A Formal Model for Describing and Evaluating Visibility Control Mechanisms

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

COINS Technical Report 86–44
August 1986

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

Reprinted from the Proceedings of the IEEE Computer Society International Conference on Computer Languages, Miami Beach, Florida, October 1986, pp. 182–189.

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

A Formal Model for Describing and Evaluating Visibility Control Mechanisms

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

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

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 formal model for describing and evaluating visibility control mechanisms. The model reflects a generalized 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 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 [5,24]. A variety of languages, such as Ada [6], Clu [13], Euclid [11], Gypsy [1], Mesa [15], MODULA-2 [20], and Smalltalk [8], have attempted to compensate for the inadequacies of nesting by offering alternative and/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 formal model for describing and evaluating visibility control mechanisms. The model reflects a generalised view of visibility in which the concepts of *requisition of access* and *provision of access* are distinguished. This model provides a means for characterising 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. Specifically, 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 formal 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 characterise the relative strengths and weaknesses 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 [21,22,23].

2. Basic Definitions

Traditionally, the concept of *entity visibility* has been defined in terms of *declaration*, *scope*, and *binding* (cf., [17]). 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 formal 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, 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 descendents. Under this

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

view, 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. Our formal model, by drawing out this distinction, is equipped to expose these 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* $g = (N, A_{re}, A_{pr})$ is a directed graph where

- N is a finite set of nodes labeled by the (unique) names of entities;
- A_{re} 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_{pr} 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_{re} and A_{pr} determine the arcs in the graph. Any pair of nodes in a visibility graph may be connected by multiple arcs resulting from the requisition 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 requisition and provision relationships. Figure 1 depicts an example visibility graph.

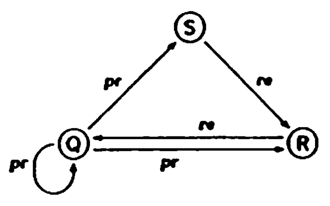
A visibility graph uniquely represents a particular set of visibility relationships among a set of entities. The visibility relationships of any entity e are defined by the arcs from a node n_e to that node's nearest neighbors in the graph (i.e., adjacent nodes). The absence of an arc between two nodes 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. A *requisition graph* $g_{re} = (N, A_{re}, A_{pr})$ is a visibility graph where $A_{pr} = \emptyset$.

Definition. A *provision graph* $g_{pr} = (N, A_{re}, A_{pr})$ is a visibility graph where $A_{re} = \emptyset$.

[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.



$$\begin{aligned} N &= \{Q,R,S\} \\ A_{re} &= \{(R,Q),(S,R)\} \\ A_{pr} &= \{(Q,Q),(Q,R),(Q,S)\} \end{aligned}$$

Figure 1: A Visibility Graph.

Two useful relationships between visibility graphs can be defined.

Definition. A visibility graph g *request-satisfies* a visibility graph h iff $h_{re} \subseteq g_{re}$.

Definition. A visibility graph g *provide-satisfies* a visibility graph h iff $h_{pr} \subseteq g_{pr}$.

where we say $h \subseteq g$ if $N_h \subseteq N_g$, $A_{re,h} \subseteq A_{re,g}$, and $A_{pr,h} \subseteq A_{pr,g}$. 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 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* $vf : R \rightarrow G$ is a function that maps a program representation $r \in R$ to a visibility graph $g \in G$.

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

The formal model uses the visibility graph to record requisition and provision relationships without insisting on a particular interpretation of the consistency/inconsistency of those relationships. For instance, an Ada interpretation of the graph in Figure 1 would view node Q as representing a *library entity*—an entity, such as a sine function, provided to all other entities—even though not all those other entities request it. Thus, the Q–R and Q–S relationships, for example, would be interpreted as consistent. The R–S relationship, on the other hand, would be interpreted as inconsistent. For Ada, 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 $g = (N, A_{re}, A_{pr})$ is *well-formed for Ada* iff $\forall (n_i, n_j) \in A_{re}, (n_j, n_i) \in A_{pr}$.

Of course, other consistency/inconsistency interpretations of the graph in Figure 1 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.

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 [10], for example, causes many problems due to the ambiguity of its prose description of entity visibility [4,19]. 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]). 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, [7]). 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 formal model 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 requisition and provision aspects of entity visibility. The *requisition function* rf_m describes requisition by mapping a program representation to a requisition graph while the *provision function* pf_m describes provision by mapping a program representation to a provision graph. Thus,

$$vf_m(r) = rf_m(r) \cup pf_m(r)$$

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

$$\begin{aligned} rf_m(r) &= rf_{m,ek_1}(r) \cup \cdots \cup rf_{m,ek_n}(r) \\ pf_m(r) &= pf_{m,ek_1}(r) \cup \cdots \cup pf_{m,ek_n}(r) \end{aligned}$$

where ek_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 [9]. 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 just give descriptions of the requisition and provision of subprograms as they are controlled by the nesting mechanism of ALGOL60. This entails the definition of the requisition function $rf_{ALGOL60,subprograms}$ and the provision function $pf_{ALGOL60,subprograms}$. The discussion is further simplified by only considering the visibility of subprograms to subprograms.

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 directed graph 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.

Notice that for this example N consists 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, each of which maps the nodes N in a nesting graph to the powerset of N as follows:

(1) $Parent(n_i) = \{ n_j \in N \mid (n_j, n_i) \in A_{pa} \}$

(2) $Siblings(n_i) = \left\{ n_j \in N \;\middle|\; \begin{array}{l} \exists n_k \in N \text{ such that} \\ (n_k, n_i) \in A_{pa} \\ \text{and } (n_k, n_j) \in A_{pa} \\ \text{and } i \neq j \end{array} \right\}$

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

For any $n_i \in N$, $Parent(n_i)$ will always be a singleton since an entity can be directly nested in only 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.

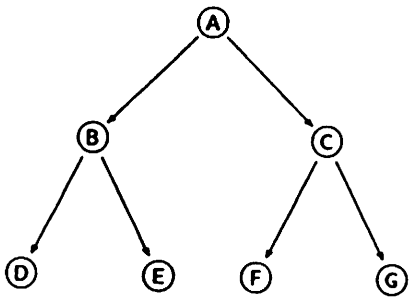
Definition. $rf_{ALGOL60,subprograms}(g) = (N', A_{re}, A_{pr})$ where

$$\begin{aligned} N' &= N \\ A_{re} &= \left\{ (n_i, n_j) \;\middle|\; \begin{array}{l} i = j \\ \text{or } n_i = Parent(n_j) \\ \text{or } n_j \in Siblings(n_i) \\ \text{or } n_j \in Ancestors(n_i) \end{array} \right\} \\ A_{pr} &= \emptyset \end{aligned}$$

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

```
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).

siblings. Figure 3 depicts a portion of the requisition graph corresponding to the nesting graph of Figure 2; for simplicity, self-recursive requisition is not shown.

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

Definition. $pf_{ALGOL60,subprograms}(g) = (N', A_{re}, A_{pr})$
where

$$N' = N$$
$$A_{re} = \emptyset$$
$$A_{pr} = \left\{ (n_i, n_j) \left| \begin{array}{l} i = j \\ \text{or } n_j = Parent(n_i) \\ \text{or } n_i \in Siblings(n_j) \\ \text{or } n_i \in Ancestors(n_j) \end{array} \right. \right\}$$

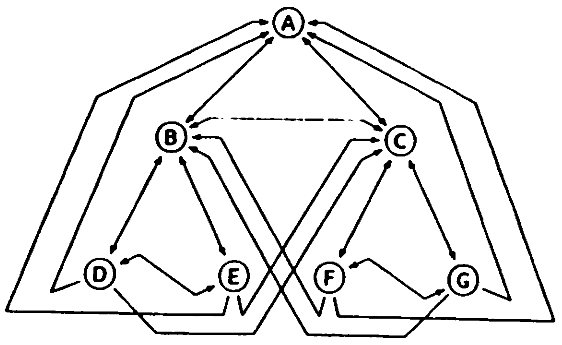


Figure 3: Partial Requisition Graph for Nesting Graph of Figure 2.

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_{pr} of the provision function is the same as the expression defining the set of tuples (n_i, n_j) in A_{re} 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 a dynamic 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 evident 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 that is possible within the framework of the formal 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 $\forall g \in G, \exists r \in R$ such that $vf_m(r)$ request-satisfies and provide-satisfies g. In other words, it should be possible to find a program representation 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.

> *Definition.* A visibility control mechanism m is *requisition-precise* iff $\forall g \in G$, $\exists r \in R$ such that $h = vf_m(r)$ and $h_{re} = g_{re}$.
>
> *Definition.* A visibility control mechanism m is *provision-precise* iff $\forall g \in G$, $\exists r \in R$ such that $h = vf_m(r)$ and $h_{pr} = g_{pr}$.
>
> *Definition.* A visibility control mechanism m is *precise* iff it is both requisition-precise and provision-precise.
>
> *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 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 the requisition relationships or provision relationships cannot be exactly realized. This suggests the following hierarchy of visibility control mechanisms based on preciseness:

$$\begin{tikzcd} & \text{PRECISE MECHANISMS} \arrow[dl, dash] \arrow[dr, dash] & \\ \text{REQUISITION-PRECISE MECH.} \arrow[dr, dash] & & \text{PROVISION-PRECISE MECH.} \arrow[dl, dash] \\ & \text{IMPRECISE MECHANISMS} & \end{tikzcd}$$

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, whose proofs are only sketched in this paper, position these mechanisms in the hierarchy.

THEOREM 1. ALGOL60 is imprecise.

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 requisition 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, 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. From the definition of $rf_{ALGOL60,subprograms}$ 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 requested by, and provided to, other ancestors of the subprograms, thus violating the desired visibility relationships. ☐

[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 language, and therefore no r can be found such that $vf_{FORTRAN}(r)$ request-satisfies or provide-satisfies a g containing a loop in either its A_{re} or A_{pr}.

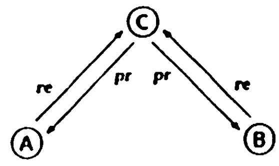


Figure 4: Visibility Graph of a Common Programming Situation.

THEOREM 2. Ada is requisition-precise but not provision-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 [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-nothing 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. 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.[8] 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 representation 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 (sketch). Gypsy does not support any degree of nesting. To control provision, Gypsy employs a construct called

[8]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 [21]. 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.

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 provision-precise. 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. 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, which is an approach to visibility control in large software systems that we have recently developed.[4] In fact, the PIC language framework was developed using the formal model described in this paper; the framework provides support for the explicit specification of both requisition and provision, and thus constitutes a precise visibility control mechanism. Before giving a proof of this, the basic visibility control features of PIC are reviewed.

There are three aspects to the PIC language framework. First, the framework provides a system structure that results in systems that are collections of nest-free modules. Second, the requisition and provision of each entity in a system can be precisely specified. The language features used to capture those two aspects of entity visibility are the *request clause*, for specifying requisition, and the *provide clause*, for specifying provision. Third, the framework provides a module structure that imposes a strict separation of inter-module visibility control information from intra-module algorithmic detail. To realize this separation, a module consists of two distinct parts: a *specification submodule*, which contains a specification of the module's contents and all of the module's *request* and *provide clauses*, and a *body submodule*, which contains the actual code sections of the module. For pre-implementation languages in the PIC family, the body takes the form of a design-language description, while for implementation languages it consists of implementation-language code.

It is our contention that precise visibility control mechanisms are of great potential value to developers and maintainers of large software systems. Such precision would permit the requisition and provision of exactly those accesses desired in a system while disallowing others. In addition, support for both precise requisition and precise provision can result in a redundancy that facilitates more rigorous analysis of the interface relationships of a system's components. For example, based on this view it is possible to formulate complementary descriptions of exactly how two modules are intended to interact, giving one description from the perspective of each of the modules, and then to analyze those interactions by checking the two descriptions for consistency [23].

[4]PIC is an acronym for "Precise Interface Control". Interface control is that aspect of visibility control that deals with inter-module relationships. Because we are concerned primarily with support for programming-in-the-large—the interaction among the modules in a program—we have chosen to concentrate on that aspect in our work.

```
package PrintQueue is
    request LinkedList.( ListElement, List );

  type Job is ...;

  procedure Enqueue ( J : in Job )
      request LinkedList.Insert;

  procedure Dequeue ( J : out Job )
      request LinkedList.Delete
      provide to Printer;

  ...;

private
  procedure Util ( ... )
      request LinkedList.Statistics;

  ...;

end PrintQueue;
```

Figure 5: PIC/Ada Specification Submodule of a Print Queue Package.

The capabilities provided by the language features of the PIC framework are relevant throughout the lifetime of a software system, and appropriate dialects of the features can be developed to make them compatible with a variety of languages, such as design or programming languages. Below, our examples are given in terms of an Ada-flavored dialect, which we refer to as PIC/Ada, and we describe only those features of this dialect that are concerned with one kind of module, namely PIC/Ada's encapsulation unit, the package. Figure 5 presents a simple example illustrating several aspects of the language features as they are realized in PIC/Ada. The example shows the specification submodule of a print queue package implemented using linked lists. The entities provided by the package realize the abstract operations of adding and removing a job from a print queue.

A module's specification submodule completely determines both the entities requested by each entity in the module and the entities provided by the module. In PIC/Ada, the *request clause* is a list of entity names beginning with the reserved word **request**. The *request clause* appearing at the top of the specification submodule in Figure 5 indicates that access to the entities ListElement and List provided by package LinkedList is requested for all the entities in package PrintQueue. The *request clause* attached to Enqueue, on the other hand, indicates that access to the entity Insert, defined in package LinkedList, is requested only for Enqueue, since Enqueue is the only entity in PrintQueue that has an attached *request clause* mentioning that entity. Similarly, the entity Delete, also defined in package LinkedList, is only to be referred to by Dequeue.

The *provide clause* in PIC/Ada is a list of entity names beginning with the reserved words **provide to**. The *provide clause* appended to procedure Dequeue indicates that Dequeue is only provided to the entities of Printer. The absence of a *provide clause* is interpreted to mean that access to the entity is provided to "all", which is the case for Enqueue. Thus, while any module in the system would be allowed to add a job to a print queue, Printer is the only module permitted to remove a job. The entities declared in the so-called *private part* of a package, such as Util, are not provided to any non-local entities.

The *request clause* is more precise and flexible than its counterparts in most other languages, including Ada's requisition construct, the *with clause*, in at least two ways. First, it does not necessarily import all the provided entities of a module but can import subsets of those entities. Second, a *request clause* can be attached to an individual entity of a module so that requisition by the entities within that module can be differentiated.

The *provide clause* is also more precise and flexible than its counterparts in other languages. In Ada, for example, provision is controlled on an all-or-nothing basis; either access to an entity is provided to every module (in a given scope), or it is provided to no module, and so the entity is hidden. While these two extremes are useful (for instance, in describing the global provision of a library module such as a package of trigonometric functions or the hiding of a low-level utility subprogram within the module needed to implement the trigonometric functions) and indeed specified the same way in PIC/Ada as in Ada, the intended provision of a particular entity often lies somewhere in between [14]. Hence, the PIC language features' *provide clause* may be appended to any of a module's entities in order to *selectively* limit their provision to specific non-local entities.

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.

The defaults for requisition and provision in PIC/Ada—that is, the minimum amounts of inter-module visibility control information that must be explicitly supplied by a developer—are designed to mimic the control available in Ada. For instance, as mentioned above, the absence of a *provide clause* on a provided entity is interpreted to mean that access to the entity is provided to "all". A *request clause*, like an Ada *with clause*, is used to import non-local entities, but requisition of all the provided entities of a package need only entail the listing of that package's name in a *request clause*, not each individual requested entity. In general, the PIC/Ada design philosophy, which we believe is in keeping with the spirit of Ada, is that the more control over module interfaces desired by a developer, the more information the developer must specify. For PIC dialects based on languages other than Ada, defaults of tighter control or requirements of more complete information may be appropriate. Thus, for example, it may be preferable in some languages to interpret an absent *provide clause* as meaning provide to "none", rather than provide to "all".

Given this brief review of the visibility control supported by the PIC language framework, we can now illustrate the highest entry in the preciseness-characterization hierarchy.

THEOREM 4. PIC/Ada is precise.

PROOF (sketch). 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. 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 four languages in terms of the preciseness of their visibility control mechanisms, as follows:

$$\begin{tikzcd}
 & \text{PIC/Ada} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{Ada} \arrow[dr, dash] & & \text{Gypsy} \arrow[dl, dash] \\
 & \text{ALGOL60} &
\end{tikzcd}$$

It is interesting to note that no other languages we are aware of reside at the same position in the preciseness-characterization hierarchy as that occupied by the dialects of the PIC language framework. Indeed, the formal model not only facilitates this comparison, but it was also instrumental in leading us to the definition of the *request* and *provide clauses* of the framework. The full benefits of this straightforward, yet powerful, mechanism for controlling entity visibility are described further in [21,22,23].

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 currently under investigation.

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 [18] 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 [18]; it was never intended as a general, descriptive formalism. Moreover, it lacks the useful concept of provision. Lipton and Snyder [12] 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. Osher [16] 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 formal model 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 formal 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 formal 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] **Reference Manual for the Ada Programming Language** (ANSI/MIL-STD-1815A), United States Department of Defense, Washington, D.C., January 1983.

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] H.L. Osher, *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.

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

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

[19] 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.

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

[21] 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.

[22] 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.

[23] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Interface Control and Incremental Development in the PIC Environment*, **Proc. Eighth Inter. Conf. on Software Engineering**, London, England, August 1985, pp. 75-82.

[24] 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).