Toward Support For Environment Prototyping

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

COINS Technical Report 87–65
July 1987

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

This work was supported in part by the following grants: Rome Air Development Corporation, F30602-86-C0006; National Science Foundation grants DCR-84-04217.

ABSTRACT

There is widespread agreement on the need for software development environments. There is much less unanimity about how such environments should be organized and what tools they should contain. This suggests that environment research must be based on experimental, exploratory, *prototyping* activities. Therefore, one important element of environment research is the development of appropriate tools to support environment prototyping.

In this paper, we describe the requirements that we see for tools supporting prototyping of environments. We then describe GRAPHITE, a tool that represents one step toward appropriate support for environment prototyping. Specifically, GRAPHITE is intended to aid in the rapid development and easy modification of tools that manipulate graphs, since graphs are a very common data structure employed in environments. We report on our experience using GRAPHITE in several projects and outline future directions toward improved support for environment prototyping.

1 Introduction

There is widespread agreement on the need for software development environments. There is much less unanimity about how such environments should be organized and what tools they should contain. This suggests that environment research must be based on experimental, exploratory, *prototyping* activities. Therefore, one important element of environment research is the development of appropriate tools to support environment prototyping.

A full-fledged software development environment will consist of a large number of components, both tools and data objects, interacting with one another in a variety of ways. The effectiveness of the environment will depend upon both the quality of the individual components and the quality of their interaction, i.e., the *integration* of the environment. In addition, evaluation of an environment will depend heavily upon performance characteristics such as response time.

These considerations impose some requirements on tools for environment prototyping that differ from the requirements for tools supporting the prototyping of other kinds of software.

In an effort to create suitable environment prototyping capabilities for use in the Arcadia consortium's software development environment research [4], we have developed a tool, called GRAPHITE, that represents one step toward appropriate support for environment prototyping. The GRAPHITE tool, whose name derives from GRAPH Interface Tool for Environments, processes abstract data type specifications for classes of graphs. These specifications are written in a specialized Graph Description Language, GDL. Given the GDL specification for a particular class of graphs, GRAPHITE produces an implementation for that abstract data type in a strongly-typed, statically type-checked and compilable language, namely Ada. The GRAPHITE-produced Ada implementation, which we call an *interface package*, defines the type of the graph object and the operations that other environment tools can use to manipulate graphs of that type. The interface package encapsulates the

actual data structures and algorithms used to implement the type and its associated operations. The way in which GRAPHITE generates these data structures and algorithms, and their particular form, results in valuable support for environment prototyping.

In this paper, we first elaborate on the requirements that we see for tools supporting prototyping of environments. We then describe the GRAPHITE tool and illustrate how it can facilitate environment prototyping. We report on our experience with using GRAPHITE in several projects. Finally, we discuss related work and alternative approaches to the support provided by GRAPHITE and outline future directions toward improved support for environment prototyping.

2 Support for Environment Prototyping

Prototyping of environments has much in common with prototyping of other kinds of software systems. We have found, however, that some of the characteristics of environment prototypes are fundamentally different from those that typify other kinds of prototype software. These differences lead to significant differences in the kinds of tools needed to support environment prototyping. In this section, we first discuss support for prototyping in general and the traditional ways in which it has been provided and used. We then elaborate on the relevant characteristics of environment prototypes and their implications for tools to support prototyping.

2.1 Prototyping in General

Prototyping as an approach to producing software has two defining goals. These are:

- **Rapid Development:** A first version of a prototype software system should be up and running as quickly as possible. In other words, a developer should experience minimal delay between conceiving of a system and being able to experiment with a first prototype of that system.

- **Easy modification:** Changes in the prototype, often suggested by the results of previous experiments, should be easy to incorporate. In other words, a developer should experience minimal delay between experiments.

One aid to achieving these goals is an *application-oriented notation*. Clearly, the better a language is suited to the problem domain at hand, the easier it will be to describe a prototype of the system. Easy description should contribute to rapid development and easy modification.

Another aid to achieving the goals of prototyping is a *software reuse capability*. In fact, two somewhat different kinds of reuse—"as-is" and "variant"—can contribute. Being able to use an existing piece of software as-is in some other system certainly contributes to rapid development. It can also facilitate modification when a pre-existing component can be added as-is to a prototype. On the other hand, a relatively minor variation on an existing piece of software may provide exactly what is needed as (part or all of) an initial prototype of a different system or in modifying a prototype after some experimentation. Thus capabilities for both kinds of reuse can be valuable support for prototyping.

Finally, an important aid to easy modification is any mechanism that helps to *limit the impact of changes*. If a developer only wishes to experiment with changes to some restricted part of the prototype system, the effects of those changes should be confined to that part of the prototype as far as possible. Moreover, the delay resulting from a change should be proportional to the size of the change; if small changes result in large delays between experiments then the goal of easy modification will be thwarted.

2.2 Traditional Prototyping Applications

Several characteristics seem to be implicitly assumed in most applications of prototyping. Prototypes are typically seen as small to medium scale programs. Generally they are constructed and

used by a single developer. They tend to be rather monolithic, consisting of relatively few distinct pieces that are relatively loosely coupled. Efficiency, in terms of execution speed or space consumption, is apparently unimportant in most traditional prototyping situations. Finally, most traditional prototypes are taken to be "throw-away" systems, destined to be replaced by a completely new, built-from-scratch version once their experimental usefulness is through.

Particularly in light of these characteristics, the traditional approach to support for prototyping is quite reasonable. That approach is based on the use of a high-level language that is interpreted rather than compiled. Often the language is weakly typed, or typeless, and any type checking that it might provide is done at run time rather than statically. For certain well-understood applications, the language may be application-oriented. More often it is a general-purpose language such as LISP.

In the context of traditional prototyping applications, this approach can be seen to achieve the goals of prototyping quite well. Interpreted languages with limited typing and run-time type checking facilitate rapid development and easy modification in part because elimination of the time required to compile or type check reduces development and modification time. Given that efficiency is not an important consideration, the delays induced by compilation and static type checking are not considered worth the improvements in speed and space utilization that they can provide. The loss of control over interfaces between components and the reduced documentation of system structure resulting from weak typing or no typing are not deemed critical in relatively monolithic, single-developer, throw-away prototypes. Furthermore, an interpreted language limits the impact of any kind of modification; changes to instructions do not require recompilation and changes to data-object definitions do not require rechecking of type definitions and their usage.

2.3 Environment Prototyping

Some of the characteristics of environment prototypes distinguish them from most traditional kinds of prototypes in ways that suggest different approaches may be needed to achieve the goals of rapid development and easy modification.

One important difference is that environment prototypes are by nature large, complex and highly interrelated collections of components. Those components include tools, such as editors, compilers, testing and debugging support systems and the like, and also data objects, such as source text, abstract syntax trees, object modules, symbol tables, test-data sets, test results and many others. Moreover, environment prototypes are generally developed by groups rather than by one individual. Partly due to their size, and partly because of the kinds of experiments they are to be used for, efficiency in both time and space is significant for environment prototypes. Finally, since they are such large and complex systems, many of whose components may be large, complex and robust tools, it is less obvious that environment prototypes should be treated as throw-away systems.

One implication of these characteristics is that a transition path from prototype version to end-user version is extremely desirable for environment prototypes. Thus a very desirable property for a tool supporting environment prototyping is a transition strategy for going from a prototype to a robust and usable version. Such a strategy would primarily involve removing any remaining inefficiencies that may have been introduced to facilitate modification and reduce inter-experiment delay. The more easily this can be done, the greater the long-term value of the prototyping support tool.

Another implication of the characteristics of environment prototypes is that they are likely to be developed and modified component-wise. That is, a developer is likely to experiment with

one component at a time, by adding one or modifying one while leaving the rest of the prototype unchanged.

The characteristics of environment prototypes also imply that managerial control will be important. Controlled and disciplined change is crucial in a multi-developer setting, especially when dealing with a large, complex system composed of highly interrelated components.

Documentation also assumes an important role due to the characteristics of environment prototypes. Managerial control depends upon the visibility provided by good documentation. Easy change of complex and highly interrelated software in a multi-developer setting also demands appropriate documentation.

The characteristics of environment prototyping and their implications make the traditional approach to support for prototyping ill-suited for this application area. An interpreted language and run-time type checking will probably not result in sufficient efficiency for environment prototypes. A weakly-typed or untyped language makes modifications to a large, complex and highly interrelated system unacceptably error prone, while reducing the documentation needed in a multi-developer situation.

These considerations suggest that prototyping support for environments should be based on a compiled, relatively strongly-typed and statically type-checked language. Unfortunately, most such languages lead to unacceptably slow development and modification. Often a small change, especially if it involves a system component that is widely used by other components, necessitates a complete recompilation. This is generally true even if the change being made actually affects only a very few components.

Ideally then, support for environment prototyping would provide the efficiency, documentation and interface control of a compiled, strongly-typed and statically type-checked language. At the

same time, it would provide means for limiting the impact of changes so as not to interfere with the rapid development and easy modification that are crucial to prototyping.

The GRAPHITE system that we discuss in the remainder of this paper represents one step in this direction, applicable to one important class of environment components, namely abstract data types for graph objects. GRAPHITE works within the context of a compiled, strongly-typed and statically type-checked language, namely Ada, but introduces an application-oriented language to aid in rapid development and a means for limiting the impact of changes that facilitates easy modification. Finally, GRAPHITE provides a transition strategy for going from prototype versions of these abstract data types and their operations to versions that are suitable for use in environments that could be used in real software development applications.

3 The GRAPHITE System

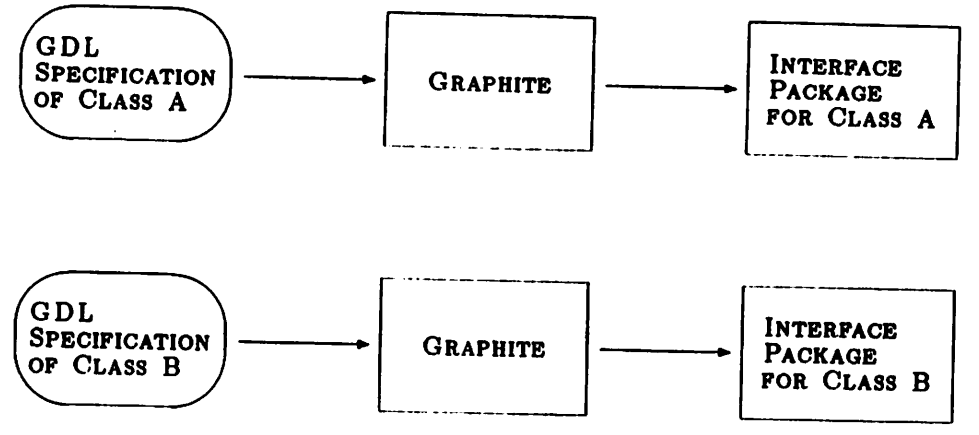
Many of the data objects manipulated by software development environment tools are *graphs*. For example, parse trees, abstract syntax trees, control flow graphs, and call graphs are all classes of graphs that are likely to be manipulated by tools in an environment. Moreover, an individual instance of a graph class may be shared by several different tools in that environment. For example, an abstract syntax tree may be initially created by a syntactic analyzer, modified by a semantic analyzer and referenced by a pretty printer, among other tools. During the prototyping phase in the design of a software development environment, experimentation with tools may dictate changes to the structure of a graph as well as changes to the graph's underlying representation. The GRAPHITE system facilitates both kinds of experimentation while minimizing the impact of that experimentation on the tools that manipulate that graph in an environment. In addition, GRAPHITE provides a medium for clearly documenting the graph structures used in an environment

so that new tools can consider these structures as candidates for reuse. Moreover, GRAPHITE provides support for moving from a prototype implementation to a production implementation once the design of the graph structures has stabilized and efficiency becomes more of a concern than flexibility. Thus, for graph structures, GRAPHITE provides a prototyping facility that satisfies the objectives described in the previous section.

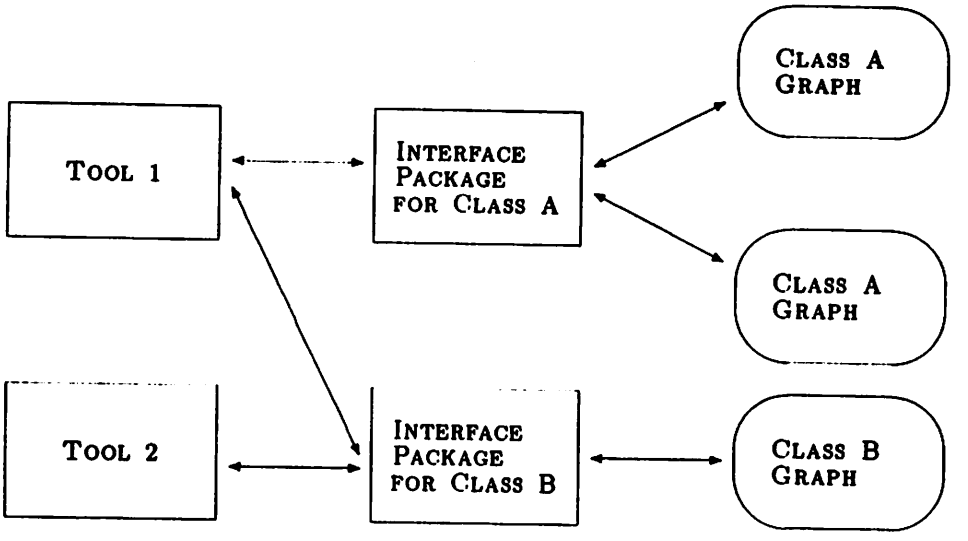
The GRAPHITE system accepts specifications of classes of graphs written in the graph description language, GDL. Given the GDL specification for a particular class of graphs, GRAPHITE produces an implementation of an abstract data type for that class of graphs, which is encapsulated in the generated *interface package*. The interface 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.

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 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.

Environments are large and complex and often even prototype environments cannot be effectively evaluated unless they are somewhat efficient. To this end, GRAPHITE produces two different kinds of interface packages. One supports software development of experimental systems. It is



(a)



(b)

Figure 1: Creating and Using Graph Interface Packages.

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 tools can be created. Thus we have support for both development and production versions of the interface package and a process for easily going from the development to the production version.

In the following subsections we describe the GRAPHITE system. First, GDL is described and an example is given. Second, the interface package is described. The capabilities provided by this package as well as the underlying design for both the development and the production versions of the abstract data type are presented. An example of how one might use the system and how to move from the development version to the production version of the system are provided. The GRAPHITE system has been implemented in Ada (the host language) for use in building environment tools in Ada (the target language). In the ensuing discussion, the examples are given in Ada but the basic concepts are independent of the host and target languages. The major requirements for the target language are that it provide support for abstract data types and for separate compilation, including separate compilation of the specification part of an abstract type from its implementation part.

3.1 Graph Description Language

In GDL, a *class* of 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 graph, which is a member of some class of graphs, is then just a set of instances of node kinds in that class.

As noted above, GDL facilitates the development of software systems by documenting the available graph classes. From this description, developers can decide if an object they need already exists or can be easily created from existing graph classes. To make this description as natural to use and easy to understand as possible, it should be modeled afer the target language whenever possible. Since our environment development work is being done in Ada, we have implemented GRAPHITE to produce an Ada abstract data type and have designed a GDL that is very Ada-like. To illustrate the information that must be provided by a GDL description, we shall use this Ada-like notation. Thus in Figure 2, the GDL specification for a given graph closely resembles, at least syntactically, an Ada package specification that contains a collection of Ada-like type declarations.

A given GDL specification defines a particular class of graphs by declaring the node kinds, attributes, and attribute value types making up the class. 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 the remainder of this section to illustrate features of GDL and the interface packages.

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

```
class ExampleGraph is
  package Example;

  with Lexical.( Comment, Position );

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

  type BranchWeight 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 Graphs.

IfStatement are two node kinds declared in Figure 2.

GDL provides a syntactic shorthand 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 attributes are the same). This syntactic shorthand allows a commonly-available attribute declaration to be made in one place and then used in various node kind declarations by simply listing the attribute's name. For example, in Figure 2, node kinds ConditionNode and IfStatement contain identical declarations for the commonly-available attribute named SourceConnection. Whether or not this 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 types, user-defined types, imported types, and GDL-specific types. Predefined types are those types defined within the target language such as Character, Boolean, and Integer. User-defined types use the target languages's type constructors to define types from the pre-defined types or other user-defined types. For imported types, we use an Ada-like *with clause* to indicate which external types are to be imported. For example, in Figure 2, types Position and Comment are imported from package Lexical and used in defining the record type LexicalInformation. Although not described here, other information, such as the definition of an assignment operator for objects of that type, must sometimes be provided with imported types.

There are three GDL-specific types, *node kind*, *node group*, and *node sequence*. Each is an attribute value type constructor that facilitates describing a graph class. 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 operations in the generated interface package.

A more complete description of GDL is given in [7].

3.2 Interface Packages

Having described the input to GRAPHITE, we now turn to the output, namely the interface packages that implement a given GDL specification of a class of graphs. As mentioned above, GRAPHITE can generate either a development version or a production version of such a package. The versions are similar in that they realize a class of graphs 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 easy modification 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 subsection presents our designs for the development and production versions of the interface package. First, an overview is given of the operations on 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.

3.2.1 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.

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. Because the get (put) operations differ in the type of the attribute returned (entered), the subprograms that implement them can be overloaded in a language like Ada, i.e., they can be 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. Compared to the alternative of having one get/put pair per attribute (e.g., Get Weight/Put Weight), our alternative results in a unique pair per attribute value type. In the extreme, these alternatives are the same 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. One consequence of our choice is that general purpose tools would need fewer arms in case statements to process all attributes of a node kind. Furthermore, we would expect that the set of attribute value types would change much less often than the set of attributes. Therefore, our approach provides greater protection against changes in class definitions.

A second thing to notice about the set of operations provided by interface packages is the

1. OPERATIONS TO MANIPULATE A NODE	
Create	creates a new node of a given kind
Delete Node	deletes a node
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
Remove	removes a node at a given position from a sequence
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.

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 a 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 a 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.

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. Such capabilities, if provided, would adversely affect reuse since the description of the graph class could not be easily determined. Thus we decided not to provide such capabilities. 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, programs. Then we may not want to incur the cost of complete re-analysis because graph representations have undergone change. Since we do not expect such loads on prototype environments, operations to perform instance-to-instance translations have not been included.

3.2.2 Ada Development Interface

Figure 3 shows a portion of the specification part of the development version of the Ada 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 and for the Ada types realizing the attribute value types of the class definition.

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 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.

```
-- imported types and subtypes
   with Lexical;
package Example is
   -- GDL-specific types
      type ExampleGraph          is private;
      type ExampleGraphSequence is private;

      NullExampleGraph          : constant ExampleGraph;
      NullExampleGraphSequence : constant ExampleGraphSequence;

   -- 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 AttributeNamePointer is access AttributeName;
      type AttributeNameList    is array ( Positive range <> ) of AttributeNamePointer;

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

Figure 3: Specification Part of Interface Package for Class of Figure 2 (Development Version).

```
        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 ( TheNodeKind : NodeKindName; TheAttribute : AttributeName )
                                    return AttributeValueTypeName;
        function NodeKindAttributes ( TheNodeKind : NodeKindName ) return AttributeNameList;
    -- operations to manipulate node sequences
        function  Create ( NodeSequence : NodeSequenceName ) return ExampleGraphSequence;
        procedure Insert ( TheSequence : ExampleGraphSequence; ThePosition : Positive;
                           TheNode : ExampleGraph );
        procedure Remove ( TheSequence : ExampleGraphSequence; ThePosition : Positive );
        function  Kind ( TheSequence : ExampleGraphSequence ) return NodeSequenceName;
        function  Length ( TheSequence : ExampleGraphSequence ) return Natural;
        function  Retrieve ( TheSequence : ExampleGraphSequence; ThePosition : Positive )
                             return ExampleGraph;
    -- operations to input and output graphs
        procedure ReadGraph ( FileName : String; TheGraph : in out ExampleGraph );
        procedure WriteGraph ( FileName : String; TheGraph : in out ExampleGraph );
    -- exceptions
        UnknownNodeKind     : exception;  UnknownAttribute  : exception;
        UnknownNodeSequence : 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;
        NullExampleGraph         : constant ExampleGraph := null;
        NullExampleGraphSequence : constant ExampleGraphSequence := null;
end Example;
```

Figure 3: (continued).

Thus, for the example shown in Figure 3, 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 same guarantee is made for node sequences, which like nodes are designated by objects of one type.

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 portions of tools that use the package. 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 graph class being managed and so insulates users of that package from most changes in class definitions and representations.

Definition Independence. In the specification part shown in Figure 3, 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 discussed 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 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.[1]

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 a 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 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 illustrated in Figure 3, 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.

[1]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.

In sum, our design for the development version of an interface package is an attempt to carefully and selectively circumvent compile-time type checking so that tools can be insulated from changes in the defintion and representation of a class of 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 those portions of tools directly interested in those changes. In addition, a change to the representation of a class of 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 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 at run time the legality of node kind, node sequence, and attribute names as well as the operations applied to instances of its class.

3.2.3 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 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.

3.2.4 Moving From Development to Production

Once a 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 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. 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 primarily requires that the *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 in Ada 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 the string-constant declarations.

4 Experiences Using GRAPHITE

GRAPHITE has been used extensively during the past two years in the development of the graph data structures of several environment tools. A short description of some of those tools and graphs follows.

- *Ada Front End Toolset.*

  A collection of tools for performing lexical and syntactic analysis of Ada programs, and for creating and displaying a general directed graph representation of those programs. The toolset consists of two kinds of lexical and syntactic analyzers, one table driven and the other hard coded, that both create program-representation graphs. The table-driven analyzer, called Athena, manipulates a second class of graphs, namely a binary tree used to organize an identifier table. The toolset also consists of a pretty printer that "unparses" a program-representation graph, and a graph walker that produces a linear, textual listing of a program-representation graph useful for debugging the other tools. The toolset was developed cooperatively by researchers at separate sites: the University of California, Irvine, and the University of Massachusetts, Amherst.

- *PIC/ADL Front End.*

  A lexical and syntactic analyzer for an Ada-based design language that focuses on the specification of module interface relationships. This tool was built by modifying Athena and its associated program-representation graph. Athena's other graph structure, the binary tree, was borrowed as-is.

- *AdaPIC Analysis Toolset.*

  A collection of tools that extract, analyze, and report on information about the consistency of software module interfaces (as specified, for example, in PIC/ADL). Two classes of graphs are shared among these tools. Both are essentially linked lists that are built using the node-sequence mechanism (see Section 3.1).

- *Loop Analyzer.*

  A tool that determines and solves a loop's recurrence relations, which describe how the values of the variables modified by a loop traversal are computed from their values on previous traversals. This tool uses two classes of graphs, one for representing loop bodies as trees and the other for representing the traversal context as linked lists of linked lists. Here too, the linked lists are built using the node-sequence mechanism.

- *CEDL Front End.*

  A lexical and syntactic analyzer for a design language that focuses on the specification of task (i.e., process) definition and interaction in concurrent systems. In addition to lexical and syntactic analysis, the front end creates graphs that are trees used to represent CEDL programs.

- *GRAPHITE.*

  The GRAPHITE processor itself uses two classes of graphs and was built by bootstrapping off an earlier version. In fact, the component of the GRAPHITE processor that constitutes the GDL front end is, like the PIC/ADL front end, a modified version of Athena. It produces, as does the PIC/ADL front end, a variation on Athena's program-representation graph and borrows as-is the binary-tree graph structure for an identifier table.

These uses of GRAPHITE exhibit a wide range of complexity, from a class consisting simply of one node kind (the binary tree of Athena) to a class consisting of 47 node kinds, 6 node groups, and 16 node sequences (the tree of the CEDL front end). They also involve a wide range of development activities, including rapid development, reuse, modification, and transition from development to production. In the remainder of this section we relate a sampling of experiences with GRAPHITE in the development of these tools. On the whole, these experiences substantiate our claim that GRAPHITE is a useful prototyping tool and that prototyping in a strongly-typed, compiled language with enforced documentation has significant advantages for environment development.

The AdaPIC toolset [8] was built as a prototyping effort to demonstrate the feasibility of the PIC approach to interface control [6]. The idea was to quickly "get something up" that exhibited the essential functionality of the system. During design, we identified the central data structures as lists and the principal operations as various kinds of traversals of those lists. We chose to use the

ready-made GRAPHITE abstraction of lists, the node sequence, because with very little effort—the specification in GDL of the components of the lists—the code for manipulating those lists could be obtained. The automatic generation of input and output operations in particular was seen as important and useful aids to rapid development, since the lists were to be shared among several tools being developed concurrently.

The table-driven Ada front end Athena was built using three tool-building tools in concert. In addition to GRAPHITE, Ada-oriented versions of the familiar Unix tools LEX and YACC, called ALEX and AYACC, were used to generate the tables used in lexical and syntactic analysis.[2] In all, Athena consists of 23 separately compilable units that total over 750 kilobytes of source code (Table 2). Compilation of a moderately-sized program such as this takes a substantial amount of computer time and, perhaps more importantly, programmer time. The component of Athena that generates program-representation graphs was actually developed incrementally by successively handling larger and larger subsets of the Ada language. Growth from one subset to the next often involved changes to the definition of the program-representation graph. The only part of the program "interested" in such changes was the set of so-called "actions" that are performed by the front end; these actions are embodied in unit Actions. By using the development version of the interface package, we were able to minimize the recompilation necessitated by changes to the definition of the program-representation graph. In particular, only the body part of RepPack (and, of course, Actions) had to be recompiled; the other units in the program were insulated from such changes. This reduced recompilation time by over half as compared to what would have been required if the definition of the program-representation graph had been exposed.

Another tool in the Ada front end toolset that benefited from our design of the interface package

[2]ALEX and AYACC were developed at the University of California, Irvine.

UNIT	KIND†	BYTES	LINES	SEMICOLONS	COMPILATION TIME‡
Actions§	S	68924	2317	1140	553.0
Athena	S	1821	53	124	9.5
BTreePac	PS	4729	161	69	5.6
BTreePac	PB	19525	697	310	94.1
GoTo	PS	53036	2798	14	552.0
RepPac	PS	7182	227	116	7.7
RepPac§	PB	352634	10575	4418	1552.1
LitTable	PS	2417	55	12	3.6
LitTable	PB	6051	154	65	18.1
MakeNode	PS	2027	43	6	3.1
MakeNode	PB	9870	335	33	109.8
Parser	PS	1502	36	9	4.3
Parser	PB	2020	52	15	25.7
ParserUtil	PS	1523	32	4	2.8
ParserUtil	PB	1387	31	5	4.7
Scanner	PS	242	14	11	4.7
Scanner	PB	153972	2873	2599	826.9
Shift	PS	50915	2754	11	557.8
SymTable	PS	1706	34	5	2.9
SymTable	PB	4553	131	56	18.1
Tokens	PS	3009	70	7	6.1
Utilities	PS	1368	28	3	2.8
Utilities	PB	1656	44	10	5.5
TOTALS		752069	23514	8932	4377.7

†S–subprogram; PS–package specification; PB–package body.

‡CPU seconds for VERDIX Ada compiler (5.2a) on a DEC VAXStationII running ULTRIX.

§A VERDIX limitation actually necessitates the breaking up of this unit into subunits for compilation; time shown is a sum of subunit compilation times and is therefore somewhat higher than expected due to the extra processing required.

Table 2: Compilation Units Composing Athena.

was the graph walker, Monkey. Monkey makes extensive use of the functions that ascertain a node's definition. As a result, it was possible to insulate this entire tool from changes to the definition of the program-representation graph as that class was incrementally developed. The only change that could cause a particular version of Monkey to become obsolete would be one to the set of attribute value types. As it happened, this kind of change never occurred and thus we were able to use the same, stable version of Monkey throughout the development of the other tools in the toolset.

As mentioned above, tools in the Ada front end toolset were developed at two different sites, making the coordinated development of their principal data structure, the program-representation graph, difficult. This problem was eased somewhat by the use of GRAPHITE. In particular, the GDL description of the data structure served as documentation that could be passed among developers. Moreover, the legitimacy of that documentation could be guaranteed, since it was the documentation itself that was used to generate actual code. Changes to the definition of the program-representation graph, because they were made by changing the GDL description, became more visible, and hence more controlable, than they would have been if the changes were reflected only in the changed code of a tool.

The PIC/ADL and GDL front ends are examples of tools whose development costs were substantially reduced by reuse of existing software. In this case, (re)use was made of Athena, particularly the inputs to the three tool-building tools ALEX, AYACC, and GRAPHITE. PIC/ADL and GDL both happen to be extended subsets of Ada. This is reflected in the changes that were made to the inputs to the tool-building tools, such as the removal of some node kinds and the addition of others to the definition of the program-representation graph. Making such changes was significantly easier and less error prone than either modifying the code of Athena directly or constructing the inputs to the tool-building tools from scratch.

We should also point out an interesting side effect of having used a GRAPHITE-generated interface package for the program-representation graphs of PIC/ADL and GDL. Monkey, the graph walker developed as part of the Ada front end toolset, can be used as-is on the program-representation graphs produced by the PIC/ADL and GDL front ends, despite the fact that those tools produce graphs of markedly different classes. This is a consequence of the fact that functions are available to Monkey that allow it to "discover" the set of node kinds and attributes defined for a given class, rather than having those definitions embedded directly in Monkey's code.

The PIC/ADL front end has recently reached a point in its development were we feel confident about the stability of its program-representation graph. We are now interested in actually using the tool, which implies that our focus has shifted from the flexibility provided by the development version of the graph's interface package to the relative efficiency provided by the production version. Our preliminary results indicate that by moving to the production version, we were able to reduce execution times by approximately one third.[3] Moreover, by adhering to the guidelines for referring to names of node kinds and attributes that are described in Section 3.2.4, we were able to move from the development version to the production version with virtually no recoding required of the tool.

GRAPHITE, even in its still early versions, has proven a valuable aid to the prototype development of a variety of tools and toolsets. It has, among other things, facilitated the rapid development of significant components of tools by automating mundane coding chores, reduced the costs of making modifications by limiting unnecessary side effects and automating code changes, increased the bandwidth of communication among cooperative developers by providing high-level

[3]In general, the speed up experienced by a given tool is highly dependent upon the kind of processing being done by that tool; the greater the percentage of a tool's code that involves directly manipulating the graph, the greater the expected speed up in the overall execution of the tool.

and enforceable documentation, which has also served to enhance the reusability of the software, and provided a mechanism for moving from flexible development versions of an interface package to more efficient, but less flexible, production versions. In future versions of GRAPHITE, we intend to concentrate on enhancing the reusability of class definitions, perhaps along the lines of the inheritance mechanisms of object-oriented languages, and on increasing the efficiency of production versions of interface packages.

5 Conclusion

In this section we look at alternative approaches for supporting the development of graphs and discuss why we believe the GRAPHITE approach is superior for prototype development. We then discuss some extensions to the system that would be beneficial, as well as discuss more long term directions of future research on managing complex objects for software development environments.

There are two major alternatives to using GRAPHITE. One is to use a traditional database approach and the other is to use an alternative graph application language, such as the IDL processor [3]. With a database management system, the classes of graphs that are to be manipulated are described using a database schema. Such systems provide a basic set of operations to create, enter, and retrieve information. For example, a relational database system provides a consistent set of operations based on relational algebra for manipulating attributed graphs.

One of the major drawbacks to a datebase approach is the lack of support for user-defined types and the run-time overhead of such systems [1]. Typically, database 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. Another drawback is the run-time overhead required for the newer, more powerful database models, such as the relational

approach. The run-time overhead for such systems is too prohibitive for even the prototype version of an environment. This overhead is due to support for dynamic redefinition, which we have argued should be avoided for our application, to support for powerful operations, which for the most part are not needed for our application, and to maintain 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 database approach is not a viable option at this point in time for large, complex system development such as an environment.

Others have recognized the limitations of current database 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 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 C and Ada interface packages. None of these organizations, to the best of our knowledge, has designed the interface package so that it is supportive of prototype 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 and provides a mechanism for moving from development to production.

The GRAPHITE system has proven to be very successful at meeting our prototyping needs. Rapid tool development has been achieved for those tools or components of tools that primarily manipulate graphs. The capabilities in GDL for describing classes of graphs and the operations for manipulating instances of graph objects have been demonstrated to be useful and appropriate for a number of projects. GRAPHITE has also been shown to facilitate modifications and reuse. By

minimizing the impact of change, GRAPHITE allows developers to quickly and easily experiment with new class objects. The GDL specifications provides documentation on the available graph classes, allowing developers to peruse these specifications to determine if object classes already exist that meet their needs. If some do, then developers can reuse their interface packages and perhaps even instances of those classes "as-is." If an available class is not identical but only similar to the desired structure, then modifications that alter the structure can easily be made to these graph class descriptions and automatically incorporated into the interface package. Such changes have a minimal impact on existing tools, usually requiring no changes to those tools that only manipulate the unchanged aspects of the modified graph class.

Graphite can be viewed as an application language for describing and manipulating graphs. For software development environments, this is a common and important object type. We have found that the current set of capabilities effectively support a wide range of tool applications. This is not to say that users will not need to build tool-specific operations on top of the GRAPHITE provided operations. It is our experience that GRAPHITE provides a reasonable set of primitive graph operations that readily support the construction of higher levels of abstraction and/or functionality.

While we have been pleased with the prototyping gains made with the GRAPHITE system, we realize that it is only an initial step toward achieving more general prototyping support. There are modifications to programming language and compiler capabilities as well as extensions to the GRAPHITE system that could greatly improve upon this prototyping support.

Our work with GRAPHITE has pointed out several ways in which programming languages and compilers could be more supportive of development work. For example, although there has been work on smart recompilation [5], most compilers still do a poor job of recognizing when recompilation can be avoided. In addition, extensions to programming languages should be considered that

would greatly improve the ability of compilers to do this task. For instance, constructs that allow a programmer to indicate the volatile and stable parts of their system could be used in determining which underlying representation should be used when such choices impact separate compilation. Our previous work on interface control [8], in which modules may indicate the objects and types they intend to reference as well as the modules to which they are making particular objects and types available, also provides information that facilitates smart recompilation.

There are many extensions to GRAPHITE that one could consider. One such extension is to support multiple views of the graph classes. Currently, GRAPHITE requires a single description of a graph class, although that description may be shared by several different tools, where each is only concerned with a subset (or view) of the structure. It would be beneficial if tools could describe the particular view of a structure that they wanted.

Related to views is the ability to partition the physical representation of an instance of a graph based on some criteria. One option would be to partition the representation based on the views, under the assumption that views represent information about locality of reference. Since environment objects can be quite large, such efficiency issues are of concern even within the prototyping domain. Another alternative is to hide the physical partitioning from the tool developer, leaving the system to make such decisions based on metrics such as frequency of use. Thus, views could be coalessed into a single structure or physically partitioned based on user or system monitored metrics.

Another concern is our model of graphs, which currently treats graphs simply as collections of nodes. A more powerful model of graphs that recognizes larger-grained graph operations would prove useful. For example, the concept of "subgraph" is clearly important in many applications. Except for subtrees, there is currently no way to read or write parts of a graph, since all nodes

reachable from the selected start node are written or read. If graphs were treated as collections of subgraphs, then developers could take advantage of that abstraction by indicating which subgraphs are to be the focus of particular operations.

As part of the Arcadia project we are working on issues related to better object management. Our work with GRAPHITE, although restricted to one type of object, namely graphs, has provide valuable insight into the difficulties of supporting complex objects. For the Arcadia environment, our goal is to support arbitrary abstract data types. In addition, we are considering the design of an object manager that would recognize views of objects, have a flexible storage mechanism that could be directed by the programmer to partition objects based on perceived locality of reference and frequency of use, but could override these directives based on experiential evidence. We are considering such capabilities as object persistence in which instances of objects can be written to and read from secondary storage without forcing the developer to explicitly program how to go from the program representation to the linear secondary storage representation. And we are incorporating interface control information into the data abstraction so that the minimal impact of change can be recognized and exploited by the object manager. Again we are focusing on the needs of software development environments and the complex, interrelated objects that they create and manipulate. The presence of such an object manager will greatly facilitate environment development by automating many of the tasks, documenting available types and objects and their interrelationships, and minimizing the impact of change. Even production environments must be designed to accommodate change since the tool set and the associated types and objects must be allowed to grow as developers' needs change. Thus the issues we are exploring should cross the boundary between prototype and production development in this domain, as it most certainly does in other problem domains.

Acknowledgements

We wish to thank Mary Burdick, Peri Tarr, William Rosenblatt, and Robert Graham for their contributions to the design and implementation of GRAPHITE.

REFERENCES

[1] P.A. Bernstein, *Database System Support for Software Engineering*, **Proceedings of the Ninth International Conference on Software Engineering**, Monterey, California, March 1987, pp.166–178.

[2] **Reference Manual for the Ada Programming Language** (ANSI/MIL–STD–1815A), United States Department of Defense, Washington, D.C., January 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, pp.137–149.

[5] W.F. Tichy, *Smart Recompilation*, **ACM Transactions on Programming Languages and Systems**, Vol. 8, No. 3, July 1986, pp.273–291.

[6] A.L. Wolf, *Language and Tool Support for Precise Interface Control* (Ph.D. Dissertation), **COINS Technical Report 85–23**, COINS Department, University of Massachusetts, Amherst, Massachusetts, September 1985.

[7] A.L. Wolf, GRAPHITE *User Manual*, **Arcadia Design Document**, 1987.

[8] A.L. Wolf, L.A. Clarke, and J.C. Wileden, *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process*, **IEEE Trans. on Software Engineering** (to appear), 1987.