A Formalism for Describing and Evaluating Visibility Control Mechanisms

Alexander L. Wolf
Lori A. Clarke
Jack C. Wileden

COINS Technical Report 83-34
October 1983

Computer and Information Science Department
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work supported in part by the National Aeronautics and Space Administration under NASA grant NAG1-115

ABSTRACT

A number of mechanisms have been designed for controlling entity visibility. As with most language concepts in computer science, visibility control mechanisms have been developed in an essentially *ad hoc* fashion with no clear indication given by their designers as to how one proposed mechanism relates to another. This paper introduces a formalism for describing and evaluating visibility control mechanisms. The formalism reflects a generalized view of visibility in which the concepts of *accessibility* and *provision* are distinguished. This formalism provides a means for characterizing and reasoning about the various properties of visibility control mechanisms. In particular, the notion of *preciseness* is defined. As an example, the formalism is used to evaluate and compare the relative strengths and weaknesses of the visibility control mechanisms found in ALGOL60, Ada, Gypsy, and an approach we have developed that specifically addresses the concerns of interface control in large software systems.

1. Introduction

For over twenty years, nesting has been the predominant visibility control mechanism found in modern programming languages. It has been informally argued elsewhere that nesting is not sufficient to describe the wide range of possible visibility relationships among the entities comprising a software system (see, for example, [CLAR80]). A variety of recently designed languages, such as Ada® [DOD83], Alphard [SHAW81], CLU [LISK79], Euclid [POPE77], Gypsy [AMBL77], Mesa [MITC79], Modula [WIRT77], and Protel [CASH81], have attempted to compensate for the inadequacies of nesting by offering alternative mechanisms for visibility control. As with most language concepts in computer science, visibility control mechanisms have been developed in an essentially *ad hoc* fashion with no clear indication given by their designers as to how one proposed mechanism relates to another.

This paper introduces a formalism for describing and evaluating visibility control mechanisms. The formalism reflects a generalized view of visibility in which the concepts of *accessibility* and *provision* are distinguished. This formalism provides a means for characterizing and reasoning about the various properties of visibility control mechanisms. In particular, the notion of *preciseness* is defined. The next section presents the basic features of the formalism. The use of the formalism for describing visibility control mechanisms is discussed in Section 3. Section 4 discusses use of the formalism in evaluating such mechanisms. Theorems are presented that serve to characterize the relative strengths and weaknesses of the visibility control mechanisms found in ALGOL60, Ada, Gypsy, and an approach we have developed that specifically addresses the concerns of interface control in large software systems [CLAR83].

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

2. The Formalism

Traditionally, the concept of *entity visibility* has been defined in terms of *declaration*, *scope*, and *binding*. In many programming languages, an entity is a language element that is given a name. Thus, entities include such things as data objects, types, statements (labels), or subprograms. A declaration introduces an entity and associates an identifier (name) with that entity. The scope of a declaration is the region of program text over which that declaration is visible and has effect. Many languages allow a single identifier to be associated with more than one declaration and the scopes of those declarations to overlap. Binding relates the use of an identifier, at a given point in a program, to a particular declaration. A description of a visibility control mechanism, then, is essentially a description of how that mechanism controls scope.

The formalism presented here is based on the concepts of *accessibility* and *provision*, which are two different, yet complementary, points of view on visibility. Accessibility is concerned with the entities that can be (potentially) accessed by some entity. For example, the accessible entities for a subprogram typically include the subprogram itself and any locally declared entities, as well as any non-local entities imported (implicitly or explicitly) into that subprogram. Provision, on the other hand, is concerned with making entities available for access by other entities in a software system. Again for a subprogram, the provided entities typically include the subprogram itself and any locally declared entities exported (implicitly or explicitly) from that subprogram.

This distinction between accessibility and provision reflects the differences in the overall approaches that language designers have taken to controlling entity visibility. In languages such as ALGOL60 and Pascal, accessibility and provision are essentially mirror images; those entities accessible from an entity are always also provided to that entity and *vice versa*. In the designs of more recent languages, particularly languages intended for the construction of large and complex software systems, the desire for greater control over entity visibility has resulted in mechanisms that address accessibility and provision in separate, and often unequal, ways. Our formalism, by drawing out this distinction, is equipped to expose these differences in visibility control mechanisms.

The formalism centers on the construction and manipulation of a unique representation of entity visibility relationships. This representation takes the form of a graph called the *visibility graph.*

Defn. A *visibility graph* vg = (N, A_a, A_p) is a labeled digraph where

N is a finite set of nodes labeled by the (unique) names of entities;

A_a is a finite set of ordered pairs of nodes (n_i, n_j) denoting the accessibility relationship n_i "can access" n_j;

A_p is a finite set of ordered pairs of nodes (n_i, n_j) denoting the provision relationship n_i "is provided to" n_j.

Any pair of nodes in a visibility graph may be connected by multiple arcs resulting from the accessibility and provision relationships. A visibility graph may also contain loops (arcs whose origin and terminus are the same node) and cycles, both resulting from recursive visibility.

A consistent set of entity visibility relationships exists when the entities that each entity can access have in fact been provided. In terms of visibility graphs, this corresponds to the following property:

Defn. A visibility graph vg = (N, A_a, A_p) is *well-formed* iff for all $(n_i, n_j) \in A_a$, $(n_j, n_i) \in A_p$.

In the remainder of this paper, we only consider the set VG of well-formed visibility graphs.

A visibility graph uniquely represents a particular set of visibility relationships among a set of entities since the visibility relationships of any entity e are defined by the arcs from the corresponding node n_e to that node's nearest neighbors in the graph (i.e., adjacent nodes), and the absence of an arc between two nodes indicates that no visibility relationship exists between the corresponding entities. Notice that the arcs could be tagged in order to indicate a special kind of accessibility or provision relationship. For instance, if a data object is provided "read-only", as is possible in a number of languages, then this fact can be recorded as a "read-only" tag on the appropriate arc.

To consider accessibility and provision separately, we refer to two spanning subgraphs of a visibility graph. One represents only the accessibility relationships of the entities in the visibility graph, while the other represents only the provision relationships.

Defn. An *accessibility graph* ag = $\{N, A_a, A_p\}$ is a visibility graph where $A_p = \emptyset$.

Defn. A *provision graph* pg = $\{N, A_a, A_p\}$ is a visibility graph where $A_a = \emptyset$.

Two functions on visibility graphs are defined that produce these subgraphs.

Defn. The *accessibility extraction function* ae: VG $\rightarrow$ VG, is a mapping from a visibility graph vg = $\{N, A_a, A_p\}$ to an accessibility graph ag = $\{N, A_a, \emptyset\}$.

Defn. The *provision extraction function* pe: VG $\rightarrow$ VG, is a mapping from a visibility graph vg = $\{N, A_a, A_p\}$ to a provision graph pg = $\{N, \emptyset, A_p\}$.

Using these functions, two useful relationships between visibility graphs can be defined.

Defn. A visibility graph vg_i *access-satisfies* a visibility graph vg_j iff $ae(vg_j) \subseteq ae(vg_i)$.

Defn. A visibility graph vg_i *provide-satisfies* a visibility graph vg_j iff $pe(vg_j) \subseteq pe(vg_i)$.

Informally stated, the desire for a set of entities s_j to be accessible from (provided to) some entity e is satisfied by e having access (being provided) to any set of entities s_i of which s_j is a subset.

A visibility graph can be derived from some *representation* of a program by applying the rules of a particular visibility control mechanism, or combination of mechanisms, to the entities in the representation. More formally, we denote the set of program representations by PR and define a function that performs this mapping as follows:

Defn. A *visibility function* vf: PR $\rightarrow$ VG, is a mapping from a program representation pr to a visibility graph vg.

A set of visibility functions VF = $\{vf_a, vf_b, ...\}$ can be defined where vf_m is the visibility function implementing the visibility control mechanism(s) m.

3. Describing Visibility Control Mechanisms

One of our goals is to provide an effective means of describing visibility control mechanisms so that one can reason about and evaluate these mechanisms. Existing informal and formal descriptive methods have proven inadequate. The Pascal Report [JENS74], for example, causes many problems due to the ambiguity of its prose description of visibility control [WELS77, BRIN81]. The few formal approaches to describing visibility control mechanisms are operational in nature. Thus, they appear to be more suitable for the implementor and verifier than for the language designer or programmer. Such formal descriptions of visibility control mechanisms have appeared primarily in operational and denotational semantic specifications, where a mechanism is typically described by the manipulation of an (identifier) environment component. The technique of employing an environment component in a formal description is unsatisfactory because the method for describing manipulation of that environment component is essentially algorithmic (despite the use of a "functional" notation; see, for example, [FDA80]). Moreover, the information in the environment component only describes entity accessibility. Employing such a description makes it difficult to understand the ramifications of using a mechanism. With nesting, for example, a subprogram's so-called "local" entities are unavoidably made visible to other subprograms nested within that subprogram, but this fact is only implicitly stated in existing formal descriptions of nesting.

In the formalism presented here, a visibility control mechanism m is described by its corresponding visibility function vf_m. Each such function has two components that *explicitly* address the accessibility and provision aspects of entity visibility. The *accessibility function* af describes accessibility by mapping a program representation to an accessibility graph while the *provision function* pf describes provision by mapping a program representation to a provision graph. Thus,

$$vf_m(pr) = af_m(pr) \cup pf_m(pr)$$

where pr is some program representation and the union of two visibility graphs vg_1 and vg_2 is the visibility graph $\{N_1 \cup N_2, A_{a,1} \cup A_{a,2}, A_{p,1} \cup A_{p,2}\}$. Component functions af and pf can be further broken down into functions operating on individual *kinds* of entities, such as subprograms and data objects, as follows:

$$af_m(pr) = af_{m,ek_1}(pr) \cup \cdots \cup af_{m,ek_n}(pr)$$
$$pf_m(pr) = pf_{m,ek_1}(pr) \cup \cdots \cup pf_{m,ek_n}(pr)$$

where ek_i denotes the entity kind upon which the accessibility function operates. Hence, for each entity kind that is of interest, there is a function that describes accessibility and a function that describes provision. Accessibility functions are similar in nature to the "binding" functions of [HENN81]. Provision functions, however, appear to have no counterpart in previous formalisms.

The full description of a visibility control mechanism is of course quite large and therefore presenting such a description in this paper is inappropriate. To illustrate the use of the descriptive method, we instead give descriptions of the accessibility and provision of *subprograms* as they are controlled by the nesting mechanism of ALGOL60. This entails the definition of the accessibility function $af_{ALGOL60,subprograms}$ and the provision function $pf_{ALGOL60,subprograms}$. The discussion is further simplified by only considering the accessibility of subprograms to subprograms.

Accessibility and provision functions, as mentioned above, operate on a representation of a program. One suitable representation for ALGOL60 is a graph we call the *nesting graph*.

Defn. A *nesting graph* ng $= \{N_{ng}, A_{ng}\}$ is a labeled digraph where

N_{ng} is a finite set of nodes labeled by the (unique) names of entities;

A_{ng} is a finite set of ordered pairs of nodes (n_i, n_j) denoting the relationship n_i "child of" n_j which means n_i is directly nested in n_j.

Notice that for this example N_{ng} consists only of nodes whose entities are subprograms. In the subsequent definition of the accessibility and provision functions, we make use of three auxiliary functions, each of which maps N_{ng} to the powerset of N_{ng} as follows:

(1) $\text{Parent}(n_i) = \{n_j \in N_{ng} \mid (n_i, n_j) \in A_{ng}\}$;

(2) $\text{Siblings}(n_i) = \{n_j \in N_{ng} \mid$ there exists $n_k \in N_{ng}$ such that $(n_i, n_k) \in A_{ng}$ and $(n_j, n_k) \in A_{ng}$ and $i \neq j\}$;

(3) $\text{Ancestors}(n_i) = \{n_j \in N_{ng} \mid n_j = \text{Parent}(n_i)$ or $n_j \in \text{Siblings}(\text{Parent}(n_i))$ or $n_j \in \text{Ancestors}(\text{Ancestors}(n_i))\}$.

For any $n_i \in N_{ng}$, Parent(n_i) will always be a singleton since an entity can be directly nested in only one other entity.

The accessibility function is now defined to transform ng, a graph representation of the nested structure of subprograms, into an accessibility graph, which explicitly describes the effect of ALGOL60's nesting mechanism on the accessibility of subprograms.

Defn. $af_{ALGOL60,subprograms}(ng) = \{N, A_a, A_p\}$ where

$N = N_{ng}$;

$A_a = \{(n_i, n_j) \mid i = j \text{ or } n_i = \text{Parent}(n_j) \text{ or } n_j \in \text{Siblings}(n_i)$
$\text{ or } n_j \in \text{Ancestors}(n_i)\}$;

$A_p = \varnothing$.

From this description it can be easily seen that 1) all subprograms are accessible from themselves, 2) all subprograms are accessible from the subprogram in which they are directly nested, 3) all subprograms are accessible from those subprograms with the same parent, and 4) all subprograms are accessible from those subprograms nested within them as well as accessible from those subprograms nested within their siblings.

For ALGOL60, the provision function is quite similar to the accessibility function.

Defn. $pf_{ALGOL60,subprograms}(ng) = \{N, A_a, A_p\}$ where

$N = N_{ng}$;

$A_a = \varnothing$;

$A_p = \{(n_i, n_j) \mid i = j \text{ or } n_j = \text{Parent}(n_i) \text{ or } n_i \in \text{Siblings}(n_j)$
$\text{ or } n_i \in \text{Ancestors}(n_j)\}$.

These descriptions reveal the fact that in ALGOL60 accessibility and provision are essentially mirror-image counterparts. In particular, the expression defining the set of tuples (n_i, n_j) in A_p of the provision function is the same as the expression defining the set of tuples (n_i, n_j) in A_a of the accessibility function, except that the i's and j's are reversed. This similarity, however, is certainly not true of all mechanisms.

We contend that accessibility and provision functions are easier to comprehend than algorithms that manipulate a dynamic environment component. For instance, the problem mentioned at the beginning of this section, concerning nesting's effect on the visibility of a subprogram's local entities, is clearly exposed using this formalism. By simply looking at the accessibility and provision functions for subprograms it is immediately evident, for example, that a subprogram's supposedly "local" child subprogram is in fact visible to any other subprograms nested within that subprogram.

4. Evaluating Visibility Control Mechanisms

Visibility control mechanisms can be characterized in a number of ways and these characterizations can then provide a basis for evaluating the strengths and weaknesses of different mechanisms. This section presents one such characterization which is possible within the framework of our formalism. Specifically, the notion of *preciseness* is defined for a visibility control mechanism in terms of the mechanism's accuracy in capturing desired accessibility and provision relationships.

It can easily be argued that the visibility control mechanisms in a language L should be such that

for all $vg \in VG$, there exists $pr \in PR$ such that $vf_L(pr)$ access-satisfies and provide-satisfies vg.

In other words, it should be possible to find a program representation that realizes any accessibility and provision relationships that a programmer might wish to devise, although additional accessibility and provision may be allowed as well. It is not surprising that the visibility control mechanisms of all modern languages that we have examined satisfy this minimum requirement.[1] Stronger properties for evaluating mechanisms are needed, however, which leads to the following definitions.

Defn. A visibility control mechanism m is *access-precise* iff for all vg $\in$ VG, there exists a pr $\in$ PR such that $ae(vf_m(pr)) = ae(vg)$.

Defn. A visibility control mechanism m is *provide-precise* iff for all vg $\in$ VG, there exists a pr $\in$ PR such that $pe(vf_m(pr)) = pe(vg)$.

Defn. A visibility control mechanism m is *precise* iff it is both access-precise and provide-precise.

Intuitively, the definitions state that if for each possible visibility graph, a program representation can be found with the property that the visibility relationships among its entities are exactly those specified in the visibility graph, then the mechanism is indeed precise. A mechanism is less than precise if only the accessibility relationships or provision relationships can be exactly realized. This suggests the following hierarchy of visibility control mechanisms based on preciseness:

$$\begin{tikzcd}
& \text{PRECISE MECHANISMS} \arrow[dl, dash, "/"'] \arrow[dr, dash, "\backslash"] & \\
\text{ACCESS-PRECISE MECHANISMS} \arrow[dr, dash, "\backslash"'] & & \text{PROVIDE-PRECISE MECHANISMS} \arrow[dl, dash, "/"] \\
& \text{IMPRECISE MECHANISMS} &
\end{tikzcd}$$

[1] Many pre-ALGOL60 languages do not satisfy this requirement since they do not support recursion. For example, the FORTRAN standard [ANSI78] excludes recursion from the language, and therefore no pr can be found such that $vf_{FORTRAN}(pr)$ access-satisfies or provide-satisfies a vg containing a loop in either its A_a or A_p.

where by *imprecise* mechanisms we mean those that are neither access-precise nor provide-precise. If we disregard self-recursive visibility, which in most languages cannot be fully controlled, then entries in this preciseness-characterization hierarchy are exemplified by the mechanisms found in ALGOL60, Ada, Gypsy, and our approach to interface control. The following theorems, whose proofs are only sketched in this paper, position these mechanisms in the hierarchy.

Theorem. ALGOL60 is neither access-precise nor provide-precise.

Proof (sketch). For a mechanism to be imprecise, a visibility graph must exist for which a program representation cannot be found that results in exactly the desired accessibility and, similarly, a visibility graph must exist for which a program representation cannot be found that results in exactly the desired provision. One such graph, which reflects a very common situation in programming, conveniently exhibits both these properties. This graph consists of two subprograms, each not callable by the other, sharing exclusive use of a third, utility subprogram. From the definition of $af_{ALGOL60,subprograms}$ and $pf_{ALGOL60,subprograms}$ it can be seen that for the two subprograms to be hidden from each other, and so not callable by each other, one cannot be nested (directly or indirectly) in the other nor can they be siblings. The utility subprogram must then be an ancestor (other than a parent) so that it is callable by both subprograms. This being the case, the utility subprogram must unavoidably be accessible from, and provided to, other ancestors of the subprograms, thus violating the desired visibility relationships.

Theorem. Ada is access-precise but not provide-precise.

Proof (sketch). Ada supports a nesting mechanism similar to ALGOL60's, but in addition offers alternatives that can be used to avoid many of nesting's shortcomings [CLAR80]. Since the previous theorem showed nesting to be inadequate as either an access-precise or provide-precise mechanism, this feature will not be considered in the evaluation of Ada for these properties. To show Ada is access-precise, first observe that Ada programs can be constructed exclusively from nest-free *packages* that are collections of data objects, types, and nest-free subprograms. Each such package employs a construct called the *with clause* to specify the entities accessible from its contents. The with clause allows the realization of any arbitrary set of accessibility relationships since, in the extreme, one package can be created for each of the entities in the system.[2] In terms of

[2] Of course, purely local entities need not be packaged, but can be left, for instance, in the subprograms in which they are used. Recursive subprograms referencing shared entities introduce some minor complications, but this can be handled by appropriate use of parameters and packages.

visibility graphs and program representations, this means that if only with clauses are used to induce accessibility arcs, then for any given visibility graph a program representation can be found that results in exactly the desired accessibility graph. Thus, Ada is shown to be access-precise. Ada does not, however, offer a provide-precise alternative to nesting. Provided packaged entities (in Ada's terminology, the *visible* packaged entities) are unavoidably provided to all other entities in the program and hence their corresponding nodes in provision graphs have arcs to every other node.

Theorem. Gypsy is provide-precise but not access-precise.

Proof (sketch). Gypsy does not support any degree of nesting. To control provision, Gypsy employs a construct called (unfortunately) an *access list* which specifies for an entity just those other entities to which it is provided. In terms of visibility graphs and program representations, this feature solely determines provision arcs. Hence, for any given visibility graph a program representation can be found that results in exactly the desired provision graph with the consequence that Gypsy is provide-precise. Gypsy does not, however, have Ada's concept of the with clause. Indeed, there is no way to control accessibility in Gypsy; all provided entities are unavoidably accessible. Therefore, aside from visibility graphs having pairs of nodes connected by both a provision arc and an accessibility arc, desired accessibility relationships cannot be realized. Thus, Gypsy is not access-precise.

Theorem. The mechanism described in [CLAR83] is precise.

Proof (sketch). This mechanism combines the essential features of Ada and Gypsy to create a means for constructing systems in which the accessibility and provision of each entity can be precisely specified. In particular, it includes an Ada-like with clause for determining accessibility and a Gypsy-like access list for determining provision. It can be shown, therefore, to be a precise visibility control mechanism.

There are other characterizations of visibility control mechanisms that are useful for performing rigorous evaluations. For instance, one would like to be able to understand the kinds of situations that lead to imprecise realizations of entity visibility when using a particular mechanism. This and other characterizations are under investigation.

5. Conclusion

It is worth noting that graphs have been used elsewhere to describe concepts related to entity visibility. For instance, graphs are used informally for describing nesting's effect on data and control flow in Ada programs [CLAR80]. Thomas [THOM76] uses graphs more formally to analyze "resource information flow." The usefulness of Thomas's approach is restricted by its strong orientation to the particular module interconnection language developed in [THOM76]; it was never intended as a general, descriptive formalism. Moreover, it lacks the useful concept of provision. Lipton and Snyder [LIPT77] use a graph model to study a particular protection mechanism, the *take and grant* system, in which arcs in a graph are labeled with the access rights one node has to another. This model is directed at analyzing the effect of rewrite rules that dynamically add and delete nodes and arcs, and thus addresses a different problem domain.

There are two limitations on our formalism that should be pointed out. First, the formalism presented here is concerned only with *static* visibility control mechanisms, that is, those mechanisms that determine bindings before execution. Second, this formalism only deals with *direct* visibility; the fact that an entity can use another entity through a third entity is not considered (for instance, if e_i and e_j are subprograms, e_k is a data object, and e_i can invoke e_j to operate on e_k, this does not necessarily imply that e_k is visible to e_i). Relaxing these restrictions is a topic for future study.

We recognize that there are other considerations that affect how a visibility control mechanism is used. For instance, the package in Ada, besides being used in the control of entity visibility, is used as a primary modularization tool; there are practical situations in which modularity and visibility control constraints conflict. In light of this, the formalism should be extended to explicitly address the sources of such conflicts. We have begun work in this direction, but discussion of that work is beyond the scope of this paper.

In summary, we have defined a formalism that can be used both to describe visibility control mechanisms and to reason about those mechanisms in order to provide characterizations of their strengths and weaknesses. In this paper, we have shown how the formalism can be used to characterize the preciseness of visibility control mechanisms. In so doing, we have pointed up the very different approaches to controlling entity visibility employed in four such mechanisms. Based on examples such as these, we believe this formalism can be a valuable aid to language designers and programmers as they try to decide on the suitability of a mechanism.

REFERENCES

AMBL77	A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch and R.E. Wells, *GYPSY: A Language for Specification and Implementation of Verifiable Programs*, **Proceedings of an ACM Conference on Language Design for Reliable Software**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, pp.1-10, March 1977.
ANSI78	ANSI X3.9 - 1978 (American National Standard Programming Language FORTRAN).
BRIN81	P. Brinch Hansen, *The Design of Edison*, **Software - Practice and Experience**, Vol. 11, No. 4, pp.363-396, April 1981.
CASH81	P.M. Cashin, M.L. Joliat, R.F. Kamel and D.M. Lasker, *Experience with a Modular Typed Language: Protel*, **Proceedings of the Fifth International Conference on Software Engineering**, San Diego, California, pp.136-143, March 1981.
CLAR80	L.A. Clarke, J.C. Wileden and A.L. Wolf, *Nesting in Ada Programs is for the Birds*, **Proceedings of an ACM-SIGPLAN Symposium on the Ada Programming Language**, appearing in **SIGPLAN Notices**, Vol. 15, No. 11, pp.139-145, November 1980.
CLAR83	L.A. Clarke, J.C. Wileden and A.L. Wolf, *Precise Interface Control: System Structure, Language Constructs, and Support Environment*, **Technical Report 83-26**, COINS Department, Unviersity of Massachusetts, Amherst, Massachusetts, August 1983 (submitted for publication).
DOD83	**Reference Manual for the Ada Programming Language** (ANSI/MIL-STD-1815A), United States Department of Defense, Washington, D.C., January 1983.
FDA80	**Formal Definition of the Ada Programming Language**, Honeywell, Inc., Minneapolis, MN, November 1980.
HENN81	J.L. Hennessy and R.B. Kieburtz, *The Formal Definition of a Real-Time Language*, **Acta Informatica**, Vol. 16, pp.309-345, 1981.
JENS74	K. Jenson and N. Wirth, *Pascal - User Manual and Report*, **Lecture Notes in Computer Science**, Vol. 18, Springer-Verlag, New York, 1974.
LIPT77	R.J. Lipton and L. Snyder, *A Linear Time Algorithm for Deciding Subject Security*, **Journal of the ACM**, Vol. 24, No. 3, pp.455-464, July 1977.
LISK79	B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Schiefler and A. Snyder, *CLU Reference Manual*, **Lecture Notes in Computer Science**, Vol. 114, Springer-Verlag, New York, 1981.

MITC79 J.G. Mitchell, W. Maybury and R. Sweet, *Mesa Language Manual Version 5.0*, **Technical Report CSL-79-3**, Xerox PARC, Palo Alto, California, April 1979.

POPE77 G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell and R.L. London, *Notes on the Design of Euclid*, **Proceedings of an ACM Conference on Language Design for Reliable Software**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, pp.11-18, March 1977.

SHAW81 M. Shaw (ed.), **ALPHARD: Form and Content**, Springer-Verlag, New York, 1981.

THOM76 J.W. Thomas, *Module Interconnection in Programming Systems Supporting Abstraction*, **Technical Report CS-16**, Computer Science Program, Brown University, April 1976.

WELS77 J. Welsh, W.J. Sneeringer and C.A.R. Hoare, *Ambiguities and Insecurities in Pascal*, **Software - Practice and Experience**, Vol. 7, No. 6, pp.685-696, November-December 1977.

WIRT77 N. Wirth, *Modula: A Language for Modular Multiprogramming*, **Software - Practice and Experience**, Vol. 7, No. 1, pp.3-35, January-February 1977.