A VHDL Compiler Based on Attribute Grammar Methodology

Rodney Farrow

Declarative Systems, Inc.

Alec G Stanculescu

Vantage Analysis Systems, Inc.

Abstract

This paper presents aspects of a compiler for a new hardware description language (VHDL) written using attribute grammar techniques. VHDL is introduced, along with the new compiler challenges brought by a language that extends an Ada subset for the purpose of describing hardware. Attribute grammar programming solutions are presented for some of the language challenges.

The organization of the compiler and of the target virtual machine represented by the simulation kernel are discussed, and performance and code-size figures are presented.

The paper concludes that attribute grammars can be used for large commercial compilers with excellent results in terms of rapid development time and enhanced maintainability, and without paying any substantial penalty in terms of either the complexity of the language that can be handled or the resulting compilation speed.

1 Introduction

This paper describes the design and implementation of a compiler for the new hardware design language VHDL. VHDL was developed by the DoD as part of the VHSIC program. In December of 1987 VHDL was adopted as an IEEE Standard [11]. VHDL is a large and semantically rich language that borrows much of its structure from the programming language Ada, including:

- separate specification of interfaces and implementations,
- packages and the ability to selectively import declarations therefrom,
- the program library and separate compilation paradigm,
- user-defined types and subtypes and implicit declaration of operators for these,
- overloaded functions and user-overloadable built-in operators,
- nested function declarations and lexical scoping rules for name resolution,
- overloaded enumeration constants,
- dynamically constrained arrays and slices of them,
- aggregate notation for specifying arrays and records in terms of their [scalar] components,
- a restricted form of Ada-renaming called *aliasing*,
- a restricted version of generic compilation units.

VHDL then adds many constructs and features of its own to support aspects of hardware design that are not reflected in a general-purpose programming language. These include:

- new compilation units: entity (interface to a device), architecture (generic body of a device), and configuration (concrete implementation of a device),
- defaults rules for configuring an architecture if no explicit configuration is given,
- user-defined attributes and specification of their values,
- signal objects: signal assignment semantics, bus resolution functions, disconnect specification,
- implicit guard signals and guarded statements,
- user-defined input- and output- conversion-functions for arguments of subprogram calls and device instantiations that can be individually and separately specified at each point-of-call or instantiation,
- simulation time synchronization: simulation cycle semantics and associated predefined attributes,
- processes and their scheduling, and wait statements.

Besides being one of the first compilers for a significant new langauge, our VHDL compiler is innovative in that it was designed as an attribute grammar[1] and is automatically generated from this attribute grammar by a commercially-available translator-writing-system [12]. This approach enhanced our productivity and enabled us to make significant changes to our code-generation strategy very late in the development process. Several compilers have been written as AGs and automatically generated [10, 7, 5] but few of these have been commercial products; most have been research projects or prototypes. To the best of our knowledge, our VHDL compiler constitutes the largest complete AG application to date, containing more than 600 productions and 9,000 semantic rules.

[1] attribute grammars are hereinafter refered to as AGs

The VHDL compiler is a component of the VantageSpreadsheet^{TM} behavioral simulation environment offered by Vantage Analysis Systems, Inc. The compiler is invoked by either a menu-based or script-driven user interface, either directly for compiling VHDL textual descriptions or indirectly in the process of compiling schematic descriptions that have a VHDL correspondence. It is a complete, tested, production-quality compiler that has compiled hundreds of thousands of lines of customer's VHDL models.

The output of the VHDL compiler is source code in the C programming langauge that is compiled and then combined with other elements of the simulation environment to simulate the circuits described by the VHDL. Thus, we need not address many aspects of code generation, such as register allocation and instruction selection. However, other code generation problems still need to be solved by the VHDL compiler; e.g. references to up-level variables from within nested subprograms is supported in VHDL but not in C, and so the code generated by the VHDL compiler must implement this construct.

In this paper we draw on our experience in this project to address two questions:

- what features of VHDL were hard to implement and why,
- what are effective programming techniques, or *idioms*, that can be used to build compilers with AGs.

The rest of the paper is organized into four sections. The first section describes the compiler. It includes an overview of the compilation paradigm used by the compiler and the software architecture we designed to implement that paradigm. It also includes a discussion of the size and composition of the compiler itself and its performance. The second section discusses problems with the language definition that we found in the course of implementing the language. Our discussion attempts to understand how these problems were introduced, but we do not propose solutions because we feel there is as yet inadequate experience with the language. The third section discusses three *idioms* for designing AGs that we found to be useful. These are *cascaded evaluation*, the creative exploitation of *implicit semantic rules*, and the construction of large applicative data-structures as a result of attribute evaluation. The fourth section presents lessons we learned about compiling VHDL and using AGs to build a compiler for a new language.

2 Compiler Architecture

The VHDL compiler is a component of the VantageSpreadsheet^{TM} behavioral simulation environment. It is invoked by either a menu-based or script-driven user interface, either directly for compiling VHDL textual descriptions or indirectly in the process of compiling schematic descriptions that have a VHDL correspondence.

The compiler accepts a file containing compilation units, a list of compiler directives, a working library where the successfully compiled units are placed and a reference library which can be referenced in addition to the work library but which can not be updated.

2.1 Target Machine

The purpose of the VHDL compiler is to produce a computer program that, when executed, simulates the circuit described by the VHDL source code. This is accomplished by generating C code that in turn is compiled and linked into a simulator for the particular circuit.

At an abstract level the compiler can be viewed as generating code for a virtual machine that is configurable and programmable in terms of C primitives.

The virtual machine consists of four modules: (1) Simulation Kernel, (2) Runtime Support, (3) VHDL I/O, (4) Name Server.

The runtime support functions perform all the predefined VHDL operations. The VHDL I/O functions support the specific VHDL I/O. The Name Server provides the means of identifying by name each object in the simulated system.

2.2 Compiler Core

The compiler is organized around an attribute grammar that describes the syntax of VHDL (both context-free and context-sensitive syntax). This AG also specifies our *simulation semantics* for the language; i.e. the C source code described in the last section. Our VHDL AG is input to the Linguist^{TM} translator-writing-system, which from it automatically generates more than 60 percent of the total source code of the VHDL compiler.

Semantic rules of the AG can be either in-line expressions, or calls to out-of-line, separately-compiled functions, or some combination of these. If a complex expression needs to be used as a semantic rule at many different places in the AG then it makes sense to abstract this into an out-of-line function. These functions are written in C and account for 18 percent of the compiler.

VHDL supports a separate compilation paradigm that is modeled after that of Ada and it is deeply embedded in the semantics of the language. Our compiler supports a machine-readable intermediate language that is generated for each separately-compilable unit and read in when that unit is referenced from another. We refer to such references as *foreign references* and to the referenced unit as a *foreign unit*. The intermediate language is called VIF, an acronym for VHDL Intermediate Format. The structure of the VIF is described in a special-purpose, declarative notation that is read by yet another special-purpose program[2] that generates declarations for this data, and generates C code that manipulates the VIF. This code:

- writes the VIF to disk,
- reads the VIF from disk, resolving any nested foreign references,
- allocates individual nodes of the VIF,

[2]this program is also written as an AG and generated by Linguist; more evidence that when one receives a hammer, one begins to see the world as a nail.

- produces a human-readable form of the VIF (used for both debugging and documentation).

The rest of the compiler consists of functions written in C that interface the automatically-generated attribute evaluator to its environment. This includes such tasks as reading command-line arguments, opening input and output files, directing outputs of the compiler to appropriate places in the filing system, supplying basic memory management facilities, etc. Figure 1 depicts this compiler organization.

Figure 2 gives the size of our compiler, and its components, in terms of the number of lines of source code in the compiler and, where applicable, the number of lines of C source code generated from this. All figures for number of source code lines reflect text that has been stripped of blank lines and comments.

The compiler compiles VHDL at a little more than 1000 lines per minute on an Apollo DN4000, unless it is processing configuration units, in which case it's not as fast.[3] This speed includes the time necessary for the host system's C compiler to compile the generated model, which accounts for 20 to 30 percent of the total time. Substantial time is spent rading, fixing up, and writing the VIF for foreign compilation units: from 40 to 60 percent of the total time. Our preliminary analysis indicates that more than 80 percent of the time is spent doing these sorts of tasks, and others such as memory management. The time spent walking the parse tree and evaluating attributes is a very small percent.

3 Issues in compiling VHDL

VHDL can be viewed as an extension of an Ada subset. The extensions are supporting the hardware description specific features. The extensions to Ada produce several interesting compiler challenges. This section describes some of them.

3.1 Partial specification of arguments versus specification by aggregate

Consider the case of formal parameters that are of composite types. The association of formal to actual parameters may be done at the level of subelements of the formal parameters. For example, a formal parameter F of a record type with two fields A and B, may be associated to two objects X and Y by separately associating F.A to X and F.B to Y. This feature is called partial specification of arguments.

VHDL supports partial specification of arguments, provided that all the associations for one argument are contiguous. However, there is no syntactic construct to lump together all partial specifications for the same argument (using the aggregate construct perhaps). Unlike the aggregate notation, the partial specification of arguments in

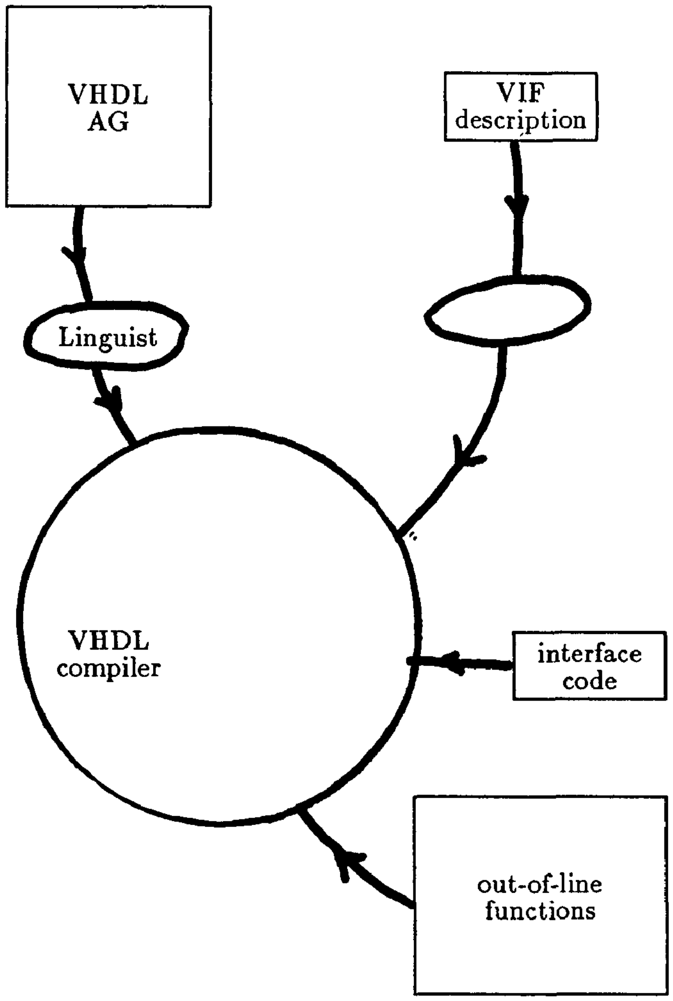

Figure 1: Organization of the VHDL Compiler

	source	[generated] C
AG	16827 (37%)	67919 (62%)
VIF description	1265 (3%)	14200 (13%)
out-of-line func	20845 (45%)	20845 (19%)
interface code	7132 (15%)	7132 (6%)
total	46069 (100%)	110096 (100%)

```
total manpower to develop compiler:
    82 man-months in 25 elapsed months
```

Figure 2: Summary of the VHDL Compiler

[3]Configuration units typically consist of very few source lines that cause large data structures built by compiling other compilation units to be read into memory and edited according to both explicit and implicit rules; the bulk of the work in processing these units is in reading and traversing these data structures rather than analyzing the source code of the configuration unit.

VHDL allows intermixing the specification of different levels in the hierarchy of the complex type. Under these circumstances, it is expensive for the compiler to enforce that the specification is complete and consistent. For the same reason, it is also difficult for the human reader to verify the correctness of the code.

3.2 Extending visibility by selection

Visibility by selection is a visibility rule by which names become visible and may be referenced at a given point in the code. In Ada there are two cases in which names may become visible by selection: after the dot in the structure field reference, and before the => token in by-name parameter association. In both these situations only a single identifier is legitimate.

The concept of visibility by selection has been extended to other areas of the language in VHDL. It applies to the position after the tic (') in the attribute reference, and to the actual designator portion of the generic and port map aspect. The problem has been further complicated by allowing both output conversion functions in by-name parameter associations and partial specifications of by-name parameters.

For example, in Ada the formal part of a by-name parameter association is a simple identifier. But because in VHDL one can have partial specification of formals and because formals can have output specification functions associated to them then `X(Y)` can denote either a partial specification of an element of an array or an output conversion function `X` applied to the formal designator `Y`.

Another example of difficulties introduced by extending the Ada visibility by selection, is represented by attribute references. User-defined attributes can be attached to many kinds of objects. The visibility by selection makes it possible for the user-defined attribute to conflict with predefined names. This makes it more difficult to identify the meaning of a name. For example, `X(T'REVERSE_RANGE)` could be an array if `T` is an aray object, or a constrained array subtype. In this case `REVERSE_RANGE` would be considered to be the predefined attribute. However, the same `X(T'REVERSE_RANGE)` could be an element of the array `X` in case `T` is an object that has the user-defined attribute `REVERSE_RANGE`.

3.3 Generic Descriptions, Configuration Information

VHDL compilation units are: package specifications, package bodies, entities, architectures, or configuration units.

Package specifications and package bodies are similar to the corresponding Ada-constructs. The main differences are that VHDL packages may not have generic parameters, and that VHDL packages may contain global signals. References to global signals in packages may be made only from within an entity or an architecture. This is conceptually different from Ada where all objects declared in the package specification are visible in the package body.

An entity can be viewed as the specification of the architecture. The architecture describes a circuit in generic terms, and the configuration provides instructions for editing the entities and architectures that describe a particular circuit. A hardware analogy, which proves to be very successful in explaining binding and default rules, is that the entity is the description of the interface (names of pins, voltage range acceptable for each pin, usage constraints such as setup and hold conditions) of a family of chips. A member of the familly corresponds to a particular set of actual values bound to the generic parameters.

Continuing the hardware analogy, the architecture can be viewed as board with sockets, in which chips can be plugged. The sockets are of different kinds. Each kind of socket is described by a component declaration in VHDL. Instances of sockets are represented by component instantiations. Configuration specifications may be present either in the architecture or in special compilation unit called the configuration unit. According to the hardware analogy the configuration specifications correspond to information regarding what actual chips to plug in the instantiated sockets. In the cases in which configuration information is apparently missing, the default rules apply.

Comparing the Ada generic capabilities to VHDL one can say that VHDL uses two layers of restricted versions of the Ada generic mechanism. The first layer consists of the VHDL-generic parameter that can be associated to entities and blocks. This is a restricted version of the Ada generic capability, where the generic parameter can only be a generic constant object. The second layer consists of the VHDL-component declaration that is similar in nature to the Ada-generic subprogram. The binding of actuals to generic formals in VHDL is done separately for each layer: using the VHDL-instantiation for the first layer, and the configuration specification for the second layer. It is important to note that the actual parameters for the first layer can be generic members of the second layer.

One of the problems that arise is how much of the generated code should be produced just by inspecting the entity and the architecture and what should be postponed until the configuration information is available. The nature of this tradeoff is similar to the one faced when compiling Ada generic packages. The details are somewhat different due to the layered approach of associating actuals to formals.

The configuration information can be obtained in three ways: from the configuration specification inside the architecture, configuration specification inside a configuration unit, and according to the VHDL default-mechanism. It is not clear at the time an architecture is compiled whether more configuration information is still to be provided since the lack of such information, at the time the configuration unit is compiled, triggers the default mechanism and the necessary information is produced by default.

A possibility which we explored was to generate code for the possible defaults and use that code in the case the default mechanism is triggered. This early code generation was intended to favor the use of the default mechanism. This solution has a more limited use than one might expect because the default mechanism is based on the usage-history of the design library, which makes the VHDL description itself non-deterministic. Specifically, the default for an architecture name in the binding of a component to an entity-architecture, is the latest compiled architecture for that entity. Therefore, any default configuration information that uses the latest compiled architecture, can be used only if the relevant set of latest compiled architectures does not change between the generation of the configuration code and the time at which the code is used.

3.4 Context Clauses

Names that may be referenced in a compilation unit must be visible at the point where they are referenced. To be visible a name must be declared either in the compilation unit itself (in an appropriate place), or in another compilation unit. If the name is declared in another compilation unit, it must be imported via some import mechanism.

VHDL differs from Ada with respect to the constructs by which names declared in foreign compilation units become visible. The `LIBRARY`-clause in VHDL and the `WITH`-clause in Ada, represent import mechanisms. In VHDL one cannot refer to a compilation unit which is in a library for which there is no library clause[4]. Similarly, in Ada one cannot refer to a name declared in a compilation unit for which there is no `WITH`-clause. The difference between the two import mechanisms is that the level of granularity for import control in Ada is the compilation unit as opposed to an entire library in VHDL.

`USE`-clauses in Ada provide a short hand notation whereby it is not necessary to specify a prefix for which there is a `USE`-clause, provided that there is no homographic collision due to the `USE`-clause.

VHDL does not have a `WITH`-clause. However, the VHDL `USE`-clause is richer in semantics, subsuming the semantics of Ada's `WITH`-clause. The VHDL `USE`-clause applies to compilation units as well as to names declared wihtin a compilation unit. Names declared within a compilation unit may be imported individually (to avoid homographic conflicts) or as a group.

When applied to all names within a compilation unit (using the suffix `.ALL`), the VHDL `USE`-clause is equivalent to a `WITH`-clause followed by a `USE`-clause in Ada. In this case the rules governing homographic conflicts apply.

The capability to import only some of the names declared within a compilation unit, and avoid homographic conflicts, is a feature present in VHDL and not in Ada. In the case in which a name that is referenced has a homograph in another imported context, and the homograph is not required by the unit under compilation, then the VHDL programmer has the possibility to import one by one exactly the referenced identifiers, thus avoiding the homographic conflicts. This VHDL feature requires the capability to import individual names rather than all the visible names of a compilation unit.

4 AG idioms for VHDL

4.1 Cascaded evaluation

One of the problems we faced in writing an AG for VHDL is that the natural, straight-forward phrase-structure for expressions is also ambiguous. An occurrence of an identifier may denote radically different things in a given syntactic context. For instance, in VHDL (as in Ada) the phrase `X ( Y )` could denote:

- a call to subprogram `X` with argument `Y`,
- a subscripted reference to array variable `X`,
- a slice of array variable `X` by range `Y`,
- a type conversion expression of the value denoted by `Y` to type `X`.

Which of these possibilities is actually denoted depends on the program objects to which the names `X` and `Y` are bound in this scope of the program. If the compiler is based on the natural phrase-structure for these expressions then `X` in `X ( Y )` should be recognized according to one of the productions `[subprogram_name ::= ID]` versus `[variable_name ::= ID]` versus `[type_name ::= ID]`. All three of these productions are legitimate in many contexts.

A similar problem, peculiar to VHDL because this language allows user-defined attributes, is the phrase `X ' REVERSE_RANGE` This could denote either a reference to a user-defined attribute named `REVERSE_RANGE`, which would produce a value, or a reference to the predefined attribute `REVERSE_RANGE`, which would produce a *range* suitable for defining a subtype or slicing an array. As above, which possibility is actually denoted depends on what `REVERSE_RANGE` means in the environment at this point in the program.

These ambiguities can not be resolved until information about the naming environment is available, i.e. until the symbol table has been built and can be consulted. However, the symbol table is itself a *result* of attribute evaluation. Thus, the accurate phrase-structure for expressions depends on the symbol table, which is built via semantic rules of the AG, and which thus depends on the phrase-structure of the program.

One way to deal with these difficulties might be to *unite*
the conflicting productions into a single production, e.g. `[name ::= ID]` instead of `[subprogram_name ::= ID]`, `[variable_name ::= ID]`, and `[type_name ::= ID]`. The semantics of such a ***united production*** would conditionally defined attributes based on which situation is later found to occur after symbol table information is available.

So long as the difference between phrase-structure is confined to a single production or a very few adjacent productions this is a feasible, although somewhat unaesthetic alternative. But for many languages, including VHDL, even a local choice such as this affects the phrase structure chosen for large parts of the adjacent source program.

For the example `X ( Y )`, the phrase structure that applies to `( Y )` can be that of:

- a one-element argument list,
- a one-element subscript list,
- the range of a slice, or
- the source expression of a type conversion.

Thus, *uniting* the possible phrase-structures for `X` leads us to *unite* as well the phrase structure for the adjacent

[4]Note: all VHDL compilation units have the implicit clauses `LIBRARY WORK; USE WORK.ALL;`

(Y). This process might result in a *united* production such as [expr ::= name (name)]. Unfortunately, this approach leads to the duplication of all the semantics for:

- an argument of a procedure call,
- a subscript element of an array reference,
- a range that is only a name,
- an expr that is just a name,
- a var_ref that is just a name.

Each of these separate sets of semantics must not only be part of the productions that derive the more general occurrences of these constructs, but also of the united production(s). Also, the united production(s) constitute a special case of other productions designed to find the phrase-structure of the general construct and cause parsing conflicts with them. Thus, the united production [name ::= ID] and the more general productions [func_ref ::= name (args)], [args ::= arg], [args ::= args , arg] together cause parsing conflicts in the LR-parser; indeed, these productions are ambiguous and the AG-author must be careful that the ambiguity is resolved in favor of the united production.

This proliferation of duplicate semantics and conflicting syntax becomes worse when, as in VHDL, these different interpretations can occur in contexts which overlap but are not identical. Thus, an array reference can be subsequently selected by a field of a record, but a type conversion expression can not. In such cases, the semantics of the united production must detect when an illegal use occurs and flag it as such.

The general problem of compiling two or more language constructs that are semantically diverse but syntactically identical is a familiar one for compiler writers who use LR parsing techniques. For us it was merely exacerbated by the size and complexity of VHDL and by the strict distinction between syntax and semantics imposed by the attribute grammar paradigm. For instance, when trying to write a Pascal compiler that uses an LR parser one would really like to use different productions to handle SUCC(X) if this denotes the predefined *successor* function for the type of X, as opposed to a call to a user-defined function named SUCC. Although the semantics involved are quite different, their syntax is identical and they can not be distinguished by the LR parser. The usual solution is to parse the construct according to one of the possibilities, but to have the associated semantic actions consult the symbol table before building the parse tree.

Compilers that don't use an LR parser often use more ad hoc techniques such as recursive descent parsing. These compilers can base their parsing decisions directly on information available in the symbol table and thus can parse according to what an identifer denotes in this particular context. Baker [2] described such a compiler for Ada. His discussion noted the same problems for Ada as we have described above for VHDL and his compiler resolved ambiguity in the grammar by having the scanner return different tokens for a name based on what the symbol table said a name was bound to at this point. Such a strategy requires very careful coordination between the scanner, the symbol table manager, and the semantic actions of the parser.

These latter two solutions are generally not available to an AG-based compiler because the semantic rules of an AG are associated with the same context-free phrase-structure as is used for parsing. Indeed, one of the reasons for using an AG-based complier generator is to avoid having to design and implement such an intermediate data structure. We originally tried to use the first solution mentioned above, that of *uniting* several conflicting productions into one and using semantic rules to distinguish between them. However, the size of VHDL and the many places where we had to do this soon grew overwhelming:

- the amount of duplicate semantics we had to write was substantial, even when as much as possible was abstracted into out-of-line functions,
- keeping track of the parsing conflicts and ensuring that they were resolved correctly was confusing and error-prone,
- the necessity that *united* non-terminals have combined sets of attributes caused the AG to be needlessly large.

The solution we finally chose was to write two AGs and then *cascade* or *pipeline* the translations they specified. The principal AG does not contain semantic rules for most of the aspects of compiling expressions; instead it merely synthesizes a simplified list of tokens that is input to the second AG. The second AG describes the semantics of an individual expression. The attribute evaluator generated for the expression AG parses the list of tokens built for an expression by the principal AG, and does attribute evaluation on that phrase-structure according to the semantics specified by the expression AG.

The intermediate language for expressions is refered to as LEF. LEF consists of a flat list of tokens with no other structure imposed on them. LEF tokens are pretty much the same as the VHDL tokens that can occur in expressions; they include literals of various flavors and punctuation such as parentheses, comma, etc. However, the symbol table is an attribute of the principal AG, not of the expression AG, and it is used to resolve identifiers so that ID is not a token of LEF; instead there are distinct tokens for *variable*, *type*, *subprogram*, *attribute*, *enum_literal*, etc. This allows the parser for the expression AG to distinguish the phrase structure of expressions based on what various identifiers denote in a particular context. Thus, very different phrase structure (of the expression AG) can be built for two identical pieces of VHDL source text, depending on to what the names in that source text are bound at this point in the program.

In illustration, consider the source fragment X (Y). If X is a subprogram and Y is a variable then the principal AG translates this to a string of LEF tokens [subprogram, '(', variable, ')'] which is parsed according to the expression AG's phrase-structure for a subprogram invocation. On the other hand, if X denotes a variable and Y denotes a type then this gives rise to the string of LEF tokens [variable, '(', type, ')'] which is parsed according to the phrase structure of an array slice.

The list of LEF tokens are built as attributes of symbols in the principal AG. These attributes are not treated at all specially by the translator-generating tool: they are undistinguished, user-declared attributes the same as any others. At those points in the AG where an expressions occur that are not themselves immediate constituents of larger expressions, the list of LEF tokens for this maximal expression is parsed and evaluated according to the semantics of the expression AG. This is written in the principal AG as an application of the separately-compiled, out-of-line function `exprEval`.

The main argument supplied to `exprEval` is the list of LEF tokens. Other arguments are the nesting level at which this expression occurs, the type expected for this expression (if this is known) the source line number of this expression, and flags indicating the context in which this expression occurs. `exprEval` returns several values; these are the result of evaluating the expression AG applied to the LEF token list. These values include a list of error messages (the null list if there were no errors), and the output translation of this expression.

A production in which this occurs is shown below. The guard expression of an `if`-statement is an example of a maximal expression.

```
stmt ::= if_KW expr then_KW stmts end_KW
             if_KW SEMI.

  stmt.CODE = TextOf("if( %t ){%t}",
                     EXPR_CODE,
                     stmts.CODE),

  stmt.MSGS = mergeMsgs( EXPR_MSGS,
                        stmts.MSGS ),

    EXPR_CODE:tp_text,
    EXPR_MSGS:tp_msgs  =
      exprEval( expr.LEF, boolean_type,
                if_KW.LINE, stmt.LEVEL,
                stmt.CONTEXT),

  stmts.CONTEXT = stmt.CONTEXT,
  stmts.LEVEL   = stmt.LEVEL,
 ;
```

The out-of-line function `exprEval` is itself a parser and attribute evaluator generated from the expression AG. Thus, we generate two different and distinct attribute evaluators, one for the principal VHDL AG and one for the expression AG. The evaluator for the former operates once per VHDL compilation unit and computes intermediate attribute values that are sequences of LEF tokens. The second evaluator operates once for each maximal expression in the source program by parsing the corresponding sequence of LEF tokens and returning a set of attribute values as a result. These values are the results, or *goal* attributes, of the expression AG. They are incorporated into the continuing attribute evaluation of the principal AG.

An important aspect of this *cascaded translation* technique is that it required no enhancement or modification of the translator-generating tool, Linguist. Constructing the lists of LEF tokens and doing attribute evaluation of them looks just like any other user-defined semantics in the principal AG. The implementation of `exprEval` consists of:

- wrapping a new functional interface around the evaluator Linguist generates for the expression AG, and
- supplying a scanner that reads tokens from the list of LEF tokens supplied as an argument to `exprEval`.

The evaluator for the principal AG is fed tokens by a scanner that reads source text from a file in the usual way. The expression evaluator is fed tokens by a trivial scanner that just takes the next LEF token off the front of the list. If L is this list of tokens then the expression scanner is just:

```
tp_list L;
scanner()
{ X = car(L);  L = cdr(L);  return(X); }
```

The Linguist tool supports a mechanism for incorporating values associated with tokens into attribute evaluation. In traditional settings such values might be the line and column number on which a token occurs, or the string that makes up the text of a token for a character literal or an identifier. We use this mechanism in cascaded translation to associate information with LEF tokens, such as the symbol table entry denoted by a **variable**, **subprogram**, or **type** token in the LEF. Thus, all the information associated with a variable by the principal AG is also available in the expression AG.

This *cascaded evaluation* paradigm is one way of importing into an AG framework the familiar technique of *multi-pass* or *staged* translations, which in turn is a way of partitioning a problem into more tractable and less complex pieces. Another way of doing this is the *attribute-coupled grammar* (ACG) approach of Ganzinger and Geigerich [8]. This paradigm also partitions a translation into two or more separate AGs in which the first AG specifies the input to the second AG. The advantage of attribute-coupled grammars is that they can be combined (by a suitably enhanced evaluator-generator) into a single AG, and the resulting evaluator need not also be partitioned into two distinct components. The disadvantage of ACGs is that the semantic rules of the first AG must specify what phrase-structure of the second AG is to be built for a given construct. Because of this feature of ACGs, the output of the first AG need not be reparsed into the phrase-structure of the second AG because that phrase-structure was already implicitly specifed by the semantic rules of the first AG. However, the ability, requirement even, to do such *reparsing* is an important feature of cascaded evaluation precisely because we find it hard to determine the phrase-structure of expressions based on the phrase-structure of the principal VHDL AG. We want to reparse portions of the source program because it is easier to find the appropriate phrase-structure that way than to write by hand semantic rules to find it. In terms of the trade-offs discussed earlier for *uniting* productions, ACGs would address the problem of proliferating duplicate semantics, but would not address the issue of syntactic conflicts between *united* productions and general-purpose productions.

The sizes of our two AGs is shown below. The expression AG is much smaller, of course, but it is of a respectable size; on the order of a simple AG for Pascal.

	VHDL AG	expr AG
productions	503	160

symbols	355	101
attributes	3509	446
rules(implicit)	8862 (6498)	2132 (1061)
max visits	3	4

4.2 Attribute classes and implicit rules.

It is often the case that an attribute is associated with many different symbols and denotes essentially the same thing for each of them. There are many examples of these in our VHDL AG, some of which are:

- `MSGS` - the list of all error messages generated for source code that is derivable from the non-terminal with which this attribute is associated,
- `LEVEL` - the number of subprograms within which this non-terminal is nested,
- `ENV` - the name-to-object binding used to resolve occurrences of identifiers in the source code derivable from this non-terminal.

Many of these attributes (although not only these attributes) serve to transfer information from one part of the semantic tree to the other, and the semantic rules that define them are often simple and don't vary much from one production to another. In fact, these simple, repetitive rules are often as many as half the semantic rules of a large AG. Because of this many AG-based translator-writing-systems have either a special notation to describe such rules [9] or will implicitly create such rules when they are omitted.

The system we used, Linguist, adopts the second approach and we found it to be both effective and widely applicable. An *attribute class* can be declared and instances of a class can be associated with various symbols, just as attributes are associated with symbols. However, if a required definition for some occurrence of an attribute class is left out of the semantic rules of a production, Linguist will supply an *implict* rule to define this attribute. The rule is based on whether the attribute is inherited or synthesized and on information supplied in the definition of the class. There are three basic kinds of implict rule that will be built to define `X.A`:

- `X.A = Y.A` where `Y.A` is some other occurrence of the same attribute class in this production,
- `X.A = u` where u is a constant specified in the attribute class declaration, or
- `X.A = m(Y.A, m(W.A, ... , Z.A) ... )` where `m` is an associative, dyadic function specified in the attribute class declaration, and `Y.A`, `W.A`, ..., and `Z.A` are other occurrences of this attribute class in the production.

The first kind of implicit rule might be supplied if `X.A` is either inherited or synthesized and is called a *copy-rule*; the other two forms can only be supplied if `X.A` is synthesized. The constant `u` is called the *unit-element* and the function `m` is called the *merge-function*.

A discussion of exactly what rules are implicitly generated and why these are good rules to implicitly supply can be found in [6, 12]. For this presentation we will just give an example that illustrates how we used this facility in the VHDL compiler.

The `MSGS` attribute is ubiquitous in our AGs because error messages may need to be issued at many different points in a source program. Such error messages must be concatenated with other messages and propagated to the root of the semantic tree as the value of various `MSGS` attributes. At the root they are used to define the `MSGS` attribute of the GOAL symbol, thus becoming a result of the translation, and hence available to be written to a file, displayed on a screen, etc. Consequently, if there is a production `[X ::= W Y Z]`, and symbol `Y` has a `MSGS` attribute, then `X` must also have a `MSGS` attribute else the value of `Y.MSGS` would not get propagated up the semantic tree. Even if production `[X ::= W Y Z]` doesn't generate any messages of its own it must still have the semantic rule

```
X.MSGS = concatMsgs(W.MSGS,
         concatMsgs(Y.MSGS, Z.MSGS))
```

It can be very tedious to supply all of these explicitly, and luckily we don't have to. By declaring `MSGS` to be an attribute class with `concatMsgs` as its *merge-function* we cause Linguist to supply *implicit semantic rules* for all the otherwise undefined `MSGS` attributes.

Our AGs for VHDL are replete with such attribute classes and Linguist uses them to create more than half of all the rules of the AGs (see table of previous section). In fact, our experience convinces us that we should go much further in this direction. Attribute classes should be expanded so that a class could contain many different attributes, both inherited and synthesized. Declarations associated with the attribute class would similarly be used to build implicit semantic rules for those productions that do not have all the rules that are needed. We think this would be useful because we found over and over that certain sets of attributes, mixtures of both inheritied and synthesized, were always occuring together in our grammars. For instance, there are the attributes needed to process an identifier, those needed for processing an expression, those needed for processing a declaration, those for processing a sequential statement, those for processing a concurrent statement, etc. If symbol `X` can derive a sequential statement then it needs to have the set of attributes for sequential statement, if `X` can derive an expression then it needs to have the set of attributes for expressions, etc.

If the declaration of these sets of attributes could be collected together in one place it would facilitate both the creation and maintenence of the AGs. Currently, if we need another attribute to process expressions we must go through and find all affected symbols and associate the new attribute with those symbols. It would be much easier and less error-prone to simply add the new attribute to a single class declaration (or delete an existing one that is no longer used) and let the toolset distribute the changes appropriately.

We were able to achieve a subset of the functionality we desire of such a mechanism by using a macro-processor preliminary to running Linguist. The VHDL AG contains macro definitions for symbolic names, e.g. `ENV_ATTRS`, `EXPR_ATTRS`, `STMT_ATTRS`, etc., to be strings that are appropriate as the declaration of a collection of attributes and attribute classes. These symbolic names are then used in the attribute declaration section of a non-terminal symbol declaration. The macro-processor expands these names into a list of attribute and attribute class declarations inside the non-terminal declaration.

This approach let us group attributes and associate all the attributes in a group with a particular symbol by simply naming the group. However, the implicit semantic rules are supplied based on individual attributes or classes; this mechanism does not allow the definition of implicit rules based on other attributes in the groups.

Something similar to this general strategy has been proposed by Alpern et.al. [1] in a formalism they propose for describing attributed *graph* grammars. They call their construct a *cable*. We think it would be well worthwhile to combine something like cables with Linguist's attribute class declarations to get a mechanism that allows one to:

- declare a set of attributes,
- associate all members of that set with a symbol, and
- have the toolset fill in implicit rules when necessary.

4.3 Applicative implementation of the symbol table.

The separate compilation mechanism of VHDL (and Ada) requires that a *foreign reference* be implemented by having the compiler read intermediate files that it created earlier when it processed the foreign compilation unit. Our intermediate language, VIF, is used for this. In most compilers a central *symbol table* or *dictionary* is the repository of information about objects and constructs declared by the user. In our VHDL compiler this is done by the VIF, both foreign VIF read from the library, and domestic VIF created as part of processing the current compilation unit. The VIF is specified in the AG and created through attribute evaluation. The value of some attributes in the AG is VIF nodes, which may contain links to other VIF nodes, which are filled in by copying the value of other adjacent attributes. A simple example is

```
sequential_statement ::=
        TGT_SIGNAL_reference LT_EQ opt_transport
                    waveform SEMI.
    ...
  sequential_statement.VIF =
    tnode_simpleSignalAsgn{
      TGT_SIGNAL_reference.VIF,
      waveform.VIF,
      opt_transport.PRESENT,
      SSS_KICK_NORMAL,
      LT_EQ.LINE
     },
    ...
```

One by-product of using the AG to build VIF is that, once built, the VIF can not be changed. Although this may seem like a serious hinderance, in practice it has worked out quite well. Scheduling the insertion, modification, and removal of information in the symbol table is not an issue for us — we only describe what information we want to know about objects and the attribute evaluator generator schedules evaluation of rules that reference the symbol table (i.e. VIF attributes) only when such information is known to be available.

For instance, consider the mapping from *identifer* to *object(s) denoted*. We refer to such a mapping as an *environment*. In many compilers this is a function of the symbol-table module. In our VHDL compiler there is an attribute called `ENV`, associated with any symbol that could produce an identifier reference, that represents this mapping. `ENV` values are themselves trees whose nodes contain both the identifier and link(s) to the object(s) that could be denoted by the identifier. `ENV` nodes may also contain information about how their corresponding objects were made visible (via USE-clause, local definition, etc.)

To *lookup* an identifier (ID) the `ENV` tree is searched according to a fixed rule until an entry is encountered whose identifier field matches ID. To build a new `ENV` value that binds ID to some other object(s) we create a new `ENV` node and insert it at the front of the tree so that it will be found first by the search rule, but so that the old `ENV` value is not changed. One simple way this could be done is to implement the `ENV` tree as a list (a tree in which each node has only one child) and to create a new `ENV` value by making the new node as the head of a list whose tail is the old list. There are *applicative* forms[14] of balanced trees, and other data-structures, that can instead be used to make the search more efficient.

`ENV` values are part of the VIF and hence are retained in the model library.

5 Lessons learned

5.1 VHDL versus Ada

At the beginning of our effort, we estimated that a VHDL compiler should require two thirds of the effort to produce an Ada compiler. There were several aspects we underestimated.

First, the number of productions in a LALR(1) VHDL grammar is comparable to the one in a LALR(1) Ada grammar. This is due to the fact that hardware description specific constructs introduced in VHDL compensated for the elimination of some of the software oriented constructs that were present in Ada.

Second, the VHDL Runtime Kernel is more complicated than its Ada counterpart. This is due to the synchronization in simulated time, that governs processes, signals, and guards. Due to the preemptive nature of signal assignments in VHDL, the effect of a VHDL signal assignment is not determinable at the time of the execution of the assignment[13].

Third, the VHDL static semantics is at places more complicated than the corresponding Ada semantics, because Ada constructs have been extended to serve hardware description purposes, as it is discussed in this paper.

Last, but not least, developing a compiler for a new and unfrozen language, is a challenge in itself. An important effort was spent in keeping up with the evolution of VHDL, up to the point when it became an IEEE standard.

5.2 AGs are monolithic

In building our compiler as an AG we ran into several problems where we had to step back and do things differently than we had originally planned. The use of cascaded evaluation is an example. However, we never found a satisfactory way to deal with the *monolithic* nature of an AG.

An attribute grammar is *monolithic* in the sense that it is hard to partition one into smaller pieces that can be processed separately. The LALR parse-table builder needs every production in order to generate tables. The dependency analysis phase of the evaluator-generator needs the dependency information for every symbol and production in order to find an evaluation order. Especially for a large language like VHDL, this is a significant problem; the VHDL AG is one 500,000-byte file whereas the rest of the compiler consists of about 50 modules of 1000 bytes to 100,000 bytes each.

There are two reasons why this is not good: it makes it hard for two or more people to work on different aspects of the AG at the same time, and it costs a lot of computer power (and wall time) to be regenerating so many evaluators all the time. The latter of these was the more serious problem in the past; we expect the former to be the more serious problem in the future.

An attribute evaluator generator such as Linguist contains some expensive, non-linear algorithms buried in it. This means that if AG1 is twice as large as AG2 then AG1 will need more than twice as much time to be processed. Thus, generation time that is reasonable for a Pascal AG can turn out to be a significant bottleneck for a VHDL AG. Compiling the generated evaluator can be even more of a problem. The monolithic AG gives rise to a monolithic block of C code that is more than 50,000 lines long and is too big to compile with our system's C compiler if symbolic information for the debugger is included in the object code. We are able to split it into pieces, some of which the C compiler can handle,[5] but one of these pieces is still so large that if it is compiled to generate information for the system's symbolic debugger then the debugger can not load it.

Although the long generation time has been troublesome in the past, we feel that the inability to decompose the AG into smaller pieces and understand it as the composition of those pieces is the more serious problem. This has always been a problem for LR parser generators, and an AG inherits this problem for its underlying context-free grammar. This exacerbates the equally monolithic nature of the semantics in an AG. A change in the dependencies of a semantic rule in one production can combine with a hitherto legal dependency in some far removed production to produce a circularity in the AG. To diagnose and correct such a circularity usually requires that one have a reasonable understanding of the global dependency structure of the AG.

The careful reader will have noticed that despite our observations above we have already discussed one way to decompose an AG — *cascaded evaluation* splits the semantics part of an AG into independent pieces, although it doesn't divide the syntactic portion. It may be that this is an effective way to decompose an AG and we hope to investigate writing separate AGs for declarations and statements, and perhaps partitioning those further by having *nested* cascaded evaluation within them.

[5]The generated attribute evaluator code contains C preprocessor macros to facilitate this splitting.

5.3 Productivity using AGs

The usual advantages claimed for AGs over more traditional compilers are that they are faster to develop and easier to maintain. The usual disadvantages claimed for AGs is that the compilers generated from them are slower and use more memory. Our experience on this project generally supports these expectations, although some of them were more fully met than others.

We think that using AGs enabled us to complete the project in substantially less time than would otherwise have been the case — but it was not an order-of-magnitude improvement. Trying to directly compare large software products will inevitably have an "apples to oranges" flavor about it, and comparisons of the engineering projects that produced them are even less illuminating. Nevertheless, we offer the figures we have gathered about our compiler and its development for other implementors to compare against their own experience. Our VHDL compiler is 46,000 lines of original source code (i.e. code written by hand) and it took us 82 man-months to complete. Of this 46,000 lines, about 12,000 or 25% is in what we consider code-generation; the rest is in parsing, semantic analysis, and management of separate compilation. We do not include in this the code for the simulation Kernel or run-time library support.

This works out to about 550 lines per man-month (MM). This is somewhat higher than the average productivity figures commonly-quoted[3] for the industry of 350 Delivered Source Instructions (DSI) per MM, but not astonishingly so.[6] If we assume that the generated C code had been written by hand then the productivity figures do become surprising: average productivity over two years of about 1350 lines per MM. This is probably not the right way to look at this issue; program generators are notorious for generating code that no person would write.

By far the most costly aspect of writing this compiler was figuring out what this new language meant and how its constructs should be implemented. From this perspective, the most advantageous property of AGs was the ease with which they can be modified; even substantially modified. Over the course of its development our compiler went from a maximum of four visits per node, to a maximum of five visits per node, to three visits[7] per node. All of this happened transparently to the AG authors, who were only aware of adding and deleting attributes and the semantic rules that defined them. In a hand-coded compiler this would have required (in addition to these decisions of what new results or intermediate values needed to be computed

[6]this assumes that a "line" is equivalent to a Source Instruction, which seems reasonable for C, but perhaps is an underestimate for an AG or the VIF description.

[7]Most symbols are only visited once; only a half-dozen symbols out of 355 are visited 3 times.

and how) substantial modifications to the code that visits sub-trees to collect and process this information in the right order. Such changes would have had to be calculated by hand in the first place and then the tree-walking algorithms changed by hand also. Most likely, the intermediate data-structures representing the tree would have had to be modified too, again by hand.

Throughout the compiler's development we:

- changed the way features were implemented,
- added optimizations to the generated code, and
- optimized some aspects of the compiler's own performance

to reflect our increasing understanding of

- the semantics of the language,
- how to best model the behavior of a source program, and
- where the bottlenecks were in our implementation as we saw how our customers were using the compiler.

In our opinion, this level of change could not have been supported if such a complex piece of software had been manually coded. That we could do so was important to our development strategy; we think it bodes well for the compiler's future maintainabilty.

References

[1] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and F. K. Zadeck. *Incremental Evaluation of Attribute Graphs.* Technical Report, IBM T.J. Watson Research Center, Yorktown Heights, NY, December 1987.

[2] T. P. Baker. A Single-Pass, Syntax-Directed Front-End for Ada. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.* ACM. June 1982.

[3] Barry Boehm. Software Engineering Economics. Prentice-Hall. 1981.

[4] D. R. Coelho and A. G. Stanculescu. A State of the Art VHDL Simulator. In *Proceedings of the IEEE CompCon 88.* IEEE. 1988.

[5] S. Drossopoulou, J. Uhl, G. Persch, G. Goos, M. Dausmann, and G. Winterstein. An Attribute Grammar for Ada. In *Proceedings of the SIGPLAN 82 Symposium on Compiler Construction.* ACM. June 1982.

[6] Rodney Farrow. Attribute Grammars and Data-Flow Languages. In *Proceedings of the SIGPLAN Symposium on Programming Language Issues in Software Systems.* ACM. June, 1983.

[7] Rodney Farrow. *Generating a Production Compiler from an Attribute Grammar.* IEEE Software, Volume 1, Number 4, Oct., 1984.

[8] H. Ganzinger and R. Giegerich. Attribute-Coupled Grammars. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction.* ACM. June 1984.

[9] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG : A Practical Compiler Generator.* Lecture Notes in Computer Science 50. Springer-Verlag. Berlin-Heidelberg-New York. 1982.

[10] U. Kastens, R. Kollner, E. Zimmermann, P. Hruschka, and A. Kappatsch. *Eine Attributierte Grammatik fur PEARL.* Fak. f. Informatik, Universitat Karlsruhe. 1980.

[11] *IEEE Standard VHDL Reference Manual.* IEEE Std 1076-1987. The Institute of Electrical and Electronic Engineers, Inc., New York, NY, March 31, 1988.

[12] Linguist User's Manual, Version 6.1. Declarative Systems, Inc., Palo Alto, CA, February 22, 1988.

[13] D. C. Luckham, A. G. Stanculescu, Y. Huh, and S. Ghosh. Semantics of Timing Constructs in Hardware Description Languages. In *Proceedings of ICCD 86.* pp. 10 - 14. 1986.

[14] Meyers, Eugene. "Efficient Applicative Data Types." In *Proceedings of the Eleventh Annual Symposium on Principles of Programming Languages.* ACM. Salt Lake City, Utah. January, 1984.