GRAPHITE: A META-TOOL FOR ADA ENVIRONMENT DEVELOPMENT

Lori A. Clarke
Jack C. Wileden
Alexander L. Wolf

COINS Technical Report 85–44
November 1985
(Revised March 1986)

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

A previous version of this report appears in the Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments, Miami Beach, Florida, April 1986.

This work was supported in part by the following grants: Rome Air Development Corporation, No. SCEEE–PDP/85–0037; National Science Foundation, No. DCR–84–04217 and No. DCR–84–08143; and Control Data Corporation, No. 84–M103.

ABSTRACT

Tools in a software development environment often manipulate objects that are instances of *attributed graphs*. Moreover, an individual attributed-graph instance may be manipulated by several different tools in an environment. During the prototyping phase in the design of a software development environment, experimentation with tools may dictate changes to the high-level structure of an attributed graph as well as changes to the graph's underlying representation. We have developed a tool called GRAPHITE to facilitate both kinds of experimentation while minimizing the impact of that experimentation on the tools in an Ada environment. This *meta-tool* and its potential contributions to an experimental effort to build an advanced Ada software development environment are described in this paper.

1. Introduction

Many of the tools composing a software development environment will manipulate (create, access, or update) complex data objects. Often tools will need to manipulate data objects created by other tools. In some cases, it may even be necessary to extend the definition of an object that is manipulated by a set of tools to accommodate new information required for some subset of those tools. For example, tools being developed at the three universities involved in the Arcadia environment project [4] all need to manipulate an internal representation of Ada programs, called IRIS. The University of California at Irvine is building the front end of an Ada compiler that creates IRIS. At the University of Massachusetts, we are creating a set of tools to analyze interface control relationships. This analysis is based on some Ada language extensions [6] and thus requires that we extend IRIS to include this information. The development group at the University of Colorado is implementing data flow analysis tools that can work with either internal representation.

While most development teams may not be as geographically separated as the members of the Arcadia consortium, the situation described above is not atypical for large environment development efforts. Often various groups within the development team must share and/or extend the definitions of data objects manipulated by the environment's tools. For experimental systems, such as the prototype Arcadia environment, one would expect that some objects' definitions will undergo considerable change throughout the life of the project until the environment's developers finally settle upon an acceptable version. In such cases, it is important to facilitate the evolution of data object definitions yet not allow that evolution to impede the development of tools that may depend upon some aspects of those definitions.

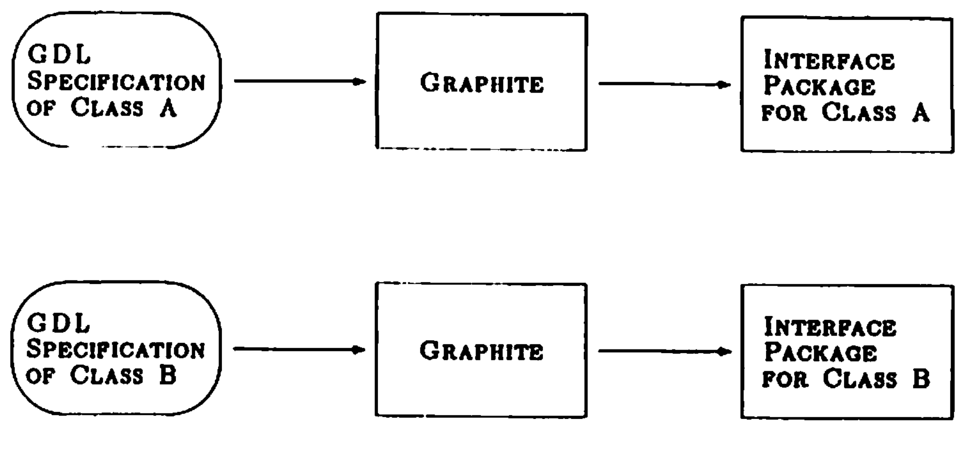
Many of the data objects manipulated by software development environment tools are, like IRIS, *attributed graphs*. For example, parse trees, abstract syntax trees, control flow graphs, and call graphs are all classes of attributed graphs that are likely to be manipulated by several tools. A *class* of attributed graphs is defined by specifying a set of *node kinds*. Each node kind is associated with a set of *attributes*. Attributes are used to describe the properties of the objects represented by the nodes in the graph and each such attribute has a type, referred to here as an *attribute value type*. Some of the attribute value types are actually node kinds, which makes it possible to connect nodes into graph structures. An instance of a node kind is a set of values, one for each attribute associated with that node's kind. A particular attributed graph, which is a member of some class of attributed graphs, is then just a set of instances of node kinds in that class. An Ada software development environment may require some complex graph structures. For example, in DIANA [2], which is another internal representation of Ada programs, there are approximately 161 node kinds, 97 attributes, and 15 attribute value types—a complex structure by any measure.

A tool supporting the definition and use of attributed graphs would greatly benefit the developers of software development environments. Such a tool should provide a uniform mechanism for defining attributed graphs and for generating an implementation of a data abstraction for such a definition. It is particularly important that this tool facilitate the redefinition of attributed graphs, minimizing the impact of such changes on all the tools in the environment. Since this is a tool for environment developers as opposed to being a tool in the actual environment (although it could be both), we refer to it as a *meta-tool*.

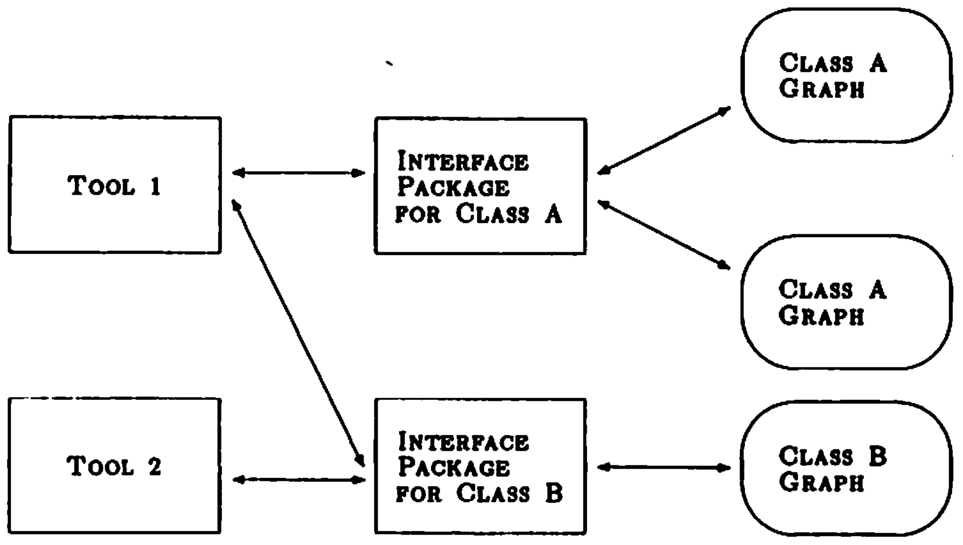
As part of our contribution to the Arcadia consortium, we have developed such a meta-tool, called GRAPHITE for GRAPH Interface Tool for Environments. The GRAPHITE system accepts specifications of classes of attributed graphs written in a graph description language, called GDL. Given the GDL specification for a particular class of attributed graphs, GRAPHITE produces an Ada package that is an implementation of an abstract data type for that class of graphs. Thus, this Ada package defines the graph and the operations that can manipulate the graph. The operations include those that allow tools to create nodes, to get and put attribute values, and to read and write nodes and graphs, as well as to ascertain the kind of a node and its associated attributes. Because it represents the interface to the implementation of a class of attributed graphs, this package is called an *interface package*.

Using the interface package produced by GRAPHITE, a tool can create and/or access a number of different graphs of a particular class. Any or all of these could be manipulated by other tools, which would also access these graphs through the operations provided in the interface package. Moreover, a tool may use more than one interface package in order to access more than one class of graphs. Figure 1a illustrates how GRAPHITE can be used to create interface packages for two different classes of attributed graphs, called Class A and Class B. Figure 1b then shows how two tools might use these packages; Tool 1 to manipulate two instances of Class A and both Tool 1 and Tool 2 to manipulate an instance of Class B.

Using an abstract data type to manipulate a data object is not a new concept. Automatically creating the abstract data type from a graph description has also been done [3]. What is innovative about the GRAPHITE system is the design of the abstract data type that is actually generated and the ways in which Ada is used to meet our objectives. Central among those objectives, which are enumerated in the next section, is support for a prototyping approach to experimental environment building that permits a straightforward transition from prototype to production quality implementation. To this end, GRAPHITE produces two different kinds of Ada interface packages. One supports software development of experimental


GDL Specification of Class A
Graphite
Interface Package for Class A
GDL Specification of Class B
Graphite
Interface Package for Class B

(a)



(b)

Figure 1: Creating and Using Attributed-Graph Interface Packages.

systems. It is designed so that when developers modify the definitions of graph classes there is a minimal effect on other tools in the system, even on those tools that use this modified class. The second interface package is designed for efficient manipulation of graphs. When the definition of a graph class has been finalized, the second interface package can be substituted for the first so that a more efficient, but less flexible, version of the environment can be created. Thus we have support for both development and production versions of an environment and a process for easily going from the development to the production version.

This paper describes GRAPHITE, emphasizing the design of the interface package that is generated. The next section discusses the goals of the GRAPHITE system. The third section describes GDL. The fourth section presents the design of the interface package and describes how to move from the development version to the production version of the package. We conclude with a discussion of related work and the current status of the system.

2. Design Objectives

The GRAPHITE meta-tool was conceived as part of our research on a suitable infrastructure for the Arcadia environment. Arcadia is to be an experimental Ada software development environment, implemented in Ada, within which we can investigate novel tools and techniques supporting development of Ada software. Both the experimental nature of Arcadia and our commitment to exploiting the features and programming style supported by Ada have influenced our objectives for GRAPHITE.

Encapsulation. A primary objective of GRAPHITE was to encapsulate attributed-graph objects into an abstract data type. Users of GRAPHITE may treat attributed graphs and their operations as primitive constructs in a higher-level programming language tailored for building software development environments. This results in a uniform mode among the tools in the environment for interacting with graphs, where graphs are manipulated not in terms of record fields or array elements, but rather in terms of getting and putting attribute values. An additional benefit of encapsulation is *information hiding*. In particular, the realization of attributed graphs and their associated operations may change with minimal side effects; whereas a change in the representation of graph instances might otherwise necessitate a reprogramming of the tools to account for a different access method (e.g., array indexing versus record-field selection), encapsulation means that such a low-level change could be performed without affecting the code of the tools. As a result, a development team can experiment relatively easily with alternative representations for attributed-graph classes.

Automatic Generation. A further GRAPHITE objective was to reduce the drudgery involved in building or modifying a software development environment prototype. Since the GRAPHITE processor automates the production of an Ada implementation of an attributed-graph class from a GDL specification, it helps to reduce the overhead associated with implementing a software development environment. This is particularly significant in an experimental setting such as Arcadia, where modifications are likely to be relatively frequent. Moreover, the automatic generation of attributed-graph implementations is less error-prone than would be the manual creation of the corresponding Ada code.

Minimizing Impact of Changes. During the prototyping phase of an environment development project there will be a great amount of experimentation in which the definitions of the attributed-graph classes will change. Thus, it should be straightforward to alter the set of node kinds, attributes, and attribute value types in the definition of an attributed-graph class. As noted previously, there may also be experimentation with the representations of attributed graphs. GRAPHITE is designed so that an alteration to either the definition or the representation of an attributed-graph class will have a minimal effect on the rest of the system. Of course, tools that do not access an attributed graph are not affected by changes of either kind. Our concern is with two cases, one in which a tool manipulates an attributed graph whose definition has been changed in ways that do not directly affect the tool (e.g., the tool does not access the new information) and a second, in which the representation generated by the GRAPHITE processor has changed. One level of insulation from either type of change is achieved by assuring that the tool need not be reprogrammed as a result of the change, although it might require recompilation, as would be implied by a reprogramming of the interface package specification part. A much greater insulation from change can be realized by assuring that not even recompilation is required. This level of insulation is attainable only if there is no reprogramming or recompilation of the interface package specification part. GRAPHITE attains this second, higher level of insulation, thereby minimizing the impact of changes.

Managerial Control. Even in an experimental setting, the definitions of the attributed-graph classes in an environment must evolve in a manageable way. Freedom to experiment with new structures must be given to developers, yet too much freedom may hamper integration. An example of excessive freedom is permitting tools to dynamically add new node kinds, attributes, or attribute value types, since individual programmers could then effect undocumented modifications to the most fundamental shared objects in an environment. Therefore, the GRAPHITE system does not support the dynamic addition of new node kinds, attributes, or attribute value types. Instead, changes to the definition of an attributed-graph class must be made by changing its GDL specification. This controls the manner in which changes can be made and provides clear documentation of the current status of each attributed-graph class. Thus, for example, when several groups have experimented with the definition of some environment object, comparison of the GDL specifications provides a clear indication of any resulting differences. Moreover, the final GDL specification becomes the medium for documenting the final, negotiated version.

In Sum—Support for Prototyping. Our immediate use for the GRAPHITE meta-tool is in building prototype, experimental environments, where frequent changes to tools and to the internal representations on which those tools will operate are to be not only anticipated but encouraged. Moreover, these changes may be made by groups working at different, and distant, locations. For example, GRAPHITE might help in the situation where a group working at one site wishes to experiment with an alternative version of one of the objects produced by a tool developed at another site. Our goal is to make it possible for this group to use the tool to produce the alternative version without needing to even recompile that tool. The original developing group would thereby retain full control over the tool's code while other groups obtain the ability to simultaneously share, experiment with, and even change the definition of the objects produced by that tool.

To foster this prototyping activity, we have sought an approach to attributed-graph definition and implementation that provides ease of use and flexibility and that also supports sharing. We have not, however, abandoned control over the environment's configuration, since we consider the structure and discipline imposed by well-defined interfaces and type checking to be particularly important in an experimental setting. GRAPHITE, by realizing the objectives enumerated in the preceding paragraphs, offers the ease of use, flexibility, and support for sharing that can facilitate environment prototyping activities.

3. Graph Description Language

This section describes the Graph Description Language, GDL, which is used to specify a class of attributed graphs. This specification is the input to GRAPHITE, which uses it to generate the interface package for creating and manipulating graphs of that class.

As noted above, GDL facilitates the development and maintenance of large software systems by providing a common medium for communicating the definition of an attributed-graph class among developers. To make this medium as natural to use and easy to understand as possible for Ada developers, GDL is strongly modeled after Ada. This is evident in the elaboration scheme, lexical rules, and syntactic style that are used. The elaboration scheme governing the order of declarations in a GDL specification is the same as Ada's linear elaboration in which entities cannot be used before they are declared. The text of a GDL specification closely follows the lexical rules of Ada. For instance, an effort was made to define reserved words for GDL that are already used as reserved words in Ada. Reserved words of Ada that are not used in GDL, such as **task**, are not permitted as identifiers in GDL to avoid any conflicts with the generated Ada interface package. The few reserved words introduced by GDL are **class**, **node**, **group**, and **sequence**. Syntactically, the GDL specification for a given attributed graph closely resembles an Ada package specification that contains a collection of Ada-like type declarations. A GDL specification is syntactically delimited by the phrases "**class** <class name> **is**" and "**end** <class name>", where <class name> is a name given to the attributed-graph data structure being specified. Figure 2 presents a skeleton GDL specification in which ExampleGraph is the name of the class of attribute graphs being defined and Example is the name of the interface package that is to be generated. This figure is used throughout this section to describe the features of GDL in more detail.

```
class ExampleGraph is
  package Example;

  with Lexical.( Comment, Position );

  type LexicalInformation is
     record
        SourceComment : Lexical.Comment;
        SourcePosition   : Lexical.Position;
     end record;

  type BranchWeigth is new Integer range -10 .. 10;

  group Statement;       -- complete definition given below
  type StatementSequence is sequence of Statement;

  node ConditionNode; -- complete definition given below
  type ConditionSequence is sequence of ConditionNode;

  node ExpressionNode is
     . . .
  end node;

  SourceConnection : LexicalInformation;
  ExecutionCount   : Natural;

  node ConditionNode is
     SourceConnection;
     Weight       : BranchWeight;
     Condition    : ExpressionNode;
     Statements : StatementSequence;
  end node;

  node IfStatement is
     SourceConnection;
     ExecutionCount;
     IfBranch        : ConditionNode;
     ElsifBranches : ConditionSequence;
     ElseBranch     : StatementSequence;
  end node;

  . . .

  group Statement is ( IfStatement, WhileStatement, CaseStatement, ... );
end ExampleGraph;
```

Figure 2: GDL Specification for a Class of Attributed Graphs.

A given GDL specification defines a particular class of attributed graphs by declaring the node kinds, attributes, and attribute value types making up the class. Node kind and attribute declarations are quite straightforward. Declarations of attribute value types are more complicated, however, because they are based upon the rich type structure found in Ada. Node kind, attribute, and attribute value type declarations are each briefly described below.

Node kinds are the primary building blocks of GDL. To define a node kind, the developer specifies the name of that node kind and the attributes that make up nodes of that kind. The format for defining a node kind is similar to that of an Ada record declaration, with attributes acting as record fields. The similarity between node kind declarations and record declarations, however, is purely syntactic. In particular, node kinds are not necessarily implemented as records, and record-oriented operations, such as field selection, cannot be applied directly to nodes. ConditionNode and IfStatement are two node kinds declared in Figure 2.

GDL provides a syntactic shorthand (not available to Ada records) intended for situations in which two or more node kind declarations contain an identical attribute declaration (i.e., the name and type of the declared attributed are the same). In Figure 2, node kinds ConditionNode and IfStatement contain identical declarations for an attribute named SourceConnection. The syntactic shorthand allows a declaration to be made in one place and then used in various node kind declarations by simply listing the attribute's name. An attribute declaration that appears outside of any node kind declaration is referred to as a *commonly-available* attribute declaration. Whether or not the shorthand is used, each attribute is only associated with one node kind; it is simply the type information that is shared using commonly-available attribute declarations.

GDL supports four categories of attribute value types: predefined Ada types, user-definable Ada types, imported Ada types, and GDL-specific types. GDL supports all the Ada predefined (sub)types, which include Character, Boolean, Integer, Float, String, Natural, and Positive. Except for private types, all the user-definable Ada types, which include subtype, derived, enumeration, character, boolean, integer, float, fixed, array, record, and access are supported. Private types must be defined separately from a GDL specification and treated as an imported type.[1]

In Ada, a *with clause* attached to a package indicates that some entities defined in other, external package(s) are to be imported. In GDL, the with clause performs a very similar function; a with clause inside a GDL specification describes a set of externally defined and packaged Ada types, including private types, that are expected to be used in, or as, attribute value types. For example, in Figure 2, types Position and Comment are imported from package Lexical and used in defining the record type LexicalInformation.

In addition to the name of the package from which a type is imported, the GDL with clause provides five pieces of information, which are necessary for automatically generating the interface package that uses the entity:

1. the name of the imported type;
2. the name of the imported type's base type, if it is a subtype;
3. the name of the assignment operator for the type;
4. the name of the "external-form" type of the imported type; and

[1]A private type requires the specification and implementation of its operations as Ada subprograms. For the GRAPHITE processor to generate an interface package that contains such a private type definition, the GDL specification would have to include not only a representation for the private type, but also the complete declarations of the subprogram operations. This would, at the very least, clutter the specification of the attributed-graph class and detract from its otherwise high-level description.

5. the names of two dual-parameter procedures that serve to convert values between the imported type and the external-form type.

The first two pieces of information are necessitated by the fact that the actual definitions of imported types (i.e., the specification parts of the packages providing the types) are not assumed to be available to the GRAPHITE processor, as they would be to an Ada compiler. As is explained in more detail in the next section, the interface package contains overloaded subprograms. Therefore, the second piece of information must be known in order to generate these subprograms, since overloading in Ada is legal for types but not for subtypes. The only operation always required by an interface package for attribute value types is that of assignment. This operator could be the standard ":=" operator (by default) or, for *limited* private types, a dual-parameter procedure whose first parameter is the left operand and whose second parameter is the right operand. The last two pieces of information are concerned with the input and output of graph instances, which are discussed in more detail in the next section. Briefly, the external-form type should have the property that it holds the same information as the imported type, but can be immediately written out to a secondary file using the Ada direct input/output package Direct_IO (see [1], Section 14.2.4). In particular, it cannot contain any access types. The imported type can, of course, be its own external-form type if it already has this property.

Figure 3 illustrates the specification of imported types in GDL. Noticed that we have modified the Ada with-clause notation so that each imported entity from a package is explicitly identified. In this figure, four types are imported from a package Pac. The first type, SomeType1, is a subtype of the Ada predefined type Integer. There is no need to explicitly indicate an assignment operator for this type, since the Ada assignment operator ":=" is the

```
with Pac.( SomeType1
               subtype of Integer;

           SomeType2/ExternalSomeType2,
             :=  => SomeType2Assign,
             in  => InternalizeSomeType2,
             out => ExternalizeSomeType2;

           SomeType3;

           SomeType4
             subtype of SomeType2 );
```

Figure 3: An Example Specification of Imported Types.

default for values of this type. Similarly, there is no need to specify an external-form type, since SomeType1 is a pure numeric type, which means it can serve as its own external-form type. Further, since SomeType1 is its own external-form type, there is also no need to specify the names of conversion procedures. The second type, SomeType2, is not a subtype. It has an external-form type ExternalSomeType2, an assignment procedure SomeType2Assign, and conversion procedures InternalizeSomeType2, which converts values of the external-form type to their "internal" form SomeType2, and ExternalizeSomeType2, which converts values of the imported type to their external form. The third type, SomeType3, is also not a subtype, is its own external-form type (and so does not require conversion procedures), and uses the standard Ada assignment operator ":=". Finally, the fourth type, SomeType4, is a subtype of SomeType2 and so by default can share SomeType2's operations.

There are three GDL-specific types, *node kind*, *node group*, and *node sequence*. Each is an attribute value type constructor that facilitates describing an attribute-graph class and, thus, has no counterpart in Ada. Node kind was described above. Node group is used as a value type for an attribute whose values can be any one of a number of different node kinds.

An example of this is given in Figure 2 where Statement is defined to range over a set of nodes representing statements. The operations appropriate to values of a node group are the same as those for a node kind, since a value of a node group is simply a node. Node sequence is used to indicate an ordered collection of nodes. Again referring to Figure 2, type StatementSequence is declared to be an ordered list of nodes, where each node must be one of the node kinds declared in the group Statement. Operations on values of a node sequence include those to create a sequence, retrieve an element of a sequence, and determine whether or not a sequence is empty. All the operations for GDL-specific types are provided as subprograms in the automatically generated Ada interface package. To insure that nodes are the primary building blocks of the GDL specificaton, node kind, node group, and node sequence type constructors are not permitted in user-defined Ada type declarations (e.g., as components of a record type).

As a final point, initial values can be given to attributes of all types. Those of type (constructors) node kind, node group, or node sequence, like Ada access objects, are given "null" initial values by default. A user-specified initial value is given as an expression of the appropriate type; syntactically, it is the same as an initialization expression for Ada objects. In the case of node kinds, the form taken is that of a record aggregate. In the case of node groups, the form is also that of a record aggregate, but the name of a particular member of the group must also be given using the "tick" notation (i.e., <node kind name>'<aggregate expression>).

An initial value for an attribute can be given in three places: in the declaration of an attribute value type, in the declaration of an attribute (within a node kind declaration or in a commonly-available attribute declaration), and in the use of a commonly-available attribute declaration within a node kind declaration. In the event that for a particular attribute more than one initial value is given, the precedence order from highest to lowest is: node kind declaration, commonly-available attribute declaration, attribute value type declaration. For example, if a value type VT is declared and an initial value *i* given, then that value holds for all attributes of that type, except for those attributes of type VT initialized to a different value *j* in their attribute declarations. If the attribute declaration of type VT is commonly available, then that second initial value *j* can be overridden by an initial value *k* given in a node kind declaration that uses the commonly-available declaration. Notice that GDL differs from Ada, which does not allow initialization in types other than for fields of record types, by allowing initialization in all user-defined Ada attribute value type declarations except those involving unconstrained arrays.

By necessity, this section can only summarize the major features of GDL. A complete description is given in [7].

4. Interface Packages

Having described the input to GRAPHITE in the previous section, we now turn to the output, namely the Ada interface package that implements a given GDL specification of a class of attributed graphs. As mentioned in the introduction, GRAPHITE can generate either a development version or a production version of such a package. The versions are similar in that they realize an attributed graph as an abstract data type and provide the same set of operations on graphs. The versions differ in how they resolve the often conflicting goals of flexibility and efficiency; while the express purpose of the development interface is to support flexibility by minimizing the impact of changes on tools, the intent of the production interface is to provide efficient access to a relatively stable class of graphs.

This section presents our designs for the development and production versions of the interface package. First, an overview is given of the operations on attributed graphs provided by interface packages. Details of the development interface are then presented and the flexibility that that design provides is demonstrated. Finally, the production interface is described and issues in moving from a development version to a production version are considered.

4.1 Operations Provided by Interface Packages

The operations provided by GRAPHITE-generated interface packages fall into four basic categories, as shown in Table 1. There are several things to notice about these operations and about the use of Ada in their implementation.

First, notice the granularity of the operations that get and put attribute values. These operations are designed to work on all the attributes of a particular type. Therefore, there will be a separate pair of get/put operations for each attribute value type in a class.[2] Because the get (put) operations differ in the type of the attribute returned (entered), the Ada subprograms that implement them can be overloaded, i.e., given the same name. The use of overloading in this situation is appealing because it underscores the similarities in the operations' functionality. For instance, tool developers can take the perspective that there is only one get operation and one put operation and that these two operations will work for any attribute. The fact that the interface package must actually provide several pairs of subprograms to realize these operations is hidden by the fact that the pairs are overloaded.

Our choice for get/put granularity has advantages beyond that of using overloading.

[2]Actually, *node kind* attribute value types are collapsed into one Ada type and so there is only one get/put pair for all attributes that are nodes. The same is true for *node sequence* attribute value types. In addition, for each set of *subtype* attribute value types sharing the same base type, there is also only one get/put pair. This is discussed further.

1. OPERATIONS TO MANIPULATE A NODE	
Create	creates a new node of a given kind
Get Attribute	gets the value of an attribute of a given type
Put Attribute	puts the value of an attribute of a given type
2. OPERATIONS TO ASCERTAIN A NODE'S DEFINITION	
Attribute Value Type	retrieves the name of an attribute's value type
Kind	retrieves the name of a node's kind
Node Kind Attributes	retrieves the names of a node kind's attributes
3. OPERATIONS TO MANIPULATE NODE SEQUENCES	
Create	creates a given sequence
Insert	inserts a node into a sequence at a given position
Kind	retrieves the name of a sequence
Length	retrieves the length of a sequence
Retrieve	retrieves a node from a sequence at a given position
4. OPERATIONS TO INPUT AND OUTPUT GRAPHS	
Read Graph	reads a graph from a file
Write Graph	writes a graph to a file

Table 1: Operations Provided by Interface Packages.

These advantages become clear when our choice is compared with an obvious alternative—one get/put pair for *each* attribute—which is the granularity used in the example implementation of a DIANA interface package in [2]. In that approach, the operations are tied to the low-level details of the data being represented, which for attributed graphs are the names of the attributes comprising the node kinds. This results, for example, in a DIANA interface package having 97 pairs of uniquely named (i.e., non-overloaded) subprograms for getting and putting attribute values, one for each of the 97 attributes in the class. This compares to the approximately 15 get/put pairs, corresponding to DIANA's approximately 15 attribute value types, that would be required under our approach. The tools most hurt by DIANA's example granularity are those that perform general-purpose functions, such as traversing a graph. These tools would very likely have to include complex constructions, such as case statements with large numbers of arms (e.g., 97). Maintaining such structures would be difficult, not only because of their sheer size, but also because they are highly sensitive to changes in a class's set of attributes.

In the extreme, our choice for granularity would reduce to the alternative discussed above if every attribute were given a unique attribute value type. This is, of course, an unlikely possibility. More typical would be classes consisting of small numbers of attribute value types compared to attributes, as is the case for DIANA. Furthermore, we would expect that the set of attribute value types would change much less often than the set of attributes. Therefore, our approach better supports the building and maintaining of general-purpose tools as well as offers greater protection against changes in class definitions.

A second thing to notice about the set of operations provided by interface packages is the inclusion of operations to retrieve details about a node's definition. This, like the granularity of the get/put operations, is intended to facilitate the development of general-purpose tools. Using these functions, tools could be written that apply to any class definition as long as the attribute value types are known. Consider, for example, the design of a tool that, given the name of an attribute and the name of the type for nodes, traverses a graph searching for all nodes containing that attribute. The basic algorithm for such a tool might be the following.

1. Retrieve the list of attribute names for the current node.
2. Note whether the desired attribute is present.
3. For each attribute in the list:
   (a) Retrieve the name of the attribute's value type.
   (b) If the type indicates that the attribute is a node, then recursively apply the algorithm to that node.

Notice that the algorithm is independent of any particular class definition because it can dynamically determine all the information that it needs.

Third, notice the operations provided by GRAPHITE to manipulate node sequences. Although support for node sequences is not strictly necessary, since such sequences can be simulated with attributes that linearly link nodes together, it is provided simply because such structures are so common in the representations used in software development environments. For example, in representing an Ada program, the nodes for the elements of a declarative part or the elements of a statement part are, by their very nature, in a sequence. The operations for manipulating node sequences were chosen for their primitive functionality. From them, practically any style of sequence manipulation, such as a LISP-like CAR–CDR–CONS approach, can be built.

The fourth thing to notice is that the read and write operations are intended to support the sharing of an attributed-graph instance among tools that may be invoked at widely different times. The write operation, given a node N in a graph, creates a secondary file and saves in that file every node reachable from N. The read operation reconstructs a graph by retrieving the nodes saved in a file by a previous write operation. The Ada direct input/output package Direct_IO provides extensive support for reading and writing values of arbitrarily complex types. The only limitation of this package is that it cannot maintain the values of Ada access types. Attributes whose types involve access types must therefore be translated, or *linearized*, into a suitable external form before being saved. The GDL specification of an attributed graph provides GRAPHITE with sufficient information to define an external form and perform any necessary linearization, even for those types imported from separate packages (see Section 3). This external form preserves the semantics of access values without actually using access types. These semantics are then used by the read operation to re-establish access values while reconstructing a graph.

Finally, notice that two possible categories of operations are completely missing. The first would consist of operations to dynamically add node kinds, attributes, or attribute value types to a class definition. As argued in Section 2, such capabilities, if provided, would adversely affect managerial control over a large project and thus we decided not to provide them. The second missing category would provide capabilities to convert an instance of a graph created from a previous class definition into an instance of a graph corresponding to the current definition. This would be an important category if, for example, an experimental environment is expected to undertake the analysis of large, or large numbers of, Ada programs. Then we may not want to incur the cost of complete re-analysis because intermediate representations have undergone change. Since we do not expect such loads in the Arcadia environment until it has somewhat stabilized, operations to perform instance-to-instance translations have not been included.

4.2 Development Interface

Figure 4 shows a portion of the specification part of the development version of the interface package that would be generated by GRAPHITE from the GDL specification given in Figure 2. The specification part contains declarations for the subprograms realizing the operations discussed above. It also contains declarations for the Ada types realizing the attribute value types of the class definition.

Two of the types declared in the specification part correspond to the GDL-specific attribute value types *node kind* and *node sequence.* (The other GDL-specific attribute value type, *node group*, is only used within the body part of an interface package and so does not require a visible Ada type to represent it.) Note that the name of the class given in a GDL specification is used as the name of an Ada private type whose objects designate nodes in attributed graphs of the class. The name of the class is also used to form the name of an Ada private type whose objects designate sequences of nodes in the class. Thus, for the example shown in Figure 4, the type for designating nodes is ExampleGraph and the type for designating node sequences is ExampleGraphSequence. Because these types are private, the only operations on nodes and sequences of nodes (other than assignment and the equality/ inequality tests) that can be performed by tools are those realized by the visible subprograms defined in the specification part. Although one type is used to designate nodes of all kinds, an interface package will guarantee at run time that a node is used in a manner consistent with its kind. For instance, if a node kind has an attribute A whose value type is another node kind NK, then only nodes of kind NK will be allowed as values of attribute A. The

```
    -- utility entities
       with ExampleGraphUtilities;

    -- imported types and subtypes
       with Lexical;

package Example is
    -- GDL-specific types
       type ExampleGraph          is private;
       type ExampleGraphSequence  is private;

    -- user-defined types and subtypes
       type LexicalInformation is
         record
           SourceComment : Lexical.Comment;
           SourcePosition  : Lexical.Position;
         end record;
       type BranchWeight is new Integer range -10 .. 10;
       . . .

    -- types for communicating names
       type NodeKindName            is new String;
       type NodeSequenceName        is new String;
       type AttributeName           is new String;
       type AttributeValueTypeName  is ( ExampleGraphVT, ExampleGraphSequenceVT,
                                          BranchWeightVT, LexicalInformationVT, IntegerVT, ... );

    -- types for listing a node's attributes
       type AttributeNameListElements is array ( Natural range <> )
         of AttributeName ( 1 .. ExampleGraphUtilities.MaxNameLength );
                                                     -- MaxNameLength is a function
       type AttributeNameList ( Length : Natural := 0 ) is
         record
           Elements : AttributeNameListElements ( 1 .. Length );
         end record;

    -- operations to manipulate a node
       function  Create ( NodeKind : NodeKindName ) return ExampleGraph;
       procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                                TheValue : ExampleGraph );
       function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                return ExampleGraph;
       procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                TheValue : ExampleGraphSequence );
       function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                return ExampleGraphSequence;
```

Figure 4: Specification Part of Interface Package for Class of Figure 2 (Development Version).

```
        procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                                 TheValue : BranchWeight );
        function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                 return BranchWeight;
        procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                                 TheValue : LexicalInformation );
        function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                 return LexicalInformation;
        procedure PutAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName;
                                 TheValue : Integer ); -- used for subtypes of Integer
        function  GetAttribute ( TheNode : ExampleGraph; TheAttribute : AttributeName )
                                 return Integer;        -- used for subtypes of Integer
        . . .
    -- operations to ascertain a node's definition
        function Kind ( TheNode : ExampleGraph ) return NodeKindName;
        function AttributeValueType ( NodeKind : NodeKindName; TheAttribute : AttributeName )
                                      return AttributeValueTypeName;
        function NodeKindAttributes ( NodeKind : NodeKindName ) return AttributeNameList;
    -- operations to manipulate node sequences
        function  Create ( NodeSequence : NodeSequenceName ) return ExampleGraphSequence;
        procedure Insert ( TheSequence : ExampleGraphSequence; Position : Positive;
                           TheNode : ExampleGraph );
        function  Kind ( TheSequence : ExampleGraphSequence ) return NodeSequenceName;
        function  Length ( TheSequence : ExampleGraphSequence ) return Natural;
        function  Retrieve ( TheSequence : ExampleGraphSequence; Position : Positive )
                             return ExampleGraph;
    -- operations to input and output graphs
        procedure ReadGraph ( FileName : String; TheGraph : ExampleGraph );
        procedure WriteGraph ( FileName : String; TheGraph : ExampleGraph );
    -- exceptions
        UnknownNodeKind      : exception;
        UnknownNodeSequence  : exception;
        UnknownAttribute     : exception;
        NodeSequenceError    : exception;
private
    -- representations; complete declarations given in body part
        type ExampleGraphRep;
        type ExampleGraphSequenceRep;

        type ExampleGraph         is access ExampleGraphRep;
        type ExampleGraphSequence is access ExampleGraphSequenceRep;
end Example;
```

Figure 4: (continued).

same guarantee is made for node sequences, which like nodes are designated by objects of one type.

The other types declared in the specification part (with the exception of NodeKindName, NodeSequenceName, and AttributeName, which are discussed below) correspond to non-GDL-specific attribute value types given in a GDL specification, such as LexicalInformation and BranchWeight for class ExampleGraph. These user-defined and Ada predefined types and subtypes move unchanged from the GDL specification to the interface package specification part.

Notice that the *imported Ada types*, which are Position and Comment in class Example-Graph, do not result in any type declarations in the specification part of an interface package. Instead, an Ada *with clause*, listing the names of the packages in which such imported types are declared, is attached to the specification part.

As pointed out in Section 2, avoiding recompilation of tools uninterested in a change is an important design goal for the development version of interface packages. This goal can be attained within the context of Ada's recompilation rules only if the specification part of an interface package does not require recompilation after a change has been made, since such a recompilation would invalidate previous compilations of the tools that use the package.[3] Toward this end, GRAPHITE generates a development version of interface packages whose specification part is nearly devoid of all definition- and representation-specific information about the attributed-graph class being managed and so insulates users of that package from most changes in class definitions and representations.

[3]Although an Ada compiler "may be able to reduce compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change" ([1], page 10-9), current technology does not support this optimization [5]. Moreover, the sorts of changes we anticipate in interface packages are not likely to be the ones that can be so optimized.

Definition Independence. In the specification part shown in Figure 4, there is no mention of particular node kinds, such as IfStatement and ConditionNode, no mention of particular node sequences, such as StatementSequence and ConditionSequence, and no mention of particular attributes, such as SourceConnection and Weight. Moreover, as disucssed above, subprograms such as GetAttribute and PutAttribute are provided for manipulating arbitrary classes of graphs. Of course, tools and the interface package still need to refer to the names of node kinds, node sequences, and attributes (e.g., to specify the desired attribute for a get operation). This is accomplished in our design through character strings; notice the declaration and use of types NodeKindName, NodeSequenceName, and AttributeName in package Example. The only information specific to a particular class of attributed graphs that must appear in the specification part of an interface package is the set of Ada types that represent the class's attribute value types. This is necessitated by Ada's use of a static type-checking mechanism.[4]

Representation Independence. Our design for interface packages provides tools with the maximum amount of insulation from changes in graph representations that is possible within Ada by using *indirection* in references to nodes. Indirection allows the details of a data structure—in this case, the representation of an attributed graph—to be confined to the body part of a package, where it can be changed without affecting the specification part of the package and, by extension, the tools using that package. While Ada's access types are the obvious choice for implementing indirect references, any type whose values can serve as "pointers", such as an integer that is an index type for an array, would be suitable for

[4]Conceivably, even this could be avoided by using string or integer types and then allowing tools to interpret them in any way that they wished. This provides a certain degree of dynamic flexibility in available types at the cost of added complexity in the code of that tool.

achieving the desired insulation from changes. Access types have an advantage over other types, however, in that they can point to dynamically created objects. Therefore, access types are used in both the development and production versions of GRAPHITE-generated interface packages. This is illustarted in Figure 4, where the private type ExampleGraph is shown to be an access type that designates the incomplete type ExampleGraphRep. The full declaration of ExampleGraphRep, which defines the actual data structure for representing graphs, would appear in the body part of package Example. Notice that the type representing node sequences also exploits this device.

In sum, our design for the development version of an interface package is an attempt to carefully and selectively circumvent Ada's compile-time type checking so that tools can be insulated from changes in the defintion and representation of a class of attributed graphs. With this design, changes in the sets of node kinds, node sequences, or attributes of a class only require the reprogramming and recompilation of the body part of an interface package and the tools directly interested in those changes. In addition, a change to the representation of a class of attributed graphs only requires the reprogramming and recompilation of the body part. The impact of a change is further reduced by the fact that GRAPHITE automates the process of reprogramming a body part. Only when a new user-defined, Ada predefined, or imported Ada type is added to a class definition's set of attribute value types will the specification part of an interface package change and so cause recompilations of all tools using that class. As noted above, this is likely to occur much less often than changes to the other aspects of a class definition. Although our design sacrifices some compile-time checking, it still permits an interface package to enforce the type consistency of the specified class definition. In particular, the body part contains all the information necessary to check the legality of node kind, node sequence, and attribute names as well as the operations applied to instances of its class.

4.3 Using the Development Interface

To appreciate some of the flexibility offered by the development version of an interface package, consider the following scenario: Tools T_1 through T_n manipulate attributed graphs of class ExampleGraph. Thus, they all refer to entities declared in package Example and are compiled against the specification part of that package. Suppose that the developer of T_1 decides that it is necessary to have a new node kind, called SpecialTool1Info, and that nodes of this kind are to be a new attribute, called Tool1Info, of node kind IfStatement. Suppose further that tools T_2 through T_n have no use for this new attribute. How extensive will be the effects of this change?

Certainly interface package Example must be reprogrammed to account for the new node kind and attribute. This activity, of course, is automated by GRAPHITE; the developer need only alter the existing GDL specification of class ExampleGraph and pass that altered specification through GRAPHITE. Tool T_1 must also be reprogrammed if it is to make use of the new node kind and attribute. The remaining tools, on the other hand, need not be reprogrammed, since they interact with IfStatement nodes, when necessary, through subprograms such as GetAttribute and PutAttribute, which allow a tool to operate exclusively on the node kinds and attributes of interest. Now, the only difference between the newly generated interface package and the old interface package (assuming all the attributes of SpecialTool1Info were of previously used types) is in their body parts; the specification parts of both packages are identical, since they do not contain any specific information about node kinds and attributes. Therefore, only tool T_1 and the body part of Example must be recompiled to account for this change in the class definition. Because the specification part of Example is not recompiled, tools T_2 through T_n do not need to be recompiled.

Using character strings to communicate the names of node kinds, node sequences, and attributes might appear to hinder the design of concise and efficient algorithms for both tools and interface packages. In contrast, enumeration literals have the advantage in Ada of being usable in more contexts than are character strings; enumeration types, of which enumeration literals are the values, are discrete and so their uses can include the expressions in case and loop statements, the indices of arrays, and the discriminants of variant records, while strings can be used in none of these contexts. This advantage argues strongly for defining node kind, node sequence, and attribute names as literals of enumeration types instead of as character strings. Unfortunately, such enumeration types cannot be declared in the specification part of development interface packages without then tying those specification parts to particular class definitions (cf., [2], page 137).[5]

A compromise scheme can be employed, however, that permits the use of enumeration literals and character strings in the places where they are most appropriate. Briefly, the scheme involves the use of the Ada predefined function Value. T'Value, for some discrete type T, converts a character string into a value of type T. For example, given an enumeration type ClassAttributes defined by

```
type ClassAttributes is ( Condition, ExecutionCount, IfBranch, ... )
```

ClassAttributes'Value("Condition") is equivalent to the enumeration literal Condition of type

[5]Since non-GDL-specific attribute value types are already fixed in the specification part of development interface packages, an enumeration type to represent them is placed there.

ClassAttributes. Thus, the function is used in this scheme to permit the names of node kinds, node sequences, and attributes to be treated as enumeration literals *within* tools and interface packages for concise and efficient local processing, while treated as character strings *between* tools and interface packages for communicating names. More specifically, each tool could have enumeration types representing the names of just those node kinds, node sequences, and attributes in which it was interested, while the body part of an interface package would have enumeration types representing the names of all node kinds, node sequences, and attributes of its class. Value is used to convert a name, from a character string to a value of the appropriate local enumeration type, for input to a tool or interface package.

4.4 Moving From Development to Production

Once an attributed-graph class has stabilized to the point where experimentation with its definition and representation is no longer a primary activity, a developer can use GRAPHITE to generate a version of the interface package that is oriented more toward efficiency than flexibility. The design of the production interface is intended to make this transition as easy as possible. Specifically, our goal was to minimize the amount of reprogramming of tools that would be needed to begin using the optimized interface. Hence, the production version is very similar to the development version; graph structures are realized as abstract data types and the same set of subprograms is provided for operating on attributed graphs. In fact, the only difference between the development and production versions that is visible to tools involves the use of enumeration types. The production version, because it is not required to be as flexible as the development version, uses enumeration literals to communicate the names of node kinds, node sequences, and attributes between tools and interface packages; whereas in the development version, types NodeKindName, NodeSequenceName, and AttributeName are character-string types, in the production version they are enumeration types. In the production version of package Example, the declarations for these types would resemble the following.

```
type NodeKindName       is ( ConditionNode, IfStatement, ... );
type NodeSequenceName is ( StatementSequence, ConditionSequence, ... );
type AttributeName        is ( Condition, ExecutionCount, SourceConnection, ..., Weight );
```

With enumeration literals, the overhead of interpreting character strings to check their legality and to use them for processing is avoided. For instance, once a single, definitive enumeration type is available for each of the various kinds of names communicated between tools and the interface package, there is no longer a need to use the translation scheme employing the Ada function Value described above. Of course, moving from character strings to enumeration literals, since it involves a change in a visible type, does require the reprogramming of tools. The *amount* of that reprogramming, however, can be made almost negligible with the use of an appropriate programming discipline during development. That discipline has two components. First, a tool that uses a local enumeration type should isolate the places where it refers to function Value. Second, *string constants*, representing node kind, node sequence, and attribute names within tools, should be used as parameters in calls to interface package subprograms. For example, a tool that refers to node kinds ConditionNode and IfStatement should, during development, use the following declarations.

```
ConditionNode : constant NodeKindName := "ConditionNode";
IfStatement    : constant NodeKindName := "IfStatement";
```

These string constants could then be used for all (relevant) invocations of graph operations, such as the following call to create a ConditionNode node.

```
X := Create ( ConditionNode );
```

Notice that references to node kind, node sequence, and attribute names through string constants are syntactically identical to references through enumeration literals. Thus, for a tool that adheres to this discipline, moving from development to production would only involve the reprogramming necessary to remove calls to function Value and eliminate access to the string-constant declarations, which may simply mean deleting those string-constant declarations or, if the declarations reside in a separate package (perhaps shared by several tools), changing a context clause.

5. Conclusion

In this section we look at alternative approaches for supporting the development of attributed graphs and discuss why we believe the GRAPHITE approach is superior for prototype development. We then report on the status and availability of the system.

There are two major alternatives to using GRAPHITE. One is to use a traditional data base approach and the other is to use the IDL processor [3]. With a data base management system, the classes of graphs that are to be manipulated are described using a data base schema, which although not similar to Ada is relatively easy to use. Such systems provide a basic set of operations to create, enter, and retrieve information. For example, a relational data base system provides a consistent set of operations based on relational algebra for manipulating attributed graphs. These operations allow new node kinds and attributes to be defined dynamically just by adding tuples to the relations that specify a class of graphs. Such extensions require no recompilation because of the underlying support provided by relational data base systems for dynamic alterations to relations. Although providing dynamic redefinition is one way to support experimental development, we have argued elsewhere that for large development efforts this reduces managerial control as well as sharing among development groups because data objects are not well documented.

The other major drawbacks to a date base approach are the lack of support for user-defined types and the run-time overhead of such systems. Typically, data base systems only provide support for a limited set of predefined types such as integers, reals, and character strings. Building a software development environment requires a much richer set of types than these. Finally, the run-time overhead for the newer, more powerful data base models, such as the relational approach, is too prohibitive for even the development version of the Arcadia environment. This overhead is due to support for dynamic redefinition, which should be avoided for our application, to support for powerful operations, which for the most part are not needed for our application, and to maintaining the assorted data objects in one monolithic structure instead of as separate objects associated with the set of tools that directly manipulate them. Thus, the data base approach is not a viable option at this point in time for large, complex system development such as an Ada environment.

Others have recognized the limitations of current data base systems and proposed alternative approaches. Probably the most popular of these is the Interface Description Language (IDL) and its processor. IDL is a BNF-like language for describing attributed graphs. The IDL processor takes an IDL description as input and creates a set of Pascal procedures that realize some of the operations available in interface packages generated by GRAPHITE. The IDL processor has been rewritten by several organizations so that it produces an Ada interface package. None of these organizations, to the best of our knowledge, has designed the interface package so that it is supportive of experimental development. In truth, the IDL system is a much more ambitious system than GRAPHITE. It is intended to be language independent and provides a number of additional capabilities. On the other hand, GRAPHITE is tailored for experimental development in Ada. Most notably, GDL is Ada-like and the development version of the interface package has been carefully designed to reduce the need for reprogramming and recompilation during prototype development.

The GRAPHITE system is currently being developed. It is being extensively tested before being distributed to the members of the Arcadia consortium. GRAPHITE is completely written in Ada and is being run on VAX/VMS, although we intend to port it to UNIX soon. We expect porting the system to be relatively straightforward since the only machine/operating-system dependencies are in reading and writing graphs and these have all been isolated to a small body of code. After the system has been exercised by the Arcadia community, we intend to make it available for public distribution.

Acknowledgements

We wish to thank Robert Graham and Mary Burdick for their contributions to the design and implementation of GRAPHITE.

REFERENCES

[1] Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), United States Department of Defense, Washington, D.C., January 1983.

[2] A. Evans, Jr. and K. J. Butler (eds.), *Diana Reference Manual (Revision 3)*, Technical Report TL 83-4, Tartan Laboratories Inc., Pittsburgh, Pennsylvania, February 1983.

[3] D.A. Lamb, *Sharing Intermediate Representations: The Interface Description Language*, Technical Report CMU-CS-83-129, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1983.

[4] R.N. Taylor, L.A. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young, *Arcadia: A Software Development Environment Research Project*, Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments, Miami Beach, Florida, April 1986.

[5] W.F. Tichy and M.C. Baker, *Smart Recompilation*, Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, January 1985.

[6] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *Interface Control and Incremental Development in the PIC Environment*, Proceedings of the Eighth International Conference on Software Engineering, London, England, August 1985.

[7] A.L. Wolf and M. Burdick, GRAPHITE *Users Manual*, Arcadia Design Document (in preparation).

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).