Extending Attribute Grammars to Support Programming-in-the-Large

Josephine Micallef
Bellcore
MRE 2D-388
445 South Street
Morristown, NJ 07960
(201) 829-4227
Fax: (201) 829-5981
micallef@bellcore.com

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
(212) 939-7081
Fax: (212) 666-0140
kaiser@cs.columbia.edu

CUCS-003-93
30 March 1993

Abstract

Attribute grammars add specification of static semantic properties to context-free grammars, which in turn describe the syntactic structure of program units. However, context-free grammars cannot express programming-in-the-large features common in modern programming languages, including unordered collections of units, included units and sharing of included units. We present extensions to context-free grammars, and corresponding extensions to attribute grammars, suitable for defining such features. We explain how batch and incremental attribute evaluation algorithms can be adapted to support these extensions, resulting in a uniform approach to intra-unit and inter-unit static semantic analysis and translation of multi-unit programs.

An Ada-specific extended abstract of this paper appeared as ‘‘Extending the Mercury System to Support Teams of Ada Programmers’’ in *1st International Symposium on Environments and Tools for Ada*, Redondo Beach CA, April 1990, pages 1-12. Special issue of *Ada Letters*, 11(3), Spring 1991.

keywords: attribute evaluation, attribute grammar, include files, programming-in-the-large, separate compilation.

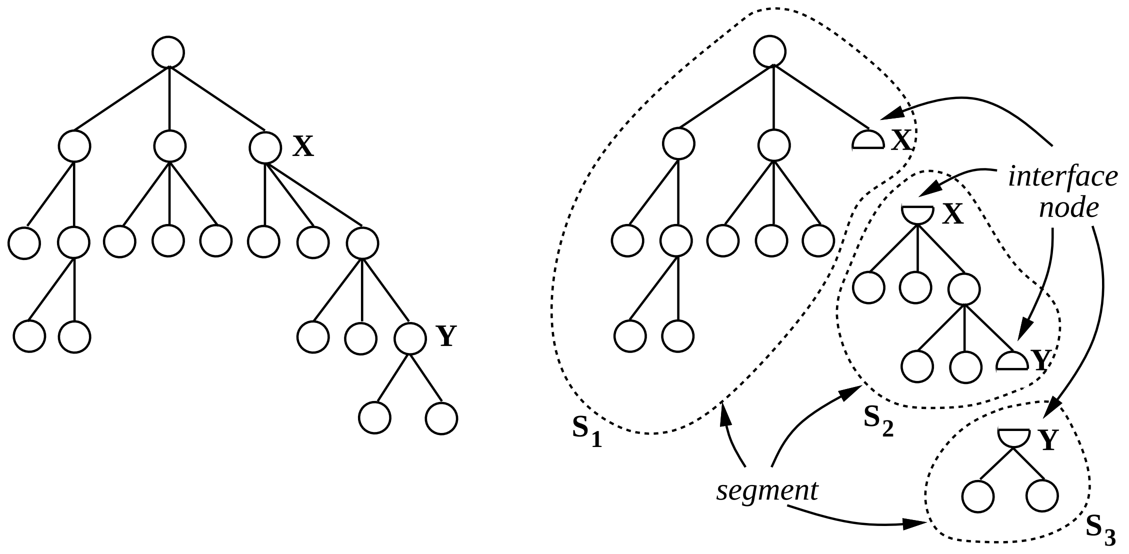
1. Introduction

The attribute grammar formalism for specifying the static semantics of monolithic programs is well-understood, and has been used extensively as the basis for generating static semantic analyzers and translators (*e.g.*, [9, 34, 40, 7, 17]). However, attribute grammars cannot express the inter-module semantics of programs that are composed of a number of modules, such as Ada library and secondary units or C source and header files. Attribute grammars written for languages with programming-in-the-large facilities, such as Ada, define only the intra-module semantics [38]. Normal English prose is typically used to describe inter-module semantics, preventing automatic generation of inter-module analysis tools. Module interconnection languages (*e.g.*, [5, 36, 10, 41, 32]) have been developed to specify inter-module connections and implement interface checking. It is not clear, however, whether such approaches can be seamlessly integrated with the analyzers generated from attribute grammars.

We propose instead to unify inter-module and intra-module static semantic specification and analysis by *extending* attribute grammars to express the programming-in-the-large constructs found in real programming languages, including textual inclusion, multiple kinds of compilation units, and nested compilation units. For example, in Ada, the inter-module static semantic analysis concerning naming involves checking that (1) the names of library units are distinct, (2) there is a subprogram body for each subprogram specification, (3) there is a package body for each package specification that requires one, (4) there is a subunit for each declared body stub, and (5) the names of all subunits that have the same ancestor library unit are distinct. It is not possible to specify any of these five requirements in the classical attribute grammar formalism because there is no way to express the structure of a multi-unit program or constraints across multiple units.

In our extended attribute grammar formalism, a program that is composed of multiple units is represented by a *segmented derivation tree*. A segmented derivation tree is a derivation tree that is decomposed into segments at some of its nodes, where each segment represents a single unit. For example, figure 1(a) shows a derivation tree that is decomposed into segments at the nodes marked *X* and *Y*, resulting in the three segments S_1, S_2 and S_3 shown in figure 1(b). A derivation tree node on the boundary of two segments is called an *interface node*, and is replicated in both of the segments. The segment where the interface node appears as a leaf node is called the *parent segment* with respect to that interface node; the segment where the interface node is the root node is called the *child segment*. Two segments connected at an interface node are called *adjoining segments*. The segment S_1 might represent an Ada package specification, S_2 might be the corresponding package implementation, and S_3 the body of a subprogram developed as a separate subunit.

The structure of the derivation tree representing a program is specified by the context-free grammar describing the syntax of the language in which the program is written. However, context-free grammars are not sufficiently expressive, and cannot directly specify segmented derivation tree structures. We extend CFGs to



(a) **(b)**

Figure 1: *Segmented Derivation Tree*

- Denote which nonterminal symbols label interface nodes, and
- Allow the specification of the following types of segment interconnections:
    - Unordered collection of segments; *e.g.*, Ada library units, C source files, Modula-2 modules.
    - Included non-shared segments; *e.g.*, Ada subunits.
    - Included shared segments; *e.g.*, C header files.

We extend the definitions of attributes, and the operators available for use in their semantic equations, to employ extended context-free grammars and express the static semantic properties of segment interfaces. We describe how existing attribute evaluation algorithms can be extended to *local segment evaluators* for attribute evaluation within segments, and combined with a *global evaluator* for inter-segment linkage and propagation of attribute values across interface nodes. Figure 2 depicts a pattern of attribute flows across segment boundaries, with the attributes of interface nodes replicated in both segments. These attributes might be used to check whether the subunit names within an Ada package are unique.

Due to the potential for shared segments, we in effect relax the underlying tree basis of attribute grammars to a special case of directed acyclic graphs. The graph aspect appears only in regards to inter-segment boundaries — within a segment, the derivation structure is still a tree.

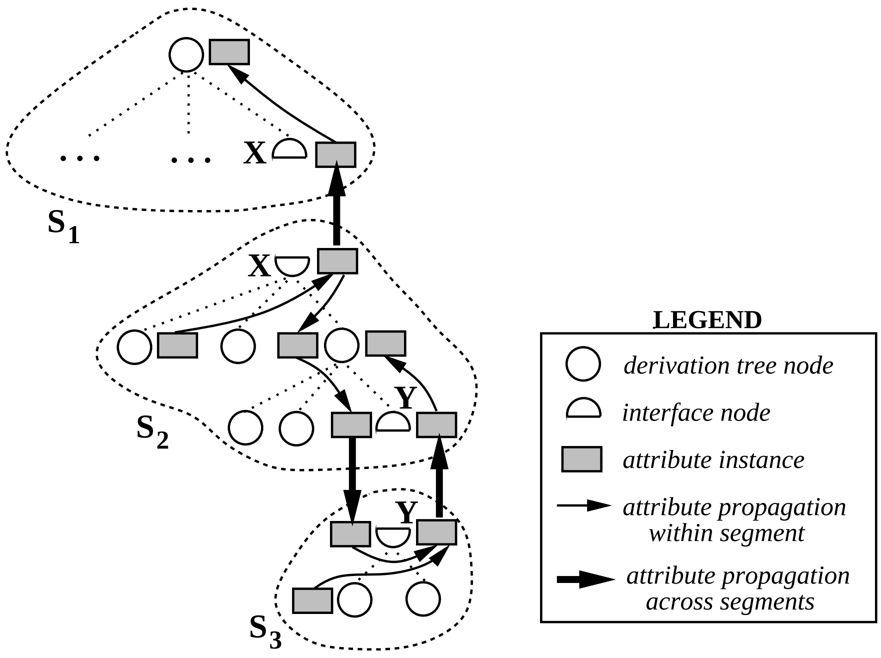


Figure 2: *Segmented Attribute Evaluation*

Therefore, local evaluators can employ conventional attribute evaluation algorithms nearly ‘‘as is’’, and new algorithms are needed only for global evaluators. This is in contrast to the general case of attributed graph grammars, where attribute evaluation is considerably more complex, even for monolithic programs [6]. Our result is a uniform approach to formal specification of both intra-unit and inter-unit static semantic properties, that is, both within and between segments, with the ability to use attribute evaluation algorithms to carry out a complete static semantic analysis or translation of a multi-unit program.

This paper is organized as follows. We first provide background on attribute grammars in section 2. In section 3, we introduce segmentable context-free grammars, illustrating with examples from Ada, Pascal and C. Segmentable context-free grammars provide the underlying substrate for segmentable attribute grammars, which is the topic of section 4. In section 5 we discuss attribute evaluation in the context of programming-in-the-large, particularly the separation of concerns between the local evaluator for each segment and the global evaluator that propagates attribute flows across segment boundaries. Section 6 describes an optimization of the global evaluator that can be achieved for a certain subclass of the segmentable attribute grammars. Section 7 discusses related work. We conclude by summarizing our contributions and briefly suggesting areas for future work.

2. Attribute Grammars

Attribute grammars were introduced by Knuth to describe the context-sensitive properties (static semantics) of programming languages [25]. An attribute grammar (AG) is based on a context-free grammar (CFG) that describes the language's syntax. A context-free grammar is denoted as $G = (N, T, P, S)$, where N and T are finite sets of nonterminal and terminal symbols, respectively, P is a finite set of productions, and S is the start symbol of the grammar. Productions in a context-free grammar are of the form $X \rightarrow \alpha$, where X is a nonterminal and α is a string of symbols from $(N \cup T)^*$.

An AG extends a context-free grammar G by associating a set $A(X)$ of *attributes* with each symbol X in G. Each attribute represents a specific property of the symbol, and can take on any of a specified set of values. The notation $X.a$ indicates that attribute a is an element of $A(X)$. *Semantic equations* defining these attributes are associated with productions of the grammar G. A semantic equation is said to *define* an attribute, a_0, as the value of a *semantic function* applied to other attributes of that production, $a_1, \ldots, a_k$. The attribute on the left side of the equation, a_0, is said to be *functionally dependent* on the attributes on the right side, $a_1, \ldots, a_k$.

Attributes are divided into two disjoint classes: *inherited* and *synthesized*. The inherited and synthesized attributes of a symbol X are denoted by $I(X)$ and $S(X)$, respectively; $I(X) \cup S(X) = A(X)$ and $I(X) \cap S(X) = \emptyset$. A semantic equation defines a synthesized attribute of the left side symbol of a production, or an inherited attribute of one of the right side symbols. The start symbol of the grammar, S, has no inherited attributes, that is, $I(S) = \emptyset$.[1]

The *output attributes* of a production p: $X_0 \rightarrow X_1 \cdots X_n$ are those attributes defined by semantic equations associated with p. These are the synthesized attributes of the left side symbol of p (*i.e.*, $S(X_0)$) and the inherited attributes of the right side symbols of p (*i.e.*, $I(X_1) \cup \cdots \cup I(X_n)$). The *input attributes* of p are those attributes that appear on the right side of the semantic equations of p.

An AG is in *Bochmann normal form* if for any production p, the input attributes of p consist of the inherited attributes of the left symbol of p (*i.e.*, $I(X_0)$), and the synthesized attributes of the right side symbols of p (*i.e.*, $S(X_1) \cup \cdots \cup S(X_n)$) [3]; that is, no attribute defined in p can be used to define another attribute in p. In this paper, we assume that attribute grammars are in Bochmann normal form. Any attribute grammar can be converted to Bochmann normal form, so

[1]In Knuth's original formulation of AGs, terminal symbols could have inherited attributes but no synthesized ones. We follow the approach of later work on AGs and make no such distinction between terminal and nonterminal symbols.

this is not a limiting assumption.[2]

Figure 3 gives an example of an attribute grammar fragment for declarations in a Pascal-like programming language. There are four productions in the context-free grammar, *p1* through *p4*. Each symbol in a production has associated attribute instances (declared in figure 3(a)), and each production has associated semantic equations that define the values of the attribute instances (shown in figure 3(b)). Occurrences of the same symbol within one production are distinguished by the use of a numerical suffix; for example, in production *p3*, there are two occurrences of *Decls*, denoted by *Decls$1* and *Decls$2* for the first and second occurrence, respectively.

The AG of figure 3 builds a symbol table for all declared identifiers, and marks identifiers that are declared more than once as erroneous. The functions *Member* and *Insert*, used in the semantic equations associated with production *p4*, are defined as part of the AG specifications; the definitions of these functions are omitted from figure 3. AG functions are pure functions, that is, they have no side effects. Their only arguments are constants or other attribute occurrences of the production. The expression language used for writing semantic equations includes conditional expressions written with the ternary operator ‘‘? :’’; see, for example, the semantic equation defining *Decl.error* associated with production *p4*.

A *semantic tree* is a derivation tree where each tree node additionally contains fields corresponding to the attributes of its labeling grammar symbol. The *dependency graph* of a semantic tree *T*, denoted by *D(T)*, represents functional dependencies among the attribute instances of *T*, and is defined as follows: *D(T)* is a directed graph, (V, E), where

- $V = \{$ attribute instances of $T \}$, and
- $E = \{ (a, b) \mid a, b \in V$, and b is functionally dependent on $a \}$.

The value of an attribute instance is computed according to its defining semantic equation. Before an attribute can be evaluated, all other attributes on which it is functionally dependent must have already received values. The functional dependencies among the attributes in the AG create a partial ordering on the attribute instances in the tree. Any attribute evaluation algorithm should obey this partial order, but since the ordering is partial, there may be more than one order for evaluating the attribute instances of the tree.

Static semantic analysis and translation of a program can be performed by computing values for the attributes of the corresponding semantic tree. Traditionally in compilers generated from attribute grammars, the error messages and/or object code is stored as the value of synthesized attributes at the root of the tree. A *batch* attribute evaluation algorithm always evaluates all the

---

[2]This conversion may require duplication of semantic functions. For example, if the semantic equations associated with the production *p*: $X_0 \rightarrow X_1\, X_2$ are $X_1.a = f(X_0.a)$, $X_2.a = X_1.a$, the equivalent normal form version would require two invocations of the semantic function *f*, resulting in the semantic equations $X_1.a = f(X_0.a)$, $X_2.a = f(X_0.a)$.

```
Decls:   { synthesized SymTab SymTabOut;
           inherited SymTab SymTabIn; }

Decl:    { synthesized SymTab SymTabOut,
           synthesized string error;
           inherited SymTab SymTabIn; }

Id:      { synthesized string Name; }

Type:    { synthesized Type TpKind; }
```

(a): *Symbols and their Attributes*

```
p1:  Program ::=  · · ·  Decls  · · · ;
         { Decls.SymTabIn = NullTbl; }

p2:  Decls ::=  /* empty rule */;
         { Decls.SymTabOut = Decls.SymTabIn; }

p3:  Decls$1 ::= Decl Decls$2;
         { Decl.SymTabIn = Decls$1.SymTabIn;
           Decls$2.SymTabIn = Decl.SymTabOut;
           Decls$1.SymTabOut = Decls$2.SymTabOut; }

p4:  Decl ::= Id ':' Type ';';
         { Decl.error = Member(Decl.SymTabIn, Id.Name)
                          ? "<-- Variable already declared"
                          : "";
           Decl.SymTabOut = Insert(Decl.SymTabIn, Id.Name, Type.TpKind); }
```

(b): *Productions and their Semantic Equations*

Figure 3: *An Attribute Grammar Example*

attributes in a semantic tree. *Incremental* algorithms, in contrast, avoid reevaluating any attributes whose values are already known to be correct. Incremental algorithms assume that a program's semantic tree is saved between changes to the program, so that it may be efficiently updated rather than completely reconstructed. Batch evaluation is typically used in compiler-compilers, while incremental evaluation is used in interactive language-based editors.

Knuth describes a simple batch attribute evaluation algorithm [25]. The algorithm makes use of the dependency graph of the semantic tree. The vertices of the dependency graph, which correspond to attribute instances in the semantic tree, are first topologically sorted. Then, the attribute instances are evaluated according to their topological order. This algorithm only works if the dependency graph is acyclic [26]. Many attribute evaluators assume that the AG is *noncircular*, that is, that the dependency graph for any string derived by the grammar is acyclic.

Although the algorithm described above is simple and works for any noncircular AG, its performance is poor with respect to both time and space: it is a *dynamic* algorithm, meaning the dependency graph must be built and stored, and all scheduling decisions made, at run-time. More efficient algorithms for performing attribute evaluation have been developed for use in practical compiler-compilers [9, 22, 7]. These algorithms perform most of the work at grammar-analysis time, once for each AG as opposed to once for each program analysis, thus improving the performance of the evaluator. The disadvantage of these evaluators, which are called *static* or *semi-static* evaluators depending on how much work is done at run-time, is that they do not work for all noncircular AGs but only for certain subclasses.

Reps devised an optimal incremental attribute evaluation algorithm for the noncircular AGs [34]. Program edits are treated as subtree replacements. After each subtree replacement, a scheduling graph is generated reflecting the dependencies among the *inconsistent* attributes, which initially are those at the root of the replacement subtree. The attributes represented in this graph are evaluated in topological order, where the scheduling graph is *expanded* when an attribute changes value to include all its dependent attributes. Attributes whose values actually change during the process are said to be *affected* by the edit. This is a dynamic algorithm; static incremental algorithms have also been devised for subclasses of the noncircular AGs [42, 34].

When the program is composed of multiple units, an intermediate position between the incremental and batch methods arises, corresponding to separate compilation. The derivation tree segment for a unit might be evaluated using either an incremental or batch approach, but other units need to be reevaluated only if there is a change in attributes that flow across the unit boundaries. We elaborate on this intermediate attribute evaluation strategy in section 5.

3. Segmentable Context-Free Grammars

We define a *segmentable context-free grammar* (SCFG) as an extension of a context-free grammar, and denote it as $G^+ = (N, T, S, D, E, P^+)$. N and T are finite sets of nonterminal and terminal symbols, respectively; these have the same meaning as for a context-free grammar. The start symbol of the grammar, S, derives the top-level segment in the segmented derivation tree representation of the program. In the typical case where a program is implicitly constructed from an unstructured collection of modules, this top-level segment may correspond to the ‘‘makefile’’ [8] rather than a language construct.

The other components of an SCFG are defined as follows:

- D is a finite set of *nonterminal property declarations*. These declarations are given in the SCFG in either of the following two forms:
  (i) **distributable** X
  (ii) **shared distributable** X

  where $X \in N$ and $X \neq S$. A nonterminal property declaration states that the nonterminal symbol *X* derives a segment; in other words, *X* labels an interface node. If the nonterminal property declaration is in the first form, **distributable**, the child segment derived from *X* interfaces to exactly one parent segment that has *X* as a leaf node. In the **shared distributable** form, a single child segment derived from *X* may interface to several parent segments, each of which has *X* as a leaf node.

In general, we say ‘‘distributable’’ when we mean either shared or non-shared distributable; when we mean specifically non-shared distributable, this should be clear from context.

- *E* is a finite set of *collection constructs* using the **set-of** operator. This construct may be used on the right side of productions in the grammar to describe an unordered collection of segments. The **set-of** operator can be applied to any distributable symbol $X \in N$: **set-of**(X).

- P^+ is a finite set of productions, of the form:

  $Y \rightarrow \alpha$

  where $Y \in N$, and $\alpha \in (N \cup T \cup E)^*$.

In the next sections, we show how segmentable context-free grammars can be written for the languages Ada, C, and Pascal, to indicate how programs in these languages may be divided into segments.

3.1. Ada Example

In an Ada program, a segment corresponds to an Ada compilation unit as defined in the Ada reference manual [1]. We identify three classes of compilation units, and define a different segment type to represent each class. The three segment types are specification segments, implementation segments, and subunit segments. A *specification segment* is either a package specification, a subprogram specification, or a subprogram body that has no corresponding specification; this segment type corresponds to the library units in the Ada reference manual. An *implementation segment* is either a package body or a subprogram body that has a corresponding specification, while a *subunit segment* is a subunit; implementation and subunit segments are collectively called secondary units in the Ada reference manual.

The SCFG in figure 4 defines the three kinds of segments in Ada and their interconnections. (Nonterminals shown in *italics* are defined in the Ada reference manual.) There are three productions defining a specification segment, one for each kind of compilation unit represented by this segment type. A specification segment that derives a subprogram specification must have a corresponding body; this is indicated in production *p2* by the distributable nonterminal *implementation-segment*, which serves as the connection point between this specification segment and the implementation segment representing the subprogram body.

```
p1:  ada-program ::= set-of (specification-segment);

     distributable specification-segment;
p2:  specification-segment ::= context-clause subprogram-declaration implementation-segment
p3:              | context-clause package-declaration [ implementation-segment ]
p4:              | context-clause subprogram-body;

     distributable implementation-segment;
p5:  implementation-segment ::= context-clause subprogram-body
p6:              | context-clause package-body;

p7:  subunit-body-stub ::= body-stub subunit-segment;

     distributable subunit-segment;
p8:  subunit-segment ::= context-clause subunit;
```

Figure 4: *SCFG for Definition and Interconnection of Segments in Ada*

A specification segment that derives a package specification may or may not have a corresponding body; in this case, shown in production *p3*, the distributable nonterminal *implementation-segment* connecting the specification segment to the implementation segment representing the package body is enclosed in square brackets [. . .], indicating that it is optional. A specification segment that derives a subprogram body has no corresponding implementation segment, and thus there is no need for a distributable nonterminal on the right side of production *p4*.

To keep the grammar in figure 4 brief, we do not distinguish between implementation segments that represent subprogram bodies and those that represent package bodies. The result is that interfacing a package body with a subprogram specification, or *vice versa*, is syntactically correct according to this grammar. To disallow this, the semantic analysis specified by the AG could check that a package (subprogram) body is only connected to a package (subprogram) specification, and flag an error otherwise. Alternatively, the grammar of figure 4 could be rewritten so that there are two kinds of implementation segments, one for subprogram bodies and one for package bodies, with the appropriate symbol used in the distributable constructs of productions *p2* and *p3*.

The connection between a subunit segment and its parent is defined in production *p7* in figure 4. Each subunit in Ada, represented as a subunit segment, must have a corresponding body stub. The body stub is a declarative item in the parent segment, which specifies that the body of a subprogram, package or task declared in the parent is to be developed as a separate unit (typically a file). Production *p7* associates with each body stub an interface to the corresponding subunit segment. All occurrences of *body-stub* that appear on the right side of productions in the syntax given in the Ada reference manual should be replaced by the nonterminal *subunit-body-stub*.

3.2. C Example

Figure 5 gives an SCFG for the C language. (Nonterminals shown in *italics* are defined in the C reference manual [23].) A C program is composed of an unordered collection of source files, as specified in production *p1*. We define a source file as consisting of a list of function definitions, data definitions, and include statements. An include statement, such as `#include "foo.h"`, specifies that the contents of *foo.h* are textually included in the source file at the position in which the include statement appears.

This is not ‘‘correct’’ C, however, since C does not have an include statement. Instead, file inclusion, along with macro substitution and conditional compilation, is handled by preprocessing and is deemed outside of the language. We treat the include directive as a statement to be able to employ attribute grammars for semantic analysis at the (preprocessor input) file granularity as opposed to the (preprocessor output) compilation unit granularity; see section 5. Note that we still require that each file (*i.e.*, segment) be preprocessed — but without evaluating the include statements — prior to construction of its derivation tree.

The connection in figure 5 between the segment for the source file containing the include statement and the segment representing the included file is specified by the distributable nonterminal *c-header-file* in production *p8*. This distributable symbol is shared since the file *foo.h* may be included in more than one source file. Production *p9* specifies that a segment that corresponds to an included file is also composed of a list of definitions.

When the shared distributable property declaration is used, the derivation structure may be a directed graph (the edges are directed from parent to child) rather than a tree. Directed graphs complicate the attribute evaluation algorithm, since input attributes for the attributes at the root of the shared segment may have different values in each of the parent segments. The implications are explored in section 5.2. In the sequel, we continue to use the terms ‘‘derivation tree’’, ‘‘semantic tree’’, *etc.*, but they should be understood as including this restricted form of graph.

3.3. Pascal Example

The SCFG for a Pascal program is shown in figure 6. (Nonterminals shown in *italics* are defined in the Pascal report [15].) It specifies that each procedure or function definition in the outermost scope of a Pascal program is a separate segment. (This requires that nested procedures or functions be defined using another nonterminal symbol, not shown in figure 6.) The segments representing outer-level procedures and functions form an ordered collection (or sequence). No new construct is necessary to specify an ordered collection of segments, as this is handled already by recursive productions in CFGs. Modeling procedures and functions as an ordered collection of segments facilitates writing an attribute grammar for the Pascal scope rules to check that a procedure or function is defined before it is used, similar to the AG given in figure 3 in section 2, which checks that an identifier is not declared more than once.

p1: C-program ::= **set-of** (c-file);

distributable c-file;
p2: c-file ::= c-defs;

p3: c-defs ::= /* empty */
p4: | c-def c-defs;

p5: c-def ::= *function-definition*
p6: | *declaration*
p7: | include-statement;

p8: include-statement ::= ’#include’ ’"’ file-name ’"’ c-header-file;

shared distributable c-header-file;
p9: c-header-file ::= c-defs;

Figure 5: *SCFG for Definition and Interconnection of Segments in C*

p1: pascal-program ::= *program-heading label-declaration-part constant-definition-part*
type-definition-part variable-declaration-part proc-or-fun-decls *statement-part*;

p2: proc-or-fun-decls ::= /* empty */
p3: | proc-or-fun proc-or-fun-decls;

distributable proc-or-fun;
p4: proc-or-fun ::= *procedure-declaration*
p5: | *function-declaration*;

Figure 6: *SCFG for Definition and Interconnection of Segments in Pascal*

Let us look in some more detail at the Pascal example to show the flexibility of SCFGs in describing segmented derivation tree structures. A Pascal program in the language specified in figure 6 must have each outermost procedure or function in a separate segment, and no outermost procedure or function may be defined in the main program segment. These two restrictions may be removed by minor changes in the SCFG, as shown in figure 7.

The grammar in figure 7(a) defines outermost procedures and functions in a Pascal program in terms of a list of lists of definitions. A segment in this grammar consists of a list of procedure and function definitions, and thus may contain an arbitrary number of procedures and functions. In contrast, the grammar in figure 7(b) specifies that outermost procedures and functions do not have to be developed as separate segments, but each may be declared either in the main program segment or as a separate segment. This is accomplished by having two different nonterminal symbols for an outermost procedure or function, *proc-or-fun-1* and *proc-or-fun-2*, one of which is distributable and the other not.

A Pascal program that has been segmented according to one grammar (such as the one shown in

```
p1:  pascal-program ::=  . . .  proc-or-fun-decls-list  . . . ;

p2:  proc-or-fun-decls-list ::= /* empty */
p3:          | proc-or-fun-decls-segment proc-or-fun-decls-list;

     distributable proc-or-fun-decls-segment;
p4:  proc-or-fun-decls-segment ::= proc-or-fun-decls;

p5:  proc-or-fun-decls ::= /* empty */
p6:          | proc-or-fun proc-or-fun-decls;

p7:  proc-or-fun ::=  . . . ;
```

(a): *Segments with More than One Procedure*

```
p1:  pascal-program ::=  . . .  proc-or-fun-decls  . . . ;

p2:  proc-or-fun-decls ::= /* empty */
p3:          | proc-or-fun-1 proc-or-fun-decls
p4:          | proc-or-fun-2 proc-or-fun-decls;

     distributable proc-or-fun-1;
p5:  proc-or-fun-1 ::= proc-or-fun;
p6:  proc-or-fun-2 ::= proc-or-fun;

p7:  proc-or-fun ::=  . . . ;
```

(b): *Optionally Distributable Procedure Segment*

Figure 7: *Other Segmentation Structures for Pascal*

figure 6) can be automatically transformed into a program with a different segmentation scheme (such as either of those shown in figure 7). This feature is important since program development often entails changing the modular structure of the program. The attribute grammar writer need only specify the simple segmentation scheme shown in figure 6; the user of the attribute grammar-based system can invoke a tool to restructure the program according to the new SCFG resulting from the transformation. The transformation algorithm is presented elsewhere [30].

4. Segmentable Attribute Grammars

We define the class of *segmentable attribute grammars* (SAGs) to consist of the following components:

1. An underlying segmentable context-free grammar, G^+, which describes the structure of the segmented derivation trees of sentences in the language $L(G^+)$.

2. Attribute declarations, specifying the names and types of the attributes that are associated with each grammar symbol. There are three new concepts: interface

attributes, segment linkage declarations, and conglomerate attributes.

3. Semantic equations associated with productions in G^+. These differ from their counterparts in classical attribute grammars in two ways: completing semantic equations and additional operators.

The attributes associated with a distributable nonterminal symbol are called *interface attributes*. The value of an interface attribute instance a associated with an interface node X are duplicated in the two adjoining segments, one where X is a leaf node and the other where X is the root node. The interface attribute a is defined in the parent or child segment, depending on whether it is inherited or synthesized, respectively.[3]

The *output attributes* of a segment S are those interface attributes of S that are defined by semantic equations associated with production instances in S. These are the synthesized attributes of the root node of S and the inherited attributes of leaf interface nodes of S. The *linkage attributes* of a segment S are the subset of the output attributes of a segment used in its segment linkage declarations. The *input attributes* of a segment S are copies of the output attributes of segments connected to S, that is, the inherited attributes of the root node of S and the synthesized attributes of its leaf interface nodes.

Segment linkage declarations specify which segment instances labeled with the same distributable nonterminal symbol are connected to each other. A segment linkage declaration has the following form:

$X \{ a_i \longleftrightarrow a_s \};$

where X is a distributable nonterminal symbol, a_i is an inherited attribute of X, and a_s is a synthesized attribute of X. The types of the two linkage attributes a_i and a_s must be the same. Two segments with an interface node labeled by the nonterminal symbol X are connected to each other if the values of the attributes a_i and a_s of X are equal. The constraints on the semantic equations defining linkage attributes are described in section 5.1.

Completing semantic equations are required for all interface attributes. Completing semantic equations specify values for interface attributes in one segment when the adjoining segment that defines the attributes does not exist (*i.e.,* it has not yet been created or has been deleted). A completing semantic equation is given as part of the declaration of the attribute. The syntax is:

<Distributable symbol>: { **synthesized | inherited** <type> <attribute name> **default** <value>; }

[3]There may be SAG grammar applications where it is desirable that some attributes associated with a distributable nonterminal symbol be local to one segment, that is, their value is accessible only in the segment in which the attribute is defined. Since this can be achieved by means of *local* attributes [16, 35] — attributes associated with a production rather than with nonterminal symbols — we keep the definition of SAGs simple by having only interface attributes associated with distributable nonterminal symbols.

A new class of attributes, called *conglomerate attributes*, is provided for nonterminal symbols deriving one or more unordered collections. A conglomerate attribute associated with an interface node in a parent segment consists of multiple components, one from each child segment connected to the parent segment through the **set-of** construct.

We introduce a new attribute type for conglomerate attributes: the *finite binary relation*. A binary relation on two sets D and R is a subset of the Cartesian product of D and R, $D \times R$. Every finite relation type declaration must specify one element of R as the $\perp$ element. A binary relation is finite if and only if the set $C = \{(d,r) \mid d \in D,\ r \in R, \text{ and } r \text{ is not } \perp\}$ is finite. All conglomerate values of interest are keyed lists representing binary mappings from a domain (the key) to a range (the information stored for this key). Conglomerate attributes are typically used to form composite symbol tables, with each entry corresponding to the individual symbol table in a multi-unit program.

The following operators are defined on a conglomerate attribute C:

- **assign(*C*, *d*, *r*)**: Assigns $C \cup \{(d,r)\}$ to C. **assign** may introduce multiple entries for the same key d. Typically adds a new module to the symbol table.

- **compute(*C*, *d*)**: If $(d,r) \in C$ and there is only one component in C whose key is d, returns r. If there is more than one component with the same key d, returns the special value μ. If $(d,r) \notin C$, returns $\perp$. Typically looks up a module name in the symbol table.

The **linkedSetSize** and **for each linked** operators are used to determine the number of child segments connected through segment linkage to a given parent segment, and to iterate over the child segments, respectively; these operators are used both for segment linkage and in semantic equations involving conglomerate attributes.

linkedSetSize(*X*) returns the number of segments with root node labeled X that are connected to a parent segment at a leaf interface node labeled X. The argument X must be a distributable nonterminal symbol. The semantics of the **linkedSetSize** operator depends on the form of the associated production instance.

- If the associated production instance is of the form p: . . . $\rightarrow$ . . . **set-of**(X) . . . , **linkedSetSize(*X*)** used within a semantic equation associated with p returns the number of segments X in the unordered collection connected to the parent segment containing production instance p.

- If the associated production is of the form q: . . . $\rightarrow$ . . . X . . . , then **linkedSetSize(*X*)** returns the number of segments derived from X that are connected to the parent segment at the interface node labeled X in the production instance q. (Typically, a value greater than 1 indicates an error.)

The **for each linked *X*[*i*]** construct iterates (in an implementation-specific order) over all segments derived from X in a specified collection of segments. Again, the collection of segments depends on the form of the associated production instance. If the associated production is defined as p above, then the operator iterates over all segments in the unordered collection specified by the **set-of** construct. If the associated production is defined as q, the iteration is over all child segments that are connected to the parent segment at interface node X. The notation $X[i]$ is used to underscore the fact that there may be multiple segments derived from X, and that the semantic equation in the body of the iterator is applied separately to the attribute instances associated with the occurrence of X in each such segment.

We now illustrate SAGs with an example of a simple modular language, shown in figure 8. A program in this language consists of a set of modules, where each module's name must be unique. The the exported facilities of a module are stored in the *exports* attribute associated with each module. The union of the facilities exported by all the modules are collected in the conglomerate attribute, *allexports*, associated with the entire program. This is accomplished by using the **for each linked** operator to iterate over all the modules and applying the **assign** operator to each module. The **compute** operator is used to check whether each module's name is unique within the set of modules comprising the program.

A module references facilities exported by other modules through the import statement. An import statement names the module from which the facility is imported, as well as the facility itself. The **compute** operator finds the component of the *allexports* attribute identified by the imported module's name. The import statement then uses this component to check the legality of the import statement (the imported module must exist and must be unique, and the imported facility must be exported).

To exemplify the use of the **linkedSetSize** operator, we show semantic equations that check that there is exactly one subunit body corresponding to each declared body stub in an Ada program.[4] The second semantic equation associated with the production in figure 9 checks that there is at most one subunit segment for each declared body stub. This equation would flag an error if an Ada program contains two or more subunits defining the body of a subprogram whose body stub is specified in a parent compilation unit. The third semantic equation shown in figure 9 checks that there is at least one subunit segment for each declared body stub. If an Ada program contains a body stub for a package in the compilation unit, but there is no subunit defining the body of the package, this semantic equation would cause an error to be reported.

---

[4] In [29], we present an SAG specification for performing inter-module semantic analysis for Ada, including (1) determining which contexts are accessible to a compilation unit and propagating context information accordingly; (2) detecting compilation units with duplicate names where unique names are required, and detecting missing bodies when they are required by specifications; and (3) checking that there is an order for submitting the compilation units comprising a program for compilation, a requirement of Ada.

```
Program:    { synthesized AllExportsList allexports; }

Module:     { synthesized string name default "?";
              synthesized ExportsList exports default EmptyExp;
              synthesized string error default ""; }

Name:       { synthesized string id; }

Import:     { synthesized string error; }

VarId:      { synthesized string name; }
```

(a): *Symbols and their Attribute*

```
p1:  Program ::= set-of (Module);
         { for each linked Module[i]
               assign(Program.allexports, Module[i].name, Module[i].exports); }

     distributable Module;
p2:  Module ::= 'MODULE' Name ';' Import Export Body Name;
         { Module.name = Name.id;
           Module.error = (compute({Program.allexports}, Name.id) == μ)
                               ? "<-- duplicate module"
                               : "";
           Module.exports =  . . .  ;
            . . . ; }

p3:  Import ::= 'FROM' ModuleId 'IMPORT' VarId ';';
         { local ExportsList single-module-exports;

           single-module-exports = compute({Program.allexports}, ModuleId.name);
           Import.error = (single-module-exports == ⊥)
                     ? "<-- imported module unknown"
                     : (single-module-exports == μ)
                               ? "<-- imported module duplicate"
                               : Member(single-module-exports, VarId.name)
                                         ? ""
                                         : "<-- variable not exported";
            . . . ; }
```

(b): *Productions and their Semantic Equations*

Figure 8: *A Segmented Attribute Grammar Example*

```
subunit-body-stub ::= body-stub subunit-segment;
        { local int no-linked-segments;

          no-linked-segments = linkedSetSize (subunit-segment);
          for each linked subunit-segment[i]
              subunit-segment[i].error =
                  (no-linked-segments == 1)
                      ? ""
                      : (no-linked-segments > 1)
                          ? "<-- duplicate subunit";
          subunit-body-stub.error = (no-linked-segments == 0)
                      ? "<-- missing subunit for this body stub"
                      : ""; }
```

Figure 9: *Checking for Duplicate or Missing Subunits in Ada*

5. Attribute Evaluation for Segmented Semantic Trees

In section 2 we sketched the distinction between batch and incremental attribute evaluation algorithms. By introducing segmentation, we add an intermediate form, which we call *separate* evaluation. In separate evaluation, an individual segment whose corresponding program unit has changed may be reevaluated without necessarily reevaluating the entire semantic tree. Only other segments whose attributes may be affected by the change need to also be evaluated; this is analogous to separate compilation of multiple program units. Evaluation within a segment, however, is done as a batch, *i.e.*, all attributes in the segment are evaluated. The alternative would be incremental evaluation, where only those individual attributes affected by the change are evaluated. The sensible evaluation strategies are indicated by Yes in table 1. In the sequel we discuss both separate and incremental attribute evaluation for segmented semantic trees.

Both separate and incremental evaluation correspond to a ‘‘smart recompilation’’ strategy [37], since the attributes in a segment are reevaluated in either case only when an input attribute actually changes in value. The gist of smart recompilation is to maintain fine-grained change and dependency information so that a program unit is recompiled only if it uses a symbol whose definition actually changes (*i.e.*, the symbol definition is created, deleted or modified). The alternative, exemplified by the make tool, is to recompile a program unit whenever it depends on a changed unit.

	Monolithic Semantic Tree	*Segmented* Semantic Tree
Batch	Yes	Possible
Separate	No	Yes
Incremental	Yes	Yes

Table 1: *Attribute Evaluation Strategies*

Any attribute evaluation algorithm for an SAG consists of two parts: (1) a *local evaluator*, which evaluates attributes within a segment, and (2) a *global evaluator*, which propagates attributes among segments and performs evaluations involving more than one segment. A local evaluator is similar to an evaluator for a monolithic semantic tree except for its actions at interface attributes. The local evaluator is initiated in the following five cases: a new segment is added to the program; an existing segment has been modified since its previous evaluation; evaluation of an adjoining segment results in new values for the input attributes of the segment; addition of a connection to a newly adjoining segment; and removal of a connection to a previously adjoining segment.

Whenever an output attribute changes in value, the global evaluator is activated. When the changed output attribute is not a linkage attribute, the new value is propagated to the local evaluator of any adjoining segments. But additional work is needed when the changed output attribute is a linkage attribute. Then segment linkage is performed first, which may cause the removal of previously adjoining segments and/or the addition of newly adjoining segments. This is followed by exchange of output attributes between any pair of newly adjoining segments, and restoration of the default values for the input attributes of no longer adjoining segments. Local evaluation is then initiated in all affected segments.

Changes to a segment's input attributes may arrive at a segment asynchronously, with a new one arriving before an evaluation triggered by a previous change has completed [19]. In the case of batch local evaluation, it might be desirable to terminate prematurely and then restart considering the new input. For incremental local evaluation, the local evaluator should *merge* the evaluation processes to avoid unnecessary multiple evaluations of the same attributes [31]. Further discussion of these *intra*-segment evaluation issues is outside the scope of this paper.

We consider several issues in *inter*-segment evaluation: Before any attributes can be propagated among segments, the global evaluator must determine which segments are connected to each other, *i.e.*, perform segment linkage. This process is described in section 5.1. Attribute evaluation for shared segments requires special facilities since the same input attribute of a shared segment may receive different values from each of the parent segments connected to it. We describe our solution in section 5.2. In section 6, we consider a problem that may arise when there are dependencies among interface attributes whose values are computed in different segments, and the evaluation of an interface attribute in one segment is delayed by the need for evaluation in the adjoining segment. We describe a subclass of SAGs for which it is possible to rewrite the semantic equations to remove such dependencies.

Finally, there is also a need for efficient propagation of conglomerate attributes, which are constructed by combining output attributes from a collection of segments into a single input attribute to these and/or other segments. The difficulty is that a change to one component of the conglomerate results in the reevaluation of all attributes that depend on any component of the conglomerate. Consider the example shown in figure 8: a change to the exported variable of a *Module* segment is propagated to all modules, including those that do not import the exported variable. This unnecessary propagation can be eliminated by employing use-lists. The local evaluator is extended to maintain a *uses* list representing the subset of the conglomerate attribute that is referenced within the segment; the global evaluator maintains a *used-by* list for each component of the conglomerate attribute. Thus when a component changes in value, the change can be propagated to just those segments that actually use that component. Our approach extends previous work by Hoover and Teitelbaum on aggregate attributes for monolithic semantic trees [12], and is described in detail elsewhere [28].

5.1. Segment Linkage

Segment linkage is performed by the global evaluator to determine which segment instances are connected to each other. As part of the linkage process, the global evaluator detects and disallows cycles and multiple paths between segments in the segmented derivation structure (*e.g.*, when a file directly or indirectly includes itself or is included multiple times in another file, respectively).[5] The SCFG describes which segment types can be connected together, and where the root of the child segment is (logically) connected to the parent segment. But it does not specify which two segment instances are involved in a particular connection. This additional information is required in order to propagate changed interface attributes between parent and child segments. Note that there may be multiple connections between the same parent segment and different children segments for the same interface node, and *vice versa*.

For a segment linkage declaration of the form

$X \{a_i \longleftrightarrow a_s \};$

the semantic equations defining the linkage attributes a_i and a_s are constrained so that a_i only depends on attributes defined in the parent segment where a_i is defined, and a_s only depends on attributes defined in the child segment where a_s is defined.

We can state this constraint precisely as follows: Let p and q denote the production instances in the parent and child segments, respectively, that apply at the interface node X.

p: $\ldots \rightarrow \ldots X \ldots$

q: $X \rightarrow X_1 \ldots X_n$

The attribute a_i (defined by a semantic equation associated with p) cannot depend, directly or transitively, on synthesized attributes of the distributable nonterminal X, and a_s (defined by a semantic equation associated with q) cannot depend, directly or transitively, on inherited attributes of X.

The reason for this constraint is obvious — no propagation of attributes between two segments connected at an interface node X can take place until segment linkage at X has occurred. In the

---

[5]In languages such as C where this problem arises, the idiomatic solution is to define a macro in the included file and then only include the file if the macro is not already defined.

appendix, we give an algorithm to check that the constraint holds for a given SAG.

We show how segment linkage is specified for an Ada program. Figure 10 extends the SCFG for Ada (shown in figure 4 in section 3.1) with segment linkage declarations and semantic equations defining the linkage attributes for implementation and subunit segments. The attributes used for linking specification segments to their corresponding implementation segments are *specification-segment-name* and *implementation-segment-name*. These attributes are assigned the simple name of the package or subprogram whose specification or body appears in the segment. We assume that the package or subprogram name is available in an attribute *name* defined in the subtree derived from the package body or subprogram body, respectively, as well as in the specification.

```
/* segment linkage declarations */

implementation-segment
        { specification-segment-name <—> implementation-segment-name };
subunit-segment
        { body-stub-name <—> subunit-name }

/* linkage attribute definitions */

p2:   specification-segment ::= context-clause subprogram-declaration implementation-segment;
                  { implementation-segment.specification-segment-name =
                            subprogram-declaration.name; }
p3:           | context-clause package-declaration [ implementation-segment ]
                  { implementation-segment.specification-segment-name =
                            package-declaration.name; }

      distributable implementation-segment;
p5:   implementation-segment ::= context-clause subprogram-body
                  { implementation-segment.implementation-segment-name =
                            subprogram-body.name; }
p6:           | context-clause package-body;
                  { implementation-segment.implementation-segment-name =
                            package-body.name; }

p7:   subunit-body-stub ::= body-stub subunit-segment;
                  { subunit-segment.body-stub-name =
                            Concatenate(body-stub.parent-name,body-stub.simple-name);}

      distributable subunit-segment;
p8:   subunit-segment ::= context-clause subunit;
                  { subunit-segment.subunit-name =
                            Concatenate(subunit.parent-name, subunit.simple-name); }
```

Figure 10: *Attribute Grammar for Matching Segments in Ada*

Segment linkage for subunit segments is similar, using linkage attributes *body-stub-name* and *subunit-name*. Each Ada subunit explicitly specifies the full name of its parent unit, starting with the simple name of the ancestor library unit. The *subunit-name* attribute of a subunit segment is assigned the full parent name specified in the subunit (*parent-name* attribute) concatenated with the simple name of the subunit (*simple-name* attribute). (The semantic equations defining the *parent-name* and *simple-name* attributes are omitted in figure 10.) The *body-stub-name* attribute is similarly defined; in this case, the *parent-name* attribute is the fully expanded name of the segment in which the body stub appears, and *simple-name* is the name of the declarative item specified in the body stub.

A simple example is shown in figure 11, where the package body *P* is developed as a separate subunit of *TOP* (not shown). *P* in turn contains the procedure *Q*, whose body is also a separate subunit. The values of the linkage attributes computed locally in each of the two segments is ‘‘TOP.P.Q’’, resulting in the connection of these two segments.

```
*** segment1 ***

separate(TOP)
package body P is
     ...
    procedure Q(Y,Z: REAL) is separate;
end P;

*** segment2 ***

separate(TOP.P)
procedure Q(Y,Z: REAL) is
     ...
begin
     ...
end Q;
```

Figure 11: *Example Segment Linkage for Ada Subunits*

In the case of Ada, it is possible to specify segment linkage using only information that is available in a program written according to the syntax defined in the language manual. This is not always possible for other languages. For instance, C header file segments cannot be connected to the segments that include them based on the contents of the files, but instead segment linkage must be performed on the basis of the included file’s name. This file name is specified in the parent segment that contains the corresponding include statement (see production *p9* in figure 5, section 3), and so the file name can be used to initialize a_i. However, the file name does not appear in the segment corresponding to the included file (production *p10* in figure 5), and there is no other indication as to which files include it.

In order to define the attribute a_s, production *p10* could be modified to contain the name of the file, as follows:

p10′: c-header-file ::= file-name c-defs;

This does not mean, however, that we require a non-standard syntax for C programs. For example, an attribute grammar-based tool might derive the *file-name* portion of the derivation tree from the file system name of the file containing the segment; it does not have to be part of the contents of the file itself.

5.2. Representation and Attribute Evaluation of Shared Segments

The existence of shared segments presents additional complexity to the local segment evaluator because the portion of the semantic tree rooted at a shared segment must contain additional information, as is described in this section.

A segment *S* whose root node is labeled with the nonterminal symbol *X*, where *X* is declared to be a shared distributable symbol, may be connected to several parent segments. We call each such segment *S* a *shared segment* and each segment connected to *S* a *shared-inclusion site of S*. For example, consider the C program in figure 12.[6] This program consists of five segments: two source files (*file1.c* and *file2.c*), and three header files (*x.h*, *y.h*, and *foo.h*). The header file segment *foo.h* will be connected to both of the source file segments that contain the statement `#include "foo.h"`, *file1.c* and *file2.c*.

A segment *S* may be shared indirectly because of an ancestor that is itself connected to multiple parents. We refer to such a segment *S* as a *shared segment* whether or not it is also shared directly, and refer to the ancestor's shared-inclusion sites as if they were *S*'s own.

The input attributes for a shared segment *S* may have different values for each shared-inclusion site of *S*. Consequently, attribute instances in *S* that depend (directly or transitively) on the input attributes have multiple values. In our example, one possible input attribute of the shared segment *foo.h*, *user-defined-types-in*, is the list of user-defined types that have been defined prior to this point in the program. This attribute has a different value depending on whether segment *foo.h* is connected to *file1.c* or *file2.c*. In the first case, *user-defined-types-in* defines the type `NAME` to be a character string; in the second case, *user-defined-types-in* defines `NAME` to be a structure. A typical output attribute of *foo.h*, *user-defined-types-out*, is the list of user-defined types that have been defined up to and including this point in the program. Since this output attribute depends on *user-defined-types-in*, its value will also be different depending on whether segment *foo.h* is connected to *file1.c* or *file2.c*: the name field of the `EMPLOYEE` type is a character string in one case and a structure in the other. The output attribute of the shared segment may be used in the shared-inclusion sites *file1.c* and *file2.c* to perform type checking, in which case a type error will be detected for *file2.c* when a character string is assigned to a structure.

Some form of *versioning* is required in order to represent a different collection of attribute values

---

[6]This example is contrived to illustrate the problems of shared segments, and is not intended to exemplify good programming style.

```
*** file1.c ***

#include "x.h"
#include "foo.h"

main() {
  EMPLOYEE empl;

  empl.name = "David";
  f();
  ...
}

*** file2.c ***

#include "y.h"
#include "foo.h"

f(){
  EMPLOYEE worker;

  worker.name = "Michelle";
  ...
}

*** x.h ***

typedef char *NAME;

*** y.h ***

typedef struct name {
  char *first;
  char middle;
  char *last;
} NAME;

*** foo.h ***

typedef struct employee {
  NAME name;
  int id;
} EMPLOYEE;
```

Figure 12: *Example Showing Why Versions are Needed for Shared Segments*

for each shared-inclusion site of a shared segment. One possibility is to version the entire semantic tree representing the segment *S*, that is, both the derivation tree and the attributes decorating the nodes of the derivation tree. This approach would require space to duplicate the derivation tree for *S* as well as time to maintain consistency among all the replicas when changes to *S* are made. A more efficient approach would be to keep a single derivation tree structure and version only its attributes. This could be further optimized to version only those attributes that actually have different values for each shared-inclusion site of *S*.

We present algorithms to create attribute versions and evaluate the attributes of a shared segment for the following five cases:

1. Creation of a new shared segment *S*;
2. A local change within a shared segment *S*;
3. A change in the value of an input attribute of a shared segment *S*;
4. Addition of a new shared-inclusion site (*i.e.*, a new segment linkage) for a shared segment *S*;
5. Deletion of an existing shared-inclusion site (*i.e.*, removal of a segment linkage) for a shared segment *S*.

There are two other cases that merit consideration, but they can be handled by one of the algorithms for the five cases above, or by completing semantic equations: (6) Deleting a shared segment, which is equivalent to removing all segment linkages between the shared segment and adjoining segments, case 5. Deleting a shared segment that has child segments, in addition, requires deleting all the shared-inclusion sites of these children inherited through the deleted segment (*i.e.*, the child segments are not themselves deleted). (7) Replacing an entire shared segment, which is equivalent to deleting the shared segment (case 6) followed by creation of a new segment (case 1).

For discussion purposes, we assume a dynamic attribute evaluation method for the general class of noncircular grammars. This method might be either batch or incremental. Static evaluation algorithms can be modified to handle versioned attributes of shared segments in a similar way.

We use the following notation for versioned attributes. Let *S* be a shared segment whose root node is labeled by the distributable symbol *X*, and $S_X^1, \ . \ . \ . \ , S_X^n$ the *n* shared-inclusion sites of the shared segment *S*. Each input attribute *a* associated with *X* has *n* distinct versions, as do all other attributes in *S* that depend on *S*'s input attributes. We denote the i^{th} version of the attribute *a* corresponding to the i^{th} shared-inclusion site S_X^i by a^i. We call an attribute that has *n* versions an *n-versioned* attribute. In a shared segment, some attributes are *n-versioned*, always for the same value *n*, while others are non-versioned.

When there are no shared-inclusion sites of a shared segment *S*, the root input attributes of *S* are defined by their completing semantic equations. In this case, there would be exactly one version of each root input attribute of *S* and of any attribute that depends on these attributes. For an attribute *a*, this special version is denoted by a^0. In the remainder of this section we assume that all versioned attributes have this additional version corresponding to the default value. The purpose of keeping the default version, which is ignored when there is at least one inclusion site, is to simplify the algorithms for cases 3 and 4 below.

Case 1, Creation of a New Shared Segment: When segment *S* is first created, all the attribute instances decorating the derivation tree for *S* have *Null* values. (This is true whether *S* is shared or not.) The evaluation/versioning algorithm is shown in figure 13. A version of each input attribute of *S* is created for each shared-inclusion site as determined by the segment linkage operation. These versions are assigned initial values from the output attributes of the adjoining segments. Then, the attribute instances in S are evaluated according to some basic algorithm, which may be either batch or incremental as explained in section 2 — although in this case both would necessarily evaluate every attribute in the segment. Attribute version creation is interleaved with evaluation by creating new versions as needed for attributes whose arguments are versioned. If a root output attribute is versioned, each version is propagated only to the corresponding shared-inclusion site; if it is not versioned, the same value is propagated to every shared-inclusion site. If a leaf output attribute is versioned, propagation to adjoining segments is handled in those segments as in case 4; if not, evaluation in the adjoining segments proceeds as in case 3.

Input:

S: semantic tree representing shared segment, with attribute instances initialized to *Null* and not versioned;
n: number of shared-inclusion sites of S;
i: number of input interface attributes of S;
input_attrs[i][n]: values for input interface attributes of S;

Output:

S: consistently attributed semantic tree representing the shared segment;

Method:

1. Make n versions of each input attribute of S.

2. Initialize input attributes of S with *input_attrs*.

3. Evaluate attributes according to a basic batch or incremental algorithm.
   For each attribute a,

   (a) If any argument of a is n-versioned
      (i) Make n versions of a, with each version initialized to *Null*.
      (ii) Evaluate each version a^i separately, using the corresponding versioned arguments.

   (b) Otherwise, evaluate a.

4. If a root output attribute b changes, then

   (a) If b is versioned, then b^i is propagated to the i^{th} shared-inclusion site.

   (b) Otherwise, the same value b is propagated to every shared-inclusion site.

5. If a leaf output attribute c changes, then

   (a) If c is versioned, then the vector of all c^i is propagated to all adjoining segments.

   (b) Otherwise, the single value c is propagated to every adjoining segment.

Figure 13: *Evaluation of a New Shared Segment*

Case 2, Local Change within a Shared Segment: Perform steps 3, 4 and 5 of the algorithm shown in figure 13.

Case 3, Change in an Existing Version of an Input Attribute: This case does not require creation of any new versions of attributes, just evaluation as in case 2.

Case 4, Creation of a Use of a Shared Segment: Modify steps 1 and 2 of figure 13 to introduce one new version of each input attribute of the root, initialized with the corresponding output attribute from the new shared-inclusion site. For incremental evaluation, modify steps 3, 4 and 5 to evaluate only this new version of affected attributes, creating corresponding new versions of dependent attributes as needed.

Case 5, Deletion of a Use of a Shared Segment: Let S_X^j denote the deleted shared-inclusion site of a shared segment S. The a^j version of each root input attribute a of S corresponding to S_X^j is deleted. Then, a traversal of the dependency graph of S is performed, starting from each root input attribute of S. When a versioned attribute b is encountered, the b^j version corresponding to S_X^j is deleted. If a version of a leaf output attribute is deleted, then this process must be repeated in the adjoining child segments.

6. Summarizable Attribute Grammars

An anomaly may arise during attribute evaluation for a segmented derivation tree, where certain attributes in a segment may remain inconsistent for a considerable amount of time. To illustrate how this may happen, consider the segmented semantic tree shown in figure 14. Two segments, *R* and *S*, are (logically) connected at interface node *X*. Interface attributes *a* and *b* are associated with *X*, and there is a transitive dependency from *a* to *b* that goes through attributes in segment *S*. Suppose that a subtree replacement in segment *R* causes the value of attribute *a* to change. Then, all the attributes in the chain between *a* and *b* in segment *S* have to be evaluated before attribute *b* can be updated. In the meantime, the attribute *b* and all of its dependent attributes in segment *R* are inconsistent. When segments reside on different workstations, the time period during which a segment has inconsistent attributes may be substantial (for example, due to machine and/or link failures). We would therefore like to avoid such scenarios.

In general, an SAG exhibits the anomaly just described if there is a direct or transitive dependency between two attributes *a* and *b* of a distributable symbol *X*, and *a* and *b* are defined in different segments (*i.e.*, one of the attributes is defined in the segment where *X* is a leaf node and the other is defined in the segment derived from *X*). Thus, the dependencies that cause the anomalous behavior must be from an inherited to a synthesized attribute of *X*, or from a synthesized to an inherited attribute of *X*.

In principle, it is always possible to rewrite the attribute grammar to avoid such dependencies across segments by using only synthesized attributes; attribute grammars that employ both inherited and synthesized attributes are no more powerful than those that use only synthesized attributes [25]. However, writing an equivalent attribute grammar without inherited attributes is often considerably harder, and the resulting grammar is more complicated — thus more difficult to understand and change.

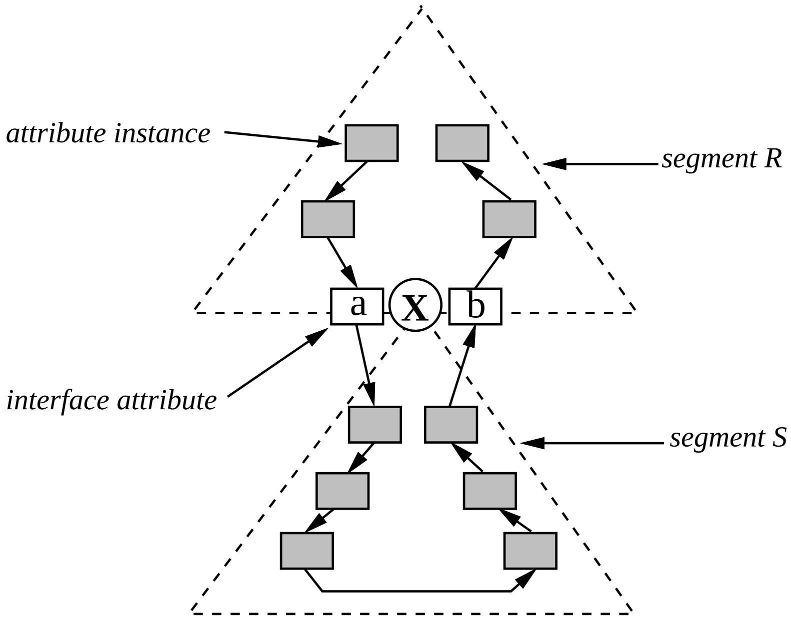


Figure 14: *Attribute Evaluation Anomaly in Segmented Derivation Tree*

We define a subclass of the SAGs, called *summarizable* SAGs, which do not exhibit anomalous behavior. A segmentable attribute grammar, AG_1, is summarizable if either of the following two conditions hold:

- **Condition 1:** For each distributable symbol X in AG_1, there are no direct or transitive dependencies from an inherited interface attribute of X to a synthesized interface attribute of X, or from a synthesized interface attribute of X to an inherited attribute of X.

- **Condition 2:** AG_1 can be mechanically transformed into an equivalent attribute grammar for which condition 1 holds by one of the transformation methods to be described in the following sections.

We identify two patterns of attribute dependencies across segments commonly found in AGs defining the static semantics of programming languages, which do not satisfy condition 1 stated above. The first pattern involves a direct dependency from a synthesized (inherited) interface attribute of X to an inherited (synthesized) interface attribute of X. This pattern can be automatically detected by checking the dependencies in the productions containing the nonterminal symbol X.

The second pattern involves a transitive dependency from a synthesized (inherited) interface attribute of X to an inherited (synthesized) interface attribute of X, where both attributes are aggregates, and the inherited (synthesized) attribute is defined to be equal to the synthesized (inherited) attribute plus some additional components. Although this pattern cannot be detected automatically — it requires knowledge of the meaning of a semantic function — a human can easily identify instances.

In the following two sections, we give example SAGs before and after the appropriate transformations have been applied. The detailed algorithms are presented elsewhere [30].

6.1. Transformation involving Direct Dependencies

The first pattern is exemplified by the SAG for procedures in a Pascal-like language shown in figure 15. Each top-level procedure in the program is defined in a distinct segment. A procedure consists of the procedure name, a list of formal parameters, and a sequence of statements.

This SAG adds a symbol table entry for each top-level procedure. A pair of attributes, *defs_in* and *defs_out*, are used to accomplish this; these attributes are associated with each of the two nonterminals *proc_list* and *proc*. *defs_in* represents the symbol table available to a procedure and *defs_out* represents the symbol table after the procedure's header has been added; *defs_in* is initialized to contain the symbol table entries for the global types and variables that are defined in the program before the top-level procedures.

```
        proc_list:
                { inherited SymTab defs_in;
                  synthesized SymTab defs_out; }

        proc:
                { inherited SymTab defs_in default NullST;
                  synthesized SymTab defs_out default defs_in; }

p1:     program ::=  . . .  proc_list  . . .
                        { proc_list.defs_in = program.sym_tab;
                            . . . ;}

p2:     proc_list ::= /* empty */
                        { proc_list.defs_out = proc_list.defs_in; }

p3:             | proc proc_list;
                        { proc.defs_in = proc_list$1.defs_in;
                          proc_list$2.defs_in = proc.defs_out;
                          proc_list$1.defs_out = proc_list$2.defs_out; }

        distributable proc;
p4:     proc ::= 'PROCEDURE' proc_name '(' formals ')' ';' stmt_seq;
                        { proc.defs_out = AddEntry(proc_name.name, formals.out, proc.defs_in);
                          formals.env_in = proc.defs_out;
                          stmt_seq. env = formals.env_out;
                           . . . ; }
```

Figure 15: *SAG with Direct Dependencies across Segments*

The entry for a top-level procedure is added to the symbol table by the first semantic equation associated with production *p4*. The *AddEntry* function adds the name of the procedure and its signature to the symbol table. (We omit the semantic equations defining *proc_name.name* and *formals.out*, which represent the procedure's name and signature, respectively.)

The nonterminal *proc* is distributable, that is, it labels an interface node between two segments. As described in section 5, the interface attributes associated with *proc* (*defs_in* and *defs_out*) are duplicated in the parent and child segments. The value of the inherited attribute *defs_in* is computed in the parent segment, and the value of the synthesized attribute *defs_out* in the child segment. When either attribute changes in value, the global evaluator propagates the change from one segment to the other.

There is a direct dependency between *proc.defs_in* and *proc.defs_out* stemming from the first semantic equation associated with production *p4*. If *proc.defs_in* changes due to a modification to a global type, variable or another procedure defined before this procedure, the value of *proc.defs_out* must be recomputed. Since *proc.defs_out* is synthesized, it is recomputed in the child segment and then propagated to the parent segment. Thus, the new value for *proc.defs_out* and any attributes dependent on it in the parent segment are not available until propagation from the child segment has occurred.

```
        proc_list:
                { inherited SymTab defs_in;
                  synthesized SymTab defs_out; }

        proc:   { inherited SymTab defs_in default NullST;
                  synthesized string proc_name default "?";
                  synthesized Signature formals_types default NullSig; }

p3:     proc_list ::= proc proc_list;
                        { proc.defs_in = proc_list$1.defs_in;
                          proc_list$2.defs_in = AddEntry(proc.proc_name, proc.formals_types,
                                                         proc.defs_in);
                          proc_list$1.defs_out = proc_list$2.defs_out; }

        distributable proc;
p4:     proc ::= 'PROCEDURE' proc_name '(' formals ')' ';' stmt_seq;
                        { proc.proc_name = proc_name.name;
                          proc.formals_types = formals.out;
                          formals.env_in = AddEntry(proc_name.name, formals.out, proc.defs_in);
                          stmt_seq. env = formals.env_out;
                           . . . ; }
```

Figure 16: *SAG of Figure 15 without Direct Dependencies across Segments*

This SAG can be automatically rewritten so that after a change to the value of *proc.defs_in*, the value of *proc.defs_out* in the parent can be recomputed without going through the child segment. The resulting summarizable SAG is shown in figure 16. *proc.defs_out* is eliminated by duplicating its computation in both parent and child segments. It is replaced by two synthesized (interface) attributes, *proc_name* and *formals_types*.

Originally, *proc.defs_out* was used in the parent segment to define the attribute *defs_in* associated with the second occurrence of *proc_list* in production *p3*. In the child segment, *proc.defs_out* was used to define *formals.env_in*. Eliminating *proc.defs_out* to remove the direct dependency between the attributes associated with the distributable symbol *proc* requires that the semantic equation defining *proc.defs_out* be duplicated in every place where this attribute was previously used, for both parent and child segments. This is shown in the two semantic equations defining *proc_list$2.defs_in* and *formals.env_in* in productions *p3* and *p4*, respectively.

6.2. Transformation involving Transitive Dependencies

An SAG with transitive dependencies across segments is shown in figure 17. It defines a list of declarations, where a sublist of declarations forms a segment. There is a transitive dependency from the inherited attribute *sym_tab_in* associated with the distributable symbol *decls_segment* to the synthesized attribute *sym_tab_out* associated with the same symbol. *sym_tab_in* represents the symbol table entries for identifiers declared earlier in the program; *sym_tab_out* contains the symbol table entries for identifiers declared prior to and including the subtree derived from the nonterminal symbol with which this attribute is associated.

Figure 18 shows a functionally equivalent SAG that has no transitive dependencies across segments. The transformation is again accomplished by removing one of the interface attributes involved in the dependency, and defining additional attributes to fill its previous role. In this case, *decls_segment.sym_tab_out* is replaced by a synthesized interface attribute *decls_segment.subtree_sym_tab_out*; the new attribute represents the symbol table for only those declarations defined in the segment derived from the *decls_segment* symbol. The union of this new attribute and *decls_segment.sym_tab_in* is equivalent to the previous interface attribute *decls_segment.sym_tab_out*. Thus, the union of the two attributes replaces all occurrences of the eliminated *decls_segment.sym_tab_out* on the right side of semantic equations in the parent segment (the second semantic equation associated with *p2* in figure 18).

The value of the synthesized interface attribute *subtree_sym_tab_out* is computed in the child segment derived from *decls_segment*. This is accomplished by defining an additional pair of attributes for the nonterminals *decls* and *decl*. These attributes, *subtree_sym_tab_in* and *subtree_sym_tab_out*, play the same role as *sym_tab_in* and *sym_tab_out* associated with the same nonterminals, except that the former represent only the declarations found in the child segment. Therefore, the *subtree_sym_tab_in* attribute associated with the topmost occurrence of *decls* in the child segment is initialized to *NullST* (in the second semantic equation associated with production *p3* of figure 18). The semantic equations defining *subtree_sym_tab_in* and *subtree_sym_tab_out* are analogous to those defining the attributes *sym_tab_in* and *sym_tab_out*.

```
        decls_list, decls, decl:
                { inherited SymTab sym_tab_in;
                  synthesized SymTab sym_tab_out; }

        decls_segment:
                { inherited SymTab sym_tab_in default NullST;
                  synthesized SymTab sym_tab_out default sym_tab_in; }

p1:     decls_list ::= /* empty */
                        { decls_list.sym_tab_out = decls_list.sym_tab_in; }

p2:             | decls_segment decls_list;
                        { decls_segment.sym_tab_in = decls_list$1.sym_tab_in;
                          decls_list$2.sym_tab_in = decls_segment.sym_tab_out;
                          decls_list$1.sym_tab_out = decls_list$2.sym_tab_out; }

        distributable decls_segment;
p3:     decls_segment ::= decls;
                        { decls.sym_tab_in = decls_segment.sym_tab_in;
                          decls_segment.sym_tab_out = decls.sym_tab_out; }

p4:     decls ::= /* empty */
                        { decls.sym_tab_out = decls.sym_tab_in; }

p5:             | decl decls;
                        { decl.sym_tab_in = decls$1.sym_tab_in;
                          decls$2.sym_tab_in = decl.sym_tab_out;
                          decls$1.sym_tab_out = decls$2.sym_tab_out; }

p6:     decl ::= 'VAR' var_name ':' type_denoter;
                        { decl.sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                                      decl.sym_tab_in);
                           . . . ; }
```

Figure 17: *SAG with Transitive Dependencies across Segments*

7. Related Work

Kaplan and Kaiser introduced the concept of attribute evaluation for multi-segment programs, and devised the first algorithms for incremental evaluation across segments [20]. They left open the problem of specifying the segmentation, and their later work assumed a simple two-level derivation tree consisting of a root with an unordered collection of segments [18, 19]. As far as we know, no other researchers have considered extending context-free grammars and attribute grammars to express complex inter-unit syntactic and semantic connections.

Perry's Inscape system also unifies the specification of inter-unit and intra-unit semantics [33]. A set of preconditions, postconditions and obligations is associated with every statement in a subroutine, with each subroutine, and with each unit (specified as a list of exported subroutines). A set of invariants are associated with every global variable and type definition. Semantic analysis is accomplished by propagation of statement preconditions to their ceilings, *i.e.*, the

```
decls_list:
            { inherited SymTab sym_tab_in;
              synthesized SymTab sym_tab_out; }

decls_segment:
            { inherited SymTab sym_tab_in default NullST;
              synthesized SymTab sym_tab_out default NullST; }

decls, decl:
            { inherited SymTab sym_tab_in;
              inherited SymTab subtree_sym_tab_in;
              synthesized SymTab sym_tab_out;
              synthesized SymTab subtree_sym_tab_out; }

p1:  decls_list ::= /* empty */
                    { decls_list.sym_tab_out = decls_list.sym_tab_in; }

p2:             | decls_segment decls_list;
                    { decls_segment.sym_tab_in = decls_list$1.sym_tab_in;
                      decls_list$2.sym_tab_in = Union(decls_segment.sym_tab_in,
                                          decls_segment.subtree_sym_tab_out);
                      decls_list$1.sym_tab_out = decls_list$2.sym_tab_out; }

     distributable decls_segment;
p3:  decls_segment ::= decls;
                    { decls.sym_tab_in = decls_segment.sym_tab_in;
                      decls.subtree_sym_tab_in = NullST;
                      decls_segment.subtree_sym_tab_out = decls.subtree_sym_tab_out; }

p4:  decls ::= /* empty */
                    { decls.sym_tab_out = decls.sym_tab_in;
                      decls.subtree_sym_tab_out = decls.subtree_sym_tab_in; }

p5:             | decl decls;
                    { decl.sym_tab_in = decls$1.sym_tab_in;
                      decls$2.sym_tab_in = decl.sym_tab_out;
                      decls$1.sym_tab_out = decls$2.sym_tab_out;
                      decl.subtree_sym_tab_in = decls$1.subtree_sym_tab_in;
                      decls$2.subtree_sym_tab_in = decl.subtree_sym_tab_out;
                      decls$1.subtree_sym_tab_out = decls$2.subtree_sym_tab_out; }

p6:  decl ::= 'VAR' var_name ':' type_denoter;
                    { decl.sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                          decl.sym_tab_in);
                      decl.subtree_sym_tab_out = AddEntry(var_name.name, type_denoter.type,
                                          decl.subtree_sym_tab_in);
                       . . . ; }
```

Figure 18: *SAG of Figure 17 without Transitive Dependencies across Segments*

earliest points in a subroutine after which they are not invalidated, and by propagation of statement postconditions and obligations to their floors, *i.e.*, the latest points in the subroutine before which they are not invalidated. All preconditions and obligations must be satisfied by corresponding postconditions. If there is no invalidation within the subroutine, then preconditions, postconditions and obligations are propagated to the interface to the subroutine; a subset of them, selected by the programmer, is propagated to the interface of the program unit. Since full theorem-proving is infeasible, Inscape performs its analysis using only simple symbol manipulation, and thus the ‘‘correctness’’ guaranteed cannot depend on deep properties and implications of the predicates that appear in the preconditions, postconditions and obligations.

Perry’s approach is orthogonal to attribute grammars and attribute evaluation techniques, since there is nothing about attribute grammars that limits them to the traditional symbol resolution, type checking and code generation. In fact, we believe that Inscape could have been implemented using our extended attribute grammar formalism. The main challenge would be representation of the propagated preconditions, postconditions, obligations and invariants in terms of attributes, but still allow the user to prune the unit interfaces. Higher order attribute grammars, where the attribute grammar can specify updates to the derivation tree as well as to the attributes, would permit the logical clauses to be presented to the user for editing [39]. This style of representation is currently used in the Inscape implementation constructed using the Gandalf system [11], where action routines rather than attribute grammars are used to express semantics processing.

More recent work on the Gandalf project sketches an approach to scaling up to large software databases and multiple users [27]. Multiple (context-free) grammars are used to describe the database organization, with segmentation of the database at grammar boundaries. Segmentation is essential for two reasons: (1) to support large databases, so that the entire database does not have to be loaded into virtual memory, and (2) to support multiple users, so concurrency control can be applied at the segment level. Multiple grammars are combined at *segment nodes*; a segment node is labeled by a terminal symbol representing an abstraction in one grammar and the start symbol in another grammar defining the abstraction. Segment nodes are similar to our distributable nonterminal symbols. The scaled up Gandalf system would still use action routines for semantics processing.

In the attribute grammar literature, segmentation or division of the derivation tree has been proposed for other purposes. Boehm and Zwaenepoel describe a distributed algorithm for attribute evaluation, exploiting parallelism to speed up the compilation process [4]. The derivation tree is divided into regions, which are evaluated in parallel on different machines. Annotations on the attribute grammar specify at which nonterminals the derivation tree may be split, and the minimum size of a region to be evaluated separately. Their augmented attribute grammar is based on a conventional context-free grammar, and the regions that are evaluated in parallel do not necessarily correspond to modular units of the language. Their distributed evaluator differs from our combined local and global evaluator because: (1) it is batch, as opposed to separate or incremental, that is, it evaluates all attribute instances in an entire semantic tree; and (2) it has no support for programming-in-the-large constructs.

Alblas extended this algorithm to support incremental attribute evaluation [2], using an approach to asynchronous changes described by Kaplan and Kaiser. Unlike Boehm and Zwaenepoel, however, Alblas does not actually distribute the regions across a network. Instead, he associates them with distinct threads of control. In effect, Alblas's algorithm parallelizes evaluation within a single segment, and does not address propagation among multiple segments.

Klaiber and Gokhale describe parallel but non-incremental evaluators for attribute grammars for execution on a multiprocessor [24]. Because simulations of their algorithm showed that productions of the form $X \rightarrow Y\,X$ severely limit the amount of possible parallelism, they propose a method for restructuring the attribute grammar to instead use the production $X \rightarrow Y^+$ (*i.e.*, X expands to one or more instances of Y). This allows clusters of several small subtrees rooted at Y to be executed in parallel, where otherwise each subtree would have been too small to justify the overhead of forking a new process. Although this form of list production resembles our **set-of** construct, it still derives an ordered list, requiring that attribute dependencies among the symbols in the list flow from left to right.

8. Conclusion

The goal of this work was to enable the application of attribute grammars to programming-in-the-large. Previously, attribute grammars could be applied only within individual program units, *i.e.*, programming-in-the-small, mandating other formalisms for interfacing among program units. Our primary results are:

- Segmentable context-free grammars, for specifying the syntactic structure of multi-unit programs. The key concepts are distributable and shared distributable nonterminal symbols together with the **set-of** construct. We explained how these facilities could be employed for Ada, C and Pascal.

- Segmentable attribute grammars, for specifying the static semantic properties of multi-unit programs. The novel ideas include interface attributes, segment linkage, conglomerate attributes, and various primitive operators for segment manipulation. We showed as an example how to specify the legality of import statements, *i.e.*, the imported unit must exist and be unique and the facilities imported from that unit must be exported.

- Separate and incremental evaluation algorithms, for efficiently attributing segmented derivation trees. The new algorithms perform segment linkage and evaluate shared segments with versioned attributes representing multiple inclusion sites. We described segment linkage for Ada subunits, and demonstrated why versioned attributes are needed for C header files.

- Summarizable attribute grammars, a restricted class of segmentable attribute grammars, where it is possible to remove dependencies that cross segment boundaries. This is important to avoid delays during attribute evaluation in a distributed environment. We give examples of both direct and transitive dependencies across segments and the transformed grammars without such dependencies.

We have assumed throughout this work that segments are statically defined, with only limited transformations possible among predefined segmenting options (see Section 3.3). This seems most practical for programming-in-the-large applications analogous to separate compilation. However, other applications might require arbitrary dynamic segmentation. One example would be to improve load balancing for Boehm and Zwaenepoel's parallel compilation. Another might be efficient propagation of triggers in an object-oriented database modeled as a semantic tree, as proposed by Hudson and King [13], but segmented according to multi-user browsing focus — introducing the possibility of overlapping segments. In fact, we anticipate that many of the most promising avenues for extended attribute grammars will arise in such non-traditional domains.

Acknowledgments

Simon Kaplan worked with one of the authors on the original idea of applying attribute evaluation to multi-segment programs. We would also like to thank Terry Boult, Yechiam Yemini, Barbara Ryder and Dan Yellin for reviewing earlier versions of this work. Tim Teitelbaum, Tom Reps and other members of the Synthesizer Generator group at Cornell University provided much of our motivation.

When this work was in progress, Micallef was supported in part by an American Dissertation Fellowship from the American Association of University Women Educational Foundation. Kaiser was supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the New York State Center for Advanced Technology in Computers and Information Systems and NSF Engineering Research Center for Telecommunications Research.

References

[1] AdaTEC the SIGPLAN Technical Committee on Ada.
Reference Manual for the Ada Programming Language
United States Department of Defense, 1982.
Draft Revised MIL-STD 1815.

[2] Henk Alblas.
Concurrent Incremental Attribute Evaluation.
In P. Deransart and M. Jourdan (editors), *Lecture Notes in Computer Science*. Number 461: *International Workshop on Attribute Grammars and their Applications*, pages 343-358. Springer-Verlag, Paris, France, 1990.

[3] Gregor V. Bochmann.
Semantic Evaluation from Left to Right.
Communications of the ACM 19(2):55-62, February, 1976.

[4] Hans-Juergen Boehm and Willy Zwaenepoel.
Parallel Attribute Grammar Evaluation.
In *7th International Conference on Distributed Computing Systems*, pages 347-354. IEEE Computer Society, Berlin, West Germany, September, 1987.

[5] Frank DeRemer and Hans H. Kron.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2):80-86, June, 1976.

[6] H. Ehrig, M. Nagl, G. Rozenberg and A. Rosenfeld (editor).
Lecture Notes in Computer Science. Volume 291: *Graph-Grammars and Their Application to Computer Science 3rd International Workshop* .
Springer-Verlag, Warrenton VA, 1986.

[7] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4):77-93, October, 1984.

[8] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[9] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland, New York, 1977.

[10] A. Nico Habermann and Dewayne E. Perry.
System Composition and Version Control for Ada.
Software Engineering Environments.
North-Holland, New York, 1981, pages 331-343.

[11] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[12] Roger Hoover and Tim Teitelbaum.
Efficient Incremental Evaluation of Aggregate Values in Attribute Grammars.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 39-50. Palo Alto CA, June, 1986.
Special issue of *SIGPLAN Notices*, 21(7), July 1986.

[13] Scott E. Hudson and Roger King.
Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System.
ACM Transactions on Database Systems 14(3):291-321, September, 1989.

[14] M. Jazayeri, W.F. Ogden and W.C. Rounds.
The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars.
Communications of the ACM 18(12):697-706, December, 1975.

[15] Kathleen Jensen and Niklaus Wirth.
Pascal User Manual and Report.
Springer-Verlag, New York, 1974.

[16] Martin Jourdan and Didier Parigot.
The FNC-2 System User's Guide and Reference Manual
INRIA, France, 1989.

[17] Martin Jourdan, Didier Parigot, Catherine Julie, Olivier Durin and Carole le Bellec.
Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System.
In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 209-222. June, 1990.
Special issue of *SIGPLAN Notices*, 25(6), June 1990.

[18] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software 4(6):58-67, November, 1987.

[19] Gail E. Kaiser and Simon M. Kaplan.
Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multi-User Software Development Environments.
ACM Transactions on Software Engineering and Methodology 2(1):47-92, January, 1993.

[20] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th Annual ACM Symposium on Principles of Distributed Computing*, pages 121-130. Calgary Alberta, Canada, August, 1986.

[21] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[22] U. Kastens, B. Hutt and E. Zimmermann.
Lecture Notes in Computer Science. Volume 141: *GAG: A Practical Compiler Generator.*
Springer-Verlag, Heidelberg, FRG, 1982.

[23] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language
Second edition, Englewood Cliffs NJ, 1988.

[24] Alexander Klaiber and Maya Gokhale.
Parallel Evaluation of Attribute Grammars.
In *IEEE Transactions on Parallel and Distributed Systems*, pages 206-220. March, 1992.

[25] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[26] Donald E. Knuth.
Semantics of Context-Free Languages: Correction.
Mathematical Systems Theory 5(1):95-96, March, 1971.

[27] Charles W. Krueger and Annalisa Bogliolo.
Scaling Up: Segmentation and Concurrency in Large Software Databases.
Technical Report CMU-CS-87-178, Carnegie Mellon University Department of Computer Science, February, 1988.

[28] Josephine Micallef and Gail E. Kaiser.
Version and Configuration Control in Distributed Language-Based Environments.
In Jurgen F.H. Winkler (editor), *International Workshop on Software Version and Configuration Control*, pages 119-143. B.G. Teubner, Grassau, West Germany, January, 1988.

[29] Josephine Micallef and Gail E. Kaiser.
Extending the Mercury System to Support Teams of Ada Programmers.
In *1st International Symposium on Environments and Tools for Ada*, pages 49-60. ACM Press, Redondo Beach CA, April, 1990.
Special issue of *Ada Letters*, 11(3), Spring 1991.

[30] Josephine Micallef.
Incremental Attribute Evaluation for Multi-User Semantics-Based Editors.
PhD thesis, Columbia University, May, 1991.
Technical Report CUCS-023-91.

[31] Josephine Micallef and Gail E. Kaiser.
Support Algorithms for Incremental Attribute Evaluation of Asynchronous Subtree Replacements.
IEEE Transactions on Software Engineering , 1993.
In press.

[32] K. Narayanaswamy and Walt Scacchi.
Maintaining Configurations of Evolving Software Systems.
IEEE Transactions on Software Engineering SE-13(3):324-334, March, 1987.

[33] Dewayne E. Perry.
The Inscape Environment.
In *11th International Conference on Software Engineering*, pages 2-9. IEEE Computer Society Press, Pittsburgh PA, May, 1989.

[34] Thomas Reps.
Generating Language-Based Environments.
The MIT Press, Cambridge MA, 1984.
PhD Thesis, Cornell University, Technical Report 82-514.

[35] Thomas W. Reps and Tim Teitelbaum.
Texts and Monographs in Computer Science: *The Synthesizer Generator Reference Manual.*
Springer-Verlag, New York, 1989.

[36] Walter F. Tichy.
Software Development Control Based on Module Interconnection.
In *4th International Conference on Software Engineering*. September, 1979.

[37] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[38] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein and W. Kirchgassner.
Lecture Notes in Computer Science. Number 139: *An Attribute Grammar for the Semantic Analysis of Ada.*
Springer-Verlag, Heidelberg, FRG, 1982.

[39] H.H. Vogt, S.D. Swierstra and M.F. Kuiper.
Higher Order Attribute Grammars.
In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 131-145. Portland OR, June, 1989.
Special issue of *SIGPLAN Notices*, 24(7), July 1989.

[40] William M. Waite and Gerhard Goos.
Text and Monographs in Computer Science: *Compiler Construction.*
Springer-Verlag, New York, 1984.

[41] Alexander L. Wolf, Lori A. Clarke and Jack C. Wileden.
Ada-Based Support for Programming-in-the-Large.
IEEE Software 2(2):58-71, March, 1985.

[42] Dashing Yeh.
On Incremental Evaluation of Ordered Attribute Grammars.
BIT 23:308-320, 1983.

I. Constraints on Segment Linkage Declarations

For a segment linkage declaration of the form

$$X\ \{a_i \longleftrightarrow a_s\};$$

the semantic equations defining the linkage attributes a_i and a_s are constrained so that a_i only depends on attributes defined in the parent segment where a_i is defined, and a_s only depends on attributes defined in the child segment where a_s is defined. Let p and q denote the production instances in the parent and child segments, respectively, that apply at the interface node X, defined as follows:

$$p\colon \ \ldots \ \rightarrow \ \ldots \ X \ \ldots$$

$$q\colon X \rightarrow X_1 \ \ldots \ X_n$$

The attribute a_i (defined by a semantic equation associated with p) cannot depend, directly or transitively, on synthesized attributes of the distributable nonterminal X, and a_s (defined by a semantic equation associated with q) cannot depend, directly or transitively, on inherited attributes of X.

For an arbitrary noncircular attribute grammar G, checking whether the constraint holds requires exponential time in the size of the grammar (*i.e.*, the number and length of the productions). We prove this by reduction. Suppose that there exists a polynomial time algorithm for deciding whether there is a transitive dependency from an inherited attribute of a symbol X to a synthesized attribute of X in any possible derivation tree of G. Similarly, for deciding whether there is a transitive dependency from a synthesized to an inherited attribute of X. Then we can check whether there is a cycle involving attributes of *X* by applying the polynomial time algorithm to all pairs of inherited-synthesized attributes of *X*; the maximum number of such pairs is a constant for any given AG. Any cycle in *D(T)* necessarily implies a cycle among some attributes of a single nonterminal occurrence. Thus by doing the same computation for every nonterminal symbol of *G*, we could decide whether the grammar is noncircular in polynomial time. But Jazayeri *et al.* proved that deciding whether an AG is noncircular requires exponential time [14].

We therefore use a pessimistic approximation that can be computed in polynomial time to check the constraint. This approximation is the relation TDS_X, which contains all the direct and transitive dependencies between attributes of a nonterminal symbol *X* that could be present in any derivation tree. (The algorithm for computing TDS_X is given by Kastens [21].) TDS_X is pessimistic because all these dependencies are assumed to be present simultaneously; that is, TDS_X could contain an edge representing a dependency from a synthesized attribute of the distributable symbol *X* to a_i (or from an inherited attribute of *X* to a_s) that could not be present in any semantic tree derived from the grammar. This is illustrated by the attribute grammar shown in figure 19, which is adapted from Waite and Goos [40].

```
p1:   Z ::= X
          { X.a = 1; }

p2:   X ::= Y
          { X.b = Y.f;
            Y.c = X.a;
            Y.d = Y.e; }

p3:   Y ::= u
          { Y.e = 2;
            Y.f = Y.d; }

p4:   Y ::= v
          { Y.e = Y.c;
            Y.f = 3; }
```

Figure 19: *A Noncircular Attribute Grammar*

This grammar derives two trees, shown in figure 20. The relation TDS_Y has the dependency edges $\{(d,f), (e,d), (c,e), (c,f)\}$. Note that both edges (d,f) and (c,e) are included in TDS_Y even though it is clear from figure 20 that only one of them can occur in any derivation tree. The result is that the relation TDS_X contains the edge (a,b) even though there is no dependency from *X.a* to *X.b* in any derivation tree of the grammar. Thus, linkage attributes involving such pathological chains of transitive dependencies as well as truly circular dependencies would be disallowed.

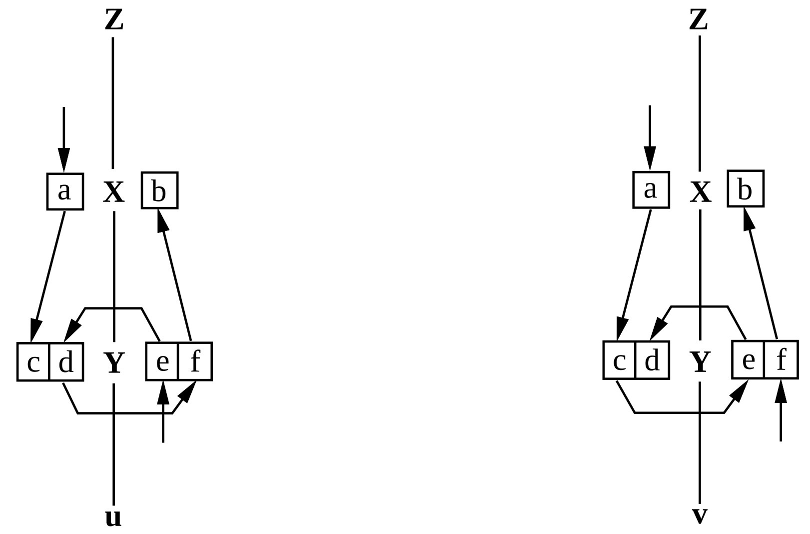


Figure 20: *Dependency Graphs of Trees Derived from AG of Figure 19*