An Overview of Door Attribute Grammars

Görel Hedin

LU-CS-TR:94-123

Also published in: Proceedings of CC'94 International Conference on Compiler Construction, Edinburgh, April 1994, LNCS 786, Springer-Verlag.

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

An Overview of Door Attribute Grammars

Görel Hedin

Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Gorel.Hedin@dna.lth.se

Abstract. An extension to attribute grammars is introduced which allows objects and references to be specified as part of a syntax tree attribution. Practical advantages of these grammars include a simpler specification of many problems in static-semantic analysis, including the specification of object-oriented languages, and a highly reduced number of affected attributes after syntax tree modifications. The resulting attributions are space-efficient and allow efficient incremental attribute evaluation in interactive language-based editors.

1 Introduction

Attribute grammars [16] and incremental attribute evaluation is a well-studied technique for implementing interactive language-based editors where static-semantic checking is performed incrementally during editing [20]. The principal idea of using AGs for incremental updates is very attractive: the attribution of a syntax tree is described declaratively , and an incremental attribute evaluator can be derived automatically from the specification. However, as is well known, there are several problems with using standard AGs [2, 3, 5, 10, 11, 12, 13, 14, 21, 24].

One problem is that standard AGs lead to low-level complex specifications for many problems in static-semantic checking. An example of this is the specification of languages with homogeneous name spaces, i.e. where declaration-sites and use-sites may appear in any order . Another example is the specification of languages with advanced scope rules, e.g. object-oriented languages where name analysis depends on the classification hierarchy and not only on block structure.

Another problem is that, even for simple languages, there are common situations where a small syntactic change results in very many affected attributes, i.e. attributes which require new values. This leads to poor performance during incremental evaluation. The typical example of this is the addition of a new global declaration which affects the environment attributes of essentially all nodes in the syntax tree.

In this paper we introduce *Door attribute grammars* , an extension to standard AGs which provides a solution to the above problems by allowing objects and reference attributes to be specified as part of an attribution. Door AGs allow complex problems to be specified in a simpler way than do standard AGs. Furthermore, the number of affected attributes after a syntax tree modification is substantially lower than for a standard AG and the potential for effective incremental evaluation correspondingly higher.

Many other researchers have also suggested solutions to the problems of using AGs for incremental evaluation, primarily to solve the problems occurring for simple block-structured languages, and to some extent modular languages. In contrast, we have addressed the more complex problems arising from specifying object-oriented languages. Our solution is more general than the earlier solutions in that more advanced attributions can be handled.

This work was done within the Mjølner project [17] and a more detailed account of Door AGs is given in the author's thesis [7]. A more elaborate example of Door AGs applied to an object-oriented language is given in [9].

This paper is organized as follows. Section 2 describes the elements of a Door AG specification. Section 3 gives an example of using Door AGs. Section 4 describes an incremental attribute evaluator for Door AGs, based on visit procedures. Section 5 describes a systematic method for constructing the visit procedures. Section 6 discusses our experience with constructing Door AG specifications and evaluators, and compares the approach to standard AGs. Section 7 discusses related work, and section 8 concludes the paper.

2 Door attribute grammars

2.1 Extended syntax trees

Door attribute grammars are based on the view of a syntax tree as a tree of objects where each object is an instance of a *node class* [6]. Rather than expressing the context-free grammar as a set of nonterminals and productions we express it as a set of node classes where superclasses correspond to nonterminals and subclasses to productions. An attributed syntax tree defined by a Door AG consists of three kinds of objects:

- syntax node objects (instances of node classes)
- door objects (instances of door classes)
- semantic objects (instances of other classes)

The *semantic objects* can be used for representing static-semantic structures, for example symbol tables. As we shall see later, it is often advantageous to model the structured attributes used in a standard AG by semantic objects in a Door AG. The *door objects* serve as interface objects between the syntax nodes and the semantic objects in order to encapsulate the non-local attribute dependencies which occur in Door AGs (this is treated in more detail below). Both door objects and semantic objects are introduced by defining them as direct or indirect *part-objects* of syntax nodes. An object denotes its part-objects by means of *static references* (references which cannot be changed to denote other objects). Part-objects and static references have the same semantics as in BETA [19].

The Door AG fragment below shows the introduction of part objects. For any A object a unique D object is created at the same time as the A object, and the

A object can refer to its D object by the static reference x. The A object is said to be the *owner* of the D object.

```
A: nodeclass
{  x: object D;     -- the static reference x denotes an object of class D
}
```

By adding part-objects, the complete set of objects forms an *extended syntax tree*, or *EST*, as shown in figure 1.

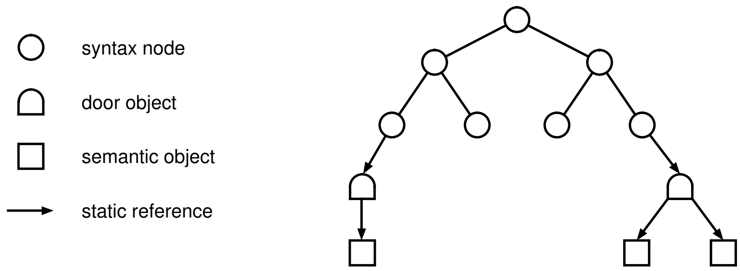

Figure 1 *An extended syntax tree*

2.2 Attributes and equations

All of the objects in the EST can have attributes. As for standard AGs the attributes are defined by equations of the form

$$a_0 := f(a_1, . . a_n)$$

where the attribute a_0 is defined by the side-effect-free function f applied to the attributes $a_1, . . a_n$. Attributes are declared as inherited, synthesized, or local. For syntax nodes, inherited attributes are defined by equations in the father node, whereas synthesized and local attributes are defined by equations in the node itself. Similarly for door objects, inherited attributes are defined by equations in the owning syntax node, whereas the door object itself defines its local and synthesized attributes. Semantic objects have only local attributes and no equations. The attributes of a semantic object are instead defined by equations in its owning door object.

We use the term “inherited” in the sense of attribute grammars, and will use the term *oo-inherited* to mean inherited in the sense of object-oriented programming. Attributes and equations defined in a class are oo-inherited by all its subclasses.

2.3 Reference attributes

Door AGs extend standard AGs by allowing attributes to be *references* to other objects. A reference attribute is declared as follows:

```
r: ref Q;
```

where Q is a class. The reference attribute r is said to be *qualified by* Q, i.e. it may only denote objects of class Q (objects of subclasses to Q are also considered to be Q-objects). The value of r is the *object identity* for a Q-object. Each object has a unique identity which is immutable and not affected by changes to the attributes of the object (the state of the object). Two reference attributes are considered equal if and only if they have the same object identity value, i.e. they denote the *same* object.

Reference attributes can be used to denote node objects, door objects, and semantic objects. To define a reference attribute, it is possible to use static references, self references, and other attributes. The example below shows the use of static references and self references. We use the "this"-notation of Simula [4] for self references.

```
P: nodeclass
{  x: object D;      -- x is a (static reference to a) part object of class D
   ↑rd: ref D;       -- rd is a synthesized reference attribute denoting a D object
   ↑rp: ref P;       -- rp is a synthesized reference attribute denoting a P object
   rd := x;          -- rd is defined to denote the object x
   rp := this P;     -- rp is defined to denote this P object
}
```

By using reference attributes it is possible to propagate a reference from one part to another in the EST. This allows objects to have references to other objects arbitrarily far away in the tree and thereby gives the possibility to define arbitrary directed graphs on top of the EST substrate. Figure 2 shows an example of such a graph. Note that the graphs may be cyclic. This possibility to use objects and reference attributes to define graphs is very powerful. It allows, for example, use sites to be connected directly to declaration sites and vice versa. In the specification of object-oriented languages it allows subclasses to be directly connected to superclasses. It also allows mutually recursive types to be conveniently described as objects containing references to each other .

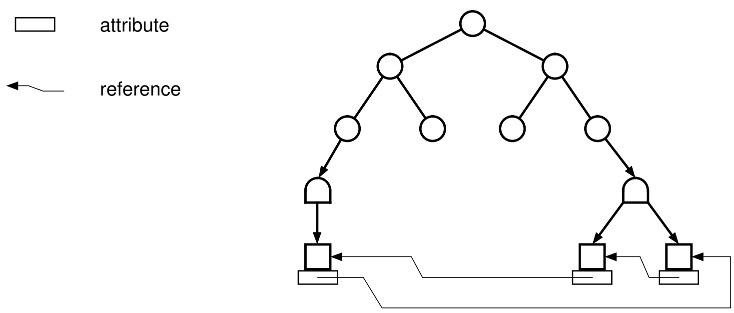

Figure 2 *Graph formed by reference attributes*

2.4 Accesses via references

An object in an EST may be connected to the following objects:

- to its son nodes (in case of a node object) via its son references
- to its part objects via its static references
- to arbitrary other objects in the EST via its reference attributes

The object may access the attributes of these connected objects using the usual dot-notation "r.a", where r is a reference and a is an attribute of the object denoted by r. The access is said to be a *local access* if r is a son reference or a static reference, and a *non-local access* if r is a reference attribute. The use of non-local accesses leads to *non-local dependencies*. Consider the following Door AG fragment:

```
D: doorclass
{ ↓ r: ref Q;        -- r is an inherited reference attribute, denoting a Q object
  ↑ b: integer;      -- b is a synthesized integer attribute
  b := r.a;          -- b is defined using a non-local access
}
```

This grammar defines the attribute b in terms of the non-local access to the attribute a. Thus, there is a dependency from a to b, but since a is located in a Q object which can be arbitrarily far away from the D object in the EST, this is a non-local dependency.

Non-local dependencies are difficult to handle in an incremental attribute evaluator: whenever the attribute a is updated we need to locate the b attribute to update it as well. To make the incremental attribute evaluation practical, Door AGs restrict the use of non-local access to occur only in the door objects. Furthermore, only attributes of semantic or door objects may be accessed non-locally. From this follows an important property:

There are no non-local dependencies involving attributes in the syntax nodes.

Thus, in order to access non-local information, or provide information for non-local access, it is necessary to introduce a door object. This design of the Door AG makes it possible to use standard attribute evaluation algorithms for the syntax nodes, while new algorithms are needed to handle the door objects.

To summarize, the communication of information between objects takes place locally from neighbor to neighbor within the EST, but can also go non-locally from a door object to another door object arbitrarily far away in the EST.

2.5 Handling large attribute values

In standard AGs it is usually necessary to have some attributes with very large structured values. Typically, such attributes are used to describe symbol tables and declarative environments in order to define name analysis. Such large attributes are problematic in many ways. From a specification point of view they are problematic because one needs to introduce auxiliary attributes which are threaded around in the syntax tree to gather all the "small" attribute values that contribute to the large value. This leads to low-level complex specifications. From the point of view of incremental evaluation, the large attributes are also problematic: Using the common evaluation technique of evaluating all attributes which depend on changed attributes, a small change to the large attribute leads to subsequent re-evaluation of all "client" attributes using the large attribute even if most of these client attributes are not affected by the change.

In Door AGs there are two mechanisms for handling the problems of large attribute values:

- Break up a large value by representing it as several small objects
- Define "collection-valued" attributes by membership declarations rather than by equations

To illustrate the first mechanism, consider the definition of a symbol table attribute. In a standard AG, a symbol table might be represented as a set of (STRING, TYPE) pairs:

ST: **set** (STRING $\times$ TYPE)

Each time the type of a declared identifier is changed, the symbol table attribute ST will get a new value. In a Door AG, a symbol table might instead be represented as a set of references to Decl objects

ST: **set** (**ref** Decl)

where each Decl object has STRING and TYPE attributes. Using this definition, a change to the type of a declared identifier does not affect the value of the symbol table attribute – its value is still the same set of references.

To split large values into small objects reduces the number of affected attributes and also the number of attributes which need to be re-evaluated (i.e., attributes which depend on affected attributes). To also simplify the specification, Door AGs have a special mechanism for defining collection-valued attributes, which we describe in the next section.

2.6 Collection-valued attributes

Symbol tables and declarative environments are usually represented by some kind of collection-valued attribute, i.e. a set, bag, sequence, finite function, or similar type. Often, there are many attributes which contribute to the collection-valued attribute independently of each other. But to define the collection-valued attribute in a standard AG, one needs to introduce auxiliary attributes which are propagated around in the tree, gathering the attribute values which should contribute to the final collection-value, leading to a complex low-level specification.

To avoid this, Door AGs have a mechanism for defining collection-valued attributes by so called *conditions* which allow objects to be declared as members of a collection. The collection-valued attribute is placed in a *collection object*, which is a semantic object but differs from ordinary semantic objects in that its attributes are defined by conditions rather than by equations in its owning door object. Collections are similar to the *set* attributes introduced by Kaiser [14] and to the *maintained* attributes introduced by Beshers [3]. See section 7 for a comparison.

The following example illustrates the mechanism. Symbol tables are represented by the class SymbolTable. A door class D1 defines a collection object of class SymbolTable and uses a synthesized attribute tbl to propagate a reference to the symbol table into the syntax tree.

```
D1: doorclass
{ collection myTable: object SymbolTable;
  ↑ tbl: ref SymbolTable;
  tbl := myTable;
};
```

The reference to the symbol table object is propagated using synthesized and inherited attributes through the tree and into zero or more door objects of class D2. A D2 object has a condition reg (for "register") to define itself as a member of the symbol table:

```
D2: doorclass
{ ↓ tbl: ref SymbolTable;
  reg: cond tbl.hasMember(this D2);
};
```

A condition has a boolean expression which must evaluate to *true* in a correctly attributed tree. In the above example, hasMember is a boolean function in class SymbolTable, and the boolean expression tbl.hasMember(**this** D2) will evaluate to true if the D2 object is a member of tbl. To maintain the condition, two operations need to be implemented in class D2:

- evalReg This operation should add the D2 object to tbl, to make the condition expression hold.
- deevalReg This operation should remove the D2 object from tbl, to undo the effects of a previous call to evalReg.

2.7 Constant objects

In addition to node, door, and semantic classes, a Door AG may also contain *constant* semantic object definitions. Such objects are declared globally and are not part of the EST. All their attributes are constant (i.e., they do not depend on any part of the EST). As an example of the use of constant objects, consider representing use-declaration bindings as reference attributes. To handle missing declarations one could declare a constant object noDecl. Each use site could have a reference attribute binding which would normally denote a declaration object in the EST. But in case there is no matching declaration for the use site, the binding attribute would denote the constant object noDecl.

3 An example

Figure 3 shows an example of an attributed EST for the following tiny Algol program:

```
begin
  integer x;
  x := 1;
end;
```

The Door AG specification of the door classes used is given in the appendix. For brevity, our example ignores multiple declarations of the same identifier. See [7] for an example of how that could be added to the specification.

- **BlockDoor** objects are used for extending the syntax tree at each block statement in the program. A BlockDoor object has two semantic part objects: a symbol table object (which is a collection of DeclDoor objects) and a so called "path" object which represents the declarative environment for use sites within the block. The path object has two attributes local and encl which connect the path object to the local symbol table (reference 1) and to the enclosing declarative environment (reference 2). In

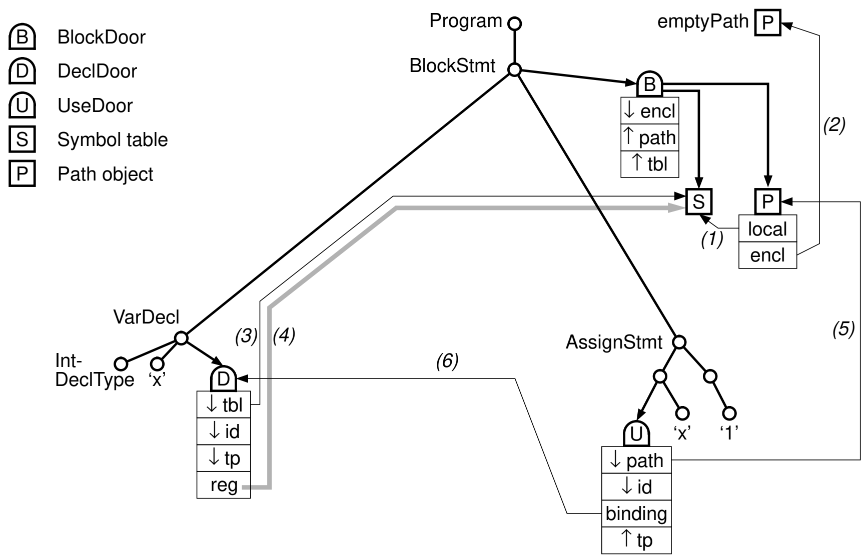

Figure 3 *Attributed EST for a tiny Algol program*

this case, the block is at the topmost level in the program and the enclosing environment is represented by the constant object emptyPath.

- **DeclDoor** objects are used for extending the syntax tree at each declaration. A DeclDoor object declares itself as a member of a symbol table tbl (reference 3) by using a condition reg (membership 4). The tbl attribute is inherited and defined by propagating the synthesized tbl attribute of a BlockDoor to each DeclDoor in the block (these propagation attributes are not shown in the figure).
- **UseDoor** objects are used for extending the syntax tree at each identifier use site. A UseDoor object has an attribute path (reference 5) which represents the declarative environment of the use site. The path attribute is inherited and defined by propagating the synthesized path attribute of a BlockDoor to each UseDoor in the block (these propagation attributes are not shown in the figure).

  The path attribute is used for defining the binding attribute (reference 6) which denotes the matching DeclDoor object. The definition of binding uses a lookup function which traverses the symbol table objects reachable from path. This lookup function performs non-local accesses to attributes in the DeclDoor objects collected by the symbol tables.

3.1 Supporting an object-oriented language

The space here is too limited to give more than a sketch of how object-oriented languages can be supported by Door AGs. For details, we refer the reader to [7] and [9].

We are considering object-oriented programming languages in the style of Simula, C++, and Eiffel which all have similar scope and type rules. The basic constructs which need to be addressed for these languages are *subclassing* (in combination with block structure), *qualified access* (e.g., message sending), and *reference assignments* (doing type checking while taking the hierarchical type system into account). We have successfully specified a small example object-oriented language containing these constructs, and constructed an efficient incremental Door AG evaluator for it.

Subclassing To handle subclassing, we attribute each class with a ClassDoor which is similar to the BlockDoor above. The Path object of the ClassDoor contains reference attributes not only to the local and enclosing symbol tables, but also to the chain of symbol tables of its superclasses. This way, the Path object describes the visibility rules for free use sites occurring inside the class: first look in the local symbol table, next in the chain of superclasses, and finally according to the Path of the enclosing block. Methods inside the class are also attributed with a door similar to BlockDoor and have a Path object combining the local symbol table of the method with the Path of the enclosing class. Methods are registered as members of the enclosing class' symboltable using DeclDoor objects, in the same way as is done for variables in the Algol example.

Qualified access To handle qualified access, each class is (via its ClassDoor) attributed with a RefType object which represents the type of references qualified by that class. The RefType object is connected to another Path object, qualPath, which describes the chain of symbol tables of the class and its superclasses. Consider a message-send "r.m" where r is a reference qualified by class C and m is a method in C. The Path object describing the environment for m is then tp.qualPath, where tp is the type of r (i.e., a RefType object). UseDoor objects are used for binding both r and m to their appropriate declarations.

Reference assignment In a reference assignment "r1 := r2", the qualifications of r1 and r2 must be compared, taking the type hierarchy resulting from subclassing into account. To support this, each RefType object is connected to the RefType object of the corresponding superclass. However, these connections may change if the user changes the class hierarchy in the program. The comparison is thus dependent on non-local information, and therefore embedded in a CompareDoor. The syntax tree propagates the reference types of r1 and r2 into the CompareDoor, and obtains the result of the comparison as a synthesized attribute of the door.

4 Incremental attribute evaluation

We have developed a systematic technique for constructing efficient incremental attribute evaluators for Door AGs. The evaluation is driven by visit procedures which are added to the node classes and door classes in the grammar. A visit procedure evaluates attributes and calls visit procedures of other objects in order to propagate the evaluation according to the attribute dependencies.

4.1 Main grammar and door package

We use the terms *main grammar* to refer to the set of node classes, and *door package* to refer to the set of door and semantic classes of a Door AG.

From an implementation point of view, the main grammar is very similar to a standard AG. Although it differs from a standard AG by allowing reference attributes, it contains no non-local dependencies, and the reference attributes can therefore be treated just like any other attributes in the dependency analysis. This allows the visit procedures for the main grammar to be constructed automatically, using standard AG methods. A door object can be treated as a special kind of son node since a syntax node communicates with its door objects in exactly the same way as with its son nodes – using inherited and synthesized attributes.

The visit procedures for the door package are more difficult to construct due to the non-local dependencies present between the door objects. We have developed a systematic method for constructing these visit procedures, but this method involves manual decisions.

The partitioning of a Door AG into a main grammar and a door package is very important from a practical point of view: the part which can be implemented automatically (the main grammar) is isolated from the part which requires manual implementation (the door package). This allows door packages to be viewed as tool boxes which extend standard AGs. Advanced facilities for common problems in static semantics can be implemented in a door package which can be used by many main grammars describing different languages.

4.2 Evaluator architecture

The attribute evaluator is implemented as a global object with operations to be called by the editor . Basic operations are: replace a subtree, insert/delete a sublist, and evaluate a whole new syntax tree. We will only discuss the replace-subtree operation since the other operations can be seen as special cases of this operation.

The evaluator starts the attribute evaluation by calling visit procedures in the syntax nodes and door objects. Syntax nodes propagate the evaluation by calling visit procedures of their neighbors in the EST . Door objects may call visit procedures of other door objects, arbitrarily far away in the EST , in order to propagate the evaluation along non-local dependencies.

The evaluator keeps a worklist of non-locally dependent doors. When evaluation propagates to a non-locally dependent door , appropriate attributes and conditions in that door are re-evaluated, but the evaluation is not immediately propagated into the owning syntax node. Instead, the door is put on the worklist and the evaluation at this site is resumed at a later stage in the evaluation. This ensures that the different evaluation threads do not collide (i.e., it is ensured that a visit procedure is never called in an object where there is already an active visit procedure).

The evaluation after a subtree replacement proceeds in the following four steps. During each of these steps, the evaluation may propagate to non-local dependent doors which are then put on the evaluator 's worklist.

1. **Exhaustive de-evaluation** The conditions in the doors of the replaced subtree are de-evaluated. I.e., objects in the replaced subtree which are members of collection objects are removed from those collections.
2. **Exhaustive evaluation** All attributes and conditions in the inserted subtree are evaluated.
3. **Local incremental evaluation** Incremental evaluation proceeds in the syntax tree, starting at the successors of the synthesized attributes of the root of the inserted subtree.
4. **Non-local incremental evaluation** For each door on the worklist, the evaluation is propagated into the owning syntax node, and from there further on into the tree.

4.3 Visit procedure protocol

The different evaluation steps make use of different visit procedures as shown in figure 4. Currently , we use a simple 1-visit algorithm for the main grammar , but this could easily be generalized to any standard AG algorithm. Below , we summarize the tasks of the different visit procedures.

- d.**exhDeEvalVisit** De-evaluates all the conditions in the door object d.

- n.**exhVisit** Evaluates all the equations in the node n.
- d.**exhEvalVisit** Evaluates all equations and conditions in the door d.

- n.**incDoorVisit**(d) Re-evaluates equations in node n which depend on the synthesized attributes of its door d.
- n.**incSonVisit**(s) Re-evaluates equations in node n which depend on the synthesized attributes of its son node s.
- n.**incFatherVisit** Re-evaluates equations in node n which depend on the inherited attributes of n.
- d.**incOwnerVisit** Re-evaluates equations and conditions in door d which depend on inherited attributes in d.

- d.**deEval**$_L$, d.**eval**$_L$ This pair of door procedures models a non-local visit to a door d from another door . They de-evaluate and evaluate equations and conditions according to a given non-local dependency labelled L.

5 Construction of visit procedures

We now show in some detail how the visit procedures for a Door AG are constructed. The full details are available in [7].

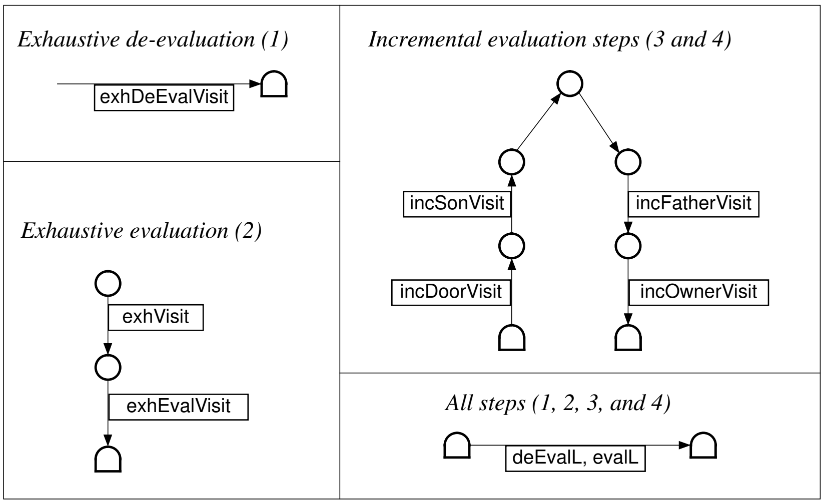

Figure 4 *Visit procedure calls*

5.1 Main grammar visit procedures

As mentioned earlier, the visit procedures for the main grammar can be constructed automatically from the grammar. In our implementation, the visit procedures implement a 1-visit evaluator (i.e. similar to an Ordered AG [15] but requiring only one visit to each node). As discussed in [7], it is possible to adapt any standard AG algorithm to the main grammars, in order to handle more advanced dependencies. This would imply merging of the exhaustive evaluation and local incremental evaluation steps. However, we have found that for our example languages we need only a 1-visit evaluator, even though the languages we have implemented would require general Ordered AGs, had the semantics been defined using a standard AG. This is because the use of references and objects in Door AGs reduces the need for complex local attribute dependencies.

5.2 Door dependency graphs

The construction of the visit procedures for the door package follows a systematic method where a dependency graph is constructed for each door class. Special *send* and *receive* vertices are added to represent the outgoing and incoming non-local dependencies. For each send vertex, a function is implemented which returns the actual set of dependent door objects. In order to implement these functions efficiently one may add so called *dependency attributes* to the door classes. These attributes are defined using equations or conditions, just like ordinary attributes, but their purpose is to make the incremental evaluator run faster. Time/space trade-offs can be made by choosing different dependency attributes.

Figure 5 shows dependency graphs constructed for our example door package. Receive vertices are added to represent the incoming non-local dependencies resulting from non-local attribute access. For example, the attribute tp in UseDoor is defined by a non-local access binding.tp. This dependency is represented by the receive vertex tpChanged. Similarly, a receive vertex lookupChanged is added to represent the non-local dependencies in the definition of the binding attribute.

For each receive vertex one or many send vertices are added, to represent matching outgoing non-local dependencies. In the DeclDoor graph, a send vertex (tpChanged, UseDoor, fUses) is added. Here, fUses is a dependency function which computes the set of UseDoors affected by a change to the tp attribute of the DeclDoor. To be able to compute this set efficiently at evaluation time, we need to add dependency attributes (see the appendix). We have added a collection uses to the DeclDoor which collects all UseDoors whose binding denotes the DeclDoor. This collection is defined by a new condition cUses in UseDoor.

To handle the lookupChanged dependency, we partly rely on the uses collection, but to efficiently handle the case of inserting a new declaration, we also add a collection attempted to the symbol tables. It collects UseDoors that have attempted to find a declaration for a given identifier in the symbol table, and is defined by the condition cAttempted in UseDoor.

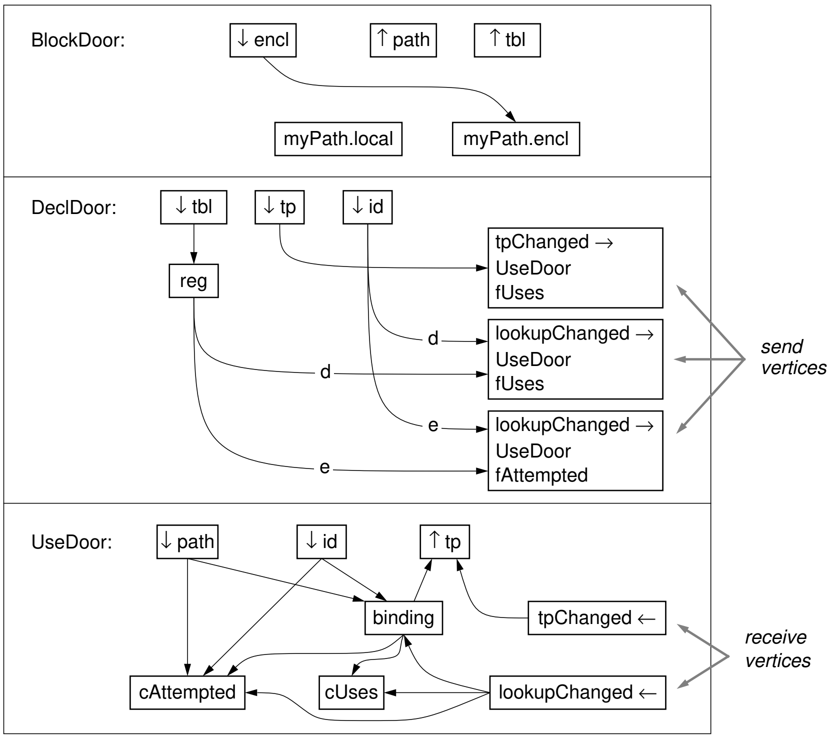

Figure 5 *Dependency graphs*

De-evaluation and evaluation edges A dependency graph edge (x,y) indicates that a change to x may affect y. An edge labelled d indicates that only the de-evaluation of x affects y. For example, id's outgoing d-labelled edge indicates that if the attribute id is changed, it is the absence of its old value which affects the UseDoors computed by uses. The outgoing e-edge indicates that the presence of the new value affects the UseDoors computed by fAffected.

Fix attributes In order to simplify the implementation of a door package, synthesized and inherited attributes of the door classes may be declared as *fix*. This indicates that the specification must be such that the attribute value will never be affected by modifications to the syntax tree, assuming that subtree replacement or list insertions/deletions are the only legal syntax tree modifications. This allows the dependencies from fix attributes to be ignored for incremental evaluation, and results in simpler dependency graphs and simpler door visit procedures. For example, we have declared the attribute encl of BlockDoor as fix. This allows us to ignore the effects of a change to this attribute, which explains why there are no send vertices in the dependency graph for BlockDoor.

5.3 Door visit procedures

Once the door dependency graphs have been constructed, the construction of the door visit procedures is straight-forward. For each of the different procedures (exhDeEvalVisit, exhEvalVisit, incOwnerVisit, $\mathsf{deEval}_L/\mathsf{eval}_L$) a characteristic subgraph of the dependency graph is considered, and the visit procedure is constructed according to the following basic outline:

1. Compute the sets of dependent doors according to the send vertices.
2. Call deEvalL for each dependent door (L is the appropriate send vertex label)
3. De-evaluate local conditions
4. Evaluate local conditions and equations
5. Call evalL for each dependent door
6. Add the dependent doors to the evaluator's worklist

For example, to construct the exhDeEvalVisit procedure one considers a characteristic subgraph containing all the condition vertices, none of the local equation vertices, and only those send vertices which have an incoming d-edge reachable from a condition vertex.

6 Practical experience

We have specified and implemented Door AGs for both block-structured and object-oriented languages with homogeneous namespaces. In addition, an earlier variant of the technique was used for implementing the incremental static-semantic analyzer for Simula in the Mjølner Orm environment [18]. Both Orm and the Door AG evaluators are implemented in Simula and run on SUN SPARC stations. Below, we summarize our experience by comparing our example Door AGs with corresponding standard AGs.

Number of attributes The number of attributes is about the same for Door AGs and standard AGs. The Door AGs have additional dependency attributes which are not present in the standard AGs. On the other hand, the standard AGs have additional auxiliary attributes used to compute the symbol tables.

Number of affected attributes For changes to declarations, the number of affected attributes in our Door AGs is proportional to the number of affected use sites, i.e. use sites which need to be rebound or re-typechecked. For the standard AGs, the number of affected attributes is much larger and grows with the size of the syntax tree. For other changes, the number of affected attributes is about the same for Door AGs and standard AGs.

Attribute dependencies The main grammars for our Door AGs have only 1-visit dependencies whereas the corresponding standard AGs are Ordered AGs. The reason for this is of course that the standard AGs build up the symbol table by using auxiliary attributes which in effect correspond to the passes of a batch compiler. The Door AGs use collection objects instead.

By considering also the non-local dependencies, Door AGs may have circular dependencies, which indeed our Door AGs have. However, although the grammar as a whole is circular, the simple and efficient 1-visit evaluation technique can still be used for the main grammar. The circular evaluation is handled during the non-local incremental evaluation (step 4) by the iteration over the worklist. A non-locally dependent door on a cycle may be added multiple times to the worklist during this iteration.

Space consumption In our Door AG implementations, we have made heavy use of *demand attributes*, i.e. attributes whose values are not stored, but instead computed each time they are accessed. As a general rule, we store only those attributes where something non-trivial is computed, whereas all attributes defined by copy rules are implemented as demand attributes.

The resulting space consumption for the Door AGs is very low, approaching that of commercial hand-coded systems. The Rational Ada system [27] (a commercial incrementally compiling programming environment) is reported to use an average of 35 bytes per syntax node for syntactic and static-semantic information. By assuming a 12 byte overhead per object for the implementation language, [1] we have calculated the space required for our Door AGs to be an average of 60 bytes per syntax node. (This includes the dependency attributes.) The space consumption in our actual implementation is higher, due to a higher object overhead in our implementation language.

Efficiency of evaluator Optimality of incremental attribute evaluators is usually defined in terms of the number of attributes re-evaluated as compared to the number of actually affected attributes [20]. Using this criterion our Door AG evaluators are close to optimal for normal programs. We have some suboptimality due the following factors:

- Demand attributes
- Dependency functions which sometimes locate a few too many dependents. [2]
- Uncoordinated evaluation threads in the non-local incremental evaluation.
- Updating of dependency attributes

While it is possible to construct pathological programs where these factors do make a difference, it is our experience that they are negligible for normal programs.

For practical purposes the usual optimality criterion is not necessarily a very useful measure. From a system perspective, there are only some of the attributes that are actually interesting, whereas many other attributes are present only in order to define the interesting attributes. The problem is that all these uninteresting attributes are included in the traditional optimality

1. This would cover one pointer to the class template, one for the static link, and one for garbage collection. For an implementation language like C++, our figures would be even lower.

2. This happens in the Door AG for the object-oriented language when the class hierarchy is changed. See [8] for details.

criterion. Thus, an optimal algorithm may then optimally evaluate a lot of uninteresting attributes, which is exactly the problem with the standard AG optimal incremental evaluators.

A more practically interesting optimality criterion is to compare the evaluator performance with the number of interesting attributes which are affected. If we consider all the attributes of the door packages (excluding dependency attributes) to be interesting, our Door AG evaluators are still close to optimal for normal programs. The evaluators for standard AGs which are optional in terms of the usual criterion, are on the other hand not anywhere near optimality using this measure.

From a practical point of view our Door AG evaluators are fast. This is because they are close to optimal, using the interesting optimality criterion, and because they are based on static dependency analysis and visit procedures which have a very low overhead during evaluation. Although our actual implementations leave much to be optimized, we have split-second response-times on changes to global declarations regardless of program sizes (the largest tested programs are around 1000 lines).

7 Related work

Nonlocal productions Johnson and Fischer suggested extending AGs with *non-local productions* [12, 13]. A nonlocal production connects a number of "interface" syntax nodes which may be distant from each other in the syntax tree, and allows attribute values to propagate directly along these connections. For example, type-changes can be propagated directly from declared identifiers to their corresponding uses. However, they do not provide any general technique for updating the non-local productions incrementally, and the technique does therefore not improve on standard AGs in the case of added and removed declarations.

Predefined finite function types Hoover and others [10, 11, 21] have developed mechanisms for improving incremental evaluation without extending or changing the standard AG formalism as such. They provide special evaluation support for a built-in abstract data type for finite functions which is useful for defining symbol table attributes. The technique allows changes, additions, and deletions of declarations to be propagated directly to the affected use sites, thus solving the basic performance problems of standard AGs in the case of simple block-structured languages. However, the finite function values are not first class values and may not be stored as a part of another value. This prevents them from being used in lookup of identifiers whose environment depends on other identifier lookups, which is precisely what is needed to handle subclassing and qualified access.

Collections The collections and conditions used in Door AGs are similar to the *set* and *membership* constructs of Kaiser [14], and to the *maintained* and *constructor* attributes of Beshers [3]. These techniques all allow the distributed definition of a collection-valued attribute. However, the two latter approaches are limited compared to Door AGs because they allow non-local access (declaration site memberships or use site lookups) only if the set/maintained attribute is located in an ancestor syntax node. Since collections in Door AGs are accessed via reference attributes, there are no such restrictions on the location of the collection objects. This is important in order to support subclassing and qualified access, where the symbol tables used in lookup are, in general, not in the use site's chain of ancestor nodes.

Visibility networks Vorthmann has developed a graphical technique called *visibility networks* (VN) for describing name visibility and bindings in programming languages [24, 25]. He has exemplified the power of the technique by specifying complex visibility rules of Ada. The technique has several similarities to Door AGs. The VN language is analogous to an advanced generic door package for name analysis which can be parametrized to support different languages. Vorthmann has also implemented an efficient incremental VN evaluator which is analogous to a door package evaluator. Combining the two approaches seems like a fruitful line of further research. If the VN language can in fact be formulated as a real generic door package it would become a very powerful library component in a Door AG-based system. From the VN perspective, Door AGs would provide an attractive way of formalizing the connections between the VNs and the syntax tree. We are currently investigating these possibilities together with Vorthmann.

Attributed Graph Specifications Alpern et al. have developed a specification technique called *attributed graph specifications* (AGS) which generalizes attribute grammars by supporting the specification of attributed general graphs rather than attributed trees [1]. This is useful if the underlying edited structure is a general graph rather than an abstract syntax tree. Example applications include hardware designs and module interconnection graphs. AGSs and Door AGs thus aim at solving different problems and extend standard AGs in different ways: AGSs by extending the edited substrate from a tree to a graph, Door AGs by extending the domain of attribute values to include references. The graph formed by the reference attributes in a Door AG syntax tree is thus derived from the syntax tree, whereas the graph in an AGS system is constructed explicitly by the user.

Higher-Order AGs In Higher-Order AGs [26, 23], a syntax tree may define subordinate syntax trees as attributes, installed as so called *nonterminal attributes*. The syntax trees of the nonterminal attributes are themselves attributed and may define their own nonterminal attributes, and so on. This scheme is useful for modelling transformations to intermediate languages, macro processing, and many other interesting applications. Door AGs may at first sight seem related to HAGs because the syntax tree is extended by additional objects. However, the existence of the door objects and semantic objects depend solely on the class of their owning syntax node, and do not at all depend on attribute values. Thus, Door AGs is a first-order formalism, and the mechanisms of HAGs are orthogonal to those presented here.

Syntactic references The Synthesizer Generator (SG) supports *syntactic references*, i.e., references to syntax nodes can be used to define attribute values [22]. This is convenient because it allows, e.g., the declaration part of a block's syntax tree to be used directly as a symboltable attribute, rather than building a corresponding structure in the attribute domain. However, an SG syntactic reference stands for a complete syntax subtree, viewed as a structured value. This is fundamentally different from Door AGs where a reference stands only for the identity of an object, and the contents of the object is not included in the reference value. Thus, the number of affected attributes does not decrease by using SG syntactic references, and they cannot be used to construct cyclic structures. Furthermore, the SG syntactic references are considered to stand for *unattributed* subtrees. An extension to view them as *attributed* subtrees was proposed in [23], but the implementational consequences of such an extension were not investigated.

8 Concluding remarks

In this paper we have given an overview of Door AGs and the implementation of incremental attribute evaluators for such grammars. Our experience is that the use of references and objects in the syntax tree attributions is very powerful and greatly facilitates the specification of complex problems, in particular name analysis. Due to limited space we had to focus on a simple example of a block-structured language, but the advantages are even more apparent when specifying more complex languages like object-oriented languages. We sketched how the technique can be used to specify subclassing, qualified access, and type checking of reference assignments. The resulting attributions have a low cost in space and the incremental attribute evaluators we have constructed are fast in practice.

There are many interesting possibilities for future work. One is to work on automatizing the implementation of the door packages, either for arbitrary packages, or for some suitable subcategories. Another important issue is to develop more examples, both in the direction of providing general door packages which are applicable to many different languages, and in the direction of supporting more advanced language constructs, e.g. the virtual classes of BETA [19]. We also believe that the use of references and objects in specifications has a wider general applicability.

Acknowledgments
I would like to thank Boris Magnusson, Bill Maddox, and the anonymous referees for useful comments on an earlier draft of this paper.

References

[1] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Graph attribution as a specification paradigm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp 121–129. Boston, Ma., 1988. ACM SIGPLAN Notices 24(2).

[2] R. A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, Computer Science Division – EECS, Univ. of California, Berkeley, 1989. TR UCB/CSD 89/548.

[3] G. M. Beshers and R. H. Campbell. Maintained and constructor attributes. In *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 34–42, Seattle, Wa., 1985. ACM. SIGPLAN Notices, 20(7).

[4] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. SIMULA 67 common base language. NCC Publ. S-2, Norwegian Computing Centre, Oslo, May 1968. Revised 1970 (Publ. S-22), 1972, and 1984. Swedish Standard SS 63 61 14, 1987.

[5] A. Demers, A. Rogers, and F. K. Zadeck. Attribute propagation by message passing. In *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pp 43–59, 1985. ACM. SIGPLAN Notices, 20(7).

[6] G. Hedin. An object-oriented notation for attribute grammars. In S. Cook, editor, *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89)*, BCS Workshop Series, pages 329–345, Nottingham, U.K., July 1989. Cambridge University Press.

[7] G. Hedin. *Incremental semantic analysis*. PhD thesis, Lund University, Lund, Sweden, 1992. Tech. Rep. LUTEDX/(TECS-1003)/1-276/(1992).

[8] G. Hedin. Incremental name analysis for object-oriented langauges. In [17].

[9] G. Hedin. Using door attribute grammars for incremental name analysis. In [17].

[10] R. Hoover. *Incremental Graph Evaluation*. PhD thesis, Cornell University, Ithaca, N.Y., May 1987. Tech. Rep. 87-836.

[11] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50, July 1986. ACM SIGPLAN Notices, 21(7).

[12] G. F. Johnson and C. N. Fischer. Non-syntactic attribute flow in language based editors. In *Proc. 9th POPL*, pp 185–195, Albuquerque, N.M., January 1982. ACM.

[13] G. F. Johnson and C. N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proc. 12th POPL*, pages 141–151, New Orleans, La., January 1985. ACM.

[14] G. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., May 1985. CMU-CS-85-131.

[15] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[16] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[17] J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object oriented environments: the Mjølner approach*. Prentice Hall, 1993.

[18] B. Magnusson. The Mjølner Orm system. In [17].

[19] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[20] T. Reps. *Generating Language-Based Environments*. MIT Press, 1984.

[21] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language-based editors. In *Proc. 13th POPL*, pages 1–13, January 1986. ACM.

[22] T. W. Reps and T. Teitelbaum. The Synthesizer Generator. *A system for constructing language-based editors*. Springer-Verlag, 1988.

[23] T. Teitelbaum and R. Chapman. Higher-order attribute grammars and editing environments. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 197–208. White Plains, N. Y., June 1990.

[24] S. A. Vorthmann. *Syntax-Directed Editor Support for Incremental Consistency Maintenance*. PhD thesis, Georgia Institute of Technology, Atlanta, Ga., June 1990. TR GIT-ICS-90/03.

[25] S. A. Vorthmann. *Modelling and Specifying Name Visibility and Binding Semantics*. CMU-CS-93-158. Carnegie Mellon University, Pittsburgh, Pa., July 1993.

[26] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 131-145, Portland, Or., June 1989. ACM SIGPLAN Notices, 24(7).

[27] T. Wilcox and H. Larsen. The interactive and incremental compilation of ADA using Diana. Internal report, Rational, 1986.

Appendix

Door package

```
Path: class
{   lookup: ref (DeclDoor) func (id: STRING);
};

BlockPath: class Path
{   local: ref SymbolTable;
    encl: ref Path;
    impl lookup
        := let res := local.lookup(id) in
                if res = none
                then encl.lookup(id)
                else res;
};

emptyPath: object Path
{   impl lookup
        := none;
};

SymbolTable: class
{   state: seq( ref DeclDoor);
    add: proc(d: ref DeclDoor) { ... };
    rem: proc(d: ref DeclDoor) { ... };
    hasMember: boolean func
        (d: ref DeclDoor) := ...;
    lookup: ref (DeclDoor) func
        (id: STRING) := ...;
};

BlockDoor: doorclass
{   ↓ encl: ref Path fix;
    ↑ path: ref Path fix;
    ↑ tbl: ref SymbolTable fix;
    collection myTable: object SymbolTable;
    myPath: object BlockPath;
    myPath.local := myTable;
    myPath.encl := encl;
    path := myPath;
    tbl := myTable;
};

DeclDoor: doorclass
{   ↓ tbl: ref SymbolTable fix;
    ↓ tp: TYPE;
    ↓ id: STRING;
    reg: cond tbl.hasMember(this DeclDoor);
};

UseDoor: doorclass
{   ↓ path: ref Path;
    ↓ id: STRING;
    ↑ tp: TYPE;
    binding: ref DeclDoor;
    binding := path.lookup(id);
    tp :=   if binding = none
            then unknownType
            else binding.tp;
};
```

Dependency attributes and functions

```
addto SymbolTable
{   collection attempted: object
    {   state: set(STRING × set (ref UseDoor));
        hasMember: boolean func (id: STRING, u: ref UseDoor) := ... ;
        attemptsAt: set(ref UseDoor) func (id: STRING)
            := ... returns the set of UseDoors associated with id;
    };
};

addto DeclDoor
{   collection uses: object
    {   state: set (ref UseDoor);
        hasMember: boolean func (u: ref UseDoor) := ... ;
    };
    fUses: set (ref UseDoor) func := uses.state;
    fAttempted: set (ref UseDoor) func(id: STRING) := tbl.attempted.attemptsAt(id);
};

addto UseDoor
{   cUses: cond
        if binding ≠ none
        then binding.uses.hasMember(this UseDoor)
        else true;
    cAttempted: cond
        the expression t.attempted.hasMember(id, this UseDoor) is true for each symbol
        table t occurring on path before the symbol table where binding is found (or for all the
        symbol tables on path in case binding = none).
};
```