Extending the MERCURY System to Support Teams of Ada Programmers

Josephine Micallef
Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027
Tel: (212) 854-8178
Internet: jm@cs.columbia.edu

December 6, 1989

CUCS-503-89

Abstract

The MERCURY system generates multi-user language-based environments from attribute grammars. The AG specifies the interface checking among modular units, to be applied by the environment to inform programmers of errors introduced by interface changes. Since AGs assume a monolithic program, we extended the formalism to support separate compilation units. Our previous work was based on an ideal language where programs consist of an unordered set of monolithic compilation units. We now augment our extensions to support Ada, to allow multiple kinds of compilation units and nested compilation units. We describe how these extensions are used to detect naming errors, determine compilation unit context, and check compilation order as mandated by Ada.

Micallef is supported in part by an American Dissertation Fellowship from the American Association of University Women Educational Foundation. Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citicorp, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

keywords: environments, analysis tools and techniques, distribution, programming-in-the-many, support for evolution

1. Introduction

We have developed a tool, called MERCURY, for generating multi-user, distributed, language-based environments from a formal specification of the desired programming language. Each MERCURY environment detects syntactic and semantic inconsistencies, as defined in its language specification, with immediate feedback regarding inconsistencies. Like other language-based environments, such analysis is applied incrementally within each modular program unit while it is being edited by an individual environment user working in isolation. The innovative feature of MERCURY environments is that they also provide this capability among units to notify all affected users of any semantic inconsistencies introduced into their own program units by changes to the interfaces of units being modified by other users.

Each MERCURY environment provides a number of levels of change propagation as desired by the individual programmers cooperating on a large software project. Each programmer editing a program unit can request that (1) his changes to the interface of his program unit be immediately propagated, at the individual editing command level of granularity, to inform any other programmers of changes that affect their units — with subsequent earliest-possible detection of newly introduced semantic errors; or (2) that such change propagation be delayed until requested, to avoid premature communication of interface changes that are only under consideration and have not yet been committed.

Further, each programmer can also select that (3) any changes made by other programmers be immediately propagated to notify him of changes to the interfaces of other program units that adversely affect his own units, again with subsequent earliest-possible detection of newly introduced interface mismatches; or (4) delay notification to avoid bombardment by error messages when many interfaces are undergoing significant changes, such as during early development stages or when it is desirable to assume the interface provided by earlier versions of other program units for an extended period of time. A small example of how a MERCURY environment works for an ideal programming language is given in appendix I.

We have developed a collection of algorithms to support this change propagation [Kaplan 86, Kaiser 87a, Kaiser 87b, Micallef 88, Kaiser 88, Micallef 89], most of which are implemented

in the current version of the MERCURY system.[1] These algorithms work very nicely for the ideal modular programming language that we imagined, but are insufficient to support the complex requirements of Ada. Ada presents two problems that we did not consider in our earlier work: there are several different kinds of compilation units, and compilation units may be hierarchically nested. These in turn present difficulties with respect to detecting naming errors, determining the contexts for compilation units, and checking the compilation order. In this paper, we present improvements over our previous work that address these problems, and make it possible to support change propagation for teams of Ada programmers. We have not yet totally solved the Ada problem, however, and leave pragmas and generic packages and subprograms for future work.

We first describe background and terminology; readers familiar with attribute grammars and the classical approach to incremental semantic analysis in single-user language-based environments can skip this section. We then sketch our previously published extensions of the classical approach to support segmentation of a parse tree and incremental evaluation across multiple segments. We then describe the new segmentation scheme devised for Ada, and present our solutions to the naming, context and compilation order problems. The paper ends with a brief discussion of related work.

2. Background and Terminology

We employ an *attribute grammar* (AG) [Knuth 68] as the formal specification of the desired programming language. An AG consists of a context-free grammar enhanced to express the language's context-sensitive constraints by (1) associating with every symbol in the context-free grammar a collection of *attributes*, or properties, and (2) associating with every production in the grammar a collection of *semantic equations*, each of which defines the value of an attribute of one symbol in the production as a function of other attributes of the same and/or other symbols in the production (and/or the values of terminal symbols in the same production). When a semantic equation defines the value of an attribute of the goal symbol of the production, the attribute is said to be *synthesized*; when an equation defines the value of an attribute of a symbol

[1] MERCURY was implemented by modifying the Synthesizer Generator [Reps 89] developed at Cornell University, which supports incremental static semantic analysis within a monolithic program in a single-user environment; MERCURY runs on a variety of Unix systems using X10 windows, and we are in the process of upgrading to the new version of the Synthesizer Generator, which uses X11 windows.

on the right hand side of the production, the attribute is *inherited.* A small attribute grammar is given in appendix II.

The context-free grammar is used to parse a program into tree form, where each node in the parse tree is an instantiation of a production of the grammar. The attributes associated with each production are then added to decorate the tree. Computing the value of an attribute by applying its semantic equation is called *evaluation* of the attribute. A syntactically correct attributed parse tree is called a *semantic tree.* A semantic tree for which all attributes have correct values is called *consistent.*

An attribute grammar specification can be used to generate a language-based editor that incrementally checks the semantic correctness of a program to provide immediate feedback after every editing command. The program is represented internally as an attributed parse tree. Editing commands are defined in terms of an editing cursor, which selects a particular subtree in the parse tree, and subtree replacements, which modify this tree. After a single subtree replacement, the tree is syntactically correct, but the attributes located at the root of the replacement subtree are inconsistent with respect to the other attributes in the tree and *vice versa.* Rather than re-apply attribute evaluation to the full attributed parse tree, an incremental evaluation algorithm is employed in order to reestablish consistency with minimal computation effort. Depending on the particular context-sensitive constraints specified in the attribute grammar, incremental evaluation can be used for symbol resolution and type checking, detection of data flow anomalies, precondition/postcondition/obligation enforcement as in Inscape [Perry 89], and code generation, among other things.

This paradigm was first proposed by Demers, Reps and Teitelbaum [Demers 81]. The classical algorithm is due to Reps, Teitelbaum and Demers [Reps 83] and is asymptotically optimal in time: its time complexity is $O(|Affected|)$, where *Affected* is the set of attributes that change. Optimality is attained because the algorithm orders the reevaluation of attributes in such a way that only attributes that are inconsistent are actually reevaluated, and a particular attribute is reevaluated only when applying the semantic equation is guaranteed to yield its correct value.

3. Segmented Evaluation

Our previously reported contribution was to extend the classical incremental evaluation algorithm and the classical AG formalism to the problem of incremental evaluation on a decentralized tree distributed among multiple user processes [Kaplan 86].[2] This approach is based on a model of software development where several programmers cooperate to develop a large program. Each program is divided into a number of *segments*, where a segment is both the unit of editing and the unit of distribution; that is, a segment represents a compilation unit (often called a module) [Kaiser 87b].[3]

There is one editing cursor per segment, and edits in different segments proceed *asynchronously*, both with respect to each other and with respect to in-progress attribute evaluation processes within the same segment generated by previous local edits or by edits to the interfaces of other segments [Kaiser 88]. Asynchronous editing is necessary to allow every programmer to carry on his work unhampered by program modifications made by others.

To support fully decentralized attribute evaluation for an ideal programming language, we extend the standard AG formalism as follows. Most of these extensions must be modified for Ada, as elaborated in the next section.

1. We add a new construct, **set**, to the context-free grammar notation as a means for defining an unordered sequence; standard context-free grammars can define only ordered sequences, by a skewed tree using a left- or right-recursive pair of context-free productions. This extension makes it possible to specify a collection of segments without requiring an explicit order among the segments. Our goal is to avoid the overhead of synchronization among multiple users that would be needed to specify a particular ordering as new segments are added to a program during development.
2. We restrict the **set** construct to appear only on the right hand side of the production that defines the start symbol of the grammar. The effect is to represent the program as a two-level hierarchy, with a program root and an unordered collection of segments. This simplifies the representation of segment interconnection information to only the one set.

---

[2]One of our previously published algorithms supports distribution across a local area network, with special facilities for availability and reliability [Kaiser 87a]; these operate the same way for Ada as for any other programming language, so are not discussed here.

[3]In this paper, we assume that multiple programmers never simultaneously modify different copies of the same segment, but elsewhere we describe facilities for dealing with multiple versions of segments [Micallef 88]. We ignore the case where one user simultaneously edits multiple compilation units within a single user process, since that is a straightforward extension.

3. We declare the goal symbol of a selected production to be a *distributable unit* (*i.e.*, the root of a segment), and this distributable unit must also be the element type of the set described above. The effect is to restrict the program to one kind of segment. This guarantees that all segment root nodes will have the same attributes.
4. We declare certain attributes of the distributable unit symbol to be *interface attributes*, meaning they represent dependencies across segments (*e.g.*, the imports and exports lists of modules). This makes it clear when incremental evaluation must propagate across segment boundaries.
5. We require that interface attributes be defined in pairs, a synthesized interface attribute and an inherited interface attribute. When a synthesized interface attribute in one segment changes in value, the corresponding inherited interface attribute in all segments must be reevaluated. This simplifies the dependencies among the interface attributes of segments.
6. We limit the semantic equation for an inherited interface attribute of a segment to the **union** of the corresponding synthesized interface attributes of all segments. An attribute that is computed in this way, by combining components from multiple segments, is called a *conglomerate attribute* (it is important to distinguish conglomerate attributes from aggregate attributes, which are composed of multiple components from the same segment and typically represent local symbol tables). Thus, there is no need for explicit attributes and semantic equations for the start symbol, and the root of the program need not be explicitly represented.

Segmented evaluation is supported by the *attribute propagation layer* (APL). The APL is implemented as a separate process where the (virtual) root of the program resides. The root of the program is represented by a set of nodes, one for each segment, with each node containing a copy of the synthesized interface attributes of the segment.

A subtree replacement within a segment initiates an incremental evaluation process to reestablish consistency, as in the classical algorithm. In many cases, operation proceeds exactly as in the classical algorithm. But sometimes a synthesized interface attribute of an edited segment changes in value, and thus becomes inconsistent with the corresponding inherited interface attributes of the other segments. In this case, the new values of the synthesized interface attribute is transmitted to the APL, which computes the new value of the corresponding inherited interface attribute and transmits it to all the segments.

When an inherited interface attribute is updated by the APL, it is treated as a simulated subtree replacement at the root of each segment; a new incremental evaluation process is initiated within each segment to reestablish consistency. Multiple evaluation processes initiated by the APL and by subtree replacements within a segment are merged to minimize costs when multiple processes may affect the same attribute or cancel each other out [Micallef 89].

When a segment is created/deleted, it is installed/removed in the set of segments maintained by the APL. The APL uses an *ad hoc* mechanism to detect whether there are two or more segments with the same name, and if so, transmits an error message to each of them and ignores their interface attributes until the conflict is removed.[4]

4. New Results Motivated by Ada

Our previous extensions to the AG formalism, designed for an "ideal language", are not sufficient for Ada for two reasons. First, our ideal language assumes a single kind of compilation unit, to be modeled as a segment; but there is more than one kind of compilation unit in Ada, and thus it is necessary to support multiple kinds of segments. Each kind plays a substantially different role in semantic analysis. Second, Ada program structure is hierarchical, not the flat set of compilation units assumed for our ideal language; thus additional mechanisms are required for representing segment interconnection information for use by the APL. Within a compilation unit, however, the assumptions we made previously still apply for the most part — and the classical attribute grammar formalism is sufficient to model almost all semantic dependencies within segments; exceptions are noted as needed in the rest of this section.

The main contribution of this paper is thus the extensions to AGs needed to perform interface checking among the segments comprising an Ada program in an interactive/incremental multi-user environment. We do not address static semantic analysis within a single compilation unit (this has been done in the Karlsruhe Ada Compiler [Uhl 82]). In particular, we consider the following issues in performing semantic analysis across the segments of an Ada program:

- **Naming**: Detection of segments with duplicate names where unique names are required, and detecting missing bodies when they are required by specifications.
- **Context**: Determining which contexts are accessible to a segment and propagating context information accordingly.
- **Compilation Order**: Checking that there is an order for submitting the segments comprising a program for compilation.[5]

---

[4]Other mechanisms, not described here, distinguish segments within different programs and prevent interference among multiple programs in the same or different languages when multiple software development projects share the same machine(s).

[5]This check may seem unnecessary when no "compilation" is being done, as we intend MERCURY environments to support interface checking among compilation units rather than code generation. But this check is necessary to generate the same error messages as an Ada compiler.

Our solutions to these problems are based on our new mechanisms to support multiple kinds of segments and hierarchical interconnections among segments.

4.1. Representation of an Ada Program

A segment corresponds to an Ada compilation unit as defined in the Ada reference manual [AdaTEC 82]. 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 nested 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 *nested segment* is a subunit; implementation and nested segments are collectively called secondary units in the Ada reference manual.

We provide two mechanisms for interconnecting segments in Ada. The first mechanism is our previously introduced **set** construct; this is used to group the collection of specification segments that make up an Ada program. The second mechanism is the new **interface** construct presented for the first time in this paper; it is used to connect implementation segments to their respective specification segments, and nested segments to their respective parent segments.

```
p1:  ada_program ::= set of specification_segment;

     distributable specification_segment;
p2:  specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
p3:          | context_clause package_declaration [ interface(implementation_segment) ]
p4:          | context_clause subprogram_body;

     distributable implementation_segment;
p5:  implementation_segment ::= context_clause subprogram_body
p6:          | context_clause package_body;

p7:  interface_body_stub ::= body_stub interface(nested_segment);

     distributable nested_segment;
p8:  nested_segment ::= context_clause subunit;
```

Figure 4-1: *Context-Free Grammar for Definition and Interconnection of Segments in Ada*

Figure 4-1 gives an extended context-free grammar that defines the three kinds of segments in

Ada and their interconnections. Nonterminals shown in italics are defined in the Ada reference manual. The keyword **distributable** indicates those nonterminal symbols that derive a segment. 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 the production by the **interface** construct, which serves as the connection point between this segment and the implementation segment representing the subprogram body.[6]

A specification segment that derives a package specification may or may not have a corresponding body; in this case, the **interface** construct 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 has no need for an **interface** part.

The connection between a nested segment and its parent is defined in production *p*7 in figure 4-1. Each subunit in Ada, represented as a nested 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 (a separate file in traditional software development environments, or a separate segment in a MERCURY environment). Production *p*7 associates with each body stub an interface to the corresponding nested segment. All occurrences of *body_stub* that appear on the right side of productions in the syntax given in the Ada reference manual are replaced by the nonterminal *interface_body_stub*. Note that the context-free grammar given in the Ada reference manual describes the syntax of individual compilation units, but does not show how the compilation units fit together to make an Ada program; thus it was necessary to define this additional syntax.

The **interface** construct 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 describe which two segment instances are involved in a particular connection. This additional information is required by the APL in order to propagate a changed interface attribute in a parent segment to the correct child segment, and *vice versa*.

---

[6]We currently do not handle the Ada INTERFACE pragma, which allows subprograms written in other languages to be called from Ada programs, and our **interface** extension to the AG formalism should not be confused with the INTERFACE pragma.

We provide this additional information in the form of two special attributes associated with each nonterminal symbol that connects two segments by means of the **interface** construct (in figure 4-1, these are nonterminals *implementation_segment* and *nested_segment*). The two attributes are *parent_interface_name* and *child_interface_name*. The value of *parent_interface_name* is computed in the parent segment, and is assigned the name of the child segment as determined from the parent segment; the value of *child_interface_name* is computed in the child segment, and is assigned the actual name of the child segment. The APL then uses these attributes to match the parent and child segments, allowing attribute propagation among these segments to proceed correctly.

```
p2:  specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
                    { implementation_segment.parent_interface_name =
                            subprogram_declaration.subprogram_name; }
p3:          | context_clause package_declaration [ interface(implementation_segment) ]
                    { implementation_segment.parent_interface_name = package_declaration.package_name; }

p5:  implementation_segment ::= context_clause subprogram_body
                    { implementation_segment.child_interface_name = subprogram_body.subprogram_name; }
p6:          | context_clause package_body;
                    { implementation_segment.child_interface_name = package_body.package_name; }

p7:  interface_body_stub ::= body_stub interface(nested_segment);
                    { nested_segment.parent_interface_name =
                            concatenate(body_stub.parent_name, body_stub.simple_name); }

p8:  nested_segment ::= context_clause subunit;
                    { nested_segment.child_interface_name =
                            concatenate(subunit.parent_name, subunit.simple_name); }
```

Figure 4-2: *Attribute Grammar for Matching Segments in Ada*

Figure 4-2 extends the context-free grammar of figure 4-1 with semantic equations defining these two attributes for implementation and nested segments. For implementation segments, the *parent_interface_name* and *child_interface_name* attributes are assigned the simple names of the specification segment and implementation segment, respectively. We assume that the segment name is available in either the attribute *subprogram_name* or *package_name*, depending on whether the segment represents a subprogram or package, respectively; we omit the semantic equations defining these attributes.

Nested segments are treated similarly, except that fully expanded names are required. In Ada, each subunit specifies the full name of its parent unit, starting with the simple name of the

ancestor library unit. The *child_interface_name* attribute of a nested segment is assigned the full parent name (attribute *parent_name*) concatenated with the simple name of the subunit (attribute *simple_name*).[7] The attribute *parent_interface_name* is similarly defined; in this case, the attribute *parent_name* is the fully expanded name of the segment in which the body stub appears, and *simple_name* is the name of the body specified in the body stub.

4.2. Naming

The inter-segment 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 it, (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. Writing an attribute grammar to perform this is a challenge because the parse tree representing the Ada program is decentralized, and units and subunits can be added to, or removed from, the program asynchronously by any of the multiple programmers.

Unlike an Ada compiler, MERCURY environments explicitly support multiple users. This requires a somewhat different interpretation of the semantic constraints described above. To illustrate this, consider the first semantic check, which states that all library unit names should be distinct. In an Ada compiler, when two library units with the same name are submitted for compilation, the last one to be compiled replaces the previous one in the program library. This action is inappropriate in a multi-user environment for two reasons. First, it requires a total order among the library units being developed, which in turn requires that the creation of library units be synchronized among the multiple programmers. Second, there may be valid reasons why library units with the same name are being developed by more than one programmer. For instance, two programmers inadvertently choose the same name for a library unit they are developing, or two programmers check out the same library unit from a version control mechanism and each start modifying it. In both examples, "throwing out" one of the units is unacceptable; rather, the environment should immediately inform both programmers so that they are aware of the problem and can negotiate to solve it.

The naming constraints specified above, when interpreted in the context of a multi-user

---

[7]The semantic equations defining the attributes *parent_name* and *simple_name* are omitted in figure 4-2.

environment, can be checked using three techniques. The first determines that the names of a specified collection of segments are unique; it checks that specification segments (which represent library units) have distinct names, and that there is at most one implementation segment for each specification segment and at most one nested segment for each body stub in a parent segment. The second technique checks for missing segments; there must be a body stub for each specification that requires it, and a subunit for each body stub. These two techniques involve extensions to the classical attribute grammar formalism. The last technique utilizes classical semantic equations, but it defines interface attributes and thus requires attribute propagation among segments; this technique checks uniqueness among the names specified in the body stubs corresponding to subunits that are descendants of the same library unit.

4.2.1. Detection of Duplicate Segment Names

We have previously presented an extension to AGs that handles detection of duplicate segment names for the flat segment organization [Micallef 88]. This work was motivated by a different problem, namely improving the performance of attribute propagation among segments for the special case of conglomerate attributes. Conglomerate attributes are collections of components from multiple program segments and are declared to be of a special type, a *finite binary relation*: the supporting algorithm limits propagation of updated components only to other segments where they are actually used.[8] We provide two operations for conglomerate interface attributes: (1) *assign*, which adds a component to the conglomerate attribute, and (2) *compute*, which returns a specified component. The *assign* operation has additional semantics for detecting duplicate names.

The general form of the *assign* operation is ASSIGN(R, d, r, *<attribute name>*, *<error string>*). This assigns $R \cup \{(d,r)\}$ to R. If the number of components in the conglomerate R whose key is equal to d becomes greater than one, then the attribute instance denoted by *<attribute name>* for each program segment defining these duplicate components is set to *<error string>*. (Programmers are informed of such errors by distinguishing the *error* attributes, and displaying them whenever their string values are non-empty.) A change in any one of a segment's error attributes is propagated as usual by the classical algorithm. The duplicately named segments are

[8]The finite binary relation is an extension of the finite function type used for aggregate attributes, where the multiple components are provided by the same segment; our algorithm is a decentralized extension of a single-user algorithm presented by Hoover and Teitelbaum [Hoover 86].

not considered as part of the program, however, and do not participate in other attribute propagation among segments.

The AG fragment in figure 4-3 shows how segments with the same name are detected for an ideal language where the program is composed of an unordered set of segments. The *assign* operator is used to build a conglomerate attribute *allexports* containing a tuple <module name, module exports> from each segment. If there is more than one segment with the same name, the error attribute of all such segments is set to a string such as "<-- duplicate module", which is displayed in the segments' texts.

```
program ::= set of module;
        { for each moduleSi, where (i = 1 .. | set of module |)
                assign(program.allexports,
                        moduleSi.name, moduleSi.exports,
                        moduleSi.error, "<-- duplicate module");
        }
```

Figure 4-3: *Checking for Duplicate Names in Ideal Language*

After an edit adds or deletes a segment, or changes the exports clause of some segment, the value of the *allexports* attribute is recomputed according to this semantic equation. This recomputation is performed incrementally by using the previous value of the attribute, the changed tuple, and whether the tuple is being added, deleted or modified.

Using the *assign* operator to detect segments with duplicate names in Ada is unsatisfactory for two reasons. First, when multiple conglomerate attributes are defined on the same set of segments, the check for duplicate segment names is repeated for each conglomerate since the *assign* operation is the only available operation for constructing conglomerate attributes. Multiple conglomerate attributes are required for the set of specification segments, as will be explained in sections 4.3 and 4.4.) Not only does this cause unnecessary attribute evaluation and propagation, but it also results in multiple semantic equations defining the error attribute that indicates duplicate segment names.

The second reason why the *assign* operator is unsatisfactory arises because of the new interconnection mechanism between two segments, namely the **interface** construct, which connects a parent segment to a child segment. In this case, no conglomerate attributes can be defined, and therefore the *assign* operator cannot be used to detect multiple child segments

(logically) connected to the same point in the parent segment.

Our new solution for detecting duplicate segment names consists of defining a new operator, *is_unique*, which checks that segment names are unique. The *assign* operator is redefined to only add components to the conglomerate attribute. The *is_unique* operator is invoked as shown in figure 4-4; in this example, the operator checks whether the segment's name is unique in the collection of segments specified in the **for each** construct. If it is not, then the segment's error attribute is set to the string "<-- duplicate segment". Note that this semantic equation involves information from multiple segments, and is thus performed in the APL; any resulting changes to a segment's attributes are propagated to the segment.

```
for each segment in a specified collection of segments
        segment.error = is_unique(segment.name)
                                ? ""
                                : "<-- duplicate segment";
```

Figure 4-4: *Using the is_unique operator*

We provide two ways for specifying collections of segments, corresponding to the **set** and **interface** constructs respectively. If, in the context-free grammar, the construct "**set of** X" appears as the right-hand side of a production *p*, then a semantic equation associated with *p* may use the construct "**for each set element** X" to refer to each segment of type X known to the APL. If in the context-free grammar the construct "**interface**(Y)" appears in the right-hand side of a production *q*, then a semantic equation associated with *q* may use the construct "**for each matching** Y" to refer to each matching segment of type Y known to the APL, where matching of segments is performed in the APL as described in the previous section.

Figure 4-5 gives an extended attribute grammar that performs the class of semantic checks that are equivalent to detecting segments with duplicate names. The semantic equations associated with production *p1* check that the names of library units (represented by specification segments) are distinct. In this case, the collection of segments used by the *is_unique* operator is the set of specification segments known to the APL. Checking that there is at most one body corresponding to a subprogram or package specification is done in the semantic equations associated with productions *p2* and *p3*, respectively. In this case, the collection of segments where the uniqueness check is performed is the set of child segments matched to the parent segment. The check that there is at most one subunit corresponding to each body stub is similar.

```
p1:  ada_program ::= set of specification_segment;
                { for each set element specification_segment
                        specification_segment.error =
                                is_unique(specification_segment.name)
                                        ? ""
                                        : "<-- duplicate specification segment";
                }

p2:  specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
                { for each matching implementation_segment
                        implementation_segment.error =
                                is_unique(implementation_segment.child_interface_name)
                                        ? ""
                                        : "<-- duplicate implementation segment");
                }
p3:         | context_clause package_declaration [ interface(implementation_segment) ]
                { for each matching implementation_segment
                        implementation_segment.error =
                                is_unique(implementation_segment.child_interface_name)
                                        ? ""
                                        : "<-- duplicate implementation segment");
                }

p7:  interface_body_stub ::= body_stub interface(nested_segment);
                { for each matching nested_segment
                        nested_segment.error =
                                is_unique(nested_segment.child_interface_name)
                                        ? ""
                                        : "<-- duplicate nested segment");
                }
```

Figure 4-5: *Checking for Segments with Duplicate Names in Ada*

The results of the *is_unique* operator are noted for each segment in the APL, to ensure that duplicate segments do not participate in attribute propagation and that conglomerates contain only components from uniquely named segments.

4.2.2. Detection of Missing Segments

The attribute grammar fragment of figure 4-6 illustrates the technique used to detect that a segment required as part of an Ada program is missing. A specification segment that represents a subprogram specification must always have a corresponding body. The semantic equation associated with production *p2* checks whether the number of matching implementation segments for the subprogram specification segment is zero, and if it is it sets an error attribute to the appropriate error message string. The semantic equation for checking that there is a corresponding subunit for each body stub is similarly defined for production *p7*.

```
p2:  specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
                { specification_segment.error2 =
                        (count(matching implementation_segment) == 0)
                                ? "<-- missing subprogram body for this specification"
                                : ""
                }
p3:         | context_clause package_declaration [ interface(implementation_segment) ]
                { specification_segment.error2 =
                        (specification_segment.requires_body == false)
                                ? ""
                                : (specification_segment.has_interface_node == false)
                                        ? "<-- package body is required"
                                        : (count(matching implementation_segment) == 0)
                                                ? "<-- missing package body for this specification"
                                                : ""
                }

p7:  interface_body_stub ::= body_stub interface(nested_segment);
                { interface_body_stub.error2 =
                        (count(matching nested_segment) == 0)
                                ? "<-- missing subunit for this body stub"
                                : ""
                }
```

Figure 4-6: *Checking for Missing Segments in Ada*

For specification segments that represent package specifications, checking for a missing corresponding body is slightly more complicated because some packages do not require a body. Each package specification segment has an attribute, *requires_body*, which is assigned true if a package body is required (this can be determined from the package specification), and false otherwise. Another attribute, *has_interface_node*, indicates whether the specification segment contains the optional **interface** to an implementation segment or not. If the specification segment requires a body (*i.e.*, the attribute *requires_body* is true) but the specification segment does not contain the optional **interface** node (*i.e.*, the attribute *has_interface_node* is false), then an error is reported indicating that a package body is required for this specification. If the specification segment requires a body and the specification segment does contain an **interface** node, then an error is reported if there are no matching implementation segments to this node.

4.2.3. Subunit Naming

Since we have already checked that there is at most one subunit for each body stub declared in a parent unit, checking that the names of subunits that are descendants of a single library unit are distinct involves checking that all the body stubs in segments that are descendants of the same specification segment have distinct names. This can be done using standard semantic equations as shown in figure 4-7, with the exception that attributes are propagated among segments.

```
p2:  specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
                { implementation_segment.subunits_in = EmptyList; }
p3:          | context_clause package_declaration [ interface(implementation_segment) ]
                { implementation_segment.subunits_in = EmptyList; }
p4:          | context_clause subprogram_body;
                { subprogram_body.subunits_in = EmptyList; }

p7:  interface_body_stub ::= body_stub interface(nested_segment);
                { interface_body_stub.error =
                                member(body_stub.simple_name, interface_body_stub.subunits_in)
                                        ? "<-- duplicate subunit name"
                                        : ""
                  interface_body_stub.subunits_out = (body_stub.error == "")
                                        ? add(body_stub.simple_name, interface_body_stub.subunits_in)
                                        : body_stub.subunits_in
                  nested_segment.subunits_in = interface_body_stub.subunits_out;
                }
```

Figure 4-7: *Checking that Subunits with Same Ancestor Library Unit Have Distinct Names*

The attribute grammar shown in figure 4-7 constructs a list containing the names of all the body stubs declared in segments that are descendants of a single specification segment. This list is ordered: body stubs in the same declarative part of a segment are inserted in the same order they appear in the declarative part, and body stubs in a segment X are inserted before those in segments nested in X. The list is initialized to the empty list at the outermost segment. Each body stub has two associated attributes, *subunits_in* and *subunits_out*. These attributes represent partial lists; *subunits_in* is the list of body stub names that appear before this body stub, not including itself, while *subunits_out* is the list of body stub names that appear before this body stub, including itself. Before adding the name of a body stub to the list, however, it is checked that the name does not already appear in the list. If it does, then an error is reported. The partial lists are propagated both among the declarative items of one segment (we omit these semantic equations), as well as from a parent segment to its children. The latter is performed by the semantic equations that initialize the *subunits_in* attribute for nested segments.

One problem with this method for checking for distinct subunit names is that an ordering among the segments is imposed. This ordering, which is derived from the text of the program units, may be different from the order in which the nested segments were added to the program. As for the other kinds of duplicate segments, it would be more appropriate to flag all subunits (or the corresponding body stubs) that have the same name as ''duplicate''. To be able to do so, a third method for specifying a collection of segments is required, which would allow the specification of the collection of implementation and/or nested segments that are all descendants of the same specification segment. Then the *is_unique* operator can be used to detect duplicate names along the same lines as described in section 4.2.1.

4.3. Context

In figure 4-8, we present an extended AG fragment for determining the *contexts* accessible to each kind of segment, that is, the collection of program entities that are not declared locally within the segment but which may be accessed from within the segment.

Each segment has a context clause at its head, which may contain with clauses naming the specification segments that are needed by the segment.[9] We call this the *imported* context of a segment. To compute the imported context of a particular segment, a conglomerate attribute *all_contexts* is constructed from the visible declarations of all (distinctly named) specification segments. The imported context of a segment, stored in its *imported_context* attribute, is computed by selecting the components of *all_contexts* for the segments specified by the with clauses. This selection is performed by means of the function *compute_subset*, which calls the operation *compute* (defined in section 4.2.1) for each distinct with clause name.

Implementation and nested segments have additional contexts accessible to them. The context clause of a specification segment also applies to its corresponding implementation segment; furthermore, all declarations of the specification segment are visible in the corresponding implementation. Therefore, the context of an implementation segment consists of the union of its imported context, the imported context of its corresponding specification, and the (visible and private) declarations of the corresponding specification. The *union* function removes duplicate occurrences of the same context imported by both specification and implementation segments. In

[9]Any segment may refer to the public declarations of the STANDARD package, and in the following we assume that this package is in the context clause of all segments.

figure 4-8, we show the semantic equations for determining contexts of packages; subprograms are handled similarly.

```
p1: ada_program ::= set of specification_segment;
                { for each unique set element specification_segment
                        assign(ada_program.all_contexts,
                                specification_segment.name, specification_segment.visible_decls) }

p3: specification_segment ::= context_clause package_declaration [ interface(implementation_segment) ]
                { package_declaration.imported_context =
                        compute_subset({ada_program.all_contexts}, context_clause.with_names);
                  implementation_segment.corresponding_spec_imported_context =
                        package_declaration.imported_context;
                  implementation_segment.corresponding_spec_declarations =
                        package_declaration.local_declarations; }

p6: implementation_segment ::= context_clause package_body;
                { package_body.imported_context =
                        compute_subset({ada_program.all_contexts}, context_clause.with_names);
                  package_body.context = union(package_body.imported_context,
                                        implementation_segment.corresponding_spec_imported_context,
                                        implementation_segment.corresponding_spec_declarations); }

p7: interface_body_stub ::= body_stub interface(nested_segment);
                { nested_segment.parent_imported_context = body_stub.parent_imported_context;
                  nested_segment.parent_declarations = body_stub.decls_out; }

p8: nested_segment ::= context_clause subunit;
                { subunit.imported_context =
                        compute_subset({ada_program.all_contexts}, context_clause.with_names);
                  subunit.context = union(subunit.imported_context,
                                        nested_segment.parent_imported_context,
                                        nested_segment.parent_declarations) }
```

Figure 4-8: *Determination of Contexts Accessible to a Segment*

Nested segments, in addition to their own imported context, have the same context effective at the corresponding body stub in the parent segment. This means that the context clause of the parent also applies to the nested segment, and all declarations of the parent that occur before (and including) the body stub are also visible in the corresponding nested segment. In figure 4-8, we assume that the attributes *parent_imported_context* and *decls_out* associated with *body_stub* contain the imported context of the segment in which the body stub appears and the symbol table for the declarations prior to and including the body stub, respectively.

4.4. Compilation Order

Determining whether there is an order in which the units and subunits comprising an Ada program can be submitted for compilation is equivalent to checking that the graph of compilation dependencies among library units is acyclic. There is a compilation dependency between two library units *A* and *B* if *A* appears in the context clause of *B*, meaning that *A* must be compiled before *B*. There is no need to consider secondary units since these can not be listed in the context clause of any other unit. (Note that the INLINE pragma, which we do not currently handle, may result in compilation dependencies involving secondary units.)

```
p1: ada_program ::= set of specification_segment;
                    { for each unique set element specification_segment
                          assign(ada_program.compilation_dependencies,
                                 specification_segment.name, specification_segment.with_names; );
                      ada_program.error = acyclic(build_graph(ada_program.compilation_dependencies)); }
                                 ? ""
                                 : "<-- cycle in compilation dependencies"; }

p2: specification_segment ::= context_clause subprogram_declaration interface(implementation_segment)
                    { specification_segment.name = subprogram_declaration.subprogram_name;
                      specification_segment.with_names = context_clause.with_names; }
```

Figure 4-9: *Checking if an Order for Compiling Units in Ada Program Exists*

Figure 4-9 illustrates an extended AG that determines whether or not an order exists for compiling the segments comprising an Ada program being developed in a MERCURY environment. Each specification segment has two attributes, *name*, which is the name of the subprogram specification, subprogram body, or package specification represented by this segment, and *with_names*, which is the list of names that appear in with clauses in the context clause of the segment. (The similar semantic equations defining these attributes for the other two productions defining specification segments are omitted.)

A conglomerate attribute, *compilation_dependencies*, is constructed by means of the **assign** operator, where each component of the conglomerate contains the name of a specification segment and the list of with clause names for that segment. A directed graph is built from this conglomerate attribute by the function *build_graph*. Each component of the conglomerate contributes a vertex to the graph (the name of the specification segment defining the component), and edges from each with clause name for that component to the segment's name.

The compilation dependency graph is then checked for cycles using another function, *acyclic*. If

there is a cycle in the graph, an error message is assigned to the error attribute of the *ada_program* node. This error attribute can be propagated to all the specification segments, or only to the one representing the main subprogram, where it is displayed for the programmers' information.

Both the conglomerate attribute *compilation_dependencies*, and the graph constructed from this attribute can be recomputed incrementally when one of the components of the conglomerate changes. If a new component is added (or removed) from the conglomerate, the vertex representing the name of this component and edges representing local compilation dependencies for this component are added (or deleted) from the graph. If a with clause in the context clause of some specification segment changes, an edge is added or deleted from the compilation dependency graph depending on whether a with clause name was added or deleted. (Changing a name is treated as a delete followed by an add.) However, the test for acyclicity cannot be performed incrementally, so this test may be expensive to compute after every affecting edit; it would be more practical to evaluate the corresponding error attributes only on demand.

5. Related Work

The Rational Environment [Archer 86] is a commercial product that supports language-sensitive editing and incremental compilation for Ada. Rational uses Diana trees [Goos 83] as the internal representation, which are similar to the attributed parse trees used by MERCURY. Like Rational, a production-quality version of MERCURY would also provide a text editing front end in addition to template editing.

MERCURY supports what Rational calls the installed state, where semantics as well as syntactic checks have been completed, both within a compilation unit and with respect to the units they reference. We do not support what Rational calls the coded state, where code generation has been completed. However, attribute grammars have been used for both research and commercial compilers [Kastens 82, Farrow 84]. Adding incremental compilation to MERCURY would primarily involve translating a full attribute grammar for Ada into our notation,[10] a tedious task, as well as the minor intellectual challenge of restricting propagation of interface attributes representing object code to when linking among compilation units is actually desired (otherwise,

[10]Our notation is essentially the same as SSL, provided by the Synthesizer Generator, except for new constructs such as **set** and **interface** described above.

the interprocess communication costs would be immense since the tiniest program change would slightly modify the object code).

Although MERCURY does not attempt to provide a complete software development environment, with execution and debugging facilities as provided by Rational, it has one primary advantage in that it supports change propagation while Rational supports *change simulation.* Assume several programmers are simultaneously editing their Ada compilation units. When one programmer makes a change, he can request that it be checked against the Ada units that it references, and he will be informed of any inconsistencies. This is one half of change simulation, and this facility is supported by Rational. The other half is for the programmer to request that his change be checked against the Ada units that reference it, and he will be informed of any inconsistencies. Although this mode is not directly supported by Rational, the same effect can be achieved through the `Show_Usage` command. We call this change simulation, because it need not be committed; the programmer can easily change his mind about making the change if adverse consequences would result.

Now consider the possibility that when one programmer makes a change, it is checked against both the Ada units that it references and that reference it. The programmer who makes the change is informed of any errors with respect to the units it references, while the other programmers are notified about any inconsistencies between their own units that reference the changed unit and this unit. This change propagation is supported by MERCURY; change simulation is a special case of change propagation where all newly introduced errors are reported only to the requesting user. The important distinction is the automatic notification of all other programmers affected by changes made by any one programmer. Although the change still need not be committed, other programmers have become involved, leading to an opportunity for negotiation as to how to best resolve the inconsistency. Since most enhancements require changes in both the referenced and referencing units, we believe it is best to make sure everyone involved is kept up to date. As stated in the Rational paper, "early detection of problems minimizes wasted effort."

As described in a Rational product brochure [Rational 86], change simulation significantly improves productivity in industrial software projects using Ada, in particular, the development of Rational itself and related tools. Adams *et al.* [Adams 89] have completed an empirical study of day to day source code changes in another industrial software project using Ada. In this project,

approximately half of all unit recompilations were unnecessary in the sense that sufficient analysis of the source code could have circumvented the recompilations. This result validates our work, since change propagation performs as a side-effect the detailed semantic analysis necessary to detect which recompilations are actually necessary, and as its main concern, ensures that all project personnel know when their units need to be recompiled as well as modified.

6. Conclusions

The extensions we have made for Ada seem sufficient to handle almost all other programming languages, and will make it possible to generate multi-user environments for more complex languages than we currently can support. There are three important missing features, however: the generic packages and subprograms mentioned previously, arbitrary nesting of subsystems with consequent use of the **set** construct at any level in the program hierarchy, and support for the C include construct. We leave these for future work.

Acknowledgments

We would like to thank Yael Cycowicz and Travis Winfrey for their work on the implementation of MERCURY, and Mark Gisi and Dewayne Perry for their aid in understanding the relevant semantics of Ada. We have developed our support for Ada in accordance with the 1982 Ada Reference Manual [AdaTEC 82]. Simon Kaplan collaborated with the authors on the original AG extensions and decentralized incremental evaluation algorithms.

References

[Adams 89] Rolf Adams, Annette Weinert and Walter Tichy.
Software change dynamics *or* half of all Ada compilations are redundant.
In C. Ghezzi and J.A. McDermid (editor), *2nd European Software Engineering Conference*, pages 203-221. Springer-Verlag, Coventry, UK, September, 1989.

[AdaTEC 82] 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.

[Archer 86] James E. Archer, Jr. and Michael T. Devlin.
Rational's Experience Using Ada for Very Large Systems.
In *1st International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1-B.2.5.11. Houston TX, June, 1986.

[Demers 81] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 105-116. Williamsburg VA, January, 1981.

[Farrow 84] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4):77-93, October, 1984.

[Goos 83] G. Goos, W. A. Wulf, A. Evans, Jr. and K. J. Butler.
Lecture Notes in Computer Science. Volume 161: *DIANA — An Intermediate Language for Ada*.
Springer-Verlag, Berlin, 1983.

[Hoover 86] 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.

[Kaiser 87a] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *6th Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg VA, March, 1987.

[Kaiser 87b] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[Kaiser 88] Gail E. Kaiser and Simon M. Kaplan.
Parallel and Distributed Incremental Attribute Evaluation.
Technical Report CUCS-412-88, Columbia University Department of Computer Science, September, 1988.

[Kaplan 86] 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.

[Kastens 82] U. Kastens, B. Hutt and E. Zimmermann.
Lecture Notes in Computer Science. Volume 141: *GAG: A Practical Compiler Generator*.
Springer-Verlag, Heidelberg, 1982.

[Knuth 68] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[Micallef 88] 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, Stuttgart, January, 1988.

[Micallef 89] Josephine Micallef and Gail E. Kaiser.
Support Algorithms for Incremental Attribute Evaluation of Multiple Asynchronous Subtree Replacements (Extended Abstract).
Technical Report CUCS-450-89, Columbia University Department of Computer Science, November, 1989.
Submitted for publication.

[Perry 89] Dewayne E. Perry.
The Inscape Environment.
In *11th International Conference on Software Engineering*, pages 2-9. IEEE Computer Society Press, Pittsburgh PA, May, 1989.

[Rational 86] Rational.
Information on Ada Software Development Products and Technologies.
1986

[Reps 83] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[Reps 89] Thomas W. Reps and Tim Teitelbaum.
Texts and Monographs in Computer Science: The Synthesizer Generator A System for Constructing Language-Based Editors.
Springer-Verlag, New York, 1989.

[Uhl 82] G. Goos and J. Hartmanis (editor).
Lecture Notes in Computer Science. Number 139: *An Attribute Grammar for the Semantic Analysis of Ada.*
Springer-Verlag, Heidelberg, West Germany, 1982.

I. MERCURY Environment Example

```
MODULE M ;
   EXPORT x ;
   FROM N IMPORT y;
   ...
END; /* M */

MODULE N ;
   EXPORT y ;
   FROM M IMPORT x ;
   ...
END; /* N */
```

Figure I-1: Skeleton of Program with Two Modules

```
MODULE M ;
   FROM N IMPORT y;
   ...
END; /* M */

MODULE N ;
   EXPORT y ;
   FROM M IMPORT x;  <-- cannot import this identifier
   ...
END; /* N */
```

Figure I-2: Error Notification after Program Change

Consider the case where *Dick* and *Jane* are editing the simple program shown in figure I. *Dick* can edit module *M* only and *Jane* can edit *N* only. Suppose that *Dick* deletes *x* from the export list of *M*, as shown in figure I. Then the ''*error*'' indication appears immediately on *Jane*'s screen, and *Jane* can scroll to the actual location of the error in module *N* and see the specific error message "<-- cannot import this identifier".

II. Small Attribute Grammar Example

Figure II-1 gives an example of an AG fragment for declarations in a Pascal-like programming language. There are four productions, *p1* through *p4*. Each nonterminal occurrence in a production has associated attribute instances and semantic equations. An attribute *a* associated with a nonterminal symbol *X* is denoted by *X.a*. Occurrences of the same nonterminal instance within one production are distinguished by the use of a numerical suffix (*e.g.*, 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 II-1 builds a symbol table for all declared

identifiers, and also marks as erroneous identifiers that are declared more than once.

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

Decl:   { synthesized attributes:  SymTabOut, error;
          inherited attributes:    SymTabIn; }

Id:     { synthesized attributes:  Name;
          inherited attributes:    Ø;  }

Type:   { synthesized attributes:  TpKind;
          inherited attributes:    Ø;  }
```

Context-free symbols of the attribute grammar 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); }
```

Productions of the attribute grammar fragment and their semantic equations

Figure II-1: *An attribute grammar example*