PGRAPHITE: An Experiment in Persistent Typed Object Management

Jack C. Wileden[†]
Alexander L. Wolf[‡]
Charles D. Fisher[†]
Peri L. Tarr[†]

COINS Technical Report 88–63
August 1988

[†]*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[‡]AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

A version of this report to appear in
Proceedings SIGSOFT '88: Third Symposium on
Software Development Environments

This work was supported at the University of Massachusetts in part by the following grants: National Science Foundation DCR–84–04217 and DCR–85–00332; National Science Foundation CCR–87–04478 with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104); and Rome Air Development Center F30602–86–C–0006. C. Fisher was also supported in part by a General Electric Foundation Graduate Fellowship.

ABSTRACT

Defining, creating, and manipulating *persistent typed objects* will be central activities in future software environments. PGRAPHITE is a working prototype through which we are exploring the requirements for the persistent object capability of an object management system in the Arcadia software environment.

PGRAPHITE represents both a set of abstractions that define a model for dealing with persistent objects in an environment and a set of implementation strategies for realizing that model. PGRAPHITE currently provides a type definition mechanism for one important class of types, namely directed graphs, and the automatic generation of Ada implementations for the defined types, including their persistence capabilities.

We present PGRAPHITE, describe and motivate its model of persistence, outline the implementation strategies that it embodies, and discuss some of our experience with the current version of the system.

1 Introduction

The Arcadia project is a collaborative software environment research program encompassing groups at several universities and industrial organizations [12, 13]. The objective of Arcadia is to develop advanced software environment technology and to demonstrate the technology through prototype environments.

One important component of Arcadia environments will be an *object management* system appropriate for handling the numerous and diverse kinds of information that will be manipulated in a software environment. We are studying two aspects of object management. One of our projects, described in a companion paper [11], is investigating abstract typing mechanisms suitable for environment object management. The other, described here, addresses persistence of typed objects in an environment. By persistence, we mean the preservation of an object beyond the execution of a tool that creates or manipulates the object.

Our investigation of persistence addresses two major topics. One of these is *models* of persistence. We wish to identify an appropriate set of abstractions through which environment designers, tool builders, and tool users can manipulate and reliably store persistent objects in a setting of possibly concurrent and distributed accesses. The other major topic is *implementation strategies* for a persistence mechanism in an object management system.

Experimentation with a prototype persistent object system provides an excellent means for investigating both of these topics. Although a few examples of persistent object systems exist (e.g., [1, 5, 6, 14, 16]), most are not particularly oriented toward supporting software environments nor are they suitable for use with our predominant implementation language, Ada. Thus, none seemed appropriate as a vehicle for our experimentation with potential persistence models and implementation strategies for use in an Arcadia object manager.

In this paper we describe PGRAPHITE, our currently operational prototype of a persistent object system. We first provide a bit of the background that motivated some of our design decisions. We then describe the set of abstractions that PGRAPHITE provides to the environment designers, tool builders, and tool users who might wish to manipulate persistent objects. Next, we outline the implementation strategies employed in our current version of PGRAPHITE. Finally, we briefly relate our experiences with using PGRAPHITE and some plans for future work.

2 Background

One of our major goals for Arcadia software environments is to integrate persistence with a suitably general typing mechanism (e.g., [11]). For purposes of our PGRAPHITE prototype, however, we have limited ourselves to automated support for only one class of types, namely directed-graph types. We feel that this limitation will not prove to be particularly severe for two reasons. First, graph objects, such as abstract syntax trees, control flow graphs, and data flow graphs, are very common in software environments. Moreover, graph types are among the most general and complex of types. Hence, a prototype providing automated support only for persistent graph types is immediately useful, can be widely exercised, and can result in experiences and insights regarding most, if not all, of the important issues related to persistence. Second, while PGRAPHITE only automates support for persistence of graph types, our abstractions of persistence, as well as our implementation strategies, are general-purpose designs applicable to persistent objects of arbitrary types. Indeed, we have already manually applied them to a number of non-graph abstract types.

In our view of persistent objects in environments, an important consideration is that persistence should be as *orthogonal* and *transparent* as possible. By orthogonal persistence, we mean that the persistence property of a type should be essentially independent of other properties of that type [1]. In fact, persistence should be a property of instances, not types, such that there is no restriction on what kinds of types can have persistent instances and any given type may have instances that persist and instances that do not persist. By transparent persistence, we mean that tools should not need to be aware of the mapping to, or the existence of, stable storage representations of persistent object instances. That is, from the tool's perspective, all objects of a given type should have the same appearance (i.e., interface) even though some may be persistent instances and others not. An object's persistence property should be something that a tool may choose to pay attention to or choose to ignore.

One could also consider trying to achieve what we term *implicit* persistence, where all aspects of persistence are completely hidden from tool developers. With implicit persistence, an object would persist for as long as the underlying persistence mechanism believed that there was a means to refer to that object: it is the simple existence of that reference that causes an object to persist and no explicit action performed by a tool is required. Some researchers have begun to pursue

this kind of persistence and it is probably the ultimate goal of research in this area. But there are some significant problems with this approach, such as the need to have one name space within which all objects must exist, the need to have a general-purpose mechanism that is so sophisticated that it can anticipate the requirements of arbitrarily complex situations, and the simple fact that programmers often want at least some control over persistence. We believe that by trying to maximize orthogonality and transparency we can achieve a reasonable approximation to the goal of implicit persistence, one that will satisfy the needs of environment developers for some time.

At a more pragmatic level, an important goal for our design was that it facilitate relatively easy experimentation with a variety of underlying support systems, such as storage managers, concurrency control systems, transaction managers, and the like. In fact, we hope to take advantage of as much existing technology as possible. At first glance this may seem to place the burden solely on the implementation strategies. But although the burden is indeed heaviest there, this goal has a significant effect on the high-level abstractions, as will become clear in coming sections.

Conveniently, we had previously implemented a system, called GRAPHITE (GRAPH Interface Tool for Environments), for automatically generating Ada implementations of directed-graph types from declarative type descriptions given in a language called GDL [3]. Thus, an obvious approach was to attempt to add persistence to the graph type implementation produced by GRAPHITE. The result was PGRAPHITE, whose properties are described in the remainder of this paper.

3 Basic Abstractions

As outlined above, PGRAPHITE provides a collection of abstractions. Viewed broadly, there are three such abstractions: one to represent persistent objects, another to represent the store for persistent objects, and a third to represent graphs as they are "normally" manipulated by tools. In this section we describe each of the abstractions and the operations associated with them. The operations are summarized in Table 1. We then illustrate how the abstractions can be used in concert by tools and, finally, discuss some of the benefits of our design. In the next section we describe how these high-level abstractions are actually realized in Ada.

PERSISTENT-OBJECT ABSTRACTION	
Get PID	retrieves a persistent identifier for a given object
Get NPR	retrieves a non-persistent reference to a given persistent object

(a)

PERSISTENT-STORE ABSTRACTION	
Create	creates a given repository
Delete	deletes a given repository
Open	opens a given repository
Close	closes a given repository
Begin Session	begins a session in a given repository
End Session	ends a session in a given repository

(b)

GRAPH ABSTRACTION	
OPERATIONS TO MANIPULATE A NODE	
Create	creates a new node of a given kind
Get Attribute	retrieves the current value of an attribute of a given type
Put Attribute	sets the value of an attribute of a given type
OPERATIONS TO ASCERTAIN A NODE'S DEFINITION	
Attribute Type	retrieves the name of an attribute's type
Kind	retrieves the name of a node's kind
Node Kind Attributes	retrieves the names of a node kind's attributes
OPERATIONS TO MANIPULATE COLLECTIONS	
Create	creates a given collection
Insert	inserts a node into a collection
Remove	removes a node from a collection
Retrieve	retrieves a node from a collection
Size	retrieves the size of a collection

(c)

Table 1: Basic Abstractions and their Associated Operations.

3.1 Persistent Objects

In striving toward the goal of orthogonal and transparent persistence, we wish to minimize—at least from the perspective of how tools manipulate objects—the differences between objects that persist and objects that do not persist. We have found that we can confine the exposure of those differences to just two situations. The first situation occurs when a tool needs to indicate that a particular object is to persist. The other situation occurs when a tool needs to preserve a reference to an object that is persistent, most likely by placing that reference into some other persistent object. In fact, it seems to us that the first situation occurs only within the context of the second. That is, an object should persist only if and when there is a desire to refer to that object at some, perhaps indeterminate, time in the future. Thus, we need only be concerned about the second situation.

Every language provides some way of referring to instances of types. But in most languages, this mechanism turns out to be somewhat restrictive. In particular, the validity of such references is typically not guaranteed outside of a single program execution. Moreover, the references form a name space that is normally not controllable by anything other than the run-time system of the programming language. There are sound reasons for this restrictiveness, mostly concerning efficiency. In order to achieve a name space of object references that is valid within and between separate program executions, one must gain control over the name space in one of two ways: by modifying the run-time system of the language or by custom-building a name space on top of that which is already provided by the language. The complexity and threat to portability of the former are prohibitive in most settings, while the latter usually results in a severe efficiency loss due to the unavoidable translations that then must occur between the custom-built name space and the language's primitive name space upon each and every object access.

In our effort to develop models of persistence, we have refused to assume that we can have such a "single" name space, either by modifying the run-time system of the language or by building our own mechanism on top. This is in contrast to most, if not all, the approaches that have been taken by other researchers in this area.

The result is that our model of persistence must admit to two, *side-by-side* name spaces and translations between them. Fortunately, this translation process can be made to occur quite infre-

quently, as the discussion below makes evident. One of the name spaces is made up of the "normal" references to objects provided by the run-time system of a language. We call such references *non-persistent references* (NPRs), since they themselves cannot be preserved. The other name space is used to refer to persistent objects in a more "enduring" way. We call references in this name space *persistent identifiers* (PIDs).

It is important to point out that NPRs can be used to refer to both persistent and non-persistent objects. In fact, this is precisely the point of having an orthogonal and transparent persistence mechanism. All "normal" operations associated with a type, such as those of the graph abstraction, can be based on the use of NPRs. PIDs are necessary only when references need to persist. Moreover, much of the manipulation of those PIDs is indeed hidden, as we demonstrate below.

There are actually only two operations associated with the persistent-object abstraction (Table 1a). One is used to retrieve a PID for an object, given an NPR. It has the appropriate side effect of indicating that the object is to persist. The other operation is used to retrieve an NPR to a persistent object, given a PID. The abstraction maintains the mapping between PIDs and NPRs, and thus equivalent results are obtained by two successive retrievals of a PID for a given object. This addresses the well-known and difficult problem of preserving shared references.

3.2 Repositories

The store for persistent objects is modeled as a collection of disjoint *repositories*. A repository can be thought of as the second name space described above, within which the names of persistent objects are guaranteed to be unique and unchanging. Although the name of an object cannot change, the value of that object can be modified. In other words, objects in repositories are mutable. This permits the modeling of both mutable and immutable stores, since immutability can be achieved by building an appropriate interface on top of the mutable store, one that does not permit modification of objects in a repository.

Repositories are disjoint in the sense that the semantics of references between repositories (i.e., references from objects in one repository to objects in another repository) are not within the purview of the model. Again, an appropriate higher-level interface can be used to achieve the missing functionality, in this case providing a mechanism for inter-repository references. There is

little concensus on what that mechanism should be and so choosing one arbitrarily for our model would have been premature. Adopting disjoint repositories allows us to experiment with a variety of such mechanisms.

The operations associated with the persistent-store abstraction include those to create, delete, and gain access to a repository (Table 1b). We have distinguised two levels of access to a repository; tools must pass through both these levels before they can actually manipulate persistent objects. The first level is delineated by the open and close operations, which broadly indicate a period of time during which access is desired, much like the open and close operations associated with files in a file system. Within that period of time, a finer granularity of access to a repository must be indicated using operations to begin and end a *session*. Several, possibly concurrent, sessions may occur during the time between the opening and closing of a repository. The advantage of this two-level granularity is that opening and closing might involve time-consuming actions, such as establishing and breaking network connections, whose costs could then be incurred less often.

The session is a notion concerned with issues of concurrency and reliability. While the exact semantics of sessions is dependent upon the underlying transaction management system, the basic intent is that tools not be permitted to manipulate persistent objects except within the context of a session; that context is used by the underlying system to guarantee the consistency of repositories. For example, the two operations associated with the persistent-object abstraction are made available only during a session in order to help guarantee consistency. In a sense, the operations to begin and end sessions are "hooks" for transaction management capabilities that allow us to experiment with a number of such systems.

3.3 Graphs

Although the previous two abstractions are quite independent of any particular object type, the prototype that we have built is primarily concerned with graph objects and so it provides an abstraction for manipulating them. Graphs are modeled as collections of *nodes*, where each node is an instance of some *node kind*. A node has zero or more *attributes*, which are determined by the node's kind. Attributes are used to describe the properties of the entities represented by the nodes, and each such attribute has a type, referred to here as an *attribute type*. Some of the attribute types actually represent references to nodes, which makes it possible to connect nodes

```
node Triangle is
   Number : Integer:
   Left   : Triangle:
   Right  : Triangle:
end node:

node Square is
   Name   : NameType:
   First  : Triangle:
   Second : Triangle:
end node:

node Circle is
   Name : NameType:
   Data : Triangle:
end node:
```

Figure 1: Examples of Node-Kind Definitions, Written in GDL.

into directed graph structures. Figure 1 presents three simple node-kind definitions written in GDL, the specification language of PGRAPHITE (and GRAPHITE). Examples of nodes that might be built according to those definitions appear in Figure 2.

The graph abstraction supports two different ways of thinking about directed graphs. The first, most general way treats a graph simply as a collection of nodes that may or may not be connected and may or may not have a root. The nodes of a graph must explicitly be placed into the collection representing that graph. A common graph structure found in software environments, however, is the rooted directed graph, which is used for such things as parse trees and call graphs. Often one would like to think of such a graph, not as an arbitrary collection of nodes, but rather as the reachability set of a root node. The abstraction described here also supports this second view, as well as supporting combinations of the two views. Such combinations would permit the straightforward representation of, for example, a forest of trees.

The operations associated with this abstraction (Table 1c) include those to create a node and to put and get an attribute value. There are also operations to ascertain information about a node's definition, such as the kind of the node and the attributes associated with the node. Finally, there are operations to create a collection of nodes, as well as add to and remove from that collection. These are exactly the same set of operations provided by GRAPHITE, which points up

the orthogonality and transparency of the persistence mechanism provided by PGRAPHITE.

It should be evident that there are really two types associated with this abstraction: one is for representing nodes and the other is for representing collections of nodes. PGRAPHITE fully supports the persistence of both types, since it allows instances of either type to be persistent objects. In other words, PGRAPHITE supports the association of PIDs with either an individual node or a collection of nodes.

3.4 Using the Abstractions

Figure 2 suggests how two tools might use the three basic abstractions supported by PGRAPHITE. This is a very simple example, one that is certainly not comprehensive and probably not typical, but it is sufficient to provide an overall impression of our approach to persistence.

In Figure 2a, we see that Tool 1 uses the graph abstraction to create a graph structure, in this case a non-rooted, connected graph evidently made up of several different kinds of nodes (cf. Figure 1). Although they are not all shown, the tool possesses NPRs with which it can refer to the nodes. The NPRs that are shown (dashed arrows) happen to be the ones that are values of attributes of the nodes.[1]

In Figure 2b, the persistent-store abstraction is used by Tool 1 to create and open a repository and then begin a session. Within that session, the tool uses the persistent-object abstraction to retrieve PIDs for two of the nodes, as shown in Figure 2c. We use shading in this figure to highlight the nodes for which PIDs have been retrieved. Presumably Tool 1 has retrieved the PIDs because it, or some other tool, has a need to refer to those nodes sometime later. Such requests for PIDs indicate the intention for the nodes to persist. Note that the two operations of the persistent-object abstraction, as mentioned above, can only be used during a session. On the other hand, the operations of the graph abstraction can be used at any time.

In Figure 2d, we see that Tool 1 ends the session. With the successful completion of the session, the nodes that were intended to persist now indeed do persist, within the safe confines of the repository. There are three important properties of our abstractions that are revealed by Figure 2d.

[1] Attributes that are not references to other nodes, such as **Name** of node kind **Triangle**, and attributes that are references to other nodes but whose values are "null", such as **Left** of node 3, are also not shown in the figure.

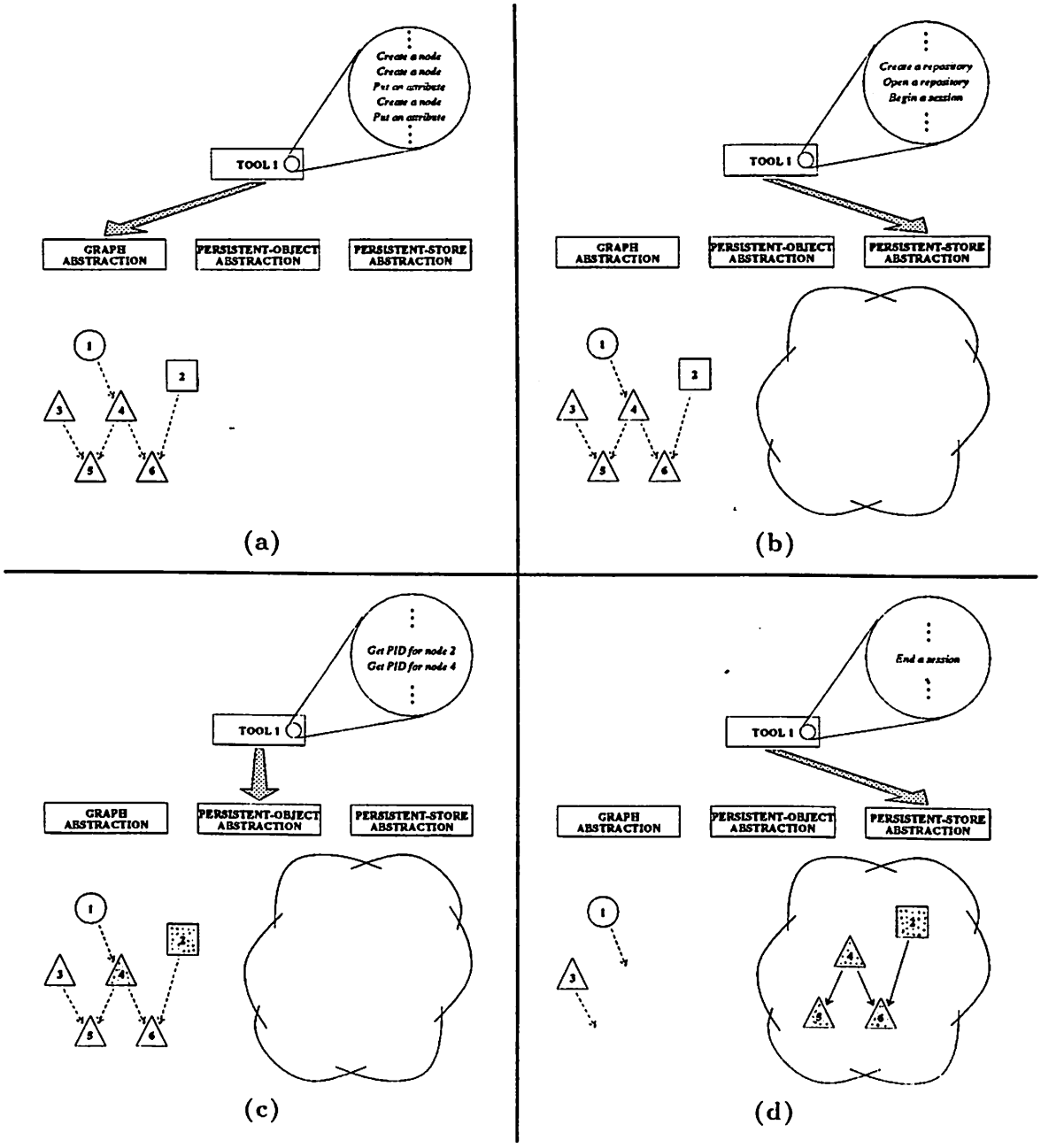

Figure 2: A Simple Illustration of the PGRAPHITE Abstractions.

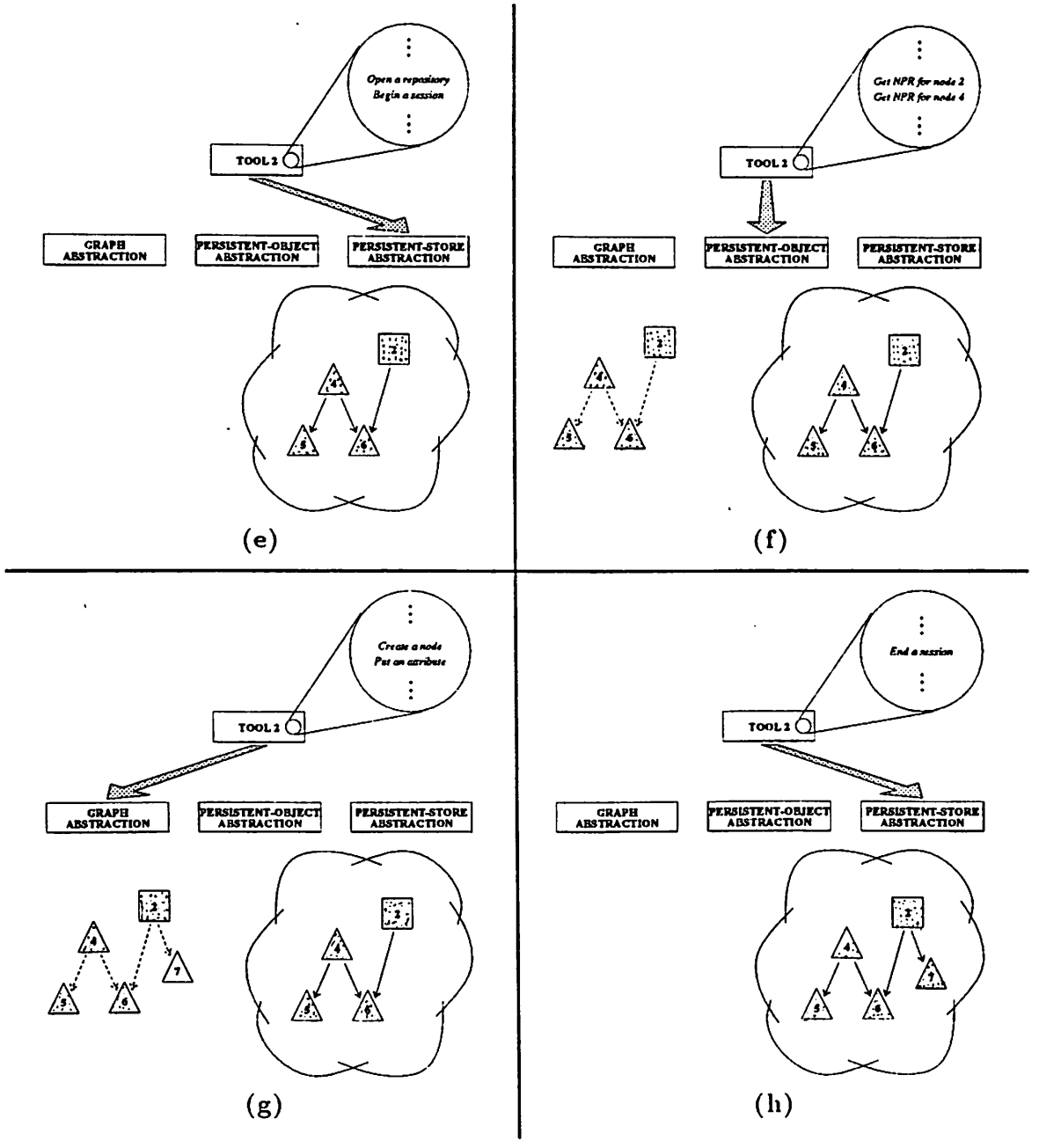

Figure 2: (continued).

First, notice that two additional nodes persist beyond those for which PIDs were (explicitly) retrieved in Figure 2c. They persist because they are *reachable* from nodes for which PIDs were retrieved. Our rationale for using reachability to (implicitly) make nodes persistent is that the meaning or value of a node includes the values of that node's attributes, and the value of an attribute whose type is a node kind is simply a node of that kind. More generally, the value of a composite object, of which nodes are an example, is derived from the values of its components.

Not surprisingly, this approach is very supportive of situations in which one views a graph as the reachability set of a node or as a collection of nodes and their reachability sets. For example, a tool need only retrieve a PID for the root of a parse tree in order to have that entire tree persist. It is arguably less convenient for the view based on explicit membership in a collection of nodes, especially when the user wishes to restrict persistence to just those member nodes, since the user would then have to appropriately "prune" the reachability set of the member nodes before ending the session. The significance of this argument is at least somewhat lessened if one accepts that superfluously persistent nodes could simply be ignored. Obviously this is another instance of the classic tradeoff between what is explicit and what is implicit. Our intuition, based on experience with developing environment tools, led us to this choice in PGRAPHITE. Only time and experience will tell whether our choice was the correct one, but we note that others have already taken a similar approach [1, 7, 15].

The second thing to notice from Figure 2d is that a transformation occurs to the inter-object references of persistent objects. In particular, the NPRs become PIDs (shown as solid arrows). One way to think of what occurs at the end of a session is to imagine that all objects for which a PID has been retrieved in turn retrieve PIDs for the objects to which they refer. It is then those PIDs that are used instead of the NPRs after the session has ended. This transformation is completely hidden (i.e., transparent), as is the inverse transformation that occurs when access to a persistent object is required, a process that we illustrate below.

The last thing to notice from Figure 2d is that the objects that were made to persist are no longer available through the graph abstraction. In particular, any attempt to access them, other than through the persistent-object abstraction, would result in an error.[2] The intent is to keep

[2]Although the picture in Figure 2d makes it look like we leave dangling references, our implementation in fact treats such references as special.

clear the fact that persistent objects can only be manipulated within the context of a session. The alternative would have been to introduce the notion of implicit non-persistent and persistent images, or copies, that somehow coexist. We prefer to have tools make explicit copies and to take responsiblity for the consistency (or, more likely, inconsistency) of those copies. Again, this is a design decision whose appropriateness can only be established with time and experience.

In Figure 2e, Tool 2 opens the repository and begins a session. The tool then retrieves NPRs for the two nodes that Tool 1 had indicated as persistent, using the PIDs associated with those nodes. (We assume that Tool 1 somehow provides the PIDs to Tool 2, perhaps through another persistent object. The exact details of this communication are irrelevant here.) This is shown in Figure 2f. There are three important things to observe from this part of the figure. First, retrieval of an NPR *logically* affords access to a persistent object and all objects reachable from that object. Any physical movement of data that might be implied by this accessibility, however, depends heavily on the implementation strategy. We explore this issue in Section 4. Second, notice that PIDs (solid arrows) used in the repository are transformed (back) into NPRs (dashed arrows). As mentioned above, this transformation is hidden. Finally, copies of the persistent nodes remain within the repository during a session.[3] Our reason for doing this is that we wish to support optimistic concurrency control mechanisms, which can provide greater availability than other concurrency control mechanisms. Although this appears to violate our ban against non-persistent and persistent copies, this is a very different situation, since the copy in this case is under the direct control of the underlying transaction management system. That is, the copy exists only within the context of a session, as opposed to between sessions.

In Figure 2g, Tool 2 uses the graph abstraction to create a new node and place that node into the graph structure. Notice that a persistent node has been modified in the process. Finally, Figure 2h shows that Tool 2 ends the session, with the state of the repository changed to reflect the presence of a new node, which persists only because it is reachable from an already persistent node.

[3] An equivalent view is that it is the copies that are withdrawn.

3.5 Discussion

There are several advantages to the abstractions that we have chosen for PGRAPHITE. For one, our abstractions for the persistent store and persistent objects are quite independent of the abstraction for graphs. In other words, they should, and indeed do, work with other types of objects. It just so happens that PGRAPHITE automates the implementation for graph types. We defer further discussion of this topic to the next section.

Another important advantage is that the notion of persistence is kept orthogonal and transparent to the "normal" manipulation of objects. In particular, objects can be treated the same whether or not they are persistent and the determination as to whether or not an object is to persist can be made at any time. The decision need not, for example, be made at the time the object is created. Moreover, the tool that creates an object need not be involved in the decision. A reasonable scenario might be to have a tool wrapped within a layer that is responsible for persistence. The tool would perform its work without having to be aware of the persistence (or non-persistence) of the objects it is manipulating. Indeed, a tool could have a number of different wrappings. Finally, translations between NPRs and PIDs and movements to and from the persistent store are implicit and automatic, with tools shielded from having to take any responsibility for, or have any knowledge of, their occurrence.

A third advantage of the abstractions, and one that is particularly important within the context of the Arcadia research goals, is that they facilitate experimentation. Specifically, they allow us to experiment with a variety of underlying storage managers, transaction managers, optimization techniques, and the like without requiring changes to the tools. There are two reasons for this. First, as just mentioned above, tools are not involved in storage management activities, which means that those activities can be modified (within reason) without affecting the tools. Second, our notion of a session is a simple, yet very powerful, abstraction that accommodates most current approaches to concurrency control and transaction management. We have therefore insulated tools from experimental changes. Unfortunately, it is still somewhat difficult for the current version of PGRAPHITE to itself be switched among alternative support systems. In Section 6 we describe our efforts at alleviating this problem.

There are several ways in which we could consider extending the functionality of the abstrac-

tions. Broadly speaking, these extensions would permit tools to exercise finer control over underlying support systems, such as storage managers and transaction managers. For example, we do not currently provide a way for tools to explicitly cluster related objects, which would perhaps give storage managers the opportunity to better optimize access. We also do not currently provide tools with the ability to commit changes to persistent objects without also ending a session. Care must be taken when considering such extensions, however, because they may conflict with our goal of producing a prototype that allows easy experimentation with a variety of underlying support systems. Our intent up to this point in the development of a prototype was to formulate and implement a basic set of capabilities with which we could gain some valuable initial experience. The next step is to consider whether and how to extend those capabilities.

4 Implementation Strategies

In this section we describe our approaches to implementing the functionality implied by the basic abstractions outlined above, concentrating particularly on the persistent-store and persistent-object abstractions. Those approaches have been, and continue to be, tested by our actual implementation of PGRAPHITE, which is already being used in several Arcadia projects. We begin by describing how the abstractions are realized in Ada, primarily in terms of how a tool developer would view the high-level modularization or architecture of the implementation. We then briefly describe our strategies for actually putting objects into, and getting objects out of, stable storage and for achieving various efficiency gains. These latter two strategies are meant to be hidden from users of PGRAPHITE, but they are of obvious importance to developers of persistence mechanisms.

It is important to reiterate a constraint under which we devised these implementation strategies, since it certainly had an effect on the outcome. That constraint, alluded to above, was our desire to use Ada *as is*. The approach being taken in many, if not all, other projects investigating persistence is either to modify an existing language or to create a new language so that the persistence capabilities become "primitive". We have chosen to implement persistence in our prototype as an application program written in an existing language. Ada, however, is a strongly-typed, statically type-checked language that does not make an object's type information available to application programs at run time. We have found, as have other researchers in this area, that a certain

amount of that information is necessary for building general-purpose mechanisms. Fortunately, since PGRAPHITE acts as a kind of preprocessor for Ada, we have the opportunity to capture the information and represent it for ourselves.

4.1 Realizing the Abstractions

As mentioned in the introduction, PGRAPHITE is an outgrowth of our earlier work on a system called GRAPHITE. GRAPHITE is a tool that takes a specification of a set of node kinds, written in the language GDL, and produces an Ada package that provides facilities for manipulating instances of those node kinds. Those facilities include a realization of the graph abstraction described in the previous section. We refer to the set of node kinds as a *class* and refer to the Ada package that effectively implements a class as an *interface package*. The interface package serves as a type-definition module, one that defines a type for references to nodes and a type for references to collections of nodes.

PGRAPHITE also takes a GDL specification and produces an interface package. But the interface package produced by PGRAPHITE incorporates a realization of the persistent-object abstraction as well as the graph abstraction (Figure 3). Figure 4 shows two GDL class declarations, containing skeletal definitions for the three node kinds of Figure 2. A skeleton of the specification part of the interface package for class C2 is shown in Figure 5. Notice that the name of the class is used as the name of the type for non-persistent references to nodes and is used in forming the name of the type for non-persistent references to collections of nodes. The type that actually implements nodes (not shown) is a union type of all the node kinds in the class. Also notice that in the current version of PGRAPHITE, collections of nodes are implemented using the somewhat less-general type *sequence*.[4] This is a temporary expedient that resulted from the fact that GRAPHITE-generated interface packages already provided such a type. Other aspects of the class declarations and of the interface package are explained below.

In general, tools might manipulate nodes from more than one class and therefore make use of more than one interface package. Moreover, there might be connections between nodes in different

[4]The collection operations described in Table 1 are realized by the same-named sequence operations provided by interface packages. The only exception is the collection operation "Size", which is realized by the sequence operation "Length".

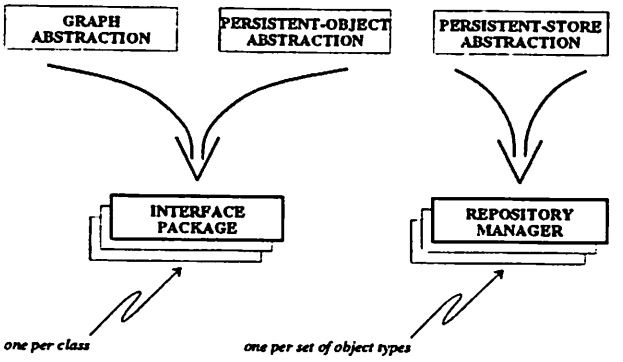

Figure 3: Mapping the Abstractions to Ada Packages.

```
class C1 is
   package C1Pack;
   . . .
   node Triangle is . . . end node;
end C1;

class C2 is
   package C2Pack;
   with C1Pack.( C1
                 /PID out => GetPID
                      in  => GetNPR );
   . . .
   node Square is . . . end node;
   node Circle  is . . . end node;
end C2;
```

Figure 4: Two GDL Class Declarations.

```
    -- imported types
        with C1Pack;
package C2Pack is
    -- utility types (NodeKindName, AttributeName, etc.)
        . . .
    -- graph abstraction
        type C2           is private;
        type C2Sequence is private;
        . . .
        function  Create ( TheNodeKind : NodeKindName ) return C2;
        procedure PutAttribute ( TheNode : C2; TheAttribute : AttributeName; TheValue : C2 );
        function  GetAttribute ( TheNode : C2; TheAttribute : AttributeName ) return C2;
        procedure PutAttribute ( TheNode : C2; TheAttribute : AttributeName; TheValue : C1Pack.C1 );
        function  GetAttribute ( TheNode : C2; TheAttribute : AttributeName ) return C1Pack.C1;
        . . .
        function Kind ( TheNode : C2 ) return NodeKindName;
        function AttributeValueType ( TheNodeKind : NodeKindName; TheAttribute : AttributeName )
                                    return AttributeValueTypeName;
        function NodeKindAttributes ( TheNodeKind : NodeKindName ) return AttributeNameList;
        function Kind ( TheSequence : C2Sequence ) return NodeSequenceName;
        . . .
        function  Create ( TheSequenceKind : NodeSequenceName ) return C2Sequence;
        procedure Insert ( TheSequence : C2Sequence; ThePosition : Positive; TheNode : C2 );
        procedure Remove ( TheSequence : C2Sequence; ThePosition : Positive );
        function  Retrieve ( TheSequence : C2Sequence; ThePosition : Positive ) return C2;
        function  Length ( TheSequence : C2Sequence ) return Natural;
        . . .
    -- persistent-object abstraction
        type PID is private;
        . . .
        procedure GetPID ( ThePID : out PID; TheNode : C2 );
        procedure GetPID ( ThePID : out PID; TheSequence : C2Sequence );
        procedure GetNPR ( TheNode : out C2; ThePID : PID );
        procedure GetNPR ( TheSequence : out C2Sequence; ThePID : PID );
        . . .
    -- operations provided only to repository managers (local forms of Create, Open, BeginSession, etc.)
        . . .
private
        . . .
end C2Pack;
```

Figure 5: An Interface-package Specification Part.

classes. This is the case for the example of figures 2, 4, and 5, where nodes of kinds Square and Circle of class C2 refer to nodes of kind Triangle of class C1. As a more concrete example in software environments, consider a symbol table organized as a binary search tree that refers to the abstract syntax tree of a program. PGRAPHITE allows the specification of the node kinds to be organized into a class for the symbol table and a different class for the abstract syntax tree. Each class is processed separately by the PGRAPHITE processor, resulting in two interface packages. The arguments for this approach are similar to those for separate compilation: better modularization, reduced reprocessing, and easier reuse.

The connections between classes are achieved through *importation*. In particular, if a node kind in one class refers to a node kind in a second class, then certain information must be imported from the second class into the first. At the GDL-specification level, a construct is provided that is similar to the Ada *with clause* but incorporates more information about the imported entities. An example of a GDL *with clause* appears in Figure 4. The information contained in the GDL *with clause* is reflected in various ways at the implementation level in the code of the interface package generated for an importing class such as C2. One trivial way that it is reflected is in an Ada *with clause* attached to the interface package. In particular, the Ada *with clause* would mention the name of the interface package generated for the imported class, as is done for C2 in Figure 5. The portion after the dot in a GDL *with clause* is what extends the Ada version of the clause. The meaning and purpose of this additional information is explained below.

Clearly there are times when one would like to have persistence of graph objects that are composed of nodes from different classes. This suggests that the persistent-store abstraction cannot be realized within any one interface package, but instead requires its own module (Figure 3). We refer to this module as a *repository manager*, which is realized as an Ada package. A portion of the specification part of such a package is shown in Figure 6. Notice that the package defines a type for repositories and so it might be used to manage several repositories.

Given that there may be more than one interface package and that the repository manager is yet another package, how is coordination among the packages achieved when carrying out the actions associated with persistence? The approach that we have taken is to define two simple protocols that dictate how the various components should communicate: one is between repository managers and interface packages and the other is between "importing" and "imported" interface

```
package RepoMan is
        type Repository       is limited private;
        type RepositoryName is new String;

        procedure Create ( TheRepositoryName : RepositoryName );
        procedure Delete ( TheRepositoryName : RepositoryName );
        procedure Open ( TheRepositoryName : RepositoryName;
                         TheRepository : in out Repository );
        procedure Close ( TheRepository : Repository );
        procedure BeginSession ( TheRepository : Repository );
        procedure EndSession ( TheRepository : Repository );
        . . .
private
        . . .
end RepoMan;
```

Figure 6: Specification Part of a Repository Manager Package.

packages. From the tool developer's perspective, these protocols are invisible and irrelevant, being hidden within the implementations of interface packages and repository managers. In particular, PGRAPHITE automatically generates the code for implementing and utilizing the protocols.

The first of these protocols simply involves a repository manager notifying the interface packages that a tool has requested an action such as the beginning or ending of a session. This gives the interface packages an opportunity to perform any class-specific actions. The other, inter-class protocol involves the transformation of NPRs into PIDs (and vice versa). Specifically, if an NPR to node n_1 of kind k_1 defined in class c_1 is the value of an attribute of a (persistent) node n_2 of kind k_2 defined in class c_2, then when the session ends, the interface package for c_2 will retrieve from the interface package for c_1 a PID corresponding to the NPR. Similarly, at certain times during a session, the interface package for c_2 will retrieve from the interface package for c_1 an NPR corresponding to the PID. (These actions are depicted in Figure 2 as the change from dashed to solid arrows and vice versa.) The two retrieval operations are specified as part of the GDL *with clause*. For the GDL *with clause* shown in Figure 4, then, we can now describe the information following the dot. It says that a type called C1 is imported from package C1Pack and that a stable way to refer to (persistent) objects of this type is through the type PID. Further, it says that to retrieve a PID, the procedure GetPID is used, and to retrieve an NPR, the procedure GetNPR is

used.[5] Notice that these two operations are the same ones used by tools to retrieve PIDs and NPRs.

Our use of a protocol approach has a beneficial side effect. Any user-defined object type, not just graph types generated by PGRAPHITE, can be fit into this scheme.[6] As long as the type-definition module for the type adheres to the protocols, then objects of that type can be stored in repositories managed by our repository managers and can be referred to by persistent objects of other types. For example, one might devise a lookup-table for a set of rooted graphs, where the entries in the table have a field for a human-readable name for the graph and a field for the graph itself. The table might be implemented by having the name field actually be a reference into a separate string table and the graph field be a reference to the root node. The persistence of all three types could be accommodated, even though only one of them presumably would be implemented using PGRAPHITE.

As it turns out, a given repository manager handles a predetermined, fixed set of object types. (The technical reasons for this have to do with Ada's static type-checking and binding rules.) If we call this set T, a repository managed by that repository manager may contain objects whose types form an arbitrary subset, S, of T. Thus, we can have repositories that have quite different makeups all managed by the same repository manager. Fortunately, there is very little overhead in using a repository manager in this way—that is, the cost of managing a repository whose (sub)set of object types S is much smaller than T is negligible. It is only when there is a need to handle a type that is not a member of T that a new repository manager must be created. To facilitate the construction of repository managers, we have built a tool called REPOMANGEN. All that this tool requires of its user is a name for the repository manager to be generated and a list of type-definition-module names. For instance, to generate a repository manager to handle the graph types of our running example, the input to REPOMANGEN would be the following:[7]

```
RepoMan

C1Pack
C2Pack
```

[5] The parameters of these procedures are fixed by convention.

[6] The types for "primitive" objects, such as integers and characters, are handled automatically.

[7] The syntax of input to the tool is actually a bit more complex than what is shown; the details of that syntax, however, are irrelevant here.

Given this input, REPOMANGEN would generate a repository manager named RepoMan (Figure 6) that could manage repositories in which objects of types C1Pack.C1 and/or C2Pack.C2 are stored.

4.2 Storage Strategy

There are two basic approaches to moving persistent objects into and out of stable storage. One approach can be taken in languages, such as Smalltalk, that provide application programs with a means to make use of information about any object's type at run time. Using this information, the mapping between any object in primary memory and its representation in stable storage can be determined dynamically. This allows the construction and use of a single, general-purpose mechanism, one that can work for arbitrary object types. Thus, we refer to this as the *centralized* approach.

Languages such as Ada, however, make no provision for the use of type information at run time. Therefore, it is not possible to provide a single mechanism to store objects of arbitrary types. Indeed, one could argue that allowing such a capability would violate Ada's strong emphasis on encapsulation. Thus, in the alternative, *decentralized* approach, the responsibility for actually moving objects to and from stable storage is left to each type-definition module. In other words, each type is responsible for its own storage. There are two advantages to this approach. First, the implementations of types are hidden: only the package in which the type is defined needs to know about the internal and external representations. Second, using a decentralized approach permits optimizations that would not be possible if a centralized mechanism were employed. For example, using a general-purpose mechanism, a table half-filled with meaningful values and half-filled with "junk" would likely occupy as much stable storage and take as much translation and transfer time as a table full of meaningful values. This is because the mechanism cannot be expected to distinguish, in general, meaningful values from meaningless ones. Under the decentralized approach, the implementor of the type for the table could allow for specialized compaction activities that a general-purpose mechanism could not hope to divine.

The approach taken for PGRAPHITE's persistence mechanism is actually a mixture of the two basic approaches. The approach is decentralized in the sense that each type-definition module (e.g., the interface package that PGRAPHITE automatically produces for graph types) is responsible for the storage of its objects. But the approach is centralized in the sense that it embodies a general-

purpose mechanism: PGRAPHITE automates the construction of storage code, freeing developers of graph types from this burden in the same way that a true centralized approach would. The mechanism works because PGRAPHITE, as mentioned above, can capture the type information it needs during the processing of a GDL specification and then make that information available to interface packages at run time.

4.3 Efficiency Strategy

Secondary-storage accesses are expensive. Furthermore, despite its growth in many computers, primary memory has a real limit. Thus, questions such as when and how much to read from the repository are significant. On one end of the range of possible solutions is an approach in which all objects reachable from an object are retrieved when access to that object is requested by a tool. In this scheme, we would accept the one-time cost of retrieving all objects, but we would then know that all objects were in memory and avoid potentially costly run-time checks for missing objects. On the other end of the spectrum is a completely "demand-driven" scheme, in which objects are retrieved from secondary storage only when they are requested. This ensures the minimum amount of data transfer from secondary storage, as well as the minimum consumption of primary memory, but it incurs the additional cost of run-time checking to determine if a requested object is already in memory.

Our approach in the current version of PGRAPHITE has been to use a transparent, demand-driven scheme for retrieval of persistent nodes and collections of nodes, employing *surrogates*. Surrogates are patterned after the *forwarders* described in [8]. In essence, a surrogate is a (non-persistent) object that contains the PID of a persistent object and, if the object is resident, an NPR for the object. Normal references are then always through surrogates. When GetNPR is called the first time for a persistent object in a particular session, a surrogate is created for that object and the NPR for that surrogate returned. However, the object itself is not retrieved from secondary storage. Instead, the NPR in the surrogate is set to a null value, indicating that the NPR has been requested, but that the information stored in the object has not been requested. The surrogate is placed in a table that is hashed by PID. All objects that reference this persistent object use the surrogate. If an explicit request is made for access to the information contained in the object, then the object is actually retrieved from secondary storage and an NPR is placed in the surrogate. Note

that the use of surrogates is completely hidden from tools. In particular, it appears to a tool using a PGRAPHITE-generated interface package as though it has direct access to persistent nodes and node collections, and that entire graphs are resident.

We are still experimenting with this completely demand-driven method of object retrieval. In the future we would like to have a scheme in which a block of objects is retrieved when any one element of that block is requested and, further, we would like to be able to flexibly define the criteria for block membership. Notice that reachability is not a sufficient criterion, since it does not, for example, work with disconnected graphs.

5 Gaining Experience with PGRAPHITE

Our initial experimentation with the current version of PGRAPHITE has been directed toward evaluating the suitability of the abstractions that it provides and the implementation strategies that it embodies. A considerable amount of additional experimentation is needed, of course, before any final judgements can be made. However, our initial experiences have been encouraging. We briefly describe one such experience in this section, namely the bootstrapped use of PGRAPHITE in developing PGRAPHITE. We also describe some plans for future uses of PGRAPHITE.

PGRAPHITE itself provided us with an excellent opportunity to study the three basic abstractions. The PGRAPHITE processor makes use of internal abstract syntax trees and symbol tables that are directed graphs. Therefore, we used an early version of PGRAPHITE to generate interface packages that allowed us to define and manipulate persistent abstract syntax trees and symbol tables, and then reimplemented PGRAPHITE using those packages.

This experiment has been successful in several ways. The earliest version of the PGRAPHITE processor was actually built using GRAPHITE. Hence, we could not easily make the PGRAPHITE internal structures persist until we replaced the GRAPHITE-generated interface packages with ones generated by PGRAPHITE. Only four additional lines of code were needed to make the structures persist, and it was not necessary to recode any part of PGRAPHITE to support the new interface packages. This indicates that we have, in fact, provided a transparent, orthogonal mechanism for persistence. Preliminary timing experiments also indicate that no degradation in performance was experienced due to the addition of persistence capabilities.

We intend to use PGRAPHITE-generated interface packages to provide persistence capabilities to other components and prototypes in the Arcadia environment. In particular, we plan to use PGRAPHITE for IRIS [2], APPL/A [4], and Chiron [17].

IRIS is a language-independent intermediate representation for programs that is used, or will be used, by a large number of tools in the Arcadia environment. Since IRIS representations are graphs, and since many IRIS graphs must persist, we are currently implementing an interface to IRIS graphs [18] on top of a PGRAPHITE-generated interface package. An interesting aspect of this use of PGRAPHITE is that it will also involve the persistence of non-graph objects within repositories. In particular, we anticipate that several non-graph object types will make reference to IRIS nodes and that objects of those types will need to persist.

APPL/A is a prototype process programming language that is being used to explore the suitability of various automated constraint-satisfaction and inferencing techniques in the domain of process programming [10]. Specifically, it provides a general framework for specifying goals in terms of "active" relationships over objects and provides mechanisms, such as backward and forward inferencing, for satisfying those goals. PGRAPHITE can be used to maintain the persistence of APPL/A relationships.

Chiron is a user interface management system. It uses a graph structure to maintain an abstract depiction of the objects being displayed. There are significant benefits to be gained from preserving such depictions, and PGRAPHITE can be used to provide that persistence.

6 Future Work

With the first prototype of PGRAPHITE available, we can begin to experiment with different models of persistence, and with different implementations of those models. There were many questions that could not be readily addressed without the prototype, such as the appropriateness of our current model of persistence or the effectiveness of our proposed implementation strategies. Our preliminary experimentation with PGRAPHITE to date, as described in the previous section, has also provided us with some insights into our current model of persistence and has raised some additional issues that must be addressed. In this section, we discuss some of these issues and some planned directions for future work with PGRAPHITE.

As we have mentioned, facilitating experimentation with a variety of underlying support systems is an important goal of this work. While we have apparently succeeded in protecting tools from such experimentation, we still need to make PGRAPHITE itself more malleable. We are therefore working closely with a group of researchers that is developing a flexible interface mechanism for storage managers, transaction managers, and the like. The prototype of this mechanism is called Mneme [9]. With a version of PGRAPHITE that uses Mneme, we will be able to experiment with different models of such things as blocking, garbage collection, and concurrency control.

The issue of concurrency is one that will be of considerable concern to us in our future work. In a software environment, it is particularly important to provide users with concurrent access to objects. To address this issue, several questions will have to be answered. We must decide if "optimistic" control is adequate, or if some other method would be more appropriate. Degrees of object locking are also an issue—should locking be based on single nodes, reachability from those nodes, whole graphs, or entire repositories? Many other questions must also be addressed, and the answers will undoubtedly affect the design of PGRAPHITE.

In Section 3, repositories are described as disjoint in the sense that the semantics of references between repositories is not addressed by our model. They are disjoint in another sense as well: our model does not now provide support for coordinating concurrent access to multiple repositories. This lack of support for inter-repository references and coordination, which in fact are interrelated concerns, may not be a desirable situation. We hope to learn through future experimentation how vital these capabilities are to tool developers and users, and what changes to our model their inclusion would require.

We would also like to address the question of version control on objects. In so doing, we face the question of what constitutes a version—is a new version of an object created when the user modifies the value of a persistent object, or is the new version not created until that object is saved in stable storage? We also need an appropriate semantics for reverting to a previous version of a particular object—is the given object reverted to a previous state by itself, or must everything reachable from that object be reverted as well? We expect to encounter these and other interesting questions in addressing the issue of version control.

Acknowledgements

We appreciate the substantial contributions made by Lori Clarke, Eliot Moss, and Steven Zeil to the work described here. We also appreciate the comments and suggestions provided by our other colleagues in the Arcadia consortium.

REFERENCES

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4), November 1983.

[2] D.A. Baker, D.A. Fisher, and J.C. Shultis. IRIS: An Internal Form for Use in Integrated Environments. Technical Report, Incremental Systems Corporation, Pittsburgh, Pennsylvania, 1987.

[3] Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf. GRAPHITE: A Meta-Tool for Ada Environment Development. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

[4] Dennis Heimbigner, Leon J. Osterweil, and Stanley M. Sutton. APPL/A: A Language for Managing Relations Among Software Objects and Processes. Technical Report CU–CS–374–87, University of Colorado, Boulder, Colorado, 1987.

[5] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *OOPSLA Conference Proceedings*, pages 472–482, September 1986.

[6] David C. J. Matthews. Poly manual. *SIGPLAN Notices*, 20(9), September 1985.

[7] David C. J. Matthews. Progress with Persistence in Poly and Poly/ML. In *Appin Workshop on Persistent Object Systems*, pages 309–316, August 1987.

[8] J. Eliot B. Moss. Implementing Persistence for an Object Oriented Language. In *Appin Workshop on Persistent Object Systems*, August 1987.

[9] J. Eliot B. Moss and Steven Sinofsky. Managing Persistent Data with Mneme: Issues and Application of a Reliable, Shared Object Interface. COINS Technical Report 88–30, University of Massachusetts, Amherst, Massachusetts, April 1988.

[10] Leon J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, California, March 1987.

[11] William R. Rosenblatt, Jack C. Wileden, and Alexander L. Wolf. Preliminary Report on the OROS Type Model. COINS Technical Report 88-70, University of Massachusetts, Amherst, Massachusetts, August 1988.

[12] Richard N. Taylor, Deborah A. Baker, Frank C. Belz, Barry W. Boehm, Lori A. Clarke, David A. Fisher, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Next Generation Software Environments: Principles, Problems, and Research Directions. COINS Technical Report 87–63, University of Massachusetts, Amherst, Massachusetts, July 1987.

[13] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, November 1988.

[14] Douglas Wiebe. A Distributed Repository for Immutable Persistent Objects. In *OOPSLA Conference Proceedings*, pages 453–465, September 1986.

[15] David S. Wile and Dennis G. Allard. Worlds: An Organizing Structure for Object-Bases. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 16–26. December 1986.

[16] David S. Wile, Neil M. Goldman, and Dennis G. Allard. Maintaining Object Persistence in the Common Lisp Framework. In *Appin Workshop on Persistent Object Systems*, pages 382–405. August 1987.

[17] Michal Young, Richard N. Taylor, Dennis B. Troup, and Cheryl D. Kelly. Design Principles Behind Chiron: A UIMS for Software Environments. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 367–376. Singapore, April 1988.

[18] S.J. Zeil. An IRIS Interface in Ada. Arcadia Design Document UM–87–06, University of Massachusetts, Amherst, Massachusetts, August 1987.