A MODEL OF VISIBILITY CONTROL

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

COINS Technical Report 87–12
February 1987
(Revised August 1987)

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

A version of this report to appear in IEEE Transactions on Software Engineering.

This work supported in part by NSF grants DCR-83-18776 and DCR-84-08143.

ABSTRACT

A number of mechanisms have been created 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 formal model for describing and evaluating visibility control mechanisms. The model reflects a general view of visibility in which the concepts of *requisition of access* and *provision of access* are distinguished. This model provides a means for characterizing and reasoning about the various properties of visibility control mechanisms. Specifically, a notion of *preciseness* is defined in this paper. The utility of the model is illustrated by using it to evaluate and compare the relative strengths and weaknesses, with respect to preciseness, of the visibility control mechanisms found in ALGOL60, Ada, Gypsy, and an approach we have developed, called PIC, that specifically addresses the concerns of visibility 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 composing a software system [28,6,5,10]. A variety of languages, such as Ada [7], Clu [15], Euclid [13], Gypsy [1], Mesa [17], MODULA-2 [24], and Smalltalk [9], have attempted to compensate for the inadequacies of nesting by offering alternative or supplemental mechanisms for visibility control. As with most language concepts in computer science, however, 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 model for formally describing and evaluating visibility control mechanisms. The model reflects a view of visibility in which the concepts of *requisition of access* and *provision of access* are distinguished. This model provides a means for characterizing and reasoning about the various properties of visibility control mechanisms. With this model, language designers can better justify new mechanisms and software developers can decide upon the suitability of a mechanism for controlling entity visibility within their application programs. We have used the model to formulate a definition of *preciseness* and applied that definition in the evaluation of several visibility control mechanisms.

The next section presents the basic features of the model. The use of the model for describing visibility control mechanisms is discussed in Section 3. Section 4 illustrates the use of the model in evaluating such mechanisms. Theorems are presented that serve to characterize the relative strengths and weaknesses, with respect to preciseness, of the visibility control mechanisms found in ALGOL60, Ada, Gypsy, and an approach we have developed, called PIC, that specifically addresses the concerns of visibility control in large software systems [25,26,27].

2 Basic Definitions

Traditionally, the concept of entity *visibility* has been defined in terms of *declaration*, *scope*, and *binding* (e.g., [19]). 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 potentially visible. 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 model presented here is based on the more general concepts of *requisition of access* and *provision of access*, which are two different, yet complementary, points of view on visibility. *Access* to an entity is the right to make reference to, or use of, that entity in declarations and statements. Requisition of access occurs when an entity (implicitly or explicitly) requests the right to potentially refer to some set of entities. Thus, for example, in most programming languages a subprogram typically requests access to itself and any locally declared entities, as well as certain non-local entities. Provision of access occurs when an entity (implicitly or explicitly) offers, to some set of entities, the right to potentially refer to that entity. Again, in most programming languages, access to a subprogram (i.e., the right to potentially invoke that subprogram) is typically provided to the subprogram itself and, in languages supporting nesting, to the subprogram's parent, siblings, and descendants. Under this view of visibility, an actual reference by an entity e_i to an entity e_j is only possible if e_i requests access to e_j and e_j provides access to e_i.[1]

A visibility control mechanism is the means for specifying requisition and provision. The distinction between requisition and provision reflects the differences in the overall approaches to controlling entity visibility taken in different languages. In languages such as ALGOL60 and Pascal, requisition and provision are essentially mirror images; those entities requested by an entity are always also provided to that entity and vice versa. In the designs of 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 requisition and provision in separate, and often unequal, ways. For instance, such a language might allow one to specify

[1]In the remainder of this paper, when the intended meaning is clear, the word "access" is dropped from certain phrases involving the terms "requisition" and "provision". Thus, a "requested entity" is one to which access is requested, and the "requisition of an entity" refers to the requisition of access to the entity. Similarly, a "provided entity" is one for which access is provided, and the "provision of an entity" refers to the provision of access to the entity.

$$
\begin{aligned}
N &= \{\mathrm{Q,R,S}\} \\
A_r &= \{(\mathrm{R,Q}),(\mathrm{S,R})\} \\
A_p &= \{(\mathrm{Q,Q}),(\mathrm{Q,R}),(\mathrm{Q,S})\}
\end{aligned}
$$

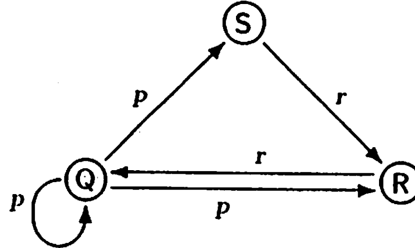


Figure 1: A Visibility Graph.

that a module is provided to any other module although only a few of those other modules may actually request it. Our model, by distinguishing between requisition and provision, can expose such differences in visibility control mechanisms.

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

> *Definition.* A *visibility graph* $g = (N, A_r, A_p)$ is a directed graph where
>
> N is a finite set of nodes labeled by the (unique) names of entities;
>
> A_r is a finite set of ordered pairs of nodes (n_i, n_j) denoting the requisition relationship n_i "requests access to" 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.

The ordered pairs in A_r and A_p determine the arcs in the graph. A visibility graph may contain loops (arcs whose origin and terminus are the same node) and cycles, both resulting from recursive requisition and provision relationships. For languages, such as Ada, that include an identifier-overloading feature, the "name" of an entity is sufficient to resolve any ambiguity; in the case of Ada, the name of an overloaded subprogram would include type-signature information. Figure 1 depicts an example visibility graph.

A visibility graph uniquely represents a particular set of visibility relationships among a set of entities. The visibility relationships of any entity e are defined by the arcs from a node n_e to that node's nearest neighbors in

the graph (i.e., adjacent nodes). The absence of an arc between two nodes indicates that no visibility relationship exists between the corresponding entities.

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

Definition. For a given visibility graph $g = (N, A_r, A_p)$, the corresponding *requisition graph* is $g_r = (N, A_r, \emptyset)$.

Definition. For a given visibility graph $g = (N, A_r, A_p)$, the corresponding *provision graph* is $g_p = (N, \emptyset, A_p)$.

These subgraphs are defined to span the visibility graph because we want them to capture the requisition and provision of each (and every) entity in the visibility graph.

Two useful relationships between visibility graphs can be defined.

Definition. A visibility graph g *request-satisfies* a visibility graph h iff $h_r \subseteq g_r$.

Definition. A visibility graph g *provide-satisfies* a visibility graph h iff $h_p \subseteq g_p$.

where we say $h \subseteq g$ if $N_h \subseteq N_g$, $A_{r,h} \subseteq A_{r,g}$, and $A_{p,h} \subseteq A_{p,g}$. Informally stated, the desire for a set of entities s_j to be requested by (provided to) some entity e is satisfied by e requesting (being provided) 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, such as its text or its parse tree, 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 collection of program representations by R, denote the collection of visibility graphs by G, and define a function that performs this mapping as follows:

Definition. A *visibility function* $v : R \rightarrow G$ is a function that maps a program representation $x \in R$ to a corresponding visibility graph $g \in G$.

A set of visibility functions $V = \{v_a, v_b, \ldots\}$ can be defined where v_m is the visibility function implementing the visibility control mechanism m.

The model uses the visibility graph to record requisition and provision relationships without insisting on a particular interpretation of the consistency/inconsistency of those relationships. In Ada, for example, a minimum condition for the consistency of a set of entity visibility relationships is that the entities that each entity requests are in fact provided. In terms of visibility graphs, this corresponds to the following property:

> *Definition.* A visibility graph $g = (N, A_r, A_p)$ is *well-formed for Ada* iff $\forall(n_i, n_j) \in A_r$, $(n_j, n_i) \in A_p$.

An Ada interpretation of the graph in Figure 1 would then view the Q–R and Q–S relationships as consistent; node Q might represent a *library entity*—an entity, such as a sine function, provided to all other entities even though not all those other entities request it. The R–S relationship, on the other hand, would be interpreted as inconsistent. Of course, consistency/inconsistency interpretations of the graph in Figure 1 other than the Ada interpretation are also reasonable. For example, node Q might represent some sort of "authorization" module; the fact that S does not request access to Q might then indicate a problem in the system. In general, the appropriateness of an interpretation can depend upon the language, the application domain, the development method, or even the managerial discipline in force.

We should point out that our model is expressly intended to capture the semantics of *static* visibility control mechanisms. Static mechanisms account for the vast majority of visibility control mechanisms found in modern languages. This is probably because static mechanisms exhibit greater security from (run-time) errors. Indeed, the trend in language design is clearly away from dynamic visibility control. This can be seen, for example, in the evolution of LISP; early dialects of LISP are based on dynamic scope rules, whereas Common LISP [21] provides a static visibility control mechanism.

3 Describing Visibility Control Mechanisms

One of our primary goals in this work is to provide an effective means of describing visibility control mechanisms so that one can reason about and evaluate those mechanisms. Existing informal and formal descriptive methods have proven inadequate. The Pascal Report [12], for example, causes many problems due to the ambiguity of its prose description of entity visibility [4,23]. The few formal approaches to describing visibility control

mechanisms are operational in nature and have appeared primarily in operational and denotational semantic specifications, where a mechanism is typically described by the manipulation of an (identifier) environment component (e.g., [3]). This technique is unsatisfactory, however, because the method for describing manipulation of that environment component is essentially imperative (despite the use of a "functional" notation; see, for example, [8]) and thus more difficult to reason about than a declarative description. Moreover, the information in the environment component is only explicitly described from the perspective of entity requisition. 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 model presented here, a visibility control mechanism m is described by its corresponding visibility function v_m. Each such function has two components that *explicitly* address the requisition and provision aspects of entity visibility. The *requisition function* r_m describes requisition by mapping a program representation to a requisition graph while the *provision function* p_m describes provision by mapping a program representation to a provision graph. Thus,

$$v_m(x) = r_m(x) \cup p_m(x)$$

where x is some program representation and the union of two visibility graphs g and h is the visibility graph $(N_g \cup N_h, A_{r,g} \cup A_{r,h}, A_{p,g} \cup A_{p,h})$. Component functions r and p can be further broken down into functions operating on individual *kinds* of entities, such as subprograms and objects, as follows:

$$\begin{aligned} r_m(x) &= r_{m,E_1}(x) \cup \cdots \cup r_{m,E_n}(x) \\ p_m(x) &= p_{m,E_1}(x) \cup \cdots \cup p_{m,E_n}(x) \end{aligned}$$

where E_i denotes the entity kind upon which the requisition or provision function operates. Hence, for each entity kind that is of interest, there is a function that describes requisition and a function that describes provision. Requisition functions are similar in nature to the "binding" functions of [11]. Provision functions, however, appear to have no counterpart in previous formalisms.

The full description of a complex visibility control mechanism can of course be quite lengthy. To provide a feel for the use of the descriptive

method, without giving an excessively long example, we present partial descriptions of the visibility control mechanisms of two very different languages: ALGOL60 and the object-oriented language Smalltalk. For ALGOL60, we describe the requisition and provision of subprograms as they are controlled by nesting. This entails the definition of the requisition function $r_{ALGOL60,subprograms}$ and the provision function $p_{ALGOL60,subprograms}$. The discussion is further simplified by only considering the visibility of subprograms to subprograms. For Smalltalk, we describe the requisition and provision of operations associated with class definitions, which entails the definition of $r_{Smalltalk,class-operations}$ and $p_{Smalltalk,class-operations}$. To keep this example simple, other aspects of visibility control in Smalltalk, such as access to instance and class variables, are not considered.

ALGOL60

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

Definition. A *nesting graph* $g = (N, A_{pa})$ is a tree where

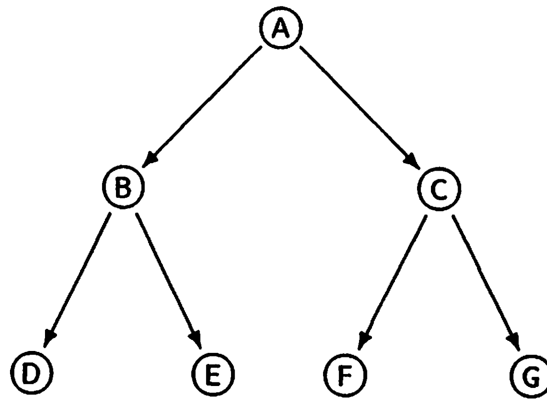
N is a finite set of nodes labeled by the (unique) names of entities;

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

For purposes of this example, N will consist only of nodes whose entities are subprograms. Figure 2 shows the skeleton of a nested ALGOL60 program and its representation as a nesting graph.

In the subsequent definition of the requisition and provision functions, we make use of three auxiliary functions defined as follows, where n_i, n_j, and n_k are elements of N, the nodes in a nesting graph.

```
begin
   procedure A;
      procedure B;
         procedure D;
         begin ... end D;
         procedure E;
         begin ... end E
      begin ... end B;
      procedure C;
         procedure F;
         begin ... end F;
         procedure G;
         begin ... end G
      begin ... end C
   begin ... end A
end
```



(a) (b)

Figure 2: A Nested Program (a) and its Representation as a Nesting Graph (b).

$$(1)\quad Parent(n_i) = \begin{array}{l}\text{if } (n_j, n_i) \in A_{pa} \text{ then } n_j \\ \text{otherwise } \bot\end{array}$$

$$(2)\quad Siblings(n_i) = \left\{ n_j \,\middle|\, \begin{array}{l}(Parent(n_i), n_j) \in A_{pa} \\ \text{and } i \neq j\end{array} \right\}$$

$$(3)\quad Ancestors(n_i) = \left\{ n_j \,\middle|\, \begin{array}{l} n_j = Parent(n_i) \\ \text{or } n_j \in Siblings(Parent(n_i)) \\ \text{or } \exists n_k \in Ancestors(n_i) \text{ such that} \\ n_j \in Ancestors(n_k) \end{array} \right\}$$

For any $n_i \in N$, $Parent(n_i)$ will always be unique, since an entity can be directly nested in at most one other entity.

The requisition function is now defined to transform a nesting graph $g = (N, A_{pa})$ into a requisition graph, explicitly describing the effect of ALGOL60's nesting mechanism on subprograms' requisition.

Definition. $r_{ALGOL60,subprograms}(g) = (N', A_r, \emptyset)$ where

$$N' = N$$

$$A_r = \left\{ (n_i, n_j) \,\middle|\, \begin{array}{l} i = j \\ \text{or } n_i = Parent(n_j) \\ \text{or } n_j \in Siblings(n_i) \\ \text{or } n_j \in Ancestors(n_i) \end{array} \right\}$$

From this description it can be easily seen that (1) a subprogram is requested by itself, (2) a subprogram is requested by the subprogram in which it is directly nested, (3) a subprogram is requested by those subprograms with the same parent, and (4) a subprogram is requested by those subprograms nested within it as well as requested by those subprograms nested within its siblings. Figure 3 depicts a requisition graph corresponding to the nesting graph of Figure 2. For simplicity, self-recursive requisition is not shown and pairs of oppositely-directed arcs have been drawn as single, bi-directional arrows.

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

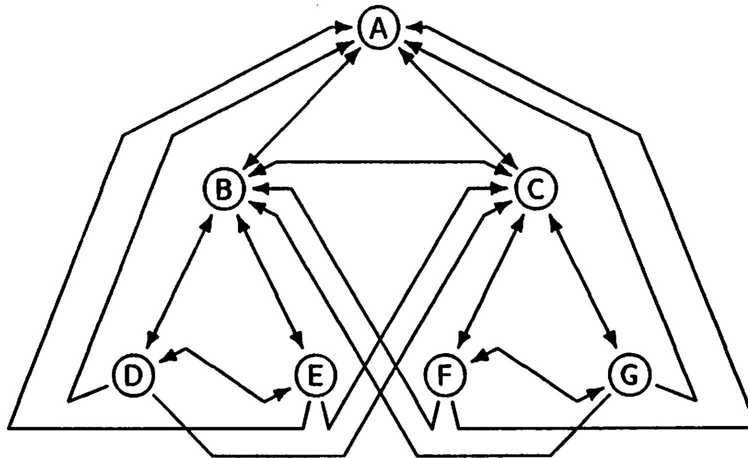


Figure 3: Requisition Graph for Nesting Graph of Figure 2.

Definition. $p_{ALGOL60,subprograms}(g) = (N', \emptyset, A_p)$ where

$$N' = N$$

$$A_p = \left\{ (n_i, n_j) \;\middle|\; \begin{array}{l} i = j \\ \text{or } n_j = Parent(n_i) \\ \text{or } n_i \in Siblings(n_j) \\ \text{or } n_i \in Ancestors(n_j) \end{array} \right\}$$

These descriptions reveal the fact that in ALGOL60 requisition 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_r of the requisition function, except that the i's and j's are reversed. As illustrated below, such a similarity is certainly not true of all visibility control mechanisms.

We contend that requisition and provision functions of the formal model presented here are easier to comprehend than the manipulation of an environment component found in other formal models. As mentioned above, for instance, those other models make it difficult to recognize that, with nesting, a subprogram's so-called "local" entities are unavoidably made visible to other subprograms nested within that subprogram. This problem is clearly exposed using the model presented here; by simply looking at the requisition and provision functions for subprograms it is immediately ev-

ident that a subprogram's supposedly "local" child subprogram is in fact visible to any other subprograms nested within that subprogram.

Smalltalk

Smalltalk is based on the notion that the major building block of a program is the class (i.e., type) definition. A class definition establishes for each instance (i.e., object) of a given class a set of operation signatures,[2] a set of code bodies (i.e., methods) to implement those operations, and some local storage (i.e., variables). Classes are related in essentially two ways. The first way is that a method in one class can "invoke" an operation/method in another class, or in Smalltalk's terminology, instances of classes communicate by exchanging messages that request operations; the operations are realized through the execution of the methods that implement the operations. The second way that classes are related is by inheritance, which is a mechanism that allows the operations and methods of one class, the so-called superclass, to be shared with another class, the so-called subclass.

Here we describe the visibility control mechanism of Smalltalk that determines the operations that are associated with a class. As always, the description is based on a visibility function that maps a program representation to a visibility graph. The program representation we choose for this aspect of Smalltalk programs is a graph we call the *subclass graph*.

Definition. A *subclass graph* $g = (N_c, A_s)$ is a tree where

N_c is a finite set of nodes labeled by the (unique) names of classes;

A_s is a finite set of ordered pairs of nodes (n_i, n_j) denoting the relationship n_i "subclass of" n_j, which means n_j is indicated as the superclass of n_i;

Notice that the subclass graph faithfully reflects how the Smalltalk language indicates the inheritance relationship among classes. In particular, Smalltalk syntax calls for a subclass to indicate its superclass in that subclass's definition through what might be referred to as a *subclass clause*. There is no way, however, for a superclass to indicate its subclasses. Notice too that the subclass graph is a tree because Smalltalk uses single inheritance as opposed to multiple inheritance.

---

[2] In the remainder of this discussion of Smalltalk, "operation signatures" are referred to simply as "operations".

Associated with a subclass graph is a finite set O of (unique) operation names. The locally-defined operations of a given class, which form a subset of O, are identified using the following function.

Definition. $L : N_c \rightarrow \mathcal{P}(O)$ is a function that maps a class $c \in N_c$ to the power set of O.

Requisition and provision are quite different in this Smalltalk mechanism. We begin by describing requisition. To do so, we make use of the following auxiliary function, where n_i, n_j, and n_k are elements of N_c, the nodes in a subclass graph.

$$Superclasses(n_i) = \left\{ n_j \;\middle|\; \begin{array}{l} (n_i, n_j) \in A_s \\ \text{or } \exists n_k \in Superclasses(n_i) \text{ such that} \\ n_j \in Superclasses(n_k) \end{array} \right\}$$

This captures the fact that the superclasses of a given class are all the classes along the path from that class to the root of the subclass graph, where the path is determined by following the directed arcs in A_s.

The requisition function is now defined to transform a subclass graph $g = (N_c, A_s)$ and its associated set of operations O into a requisition graph, explicitly describing the effect of Smalltalk's inheritance mechanism on the association of operations with a class.

Definition. $r_{Smalltalk,class-operations}(g) = (N, A_r, \emptyset)$ where

$$N = N_c \cup O$$

$$A_r = \left\{ (n_i, n_j) \;\middle|\; \begin{array}{l} n_i \in N_c \text{ and } n_j \in O \\ \text{and either } n_j \in L(n_i) \text{ or} \\ \exists n_k \in N_c \text{ such that} \\ n_k \in Superclasses(n_i) \text{ and} \\ n_j \in L(n_k) \end{array} \right\}$$

The requisition function thus indicates that the subclass clause appearing in a class definition essentially serves as a request for all operations appearing in the chain of superclasses leading up to the root of the subclass graph.

The provision function is almost trivial, since Smalltalk does not support a capability for a class to itself limit its subclasses. All operations are (implicitly) provided to all other classes.

Definition. $p_{Smalltalk,class-operations}(g) = (N, \emptyset, A_p)$ where

$$N = N_c \cup O$$
$$A_p = \left\{(n_i, n_j) \,\middle|\, n_i \in O \text{ and } n_j \in N_c\right\}$$

Our model highlights several interesting properties of Smalltalk. In particular, visibility control in this context amounts to the ability of a class to indicate its superclass (requisition) but not the ability to indicate its subclasses (provision). Moreover, while the only operations that can be requested are those belonging to classes along the subclass path, *all* operations along that path are unavoidably requested. In other words, Smalltalk supports no means for selective inheritance. Observe too that provision has no effective impact on visibility, since all operations are provided to all classes, and so the requested operations are always a subset of the provided operations. Some have suggested that finer control over inheritance is desirable (e.g., [20]). We have begun to investigate the utility of such controls in supporting different views of the abstraction presented by a class definition and are using our model to guide the design of appropriate language features.

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 that is possible within the framework of the model presented above. Specifically, the notion of *preciseness* is defined for a visibility control mechanism in terms of the mechanism's accuracy in capturing desired requisition and provision relationships.

It can easily be argued that a language's visibility control mechanism(s) m should be such that $\forall g \in G, \exists x \in R$ such that $v_m(x)$ request-satisfies and provide-satisfies g. In other words, it should be possible to find a program in the language that realizes any requisition and provision relationships that a developer might wish to devise, although additional requisition 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.[3] Stronger properties for evaluating mechanisms are needed, however, which leads to the following definitions.

[3] Many pre-ALGOL60 languages do not satisfy this requirement since they do not support recursion. For example, the FORTRAN standard [2] excludes recursion from the

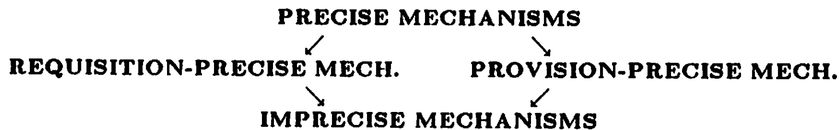
Definition. A visibility control mechanism m is *requisition-precise* iff $\forall g \in G$, $\exists x \in R$ such that $r_m(x) = g_r$.

Definition. A visibility control mechanism m is *provision-precise* iff $\forall g \in G$, $\exists x \in R$ such that $p_m(x) = g_p$.

Definition. A visibility control mechanism m is *precise* iff it is both requisition-precise and provision-precise (i.e., $\forall g \in G$, $\exists x \in R$ such that $v_m(x) = r_m(x) \cup p_m(x) = g$).

Definition. A visibility control mechanism m is *imprecise* iff it is neither requisition-precise nor provision-precise.

Intuitively, the definitions state that if for each possible visibility graph, a program 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 the requisition relationships or provision relationships cannot be exactly realized. This suggests the following hierarchy of visibility control mechanisms based on preciseness:



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 PIC. The following theorems position these mechanisms in the hierarchy.

THEOREM 1. ALGOL60 is imprecise.

PROOF. For a mechanism to be imprecise, a visibility graph must exist for which a program cannot be found that results in exactly the desired requisition and, similarly, a visibility graph must exist for which a program cannot be found that results in exactly the desired provision. One such graph, which reflects a very common situation in programming, conveniently

language, and therefore no x can be found such that $v_{FORTRAN}(x)$ request-satisfies or provide-satisfies a g containing a loop in either its A_r or A_p.

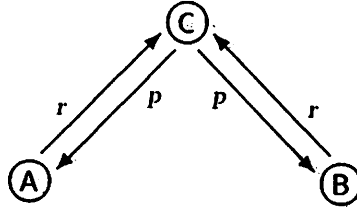


Figure 4: Visibility Graph of a Common Programming Situation.

exhibits both these properties. This graph, depicted in Figure 4, represents two subprograms A and B, each not callable by the other, sharing exclusive use of a third, utility subprogram C. This graph could correspond to a complete program or could, and typically would, be a subgraph of the desired visibility graph for a larger program. In that case, this subgraph would capture not only all the visibility relationships among A, B, and C, but also all the relationships between C and any entity in the entire program. From the definition of $r_{ALGOL60,subprograms}$ and $p_{ALGOL60,subprograms}$ it can be seen that for the two subprograms A and B 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. If this graph corresponds to a complete program, then it is impossible for the utility subprogram to be an ancestor other than a parent, and hence we have a contradiction. If the graph is a subgraph, then being an ancestor other than a parent to A and B means that the utility subprogram must unavoidably be requested by, and provided to, other ancestors of those subprograms, thus violating the desired visibility relationships. □

THEOREM 2. Ada is requisition-precise but not provision-precise.

PROOF. 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 [5]. These alternatives are the *private/visible* mechanism of Ada's encapsulation construct, the package, and the *with clause*. The first can be used in combination with nesting to achieve a greater degree of provision control than is possible in ALGOL60. In particular, it can be used to selectively hide nested entities that would otherwise be undesirably

provided. That selection, however, is on an all-or-none basis; either an entity is provided to all entities in the scope of the package or it is provided to no entity. Therefore, Ada's version of nesting is still not provision-precise. This shortcoming with respect to provision extends to nest-free packages, where the "scope" of a package is then the entire program. Entities provided by a package (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. (For example, if either of the provision arcs in Figure 4 were omitted, then it would be impossible to represent the resulting visibility relationships in Ada.) Thus, Ada is shown not to be provision-precise. To show Ada is requisition-precise, first observe that Ada programs can be constructed exclusively from nest-free packages. Each such package employs the second alternative mentioned above, the *with clause*, to specify the entities requested by its contents. The *with clause* allows the realization of any arbitrary set of requisition relationships since, in the extreme, one package can be created for each of the entities in the program.[4] (Figure 5 illustrates how such an approach can be used to achieve the desired requisition relationships shown in Figure 4.) In terms of visibility graphs and program representations, this means that if only *with clauses* are used to induce requisition arcs, then for any given visibility graph a program can be found that results in exactly the desired requisition graph. Thus, Ada is shown to be requisition-precise. □

THEOREM 3. Gypsy is provision-precise but not requisition-precise.

PROOF. Gypsy does not support any degree of nesting. To control provision, Gypsy employs a construct called 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 can be found that results in exactly the desired provision graph, with the consequence that Gypsy is provision-precise. (Figure 6 illustrates how Gypsy can be used to

[4]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 these can be handled by appropriate use of parameters and packages [25]. Finally, types that are mutually dependent cannot be separately packaged. It can be argued, however, that while this special case involves two or more syntactically separate type declarations, only one *genuine* type is being defined; it makes no sense to request (or provide) access to one component of the definition without requesting (or providing) access to the others.

```
package CPack is
   procedure C ( ... );
end CPack;

   with CPack; -- request for C
package APack is
   procedure A ( ... );
end APack;

   with CPack; -- request for C
package BPack is
   procedure B ( ... );
end BPack;
```

Figure 5: Use of Ada to Achieve Requisition Relationships Shown in Figure 4.

achieve the desired provision relationships shown in Figure 4.) Gypsy does not, however, have Ada's concept of the *with clause*. Indeed, there is no way to control requisition in Gypsy; all provided entities are unavoidably requested. Therefore, aside from visibility graphs having pairs of nodes connected by both a provision arc and a requisition arc, desired requisition relationships cannot be realized. (For example, if either of the requisition arcs in Figure 4 were omitted, then it would be impossible to represent the resulting visibility relationships in Gypsy.) Thus, Gypsy is not requisition-precise. □

The languages positioned by the previous three theorems illustrate each of the entries in the preciseness-characterization hierarchy except the highest. That entry is illustrated by the family of languages based on the PIC language framework [25]. The framework was developed using the model described in this paper. Thus, it should not be surprising that it constitutes a precise visibility control mechanism.

PIC, which stands for Precise Interface Control, was designed to support the variety of inter-module relationships required in large software systems. Interface control is concerned with that aspect of visibility control that addresses inter-module relationships. The framework provides support for the explicit specification of both requisition and provision across mod-

```
procedure < A, B > C ( ... ) =  (* C provided only to A and B *)
   ...
end;

procedure A ( ... ) =
   ...
end;

procedure B ( ... ) =
   ...
end;
```

Figure 6: Use of Gypsy to Achieve Provision Relationships Shown in Figure 4.

ule interfaces. The language features used to capture these two aspects of entity visibility are the *request clause*, for specifying requisition, and the *provide clause*, for specifying provision. Below, we refer to a language called PIC/Ada, which is a version of Ada enhanced with (among other things) request and provide clauses.

PIC/Ada, just like Ada, provides little control within a module over the visibility of entities declared in that module. This lack of control, which is based on the presumption that entities are declared together because they are strongly interrelated, can be viewed as a notational shorthand for a commonly occurring situation. If more control is desired, then it can be achieved through the creation of additional modules to hold the appropriate entities. This limitation in PIC/Ada is not one that is inherent in the PIC language features. For example, it would be feasible to extend the semantics of *request* and *provide clauses* to support intra-module control. Doing so in PIC/Ada, however, would not be in keeping with the spirit of Ada; a level of control such as that might be more appropriate in a language based, for example, on Euclid.

Details of the PIC language framework and the benefits of this straightforward, yet powerful, mechanism for controlling entity visibility are given in [25,26,27]. The aspects of PIC/Ada described above are sufficient, however, to illustrate the highest entry in the preciseness-characterization hierarchy.

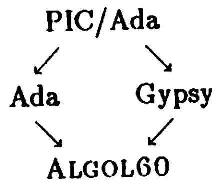
THEOREM 4. PIC/Ada is precise.

```
package CPack is
   procedure C ( ... )
      provide to A, B;
end CPack;

package APack is
   procedure A ( ... )
      request C;
end APack;

package BPack is
   procedure B ( ... )
      request C;
end BPack;
```

Figure 7: Use of PIC/Ada to Achieve Both the Requisition and Provision Relationships Shown in Figure 4.

PROOF. PIC/Ada permits the description of arbitrary graph structures of requisition and provision relationships using *request* and *provide clauses*. By definition, these clauses can be used to completely determine the requisition and provision of individual entities and, therefore, can be used to realize the requisition and provision of any desired visibility graph. (Figure 7 illustrates how PIC/Ada can be used to achieve both the desired requisition and desired provision relationships shown in Figure 4.) This is true despite the fact that PIC/Ada provides little control within a module over the requisition and provision of entities declared in that module, since, in the extreme, one package can be created for each of the entities in a system (cf., proof of Theorem 2) and the requisition and provision of those entities controlled individually and precisely using *request clauses* to determine requisition arcs and *provide clauses* to determine provision arcs. PIC/Ada is thus shown to have a precise visibility control mechanism. □

To summarize, the theorems given in this section allow us to rank the four languages in terms of the preciseness of their visibility control mechanisms, as follows:



It turns out that the Smalltalk mechanism described in Section 3 falls in the same position as ALGOL60.

This characterization clearly demonstrates and justifies how Ada and Gypsy have managed to improve upon the visibility control found in ALGOL60, and reveals the differences in their approaches to this improvement. It also shows that the PIC language framework is essentially a combination of the approaches taken in Ada and Gypsy, one that allows precision from either or both the requisition and provision perspectives. Our model not only facilitates this evaluation, but it was also instrumental in leading us to the definition of the *request* and *provide clauses* of the PIC language framework.

In addition to preciseness, 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 would aid the development of appropriate programming styles for use with that mechanism. We also 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. The proof of Theorem 2 in particular suggests that precision of requisition in Ada can be achieved by placing entities in separate packages. Such a separation may interfere with the colocation of entities that, while perhaps not requested in the same way, are otherwise logically related. The implications of this and other considerations, as well as the development of additional characterizations of visibility control mechanisms, are topics for future study.

5 Conclusion

Graphs have been used elsewhere to describe concepts related to visibility. For instance, graphs are used informally for describing nesting's effect on data and control flow in Ada programs [5]. Thomas [22] 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 [22]; it was never intended as a general, descriptive formalism. Moreover, it lacks the useful concept of provision. Lipton and Snyder [14] 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. Although oriented toward control of access, the purpose of this model is to understand the effect of rewrite rules that dynamically add and delete nodes and arcs, and thus addresses a different problem domain. Osshcr [18] presents an extremely complicated, albeit general, graph model for describing entity relationships at multiple levels of abstraction, which was developed for specifying VLSI fabrication processes. While it would certainly be possible to recast that model to describe requisition and provision relationships, the complexity inherent in the model hinders its usefulness for our current purposes.

We have defined a model that can be used both to formally 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 model can be used to characterize the preciseness of visibility control mechanisms. In so doing, we have pointed up the different approaches to controlling entity visibility employed in four such mechanisms. Based on examples such as this, we believe that the model can be a valuable aid to software developers and language designers as they try to decide upon the suitability of visibility control mechanisms.

REFERENCES

[1] 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*, **Proc. ACM Conf. on Language Design for Reliable Software**, appearing in **SIGPLAN Notices**, Vol. 12, No. 3, March 1977, pp. 1–10.

[2] **ANSI X3.9–1978** (American National Standard Programming Language FORTRAN).

[3] D. Bjørner and C.B. Jones (eds.), *The Vienna Development Method: The Meta-language*, **Lecture Notes in Computer Science**, Vol. 61, Springer–Verlag, Berlin, 1978.

[4] P. Brinch Hansen, *The Design of Edison*, **Software–Practice and Experience**, Vol. 11, No. 4, April 1981, pp. 363–396.

[5] L.A. Clarke, J.C. Wileden, and A.L. Wolf, *Nesting in Ada Programs is for the Birds*, **Proc. ACM-SIGPLAN Symp. on the Ada Programming Language**, appearing in **SIGPLAN Notices**, Vol. 15, No. 11, November 1980, pp. 139–145.

[6] E.W. Dijkstra, **A Discipline of Programming**, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[7] **Reference Manual for the Ada Programming Language** (ANSI/MIL–STD–1815A), United States Department of Defense, Washington, D.C., January 1983.

[8] **Formal Definition of the Ada Programming Language**, Honeywell, Inc., Minneapolis, MN, November 1980.

[9] A. Goldberg and D. Robson, **Smalltalk-80: The Language and its Implementation**, Addison-Wesley, Reading, Massachusetts, 1983.

[10] D.R. Hanson, *Is Block Structure Necessary?*, **Software–Practice and Experience**, Vol. 11, No. 8, August 1981, pp. 853–866.

[11] J.L. Hennessy and R.B. Kieburtz, *The Formal Definition of a Real-Time Language*, **Acta Informatica**, Vol. 16, 1981, pp. 309–345.

[12] K. Jenson and N. Wirth, *Pascal—User Manual and Report*, **Lecture Notes in Computer Science**, Vol. 18, Springer-Verlag, New York, 1974.

[13] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, *Report on the Programming Language Euclid*, **Tech. Rep. CSL-81-12**, Xerox PARC, Palo Alto, California, October 1981.

[14] R.J. Lipton and L. Snyder, *A Linear Time Algorithm for Deciding Subject Security*, **Journal of the ACM**, Vol. 24, No. 3, July 1977, pp. 455-464.

[15] 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.

[16] N.H. Minsky, *Locality in Software Systems*, **Conf. Record Tenth Annual ACM Symp. on Principles of Programming Languages**, Austin, Texas, January 1983, pp. 299-312.

[17] J.G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual Version 5.0*, **Tech. Rep. CSL-79-3**, Xerox PARC, Palo Alto, California, April 1979.

[18] H.L. Ossher, *Grids: A New Program Structuring Mechanism Based on Layered Graphs*, **Conf. Record Eleventh Annual ACM Symp. on Principles of Programming Languages**, Salt Lake City, Utah, January 1984, pp. 11-22.

[19] B. Schwanke, *Survey of Scope Issues in Programming Languages*, **Tech. Rep. CMU-CS-78-131**, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1978.

[20] A. Snyder, *Encapsulation and Inheritance in Object-oriented Programming Languages*, **Proc. OOPSLA '86**, appearing in **SIGPLAN Notices**, Vol. 21, No. 11, November 1986, pp. 38-45.

[21] G.L. Steele, **Common LISP, the Language**, Digital Press, Maynard, Massachusetts, 1984.

[22] J.W. Thomas, *Module Interconnection in Programming Systems Supporting Abstraction*, **Tech. Rep. CS-16**, Computer Science Program, Brown University, April 1976.

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

[24] N. Wirth, **Programming in MODULA-2** (second edition), Springer–Verlag, New York, 1983.

[25] A.L. Wolf, *Language and Tool Support for Precise Interface Control* (Ph.D. Dissertation), **Tech. Rep. 85–23**, COINS Department, University of Massachusetts, Amherst, Massachusetts, September 1985.

[26] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Ada-Based Support for Programming-in-the-Large*, **IEEE Software**, Vol. 2, No. 2, March 1985, pp. 58–71.

[27] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*, **IEEE Trans. on Software Engineering** (to appear), 1987.

[28] W.A. Wulf and M. Shaw, *Global Variable Considered Harmful*, **SIGPLAN Notices**, Vol. 8, No. 2, February 1973, pp. 28–34.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

Smalltalk is a trademark of Xerox Corporation.