They are equivalent to the simple syntax-directed translation schemata of Aho and Ullman [1, Vol. I].

Notationally speaking, it is convenient for $V_I \cap V_O$ to be nonempty, thus we establish the convention of double quotes delimiting the members of $V_I$ (as in earlier sections of this paper) and single quotes delimiting the members of $V_O$.

$$E = E \ "+" T \ '+'$$

$$E = T$$

$$T = T \ "*" \ "a" \ 'a' \ '*'$$

$$T = "a" \ 'a'$$

**Table 8.1:** A mixed grammar describing the translation from infix to Polish.

For example, the mixed grammar in Table 8.1 has the same effect as the coupled grammars in the preceding examples. It will also accept other strings but its behavior is then of no interest.
The advantages of mixed grammars are that they are directly executable on a simple pushdown store machine, and that their notation makes implicit, unavoidable and natural the constraints for simple syntax-directed translation schemata. One can produce a logic program similar to PSI and FI to translate mixed grammars to logic.

9. Conclusions

We have established some relationships among context-free grammars, translation schemata, and logic. The interpretation of paired grammars has been extended in several ways. We have defined a set of translation programs that are actually sets of logic theorems. They are concise, and the correctness of each program can be established by proving each theorem individually. The process of parsing and the process of generation which are inverses are shown to have the same formal specification.
REFERENCES


OFFICIAL DISTRIBUTION LIST
Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies