Modelling and Specifying Name Visibility and Binding Semantics

Scott A. Vorthmann

July 1993

CMU-CS-93-158

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Revised from a draft written in November 1990

Abstract

This paper describes *visibility networks*, a graphical model of static name visibility and binding. A visibility network is a visual representation of the search algorithm performed when binding a name reference to a declaration. In conjunction with an extended attribute grammar mechanism, visibility networks allow clear and precise specification of the naming semantics of programming languages. The power of the model is demonstrated through its description of several examples of complex visibility constraints in Ada. As a specification technique, the model has several advantages for the language designer, including support for prototyping, analysis, formal description, and documentation of naming semantics. Similar advantages make the visibility network model pedagogically attractive. Finally, it has been demonstrated that the specifications can be used to automatically generate naming semantics modules for compilers and language-based editors, reducing the burden of the language implementor.

This research was sponsored by the National Science Foundation under Grant No. IRI-9157643.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: Software Engineering: Coding - Program editors; Programming Environments - Interactive; Programming Languages: Formal Definitions and Theory - Semantics; Language Constructs and Features - Modules, Packages; Logics and Meanings of Programs: Semantics of Programming Languages - algebraic approaches to semantics;

1. Introduction

This paper describes *visibility networks*, a graphical model of static name visibility and binding in programming languages. A visibility network is a visual representation of the search algorithm performed when binding a name reference to a declaration. Section 2 will introduce visibility networks as a way to model the naming semantics of specific programs, then will described how specific networks can be generalized to describe the visibility rules of an entire language. The power of the model will be demonstrated in Section 3, through its description of several examples of complex visibility constraints in Ada. Section 4 will describe an extended attribute grammar notation which, when combined with the graphical visibility network diagrams, allows clear and precise specification of the naming semantics of programming languages.

As a specification technique, the visibility network model has several advantages for the language designer, including support for prototyping, analysis, formal description, and documentation of naming semantics. Similar advantages make the visibility network model pedagogically attractive. Finally, the specifications can be used to automatically generate naming semantics modules for compilers and language-based editors, reducing the burden of the language implementor. These advantages and others will be elaborated in Section 5.

2. The Visibility Network Model

This section will introduce visibility networks, both as a technique for modelling the naming semantics of individual programs, and as a technique for specifying the naming semantics of programming languages. Section 2.1 will present a visibility network that models a simple C program, with very basic visibility semantics. Section 2.2 will then generalize from that example, looking at the same network as a specification of the visibility semantics of a particular type of scope in C.

2.1. Modelling a Specific Program

Figure 1 shows a simple C program containing a single nested block, and a network representation of the visibility and binding among identifier sites (highlighted) in the program. Note first that the set of all identifier sites in the program has been partitioned into four separate sets according to two criteria. The first criterion, declaration vs. reference, does not require an explanation. The second criterion, local vs. global, simply partitions the identifier sites according to the immediately enclosing scope, with *local* referring to the nested block, and *global* referring to the program scope. Each set of identifier sites in the program is associated with a *namesite* in the visibility network (represented by a solid rectangle). A namesite can be looked at as a table of program identifier sites, indexed by the identifier string values.

Both of the criteria mentioned above are essential for isolating the characteristics that determine the treatment of individual identifier sites by the binding algorithm; all identifier sites associated with a particular namesite will be treated in a similar manner. Declaration sites must be unique in a scope (in C), and reference sites must each be bound to a declaration site. Global references can only be bound to global declarations, whereas local references can bind either to local or global declarations, with local declarations preferred. The algorithm that constructs these bindings can be viewed as a search of the available declaration namesites, initiated from a particular reference namesite. This search is visually represented by the network edges connecting the namesites in Figure 1. The implied direction of the search is right-to-left,

```
main ()
{ int x, y, z ;                  /* global declarations */
  z = 3 ; y = z ; x = z ;        /* global references */
  { int y, a, b ;                 /* local declarations */
     y = 4 ;  a = x ; b = z ;     /* local references */
}}
```

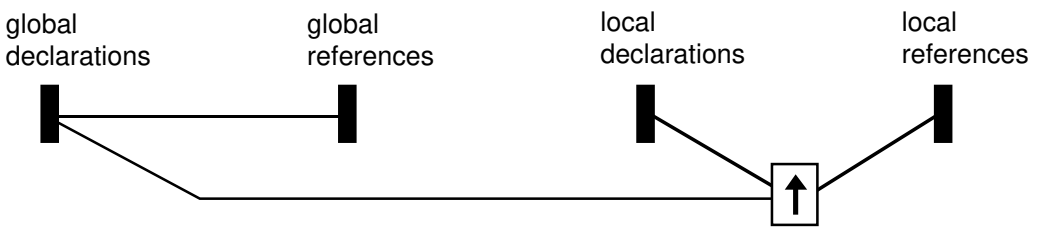

Figure 1: Visibility network model of a C program with a single nested block.

representing reference-to-declaration. This implicit orientation persists in all visibility network diagrams, to avoid confusion.

The visibility rules of a programming language usually require that a particular reference site can be bound to any of several declaration sites, made visible in different ways, with some means of unambiguously selecting among the alternatives. This situation is explicitly represented in the visibility network model by a multiplicity of paths converging on a *selector*, as in Figure 1. The selector in this case is a *preference* selector, indicated by a vertical arrow in a box. This selector orders the outgoing edges according to increasing preference, indicated by the arrow direction. When a binding search reaches the selector, each outgoing edge is searched, starting with the highest-preference edge, until a matching declaration is found. It is therefore clear from Figure 1 that, when binding a local reference site, local declarations are preferable to global declarations. We shall encounter other types of selectors later.

2.2. Modelling Programs in General

The preceding section showed how the visibility semantics of a specific C program could be modelled as a visibility network. If this modelling technique is to be useful in general, it must be possible to associate with *every* C program, in a well-defined way, a network that models that particular program. This section will abstract from the notion of a visibility network model of a specific program, to arrive at the notion of a network diagram as a specification of the set of possible visibility network representations of programs in a particular language.

Clearly, these networks will exhibit regularities of structure corresponding to certain syntactic constructs — exactly those regularities that allow us to write down a natural language description of the visibility rules of C. The overriding majority of such regularities correspond to some notion of a "scope," or a region of the program with more-or-less uniform visibility of declarations. The program of Figure 1 has two scopes; correspondingly, the visibility network model of the program can be adjusted to reflect the boundary between the two scopes, as in Figure 2. Note that the network has also been augmented by an additional preference selector in the global scope, and by connections to additional contained and containing scopes. (These

augmentations have no effect, since the additional connections do not lead anywhere in this example.) With these augmentations, it is easy to see that the two scopes are represented by fragments of the network with identical structure. The term 'scope' will be used to designate a fragment of a visibility network with fixed structure, as well as the region of the program that is modelled by the fragment.

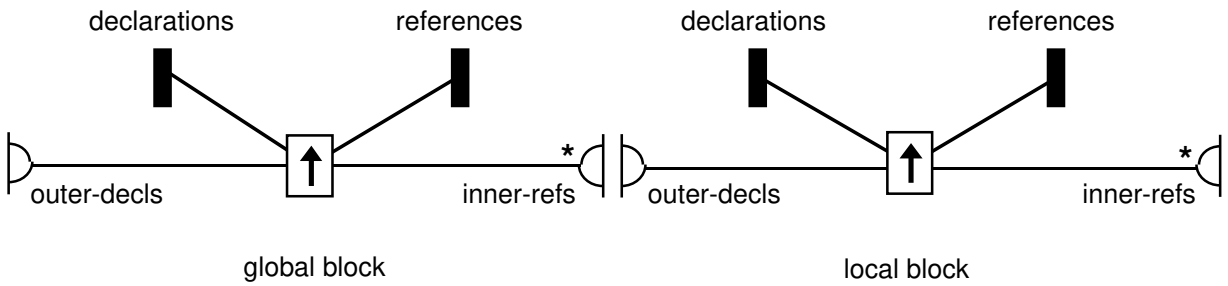

Figure 2: The previous visibility network, subdivided into scopes.

While there are a potentially infinite number of visibility network configurations for programs of a particular language, these configurations must all be composed of scope subnets from a fairly small set of 'types' defined for that language. The network of Figure 2 is composed of two scope subnets of the same type. Notice that now the only distinction between the global namesites and the local ones is the subnet in which they are found. The association between identifier sites in the program of Figure 1 and namesites in the visibility network must now be made in two parts, first selecting a subnet, then selecting a namesite type (declaration or reference). More will be said about this later.

Just as the structures of scope subnets are constrained, so too are the ways in which they can be composed. We will say that two scopes that can be connected meaningfully participate in a *view relationship*, and the connection between them is a *view*. A view consists of connections between one or more pairs of *ports* in the two scope subnets. A port is designated in the figures by a semicircle attached to a vertical bar. The scopes of Figure 2 are participating in a view relationship reflecting lexical containment, and the single view connection is between the ports labelled **inner-refs** and **outer-decls**. Ports will usually be labelled according to what can be found across that connection, outside of the scope. Some ports are designated as *multiple* ports, by the presence of an asterisk (*), as are the *inner-refs* ports in the figure. Such ports essentially can be duplicated as many times as necessary, to accommodate view connections to multiple scopes. For example, a single block scope can contain, and therefore be connected to, an arbitrary number of inner block scopes.

By now, the reader should be able to view Figure 2 either as a network model of the visibility semantics of the program in Figure 1, or as two copies of a scope subnet specification for the C programming language. These *scope type* specifications are actually *class* descriptions, with multiple inheritance between classes possible. This inheritance mechanism turns out to be invaluable for expressing the aspects common to several scope types in a particular language description.

Visibility network diagrams, as a graphical specification technique, form a major part of a complete specification language for describing the naming semantics of programming languages. The remainder of the language, pieces of which are described in later sections, is more conveniently expressed in traditional, textual

fashion. The visibility network specifications can be represented textually as well, if necessary; naturally, they are more understandable in graphical form.

3. More Complex VN Specifications

This section presents several examples of visibility network specifications of the naming semantics of Ada. The purpose of these examples is threefold: 1) to elaborate upon the specification features already introduced, 2) to introduce and explain additional specification features, and 3) to illustrate the power and flexibility of the visibility network model as a specification mechanism. The last purpose is well served by examples from Ada, which has unusually complicated name visibility and binding semantics. This is not to say that more common languages would not benefit from visibility network specifications as well.

3.1. Ada *With*- and *Use*-clauses

The first example concerns the visibility effects of *with*- and *use*-clauses. Briefly, a *with*-clause is attached at the beginning of any compilation unit, and indicates the names of library units (top-level packages and subprograms) that are to be visible within that compilation unit. A *use*-clause makes directly visible the public declarations of the indicated packages, otherwise visible only by explicit selection (dot-notation). A *use*-clause may appear directly after a *with*-clause, or within any basic declarative region. Figure 3 shows the visibility network specification of the **basic-decls** scope type, from which most other scope types in Ada are derived (as described below). The top portion of the network largely repeats the basic structure already seen in previous examples, with declaration and reference namesites, and ports connecting inner and outer scopes.

3.1.1. *With*-clause Visibility

Ignoring for a moment the additional selector boxes and inner-outer connections, let us concentrate on the *with*-clause semantics. Note first the namesite labelled **with**, which is associated with the identifiers that appear in the *with*-clause. This namesite is at once a declaration site and a reference site, since it makes a name visible for other references, but itself must be bound to a declared library unit name; hence the namesite has both an incoming and an outgoing edge. Here we begin to see the generalization of the concept of namesite: there are many more roles that an identifier can take on than just that of declaration or reference. These roles will be defined on a per-scope-type basis, in terms of their effect on visibility and binding.

The Ada language definition [U. 83] stipulates that the library unit names made visible by a *with*-clause are to be visible as though they were declared immediately outside of the scope of the compilation unit. In terms of the visibility network, this behavior would be most directly achieved by inserting a single **with** namesite on the **outer-decls** edge of the outermost scope. However, the presence of body stubs and subunits complicates the situation. Although a *with*-clause can only appear at the outermost scope boundary of a compilation unit, that compilation unit may be a *subunit*, a separately-compiled body of a subprogram, package, or task declared within another compilation unit. In that case, the visibility within the subunit must be as if it appeared lexically in the place of the corresponding body stub, except that any attached *with*-clause should be treated as though it appeared at the beginning of the containing unit. Therefore,

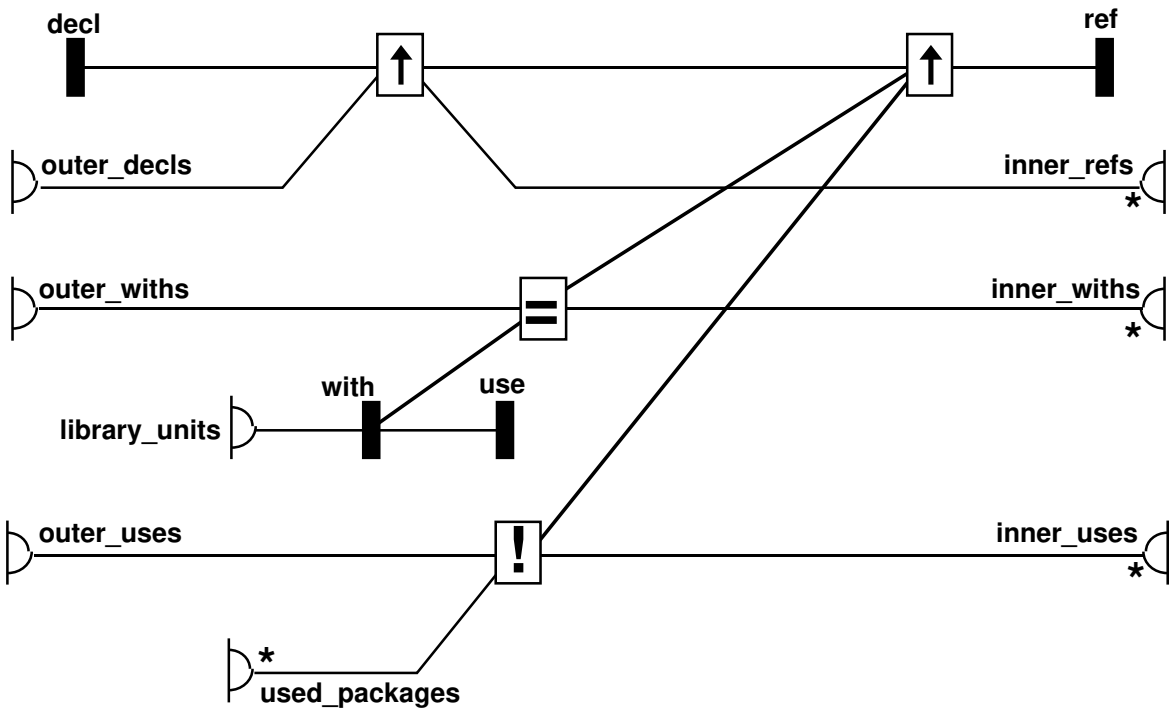

Figure 3: Visibility network specification of Ada's **basic-decls** scope type.

the visibility network model of the subunit is unaffected by its textual separation from its 'outer' scope, except that the **with** names visible in the outer scope must be pooled with those visible from the subunit's *with*-clause. Effectively, any 'body' scope can add its own **with** names to the accumulating pool, regardless of its nesting depth. Local references can then tap that pool, with a lower preference than local and outer declarations, as seen in the second preference selector, simulating the presence of the **with** names outside the outermost scope.

The local **with** names are pooled with those in outer scopes using the 'equal' selector seen in Figure 3. This type of selector indicates that all source paths are to be searched, but successful binding occurs only if all visible identifier sites are the same, or bound to the same declaration. This, too, is consistent with the definition of *with*-clause visibility.

3.1.2. *Use*-clause Visibility

So far we find two separate 'streams' of visible names accumulating as we proceed inward in the scope nesting hierarchy, with the two streams resolving name conflicts differently. A third stream, with yet another method of conflict resolution, is created by the presence of *use*-clauses in the scopes. A *use*-clause does not represent a set of explicit name propagation sites, as does a *with*-clause. The (possibly qualified) names in a *use*-clause are simply reference sites, but each such reference opens a visibility path to some package declaration scope. Note that the references in a *use*-clause at the beginning of a compilation unit must be simple references to package names in the corresponding *with*-clause, hence the special **use** namesite. In terms of the visibility network of Figure 3, each reference in a *use*-clause joins an additional package scope to the **used-packages**

port. We will encounter the other end of this connection later.

The names made visible by all applicable *use*-clauses (including those in outer scopes) must be unique: any name declared in more than one of the 'used' packages is not visible (except by qualification). In the visibility network of Figure 3, this is accomplished by the 'unique' selector, designated by an exclamation point. This selector requires that all source paths be searched, but binding only succeeds if a name is visible along just one of those paths. As with the 'equal' selector, there is no ordering on the outgoing edges of the 'unique' selector. Multiple-source ports, like **used-packages**, can only be connected to selectors with this property.

The pool of names visible through *use*-clauses is tapped for visibility by local references at the lowest preference level. This is to say that any declaration visible from an outer scope can mask a name that would otherwise be made visible by a local *use*-clause; this is consistent with the Ada language definition.

3.1.3. View Relationships

Figure 3 is drawn to suggest that an **inner-withs** port, for example, is always connected to the **outer-withs** port of any inner **basic-decls** scope. However, no formal assertion of that invariant has yet been presented. In the interest of having a modelling and specification formalism that supports analysis of visibility constraints, it is essential that such an assertion be made. This role is filled by *view relationship specifications*, as illustrated in Figure 4.

```
VIEW containment [ basic_decls - * basic_decls ]
    SERVER basic_decls       CLIENT basic_decls :
        inner_refs          -    outer_decls
        inner_withs         -    outer_withs
        inner_uses          -    outer_uses

VIEW export [ * pkg_spec_public - * basic_decls ]
    SERVER pkg_spec_public   CLIENT basic_decls :
        client_uses         -    used_packages

VIEW context [ standard_package - * basic_decls ]
    SERVER standard_package  CLIENT basic_decls :
        context_clause      -    library_units
```

Figure 4: View relationship specifications for the scope type of Figure 3.

Each `VIEW` specification defines a view relationship, such as `containment`, in terms of the way in which participant scope types are connected by that relationship. This is nothing more than a correspondence between ports in the scope types involved. The `CLIENT/SERVER` designations establish the directionality of the connections, and asterisks define whether the connections are one-to-one, one-to-many, *et cetera*. These annotations can be checked against the usage of the ports in the participant scope types. Naturally, any scope type that is a descendent of **basic-decls** can participate in these view relationships. Note that, due

to the naming convention for view ports, the port names in each connection may seem reversed with respect to the client and server scope type names. For now, view relationship specifications will serve to define and constrain the ways in which scope subnet type can be composed into visibility networks. We shall find another use for them later, in Section 4.

3.2. Ada Package Visibility

This section further illustrates the descriptive power of the visibility network model, by presenting VN descriptions of the visibility and binding semantics of Ada packages. There will be less discussion of the mechanics of the networks. The discussion will be limited to describing the semantic effects; the reader is left to convince himself that the networks pictured implement those effects.

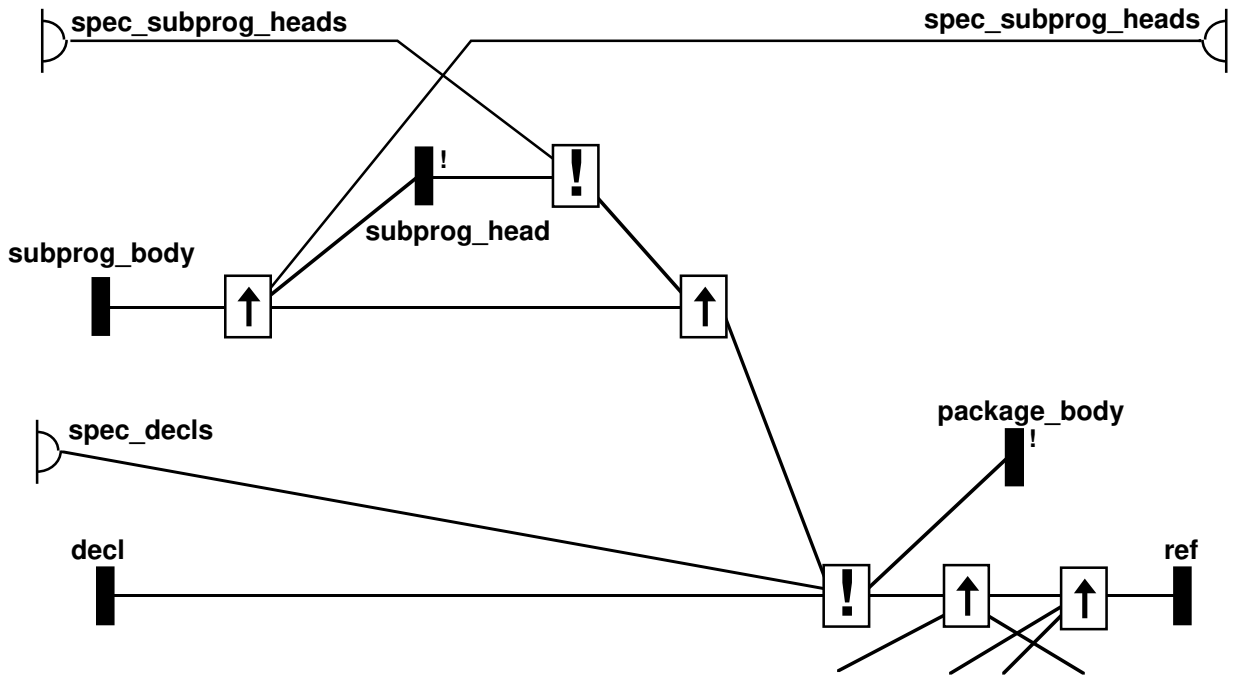

Figure 5: Visibility network specification of the **package-body** scope type.

Figures 5 through 7 depict three scope types, corresponding to the body of a package and the public and private parts of its specification, respectively. Three separate scope types are required to represent the different visibility semantics of the various regions, even though they embody a single 'declarative region' in the terms of the language description [U. 83]. All three scope types are descendents of the **basic-decls** type seen in Figure 3; each is depicted here as an extension of that diagram, which is shown in elided form at the lower right corner of each network.

In a package body scope (Figure 5), we can find nested package bodies and subprogram bodies. If a nested package body is present, the corresponding specification must be found in the current package body scope, or its specification scope. Subprogram bodies can have an optional 'head' declaration, in the current body scope or the specification scope. All declarations in the combined package specification and body scope must be unique, and are equally visible to references in the body scope.

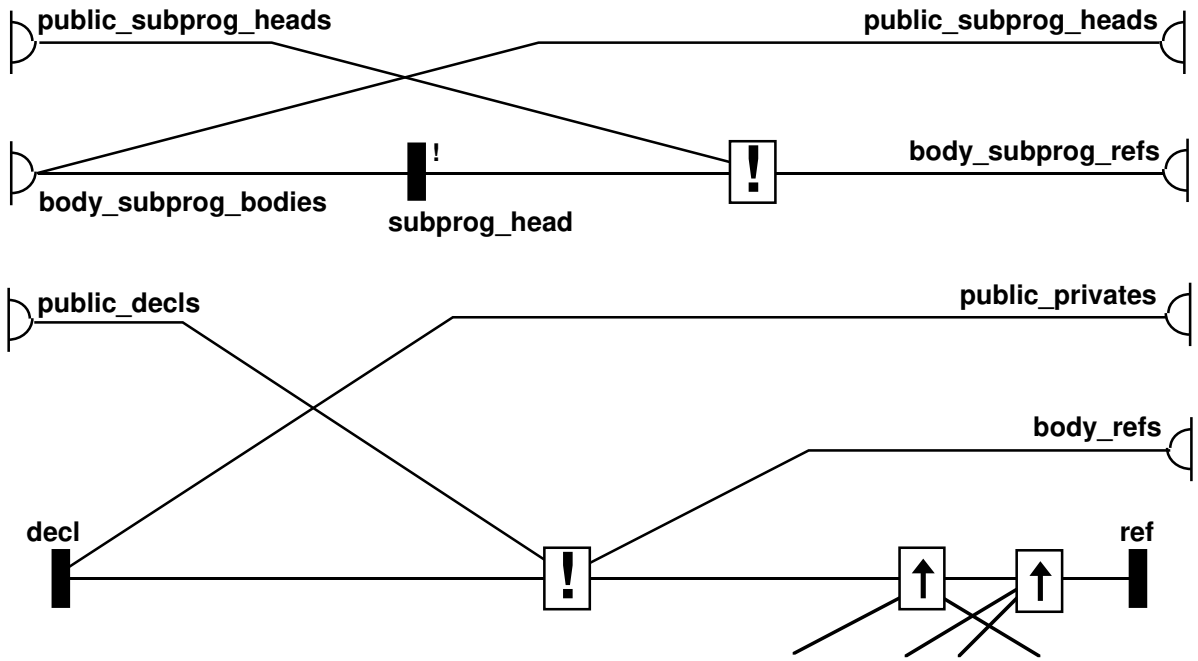

Figure 6: Visibility network specification of the private part of a package specification scope.

The small exclamation point (!) next to the **subprog-head** and **package-body** namesites in Figure 5 indicate that the same identifier may not be duplicated at multiple syntactic sites associated with those namesites. By default, any 'reference' namesite (one having an edge to the left) allows duplications, as does the **ref** namesite. However, nested package bodies and subprogram heads must have unique names.

A package specification is associated with two separate scope subnets, one for the public declarations (Figure 7), and one for the private ones (Figure 6). All declarations in the public part are visible in the private part and the package body. Any private type or deferred constant declarations in the public part must be bound to a full declaration in the private part. Any subprogram head in the public or private part must be bound to a corresponding subprogram body in the package body.

The declarations in the public part of a package specification are visible to any client of that package, whether that client uses qualified references (dot notation) or a *use*-clause to access the declarations. Note that a qualified reference may appear textually within the client scope, yet still be associated with the **client-ref** namesite in the package specification public scope.

Finally, any *with*- and *use*-clauses that apply to the two parts of the package specification must apply to the package body as well. The reader can imagine how the **basic-decls** network, elided in these diagrams, can be augmented with additional view ports to accomplish this.

4. Associating Syntax with VN Semantics

It has been indicated in the preceding sections that a visibility network model can be associated with any program. Each scope subnet is associated with some syntactic construct, and each identifier site in the program is associated with some scope subnet and namesite thereof. However, thus far no mention has been made of how these associations are specified. This will be described in this section.

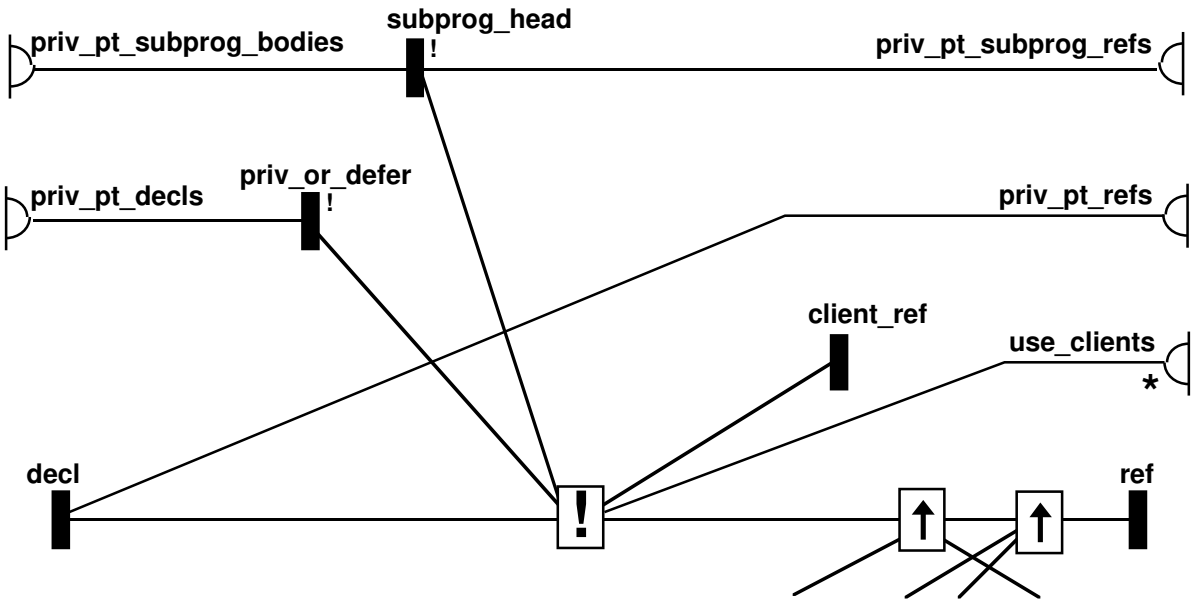

Figure 7: Visibility network specification of the public part of a package specification scope.

For simple naming semantics, like that in the example of Figure 1, the mapping between the syntactic structure and the visibility network model is readily apparent: each scope subnet is associated with a subtree, and identifier sites are associated with the nearest scope along the path to the root. All that remains is to indicate, for each type of identifier site, which namesite to use in the network. In general, however, the situation can be quite a bit more complicated. Since we are concerned only with *static* visibility and binding, the visibility network state can always be derived from the syntax, but the derivation might be complex.

A slightly modified form of *attribute grammar* will prove to be an adequately powerful mechanism for describing the mapping between syntax and naming semantics. Briefly, an attribute grammar is a context-free grammar, augmented by sets of attributes attached to nonterminal symbols, and attribute equations attached to productions. The attribute grammar describes the flow and computation of semantic information on the substrate provided by the abstract syntax tree.[1]

For our purposes, the attribute grammar is used primarily to propagate scope subnets (or pointers to them) around the tree. The sources of these propagations are *scope-defining* productions, and the destinations are *view-defining* productions and identifier terminal sites. A scope-defining production instantiates a scope subnet, with the scope type specified. A view-defining production connects two or more scope subnets according to a particular view relationship. An identifier site installs a pointer to itself in the specified namesite table of the given scope subnet. All scope attributes of nonterminal symbols are typed, so that it is possible to statically check that the namesite in question can be found in that type of scope.

Once a reference site has been installed, and a binding has been constructed, that binding can be used to provide access to attributes of the indicated declaration site. In order to provide this access in a safe way, names and bindings must be given a type that defines a particular attribute signature. These types are called *name types*, and must be distinguished from name*site* types. Common examples of name types

[1] In the complete paper, an appendix will show parts of an AG specification.

include *procedure*, *variable*, *constant*, etc.. Each defines a particular set of *public* attributes, whose values must be determined by equations associated with the declaration production. Often these public attributes are used as sources of scope subnets, as in qualified package references and record field references.

Note that the visibility network specifications are defined independently of the name types defined for the language. However, binding of a reference to a visible declaration is conditional upon the declaration name type satisfying a name type constraint on the reference site. This is the most common version of a more general concept, a *binding constraint*, attached to an identifier site. In the most general case, a binding constraint can be a boolean expression over values of local attributes at the reference site, the name type of the visible declaration, and the values of public attributes of the declaration. Binding occurs only when the constraint is satisfied.

Binding constraints are part of the machinery required to support general overloading of declarations. In addition to binding constraints, visibility network selectors and declaration namesites must be annotated with *collision* and *error constraints*. A collision constraint dictates when two visible declarations collide, making neither visible. An error constraint specifies the conditions under which a collision should produce an error message, and what that error message should be.

Binding, collision, and error constraints can be use to provide declaration-before-use enforcement. This rule is not enforced by the default mechanisms of visibility networks: a scope is a region in the program with homogeneous, uniform visibility. There are many reasons for this default behavior, documented in [Gar87, Vor90]. Declaration-before-use can be enforced by defining an integer-valued "declaration-order" attribute for every nonterminal in the grammar. This attribute is incremented at every declaration, or any other construct that introduces visible declarations. Binding and error constraints can then be used to guarantee that no reference is ever bound to a later declaration.

5. Putting Visibility Networks to Use

The visibility network model offers a number of advantages for language designers, implementors, users, and students. These advantages will be described in this section. They accrue from its visual nature, as well as its precision in describing visibility semantics.

While the advantages can be realized using the visibility network model as a 'chalkboard' or 'pencil-and-paper' technique, they will be more strongly realized when the model is implemented as a set of interactive software tools. One tool will allow prototyping of VN specifications, using a direct manipulation interface. This tool will support abstraction of scope type specifications from specific networks examples, as we did in Section 2.2, as well as network layout assistance, to help produce network diagrams with the most intuitive presentation. (Together, these two capabilities would have reduced by an order of magnitude the amount of time spent by the author in producing the examples for this paper.) Due to the strongly-typed nature of VN specifications, the specification prototyping tool can support analysis of the consistency and completeness of naming semantics specifications. Finally, this tool will have a 'test' mode, in which scopes can be instantiated, connected, and decorated with name instances.

The advantages here for the language designer should be obvious. This tool will have pedagogical applications, as well, finding use to explain visibility in general, or the visibility rules of a specific language, when used in a classroom demonstration setting. Similarly, the prototyping and testing tool can serve as an online reference manual, with specifications of individual languages serving as their own documentation.

Another tool will operate in conjunction with a language-specific editor, allowing the user to display and query the visibility network associated with the program being edited. Various operations will allow the user to explore the source of visibility of particular identifiers, the reasons that a particular declaration is masked, and the correspondence between the network namesites and the identifier sites in the program. This tool will offer its advantages primarily to software developers, and to students of specific programming languages, programming languages in general, language design, and language implementation.

Finally, the visibility network model as a specification technique offers a particularly attractive advantage to language implementors, whether they develop compilers or language-based editors. The specifications can be translated into a set of tables that drive a language-independent naming semantics module. In other words, an implementation of the naming semantics of a particular language can be automatically generated from a visibility network specification. This is being done in the MacGnome project at CMU, where an incremental naming semantics analysis facility has been incorporated into the existing Pascal environment, although the tables are currently hand-crafted rather than generated. The scope and binding information maintained by this 'naming layer' will support not just the error checking facility, but also tools such as a code browser, cross-referencing tools, module dependency analysis tools, and various navigation aids.

6. Related Work

The visibility network model has its roots in the author's thesis work [Vor90], which had similar goals, but used a much more limited, *ad hoc* model of visibility. To the author's knowledge, the visibility network model is a novel approach to the problem. Garrison's *inheritance graph model* of visibility [Gar87] is related in that it has a graphical basis, but that model is much less intuitive with respect to the specification of masking and collision. However, Garrison applies his model to dynamic as well as static visibility and binding features, while the visibility network model makes no attempt to describe dynamic features.

The extended attribute grammar specification presented in Section 4 is quite similar to a number of previous efforts [Hed88, HM88, BC85, MK88, Cap85a, Cap85b], and does not represent a significant new contribution. However, the use of attribute expressions to constrain binding and visibility is unique to the approach described here.

References

[BC85] G. Beshers and R. Campbell. Maintained and constructor attributes. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 34–42. ACM, July 1985.

[Cap85a] M. Caplinger. Structured editor support for modularity and data abstraction. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 140–147, Seattle, WA, June 1985. ACM.

[Cap85b] M. A. Caplinger. *A Single Intermediate Language for Programming Environments.* PhD thesis, Rice University, Houston, TX, 1985. Rice COMP TR85-28.

[Gar87] Phillip E. Garrison. *Modeling and Implementation of Visibility in Programming Languages.* PhD thesis, University of California, Berkeley, CA, December 1987. Tech. Report # UCB/CSC 88/400.

[Hed88] G. Hedin. Incremental attribute evaluation with side-effects. Technical Report LUTEDX(TECS-3019)/1-17/(1988) & LU-CS-TR:88-37, Lund University, Lund, Sweden, 1988.

[HM88] G. Hedin and B. Magnusson. The Mjølner environment: Direct interaction with abstractions. Technical Report LUTEDX(TECS-3018)/1-14/(1988) & LU-CS-TR:88-36, Lund University, Lund, Sweden, 1988.

[MK88] Josephine Micallef and Gail E. Kaiser. Version and configuration control in distributed language-based environments. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 119–143. B. G. Teubner Stuttgart, 1988.

[U. 83] U. S. Dept. of Defense, Washington, D.C. *Reference Manual for the Ada Programming Lanaguage*, January 1983. (ANSI/MIL-STD-1815A).

[Vor90] Scott A. Vorthmann. *Syntax-Directed Editor Support for Incremental Consistency Maintenance.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, March 1990. Tech. Report GIT-ICS-90/03.