Using Graph Deltas to Implement Programming Support Libraries

David Alex Lamb

June, 1988
Technical Report 1987-197
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
CANADA
(613) 545-6067

Version 2.3
Document prepared Friday, March 27, 1998

Abstract

Many programs read an attributed graph and manipulate it to produce an output graph that resembles the input. This paper shows how to organise such a program so that it can produce a ''delta'' showing how the output differs from the input. Storing deltas instead of full graphs saves space; it also gives a way to annotate ''frozen'' data structures that programs cannot modify directly. The scheme is especially suitable for modern programming environments, where collections of diverse tools, written and maintained by different groups, operate on libraries of graph-structured program representations.

Keywords and phrases: graph differences, graph structures, input/output, information hiding, program libraries, software engineering

Computing Reviews categories: D.2.2 (software - tools and techniques), D.2.6 (programming environments) D.3.3 (modules, packages)

General terms: Algorithms, Design, Performance

1. Introduction

We can view many programs as instances of the *graph transformation paradigm*: they read a graph and transform it into a modified but related graph. In general the output graph may be an instance of a slightly different *schema* (set of node types and corresponding attribute descriptions) than the input. For example, the DIANA representation for Ada[1] characterises the semantic analyser of a compiler as transforming an abstract syntax tree into an augmented attributed graph that contains a slight transformation of the syntax tree as a spanning tree; the added attributes record the results of semantic analysis [3]. This paradigm is becoming more common; much work on modern programming support environments uses graph-like representations of programs, composed of objects and links between objects (for several examples, see the proceedings of the conference on Practical Programming Environments [4]).

If an output graph is a transformation of an input graph, we could represent the output graph as a *delta*: a record of how to change the input to create the output. Storing graph deltas is important for two reasons:

1. It reduces storage requirements when users need to keep both the input and output of a program.
2. It gives a way to represent annotations to a data structure whose representation is immutable (such as one frozen by configuration management disciplines).

Unfortunately, determining a set of edits to transform one graph into another, given only the two graphs, is difficult. The apparently related problem of determining whether two graphs share an isomorphic subgraph is NP-complete[2]. However, given control over how the application program accesses its data, it is easy to detect what changes it makes and what portions of the original structure remain untouched.

Section 2 shows methods for detecting and recording deltas; the remainder of this section discusses constraints on the environment in which you can use them. Section 3 discusses other issues about using these methods and suggests some extensions; Section 3.2 discusses the special case where the input and output schemas are identical, and one is only changing instances of nodes and attributes.

1.1. Constraints on the Program

Figure 1 shows the overall organisation of a program that can use the techniques of Section 2. The program views its data structures as typed, attributed, directed graphs. The type of a node in the graph determines what attributes it has; each attribute has a specific type that determines what types of nodes it may reference. The program treats internal representations as abstract data types, accessing them through an interface package (Interface II, hereafter called the application interface). The operations the application may use are

- Create a node of a particular type.
- Fetch the value of a particular attribute of a particular node.
- Store a new value in a particular attribute of a particular node.

The package implementing the abstract data type segregates old data (from the input file) from new data (created by the application). Graph readers and writers take care of input and output; they use a wider interface (Interface I, hereafter called the reader/writer interface) that gives them access to more information about the representation. Using two interfaces lets us hide representational details from the application and the input/output routines, and also lets us hide from the application whether we are reading full graphs or deltas.

Mentioning graph readers and writers presumes some method of reading and writing general graphs. The following discussion presumes you begin with a writer that can output a representation of any connected subgraph as a single recognisable unit, and a reader that can read such a subgraph and create an internal representation. Each might have to deal with labels to represent references to shared sub-graphs;

[1] Ada is a registered trademark of the U.S. Government (OUSDRE-AJPO).

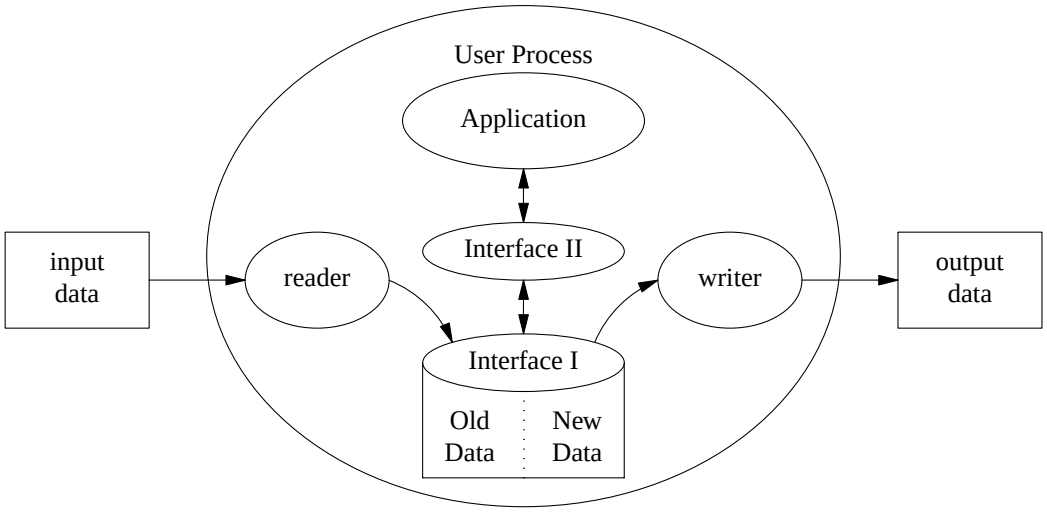

Figure 1: General Process Organisation

Arrows indicate direction of data flow.

sub-graphs without sharing must be trees, and can be represented via nesting to avoid labels. For reasons that Section 2.1 discusses, it should be possible for the graph writer and the graph reader to traverse the graph in a canonical order. Practical versions of readers and writers with these properties already exist; the Interface Description Language (IDL) system is one example [6, 10].

1.2. Automation

Coordinating readers, writers, and interfaces can be a tedious and error-prone task, especially since writers and readers must traverse the graph in the same order. It is much easier to use these methods if the environment contains a tool (hereafter called the package generator) which can generate the interface packages (as well as readers and writers) from descriptions of the data structures. Existing IDL systems have such a tool; extending them to handle the methods of this paper should be conceptually straightforward but would likely require considerable implementation effort. To be as widely useful as possible, this paper discusses a generic data definition language and corresponding environment; only the concrete examples use IDL.

The package generator and runtime system should have five characteristics.

1. The input to the package generator (hereafter called the DDL, for Data Definition Language) describes schemas for data structures, stating what attributes each type of node contains, and what types of node each attribute may reference.
2. The DDL has mechanisms for succinctly describing the ways in which the schema for the output of a tool differs from the one for its input.
3. The DDL has mechanisms for saying what attributes the application might change and what types of nodes it might create; the package generator can enforce these constraints by deleting operations from the application interface.
4. Given a reference to a node of any data structure, the readers and writers can determine its type, the name and nature of its attributes, and some boolean properties such as whether it came from

the input file or was created afterward.

5. It is possible for a writer to distinguish nodes the corresponding reader creates from nodes the application creates.

IDL and its translator are one such DDL and package generator. In previous work I have shown that the IDL-generated graph readers and writers are practical tools for communicating between particular pairs of tools [6]. Nestor et al. [7] have described a representation for IDL that satisfies property 4; such representations are part of several IDL implementations. Biyani [1] has designed a runtime system with similar properties. There are several possible mechanisms for satisfying Property 5, including using different regions of memory, or tagging individual nodes.

Figure 2 shows an IDL declaration for a simple tree structure representing programs in a trivial programming language. The names program, const_decl, var_decl, binary, and unary are *node types*; objects of these types are nodes in a graph. Declarations with left arrows (::=) define *strict classes*, which resemble union types. The names statement, expression, and declaration are strict classes. The term *class* includes both node types and strict classes. Declarations with right arrows (=>) say that each node that is an instance of the class whose name precedes the arrow has all the *attributes* after the arrow (up to the semicolon). An attribute has a value of a basic type, such as integer or string, a sequence type (Seq), or a reference (pointer) to another type of object.

Figure 3 shows a new schema derived from that of Figure 2. The new structure, graph, inherits all the declarations of the old structure, lang, except those explicitly named in **without** clauses. Thus, for example, the graph has no const_decl nodes, and var_decl nodes no longer have initial_value attributes. The identifier class splits into two sub-classes: defining occurrences (in declarations) and used occurrences

---

```
Structure lang Root program Is
   program =>
      decls : Seq of declaration,
           stmts : Seq of statement;
   declaration ::= const_decl | var_decl | ... ;
   const_decl =>
      name : identifier,
           value : expression;
   var_decl =>
      names : Seq of identifier,
           type : type_specification,
           initial_value : expression;
   statement ::= ...; -- several types of nodes representing statements
   expression ::= binary | unary | primary;
   expression => op : operator;
   binary =>
      lhs : expression
           rhs : expression;
   unary => operand : expression;
   primary ::= identifier | literal | ...;
   identifier => token : string;
End
```

Figure 2: IDL Structure for a Simple Language

```
Structure graph From lang Is
   Without const_decl;
   Without var_decl=>initial_value;

   identifier ::= defining_occurrence | used_occurrence;
   used_occurrence => def_occ : defining_occurrence;

   For statement Use Forbid(create,delete)
   For var_decl.type Use Forbid(store)
End
```

Figure 3: Example of Derived Schema

(in expressions and statements). Used occurrences include pointers back to the corresponding defining occurrences. The **for** clauses say that we will not create new statements in the graph, and that we will not change (store into) the type attribute of var_decl nodes.

2. The Scheme

This section shows how to modify a program that reads and writes entire graphs so that it instead writes a ‘‘delta’’ consisting of a reference to the input file followed by differences between the input and the full output graph. Later sections will refer to Figure 4. Program 1 and Program 2 are each instances of the general organisation of Figure 1; the dotted lines between readers and writers stand for the omitted portions of

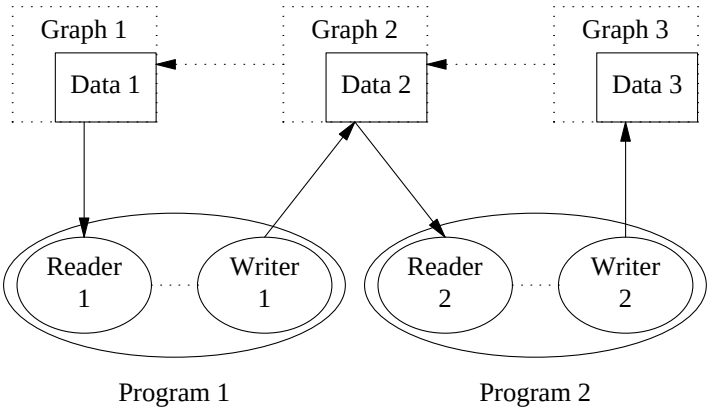

Figure 4: A Pair of Cooperating Tools

the organisation. The boxes labelled ‘‘Graph i’’ represent logical input and output files; ‘‘Data i’’ represent the physical implementation. The dotted arrow from Graph i to Graph i-1 represents a reference in Data i (the delta) to the input file it modifies to give a full graph. Reader 1 might simply read an entire graph (if Data 1 is a full graph rather than a delta); Reader 2 is necessarily more complex. As it begins to read Graph 2, it finds the reference to Graph 1 and reads it in the same way that Reader 1 did. It then reads Data 2, modifying its internal representation appropriately. A reader for Graph 3 would go through two levels of this, first reading Graph 1, then modifying it with Data 2, then modifying that with Data 3.

The following subsections describe how to handle increasingly more complex kinds of deltas. We begin by showing how to handle deleting attributes, then work up to allowing arbitrary changes in several steps:

- adding simple (non-pointer) attributes,
- adding pointers to new types of node not present in the input,
- allowing back-pointers from new nodes to old ones,
- adding pointers to new instances of types of objects present in the input,
- changing simple attributes,

and changing pointer-valued attributes.

‘‘Adding’’ means adding new attributes to nodes from the input file, and ‘‘changing’’ means modifying the value of an old attribute from the input file. Changing attributes of newly-created nodes introduces no problems, since the writer will always emit a full representation of any new node.

2.1. Representation of Deltas

A delta file may contain some fixed overhead (such as a reference to the file containing the original graph). The remainder of the file is a sequence of directives of the form

```
identification of changed node
  set attribute A1 to value V1
  set attribute A2 to value V2
```

Given a way to write out a connected subgraph as a unit (so that V1 and V2 could represent graph-valued attributes), this suffices to describe any set of changes to a graph.

One way to represent a connected subgraph is to represent a node as a unique label (if there is more than one pointer to the node), followed by a tag representing the node’s type, followed by the values of its attributes in a fixed order. We represent a pointer-valued attribute as the representation of the node it points to; if a node is shared, all values but the first use a reference to the node’s label. In the delta file, we represent any node that occurs in the original input by a reference to a suitable label for the original node.

With certain assumptions, we can reduce the directives to just the sequence of values V2, V2, and so on. The main such assumption is that a delta writer and delta reader can traverse the nodes of the original graph in the same order. With a suitable language for defining the graph type schemas, it is possible to declare that no instances of certain attributes would ever change. If a suitably high fraction of instances of changeable attributes really change, we can pretend that all the instances of such attributes change without incurring much overhead by writing the unchanged attributes. Thus the sequence of node identifications would simply be the list of changeable nodes in canonical order; for each node, the attribute identifications would be all the changeable attributes in canonical order. Thus we could omit these identifications, leaving only the sequence of values. Section 2.7 discusses what happens if we cannot afford to treat all changeable attributes as really changed.

Ensuring a canonical order is straightforward. We could ensure that each graph has a unique root node from which we can reach all the others, and traverse the attributes of each node in a canonical order (such as alphabetically by name of attribute). For programming environment applications, where graphs represent program abstract syntax trees, flow graphs, and so on, the requirement for such a root is not restrictive. Given the way we handle changes to pointer-valued attributes (see Section 2.8), we can ensure

that the delta writer and delta reader traverse the original data structure in the same order.

If the root of the new data structure might be different from the root of the old, the delta file needs to begin with a representation of the new root node. This creates no new difficulties; the rest of the paper assumes the new and old roots are the same.

2.2. Deleting Attributes

The simplest way for the output of a process to be different from its input is for the process to eliminate certain attributes from the graph's type schema. An easy implementation is to eliminate procedures for manipulating the ''deleted'' attributes from the application interface; the application cannot tell the difference between this implementation and one where the attributes are absent from the internal representation, except for resource constraints.

A slightly more complex Reader 2 could avoid storing a representation for omitted attributes. For reasons discussed in later sections, it may need to keep representations of some pointer-valued attributes.

2.3. Adding Simple Attributes

A simple attribute is one whose representation requires no pointers. Typical simple attributes are those whose values are integers, strings, booleans, and rationals (which include all typical fixed point or floating point numbers). When the internal representation has attributes missing from the input, the reader ignores them and lets the application fill in values later.

The scheme for writing a minimal representation of Graph 2 depends on Property 2: the DDL has some way to distinguish what attributes of nodes in Graph 2 are absent in Graph 1. Furthermore, because the package generator creates the readers and writers, it knows what order the writers will store information in the files, and thus in what order the readers will fetch it. Writer 2 can traverse the internal data in the canonical order. Whenever it reaches a type of node that has new attributes, it writes the values of the attributes in a fixed order to Data 2. Similarly, Reader 2 reads information from Graph 1, then traverses it in the same order as Writer 2. Whenever Reader 2 finds a type of node that has new attributes, it reads the appropriate number of values from Data 2 and stores them in the space reserved for them in Program 2's internal data.

2.4. Adding Pointers to New Data Types

The simplest pointers to deal with are those that reference new data not present in the original input file, because such pointers cannot interfere with the pointers we use to ensure the readers and writers traverse the original data in the same order.

Given the external representation for subgraphs mentioned in Section 2.1, the writer follows almost the same procedure as for simple attributes. It walks over the original data, emitting values of newly-added attributes in a fixed order. Figure 5 shows a sample internal data structure with which we will illustrate how this works. Original data are to the left of the dotted line; new data are to the right. Writer 1 makes a pass over the data and discovers that nodes B and D are shared. It then starts at node 1, which contains one attribute that points to an object of a new type. It therefore writes a representation of node B, which involves a label for B, the representation of B, the representation of C, a label for D, the representation of D, a reference to the label for B, and finally the representation of E. This constitutes the first unit of information in Data 2. The writer continues over nodes 2, 3, and 4. For node 4 it writes a reference to label D; this is the second unit of information in Data 2. For node 5 it writes the representation of node A, with a reference to label B. This is the third unit of information in Data 2.

Reader 2 can build a corresponding internal data structure. It opens Graph 2 and finds the reference to Graph 1. As it reads node 1, it discovers that its type requires reading a unit of information from Data 2. It reads the first unit, creating internal representations of nodes B, C, D, and E and recording labels for B and D. It reads nodes 2 and 3 from Graph 1. On reading node 4, it discovers it must read the next unit of information from Data 2, which is the label reference for D; it fetches the corresponding pointer by searching its label table. On reading node 5, it discovers it must read the last unit of information from Data 2,

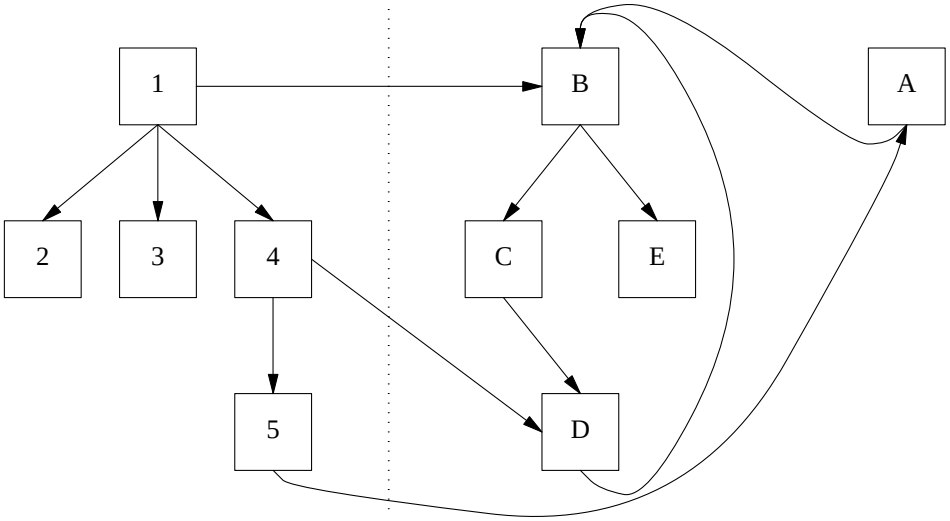

Figure 5: Handling Pointers to New Data

which is the representation of A; it also fills in the reference to B by searching its label table.

2.5. Allowing Back-pointers

At first glance, allowing back-pointers from new data types to old ones is simple. Imagine that in Figure 5 we add a back-pointer from node E to node 4. While writing out the representation of node E, Writer 1 emits a reference to a label for node 4. When reading in this representation, Reader 2 fills in the pointer from its label table.

Unfortunately, node 4 might not have had a label in Graph 1, since node 1 might have had the only reference, and labeling all nodes might cost too much in storage space and reader time. Furthermore, since we presume we cannot modify Graph 1, we cannot add a label to it. Fortunately, since readers and writers can traverse nodes from Graph 1 in the same order, we can use the position of node 4 in canonical order as a label; this may require a second type of label reference. If we can afford the space within Program 1, we can make Writer 1's job easier by adding an integer-valued attribute to each node to record its position. Otherwise, we can add a pass to Writer 2 between the one that detects sharing and the one that writes out Data 2. This pass traverses the original data, keeping track of what position each node would have come from. Whenever it finds an unlabeled that now needs a label, it can create a label, write the label and input position to Data 2, and record the label and node address in a hash table. When the output pass encounters a reference to an unlabelled node, it fetches the label from the hash table.

2.6. New Instances of Old Data Types

Given the mechanisms of Sections 2.4 and 2.5, it is easy to allow Graph 2 to contain new instances of old data types from Graph 1. By property 5, we can distinguish nodes Reader 1 creates from nodes the application creates. When Writer 1 encounters a pointer to an old data type, it tests whether the node is new. It handles old nodes as before (writing out only new attributes), but writes out newly-created nodes as if they were new types of nodes as in Section 2.4.

2.7. Changing Old Simple Attributes

If the schema says that Program 1 might modify certain simple attributes from Graph 1, then Writer 1 can do one of two things. If the application will likely change most nodes with the given attributes, the writer can treat them as if they were new attributes, and write them all out as in Section 2.3. If only a small fraction of the changeable attributes really will change, then the internal representation should flag attributes that really change. Writer 1 writes out only flagged attributes, labeled (as discussed later) with what node they came from; Reader 2 reads the original data from Graph 1, then overwrites the attributes of those particular nodes mentioned in Data 2.

There are several methods for flagging what attributes change.

1. For each changeable attribute, add a corresponding boolean flag in the same node. Storing the attribute via the reader/writer interface sets the flag to false; storing via the application interface sets it to true. The writer uses the reader/writer interface to check the value of the flag.
2. Store a second attribute representing the original value of the changeable attribute. The writer compares the two to detect changes. This has the benefit of eliminating from the delta those attributes whose values changed, then changed back to their original values.
3. To minimise space, use a table hashed by node address to represent the new value of the attribute. If the node address is in the hash table, the value in the hash table is the changed attribute value; otherwise the value stored in the node is the attribute value.

When Reader 2 reads a node from Graph 1 that contains changeable attributes, it must be able to tell whether the next value in Data 2 is a value for the changeable attribute. There are several possible schemes for doing so.

1. The delta file might contain entries of the form

   node identification
   values for always-changed attributes
   attribute name/value pairs for sometimes-changed attributes.

   When Reader 2 is trying to read a sometimes-changed attribute, the next item in Data 2 must be either a node identification or an attribute name; it must be possible to distinguish the two. This requires node identifications even for nodes with only always-changed attributes.
2. For nodes with sometimes-changed attributes, the delta file would contain entries of the form

   values for always-changed attributes of this node
   attribute name/value pairs for sometimes-changed attributes of this node
   end-of-node marker

   This eliminates the need for node identifications, but forces an end marker for all nodes with sometimes-changed attributes. If the end marker is no larger than a node identification, this requires no more space than the previous scheme, and requires less space if the graph contains some nodes with always-changed attributes and no sometimes-changed attributes, since they would need no end markers. If all sometimes-changed attributes of a particular node instance really changed, we could eliminate the end marker for that node (at a cost of slightly more complex processing in the reader and writer).

2.8. Changing Old Pointers

To coordinate Reader 1, Writer 1, and Reader 2, all three had to be able to walk over the data structure in the same way. If you change a pointer-valued attribute, this would not be possible. However, you can easily apply some method from Section 2.7 to keep both the old and the new value of pointer-valued attributes. The writer traverses the structure using the old value, and writes out the new value using the methods of Sections 2.4, 2.5, and 2.6.

You need not duplicate all pointer-valued attributes. The DDL might have facilities for declaring that some pointer-valued attributes are *structural*; that is, you can reach all nodes in the structure by following

structural attributes. Other attributes would simply be cross-references. The readers and writers could limit themselves to traversing only the structural attributes, allowing you to use the overwrite-and-set-flag method for the others.

3. Discussion

The method of Section 2 depends on coordinating readers and writers of separate tools. It thus implies that designers of individual tools must use the same DDL and compatible generators of readers, writers, and interfaces. However, it does not require that designers write tools in the same language, or that tools operate on the same machine, or that they have the same internal representation of their data. In prior work I have shown how to use IDL to avoid such constraints in the simple case of programs that write full graphs [5, 6].

3.1. Efficiency

Section 2 shows that detecting and recording graph deltas is possible. The following rough analysis argues that it should often be practical. We need to compare the efficiency of reading and writing graph deltas with that of reading and writing full graphs.

First, the graph delta clearly requires less disk storage space. Section 2 showed that a delta file requires a small amount of overhead to reference the original input file, followed by representations of exactly those attributes that changed. Writing out such a file is more efficient than writing out a full graph. The writer must traverse the internal data in the same way it would traverse the full graph, performing almost the same operations on each node, except that it does not write out any attributes or nodes already present in the original input. The only extra work the writer must do is to create labels for old nodes that were unshared in the input but become shared in the output; creating such labels may be somewhat more expensive than creating labels for new nodes.

Recording what attributes change requires some additional internal overhead.

- It must be possible to distinguish reader-created nodes from those the application creates. This may require an overhead bit per node, or a more complex storage manager that allocates the two kinds of nodes in different areas of memory.
- The internal representation must keep the reader's label table so the writer can use it. When writing full graphs, the reader could discard this table and let the writer regenerate it as necessary.
- At first glance it appears that the internal representation must waste space for old, unused attributes from the file to which the reader applies the deltas. However, since a reader can do some simple mappings from the disk representation to an internal representation, the only unused attributes it needs to keep in memory are the structural ones (the ones that guide reader and writer traversals). Thus tools that avoid transforming the underlying tree structure need not incur any internal space overhead.

Thus the only real overheads appear to be in the reader. We can ignore the time taken to read information from the delta file, because the reader uses the same methods to read items from this file that it would to read them from a full input file. A reader must open a second input file (which is a slow operation on some systems), and must read some unnecessary attributes (those deleted or changed by the delta). This costs extra input time, and may cause extra work in the storage allocator for pointer-valued attributes that reference discarded subgraphs. However, in a well-engineered system both of these overheads are proportional to the amount of deleted information.

In earlier work on graph readers and writers [6], I showed that:

- The time taken by a simple graph reader was linear in the size of the graph and the number of labels if the label table uses an O(1) method such as hashing.

- Several kinds of graph readers were about 2-3 times slower than the corresponding kinds of graph writers.
- The cost of traversing a tree in memory is much smaller than the cost of reading or writing the tree.

Suppose the size of an input graph is s(shared) + s(deleted) and a full output graph is s(shared) + s(new), the cost of reading a unit of input with a full graph reader is c(full), and the cost of reading a unit of input from a delta file is c(delta); c(delta) > c(full) because of the somewhat more complex processing. Then the cost of reading the output graph in another program is c(full) * (s(shared) + s(new)) with a full graph reader, and c(full) * (s(shared) + s(deleted)) + c(delta) * s(new) with the delta reader. Thus the extra processing for a delta reader is c(full) * s(deleted) + (c(delta) – c(full)) * s(new). As expected, this overhead would be low if the fraction of deleted information is low, and either the amount of new information is low or c(delta) is close to c(full) (as it would be in our optimised representation where we can omit node identification and attribute name information). These costs are offset somewhat by the saving in the writer; they are not completely offset because the savings in the writer are smaller (because writers are faster than readers), and because in many applications a data structure is written once and read several times.

Other researchers (for example, Tichy [12]) have proposed methods for finding efficient string-to-string deltas. It might be possible to apply such methods to the linear representation of a graph as an alternative to the methods I propose. The natural operations to use on string transformations are block copies from the source to the destination, and adding entirely new strings. It seems likely that it would take considerable effort (comparable to that required to design and implement the methods of this paper) to design an efficient way to turn such string operations into changes to an internal graph structure.

Full validation of practicality requires building and measuring a prototype system. A reasonable experiment requires three test systems: one to produce an original graph, one to produce a delta, and one to read the delta. For example, a parser might emit an abstract syntax tree, a semantic analyser might annotate it, and a metrics tool might read the annotated graph. I expect to begin such work for IDL in 1990; I hope other researchers with similar DDLs will apply these methods to their own systems.

3.2. Instance Changes

I phrased most of the earlier sections as though the input and output of a tool had to have different schemas. Such a restriction is unnecessary; the techniques allow you to store graph deltas representing the edit history of a graph, similar to the way tools like SCCS[8] and RCS[11] store source deltas, or the way database systems store differential files [9]. John Nestor and Joseph Newcomer told me that they used most of these methods in the GNAL separate compilation mechanism at Tartan Laboratories. One could view this work as formalising their collection of ad-hoc techniques (‘‘hacks’’).

At first glance, it might appear that the techniques of this paper are unnecessarily complex for representing changes to a graph instance where the type structure does not change. However, in such a structure, the only operations of the general scheme we cannot do are

- delete attributes from the schema,
- add new attributes to the schema, and
- create instances of new types of nodes not present in the input.

All the other types of change are still possible. The most complex techniques keep track of changing attributes and newly-created nodes; this still necessary for changing graph instances. Thus all the mechanisms of this paper are still necessary.

3.3. Other Issues

Using graph deltas to implement a programming support library requires additional library management tools beyond the scope of this paper. For example, the library would need some form of garbage collection to eliminate unreferenced old files. If you have a chain of deltas where earlier points in the chain do not change frequently, you might wish to collapse a series of deltas into a single one; you could build a

collapsing tool from the automatically-generated readers and writers with a trivial application program to glue them together. Indeed, given any sequence of graphs $G_0, \cdots, G_k$ represented by storing G_0 and a sequence of deltas $D_1, \cdots, D_k$, it is easy to select several indices $I_1, \cdots, I_m$ ($m \leq k$) and collapse all the deltas to a sequence of m changes that take us from G_0 to G_{I_1}, from G_{I_1} to G_{I_2}, and so on. One reads in G_0, applies all the deltas up to D_{I_1}, then writes out the difference between the original G_0 and the result. Then one marks the current internal representation of G_{I_1} as ''old'', and applies D_{1+I_1} through D_{I_2}, and repeats the process.

If the data definition language allows sequence-valued attributes, the scheme will work without change if we treat any change to a sequence as changing the value of the entire sequence. A naive representation of inserting a new node at the beginning of a sequence of nodes would be a representation of the new node, followed by references to the old nodes (which would still exist in the original input file). It might be possible to do better by retaining both the old and new value of the sequence and applying a variant of a string-to-string correction algorithm.

When input and output schemas are different, the length of the chain of references from a delta file back to an original graph tend to be short. The limit is the maximum number of tools between an original source file and final output. Once you start storing deltas of graphs with the same structure, there are no *a priori* limits on the lengths of chains. Users of source deltas find that it takes a long time to work through the entire chain to get the most recent version of a source program; similar problems might show up in a graph delta system. It is technically easy to modify the techniques of Section 2 to give *backwards deltas* (that is, deltas that show how to change the output graph into the input graph). You merely keep two versions of each changeable attribute, and chose a different set of attributes to write out. For example, if a program deletes an attribute from its input to produce its output, the writer would emit a full graph for its output and replace its input by a delta that showed what information to add to the output to recreate the original input. This has two inherent disadvantages. First, you must write an entire graph plus a delta, which costs extra time. Second, in many programming environments you could not use backward deltas to represent changes between different structures, because the (compiled) readers of earlier tools would not necessarily understand the format of the new files. A third disadvantage in some circumstances is that it presumes that one input leads to only one output, so that there is always a unique output file to which to apply the backward delta to regenerate the input; however, it should be possible to extend these techniques to represent an output file as a delta of a set of input files, rather than just one.

3.4. Acknowledgements

I presented a talk on this material at the meeting of IFIP Working Group 2.4 in Niagara-on-the-Lake, Ontario, Canada in May, 1987. Several members and observers gave helpful feedback. Margaret Lamb, David Parnas, Richard Snodgrass, and the anonymous referee made helpful comments on earlier versions of this paper. The Natural Sciences and Engineering Research Council of Canada (NSERC) supported my work under grants A0908 and OPG0000908.

References

1. Vikram Biyani, ''An Efficient Runtime System for IDL,'' Technical Report TR87-029 (Master's Thesis), Computer Science Department, University of North Carolina (Chapel Hill) (October 1987).
2. Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco (1978).
3. G. Goos, W. A. Wulf, A. Evans, and K. J. Butler, *DIANA: An Intermediate Language for Ada*. Springer-Verlag (1983). Lecture Notes in Computer Science #161.
4. Peter Henderson, *Proceedings of the ACM SIGPLAN/SIGSOFT Software Engineering Symposium on Practical Software Development Environments*. Association for Computing Machinery (January 1987). SIGPLAN Notices 22(1)

5. David Alex Lamb, *Sharing Intermediate Representations: The Interface Description Language*, Ph.D. dissertation, CMU-CS-83-129, Computer Science Department, Carnegie-Mellon University (May 1983).
6. David Alex Lamb, ‘‘IDL: Sharing Intermediate Representations,’’ *ACM Transactions on Programming Languages and Systems* **9**(3):267-318 (July 1987).
7. John R. Nestor, Joseph M. Newcomer, Paola Giannini, and Donald Stone, *IDL: The Language and its Implementation*. Prentice-Hall, Inc. (1990).
8. M. J. Rochkind, ‘‘The Source Code Control System,’’ *IEEE Transactions on Software Engineering* **SE-1**(4) (December 1975).
9. Dennis G. Severance and Guy M. Lohman, ‘‘Differential Files: Their Application to the Maintenance of Large Databases,’’ *ACM Transactions on Database Systems* **1**(3):256-267 (September 1976).
10. Richard Snodgrass, *The Interface Description Language: Definition and Use*. Computer Science Press, Rockville, MD (1989).
11. Walter F. Tichy, ‘‘Design, Implementation, and Evaluation of a Revision Control System,’’ in *Proceedings of the 6th International Conference on Software Engineering* (September 1982).
12. Walter F. Tichy, ‘‘The String-to-String Correction Problem with Block Moves,’’ *ACM Transactions on Computer Systems* **2**(4) (November 1984).