A Different Tack to Providing Persistence in a Language

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

COINS Technical Report 89-66
July 1989

[†]*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

An earlier version of this paper appeared in
Proceedings of the Second International Workshop on Database Programming Languages, pp.41-60,
published by Morgan Kaufmann

At the University of Massachusetts, this work was supported in part by the following grants: National Science Foundation DCR-84-04217 and DCR-85-00332; National Science Foundation CCR-8704478 with cooperation from the Defense Advanced Research Projects Agency (ARPA Order No.6104); and Rome Air Development Center F30602-86-C-0006.

A Different Tack to Providing Persistence in a Language

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

[†]Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

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

Abstract

Persistence is the preservation of a data object beyond the execution of a program that creates or manipulates that object. The need for persistence gave rise to file systems and databases. A fresh approach to persistence is emerging that seeks to make the programming of applications relying on persistence easier, less prone to error, and more flexible. That approach is based on incorporating the persistence mechanism into the programming language itself.

Looking at current research efforts, we see that the predominant methods of incorporating persistence into a language involve the development of a completely new language, the modification of an existing language in some substantial way, and/or the use of a language that has at best dynamic type checking and at worst no type checking at all. We chose to take a different tack, namely to use the inherent abstraction capabilities of an existing strongly typed, statically type-checked language as the means of providing a persistence mechanism to developers of applications written in that language.

We have described elsewhere the model of persistence that we have adopted. In this paper we concentrate on the implementation of that model. Specifically, we describe our prototype preprocessor-based implementation of persistent graph data types in Ada, its mechanism for incorporating non-graph types, and a software architecture that facilitates experimentation with various aspects of the implementation.

At the University of Massachusetts, this work was supported 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.

1 Introduction

Persistence is the preservation of a data object beyond the execution of a program that creates or manipulates that object. The need for persistence gave rise to file systems and databases. A fresh approach to persistence is emerging that seeks to make the programming of applications relying on persistence easier, less prone to error, and more flexible. That approach is based on incorporating the persistence mechanism into the programming language itself [3].

Looking at current research efforts, we see that the predominant methods for incorporating persistence into a language involve the development of a completely new language (e.g., [1, 10]), the modification of an existing language in some substantial way (e.g., [2, 6, 7]), and the use of a language that has at best dynamic type checking and at worst no type checking at all (e.g., [4, 8]).

We chose to take a different tack, namely to use the inherent abstraction capabilities of an existing strongly typed, statically type-checked language as the means of providing a persistence mechanism to developers of applications written in that language.

In our view, 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 [2]. In fact, persistence should be a property of instances, not of 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 programmers 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 programmer'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 programmer may choose to pay attention to or choose to ignore.

Another important consideration is that the design of a persistence mechanism should facilitate relatively easy experimentation with a variety of underlying support systems, such as storage managers, concurrency control systems, transaction managers, and the like. While database research over the years has made significant advances in these areas, there is still no consensus on what constitutes the "best" techniques, particularly for such applications as office automation, computer-aided design, and software development environments.

These considerations place great demands on the design of the *model* of persistence presented to programmers, as well as on the designs of the *implementations* of that model. We have described our model elsewhere [9], identifying an appropriate set of abstractions through which programmers can manipulate and reliably store persistent objects in a setting of possibly concurrent and distributed accesses. Here we concentrate on the novel features of a prototype implementation of the model in the programming language Ada. Specifically, we describe PGRAPHITE, our preprocessor-based

implementation of persistent graph data types in Ada, its mechanism for incorporating non-graph types, and a software architecture that facilitates experimentation with various aspects of the implementation. To provide a suitable context for this description, we first briefly review our model of persistence.

2 Model of Persistence

2.1 Side-by-side Name Spaces

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 these restrictions, 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 a model 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 infrequently, as the discussion below demonstrates. 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 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.

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)

Table 1: Persistence Abstractions and their Associated Operations.

2.2 Persistence Abstractions

As described in [9], our model provides one abstraction that represents persistent objects and another that represents the store for persistent objects. The operations associated with these abstractions are summarized in Table 1.

In striving toward the goal of orthogonal and transparent persistence, we wish to minimize—at least from the perspective of how programs 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 program needs to indicate that a particular object is to persist. The other situation occurs when a program 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.

This simplification leaves us with needing only two operations for 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.

The store for persistent objects is modeled as a collection of *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.

The operations associated with the persistent-store abstraction include those to create, delete, and gain access to a repository (Table 1b). We have distinguished two levels of access to a repository; programs 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. Although a particular implementation of this notion may impose various semantics concerning such things as locking protocols, granularity, rollback mechanisms or notification schemes, from the user's perspective the only semantics attached to sessions is that they represent units of consistency. Therefore, the notion of session in our model of persistence imposes the minimum semantics necessary to ensure the consistency of persistent objects, namely that programs are not permitted to manipulate persistent objects except within the context of a session. In our model, this amounts to a requirement that the two operations associated with the persistent-object abstraction are made available only during a session, and that objects that have already been made persistent cannot be manipulated except during a session. We leave any more detailed semantics for sessions (e.g., locking or rollback semantics) to be defined by whatever underlying transaction management system may be implementing our session abstraction. We do not believe that it is appropriate to include such semantics at the user level, particularly because different transaction management systems impose different low-level semantics. On the other hand, we believe that any transaction management system must provide some unit of consistency that is the logical equivalent of a session. Therefore, by encapsulating the specific semantics of transaction management within operations to begin and end sessions, we have given ourselves the freedom to experiment with a number of underlying transaction management support systems without impacting the user.

2.3 The Resulting Model: Using The Abstractions

Every programming language provides a means by which users can define object types and operations to create and manipulate instances of those types. Our approach to adding persistence is based on augmenting the set of operations to include those needed for the persistent-object and persistent-store abstractions. From the point of view of a user (i.e., a programmer writing code to manipulate objects of a given kind), the result is the following:

> There is a set of types for objects. Individual instances of those types can optionally be made persistent, by invoking one of the persistent object abstraction operations associated with that object type. Manipulation of all object instances, whether persistent or not, can be done via the usual operations that the programming language provides for such manipulation. Making an object persistent results in a second "name" (the PID) for that object, which, unlike the name provided by the programming language itself, can be used to access the object after the currently executing program has terminated. Making an object persistent also preserves the reachability (by the usual dereferencing operations of the programming language) of all objects reachable from that object. Objects that have been made persistent reside in a repository. During a session, those objects can be manipulated in the same way as any other object in the program. Outside of a session, however, the objects in the repository are unavailable for manipulation and no new objects can be added to the repository. At the beginning of a session, the objects in a repository will be in exactly the same state that they were at the end of the last previous session. To manipulate such objects, a program must have a PID for at least one object in the repository. Converting that PID to an NPR will provide the program a means of accessing that object and any objects reachable from it.

In order to clarify the above-described user view of our model of persistence and its implications, we now present a brief example of how a user would use the model's abstractions to define and manipulate a potentially persistent object. (A more complete example appears in [9].)

Consider a simple binary search tree object type. Each node in the tree contains a string and references to the node's left and right children. The definition of such a type would include a type for nodes in the tree, plus operations to create nodes and to set or retrieve the values stored in nodes. With these, the user can define a (potentially persistent) binary search tree.

At some point, the user might decide that he/she wants the tree to persist (i.e., that he/she needs to retain a persistent reference to the tree). This can be accomplished through the persistent-object operation Get PID. Recall, however, that operations in the persistent-object abstraction are not available outside the context of a session. Therefore, the user must first create or open a repository and begin a session, using the persistent-store operations. Note that these actions could have been taken at any point prior to the first use of a persistent-object operation, i.e., before, during or

after use of the operations that constructed or modified the tree. So, at some point following a Begin Session, the user requests a persistent reference to the root of the tree. As a side effect, the root, and all objects reachable from it, will persist.

Although the tree is now a persistent object, it can still be modified by the operations for creating nodes and setting or getting values of nodes, as long as the session has not ended. New nodes can be added to the tree, all of which will persist, and the values stored in any node of the graph may be changed normally. When the user no longer wants to manipulate the tree, he/she ends the session with the persistent-store operation End Session. Once the session is over, the persistent objects are no longer available for manipulation. These objects are preserved in the repository, however, in whatever state they were in at the time the session ended.

At some later time, another user (or even the same one) might want to manipulate the persistent tree. Again, he/she must first open the repository and begin a session before being permitted to manipulate any persistent object in, or add any other objects to, the repository. Once a session is active, persistent objects within the repository may be accesssed. There are actually two possible ways in which to interact with persistent objects, depending on whether or not the object became persistent explicitly (i.e., via a call to Get PID). If the object was made persistent explicitly, then there exists a PID for it,[1] and hence a way of directly accessing that object in the repository. Before a user can manipulate such an object, however, he/she must use the Get NPR operation to obtain a non-persistent reference to the persistent object.

Once the user has an NPR for the root of the binary search tree, he/she can manipulate the tree as before. Note in particular that if the user asks for the value of the left child of the root, the appropriate node will be returned, even though it is also a persistent object. The user does not have to go through the persistent-object abstraction to gain access to those objects reachable from the root, but instead can use the object abstraction operations directly. Just as requesting a persistent reference to the root object was sufficient to make all reachable objects persist, requesting an NPR to the root is sufficient to make all reachable objects available again via the "normal" object manipulation operations. This satisfies our requirement that persistence be an orthogonal and transparent property of instances of types. We note that if the user had asked for a PID for a node that is reachable from the root, he/she could request an NPR for that node instead of traversing the tree to reach it, and so has two ways of interacting with such persistent objects.

3 Implementing the Model in Ada

As mentioned in Section 1, our goal is to incorporate persistence into a strongly typed, statically type-checked language without modifying the language itself. This is an especially challenging test of the thesis that language features for defining procedural and data abstractions are sufficient to

[1]Note, however, that implicitly persistent nodes also have PIDs; see below.

attain language extensibility. In this section, we describe the implementation strategies that can be employed to realize our model of persistence, as illustrated by our existing Ada implementation.

3.1 The User Interface

The Ada implementation of our model of persistence is based on the assumption that all object types are defined as Ada packages.[2] We refer to these packages as *interface packages* because they provide or define the abstract interface to the objects of that type. To implement our persistence model, we add to each interface package a realization of the persistent-object abstraction described in Table 1a. This amounts to the addition of a type for referring to persistent objects (i.e., PID types) together with the Get PID and Get NPR operations. The persistent store abstraction is realized in a separate Ada package, which we refer to as a *repository manager*. This package provides the operations shown in Table 1b. It also provides a type for repositories and thus can be used to manage several repositories. By incorporating the persistent object abstraction into the interface packages of any and all abstract types defined in an Ada program, while providing the persistent store abstraction through the repository manager package, we can provide persistence capabilities to Ada programmers in a manner consistent with the Ada style of programming.

For the simple binary tree example of Section 2.3, an interface package defining the (potentially persistent) binary tree type, along with the operations to manipulate instances of that type, might look like that shown in Figure 1. (A more detailed example appears in [9].) This interface package represents the view that a user has of our persistence model and its integration with an object type. Via this interface package the user can create nodes (using the Create operation), link and fill nodes by putting values into the attributes of the nodes (PutAttribute), and navigate the tree and interrogate nodes by getting the values of attributes (GetAttribute). Having created a binary tree, such as that shown in Figure 2, the user can make it persist by opening a repository, beginning a session, and requesting a persistent reference to the root of the tree (GetPID), thereby making all nodes reachable from the root persist. If the user needs a persistent way to refer directly to any of the other nodes in the tree, he/she can request additional PIDs for those nodes. The user can continue to modify the tree after requesting the PIDs, adding or deleting nodes, changing node attribute values, etc. Note that the signatures of GetAttribute and PutAttribute indicate that all of these operations are performed via NPRs (type Binary_Search_Tree in this example), and hence are equally applicable to persistent or non-persistent nodes.

When the user decides that he/she will not be interested in manipulating the tree for some time, he/she ends the active session. At the end of the session, all persistent objects become inaccessible. The use of GetNPR in subsequent sessions will make the persistent tree/nodes available again, and

[2]This is a slight abuse of the Ada terminology. An Ada package actually encapsulates declarations for a type and that type's operations, and so is not itself "the type". However, viewing a package as the type is convenient for expository purposes.

```
package Binary_Search_Tree is

   -- utility types (NodeKindName, AttributeName, etc.)
      ...
   -- object abstraction
      type Binary_Search_Tree is private;
      ...
      function   Create ( TheNodeKind : NodeKindName ) return Binary_Search_Tree;
      procedure PutAttribute ( TheNode : Binary_Search_Tree; TheAttribute : AttributeName;
                               TheValue : Binary_Search_Tree );
      function   GetAttribute ( TheNode : Binary_Search_Tree; TheAttribute : AttributeName )
                               return Binary_Search_Tree;
      procedure PutAttribute ( TheNode : Binary_Search_Tree; TheAttribute : AttributeName;
                               TheValue : String );
      function   GetAttribute ( TheNode : Binary_Search_Tree; TheAttribute : AttributeName )
                               return String;
      ...
      function Kind ( TheNode : Binary_Search_Tree ) return NodeKindName;
      function AttributeValueType ( TheNodeKind : NodeKindName; TheAttribute : AttributeName )
                                    return AttributeValueTypeName;
      function NodeKindAttributes ( TheNodeKind : NodeKindName ) return AttributeNameList;
      ...
   -- persistent-object abstraction
      type PID is private;
      ...
      procedure GetPID ( ThePID : out PID; TheNode : in Binary_Search_Tree );
      procedure GetNPR ( TheNode : out Binary_Search_Tree; ThePID : in PID );
      ...
   -- operations provided only to repository managers in support of the decentralized
   -- implementation of the persistent-store abstraction (local forms of Create, Open,
   -- BeginSession, EndSession, etc.)
      ...
private
      ...
end Binary_Search_Tree_Definitions;
```

Figure 1: A Binary Search Tree Interface Package.

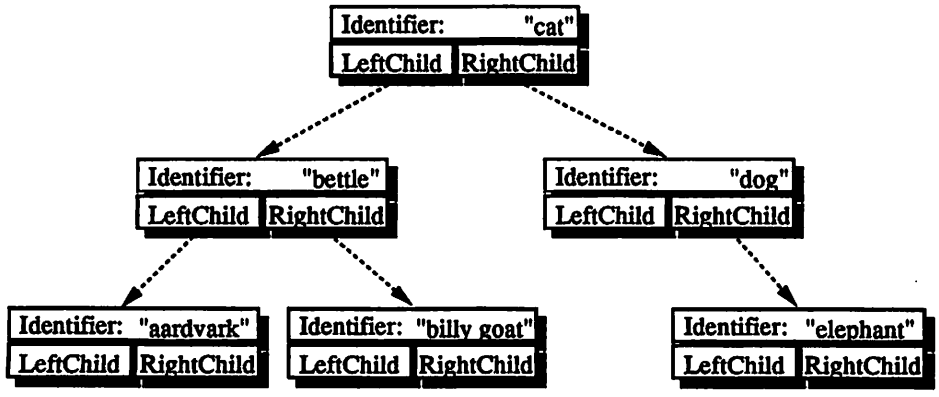


Figure 2: A Simple Binary Search Tree.

they will be in the same state they were in at the end of the most recent previous session. The user will then be able to create new nodes and link them to nodes in the tree, fill in node attribute values, navigate through the tree and interrogate nodes as before.

3.2 Implementation of the Ada Interface Package

In Section 3.1 we described the user view of the interface package. In this section, we provide some lower-level details about how the abstractions that the user sees are realized in our Ada implementation.

Because Ada provides us with a way to define abstract data types, defining NPRs and the operations on them (i.e., realizing the object abstraction) is straightforward. Realizing the persistent object abstraction, however, is considerably more complicated. One question that we faced immediately was when to move persistent objects to stable storage. We decided for the present to defer such movement until the end of sessions. Clearly, this is not the only strategy available — we could have decided as easily to store persistent objects as soon as they are designated as such (i.e., by a Get PID operation), and to update the stable storage representation as persistent objects are modified. We felt, however, that deferring storage until the end of the session would provide us with the most efficient implementation possible, since we would only have to store each persistent object once per session.

Deciding that the storage of persistent objects should occur at the end of sessions had another interesting effect on our implementation. Because every interface package provides the realization

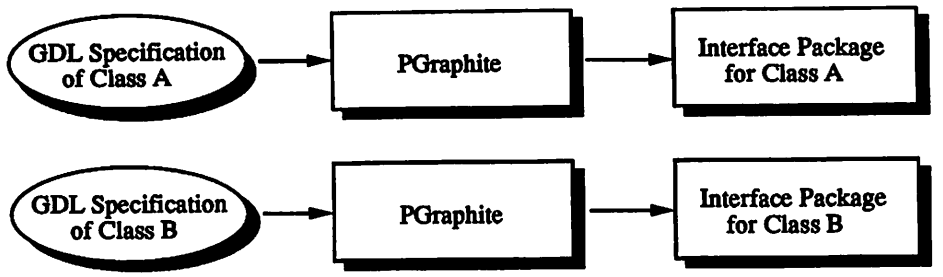
of the persistent-object abstraction for objects of the type(s) it defines, only the individual interface packages are notified when instances of the object types they define are to persist. The End Session operation, on the other hand, has no way to determine which objects need to be stored. Therefore, we employ a *decentralized* approach to end-of-session processing — that is, each interface package is told when the session has ended so that it can store copies of its persistent objects in secondary storage. This process involves generating and storing what we term "external forms" of the information in each object. For most types of information, this is straightforward; in fact, except for references to other persistent objects, Ada provides a direct means to store any kind of data. In Section 3.4, we discuss how we store references to other objects.

While it was only necessary to use decentralized processing for the End Session operation, we opted to implement all of the persistent-store operations in a decentralized fashion. Therefore, each interface package is also responsible for communicating directly with the underlying storage management system(s). We chose this implementation strategy to allow ourselves the greatest amount of flexibility in choosing underlying storage management systems to meet the particular storage needs of each object type. Since no single storage management system adequately supports every kind of object, we felt that it would be an advantage to permit different interface packages, even within the same program, to use different underlying support systems, thereby maximizing storage efficiency.

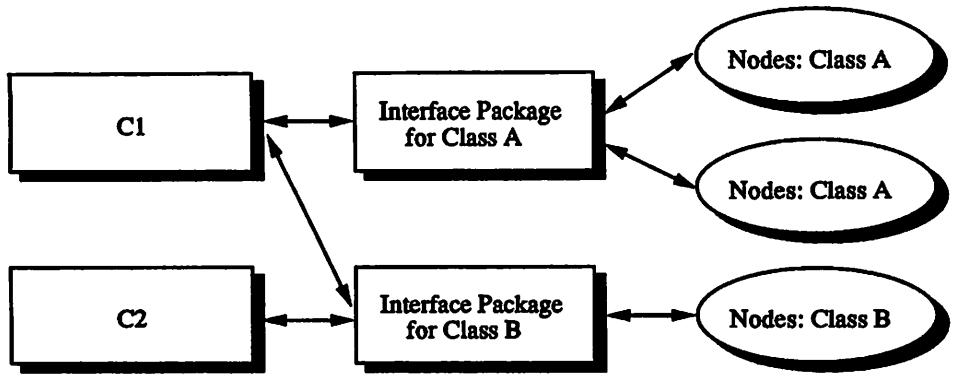
3.3 Generating the Interface Package

Of course, programming an interface package for a potentially persistent object type such as the one described above is by no means a trivial exercise. Clearly, such packages can be programmed by hand, and in fact, we did so initially. It has been well documented in the literature, however, that a significant portion of programming errors arise from the need to implement input and output functions. For "standard" or heavily used types, it would nevertheless be worthwhile to provide such operations, even if they had to be hand coded, since the operations would only need to be written once, and could be reused thereafter. It would be preferable, however, to have the input/output operations automatically generated from some declarative definition of the (potentially) persistent type. PGRAPHITE is an instance of such a generator for one class of types, namely directed graphs. Since graph objects are very common in software applications in our setting, and since graph types are among the most general and complex of types, a prototype providing automated support for persistent graph objects is immediately useful, can be widely exercised, and can result in experience with and insights regarding most, if not all, of the important issues related to persistence. We have taken care to design general-purpose implementation strategies for persistence operations, and as we show in Section 3.5, we have in fact been able to demonstrate that our strategies can be extended to non-graph objects types.

Figure 3 depicts the use of PGRAPHITE. The specification language that is used for input to PGRAPHITE is called GDL (Graph Description Language). A GDL specification describes a set of



(a)



(b)

Figure 3: Creating and Using PGRAPHITE-generated Interface Packages.

```
class Binary_Search_Tree is
    package Binary_Search_Tree_Definitions;


    node BST_Node is
        Identifier : String;
        LeftChild : BST_Node;
        RightChild : BST_Node;
    end node;


end Binary_Search_Tree;
```

(a)

```
class Binary_Search_Tree is
    package Binary_Search_Tree_Definitions;


    with Abstract_Syntax_Graph_Definitions.
        ( Abstract_Syntax_Graph / ASG_PID
          in  => Get_ASG_NPR
          out => Get_ASG_PID );

-- Imports the definition of type Abstract_Syntax_Graph
-- from interface package Abstract_Syntax_Graph_-
-- Definitions.
--
-- Other parameters:
--   ASG_PID        -- The name of the PID type for
                       Abstract_Syntax_Graphs
--   Get_ASG_NPR    -- The Get NPR operation
--   Get_ASG_PID    -- The Get PID operation

    node BST_Node is
        Identifier : String;
        Declaration : Abstract_Syntax_Graph_Definitions.
                      Abstract_Syntax_Graph;
        LeftChild : BST_Node;
        RightChild : BST_Node;
    end node;


end Binary_Search_Tree;
```

(b)

Figure 4: GDL Descriptions of Binary Search Trees.

node kinds and we refer to that set as a *class* of node kinds. Using the interface package produced by PGRAPHITE, an application program can create and/or access a number of different graphs. Any or all of these could be manipulated by other programs, which would also access these graphs through the operations provided in the interface package. Moreover, a program may use more than one interface package in order to access more than one class of node kinds. Figure 3a illustrates how PGRAPHITE could be used to create interface packages for two different classes of node kinds, called Class A and Class B. Figure 3b then shows how two programs, C1 and C2, might use these packages; C1 to manipulate two graphs consisting of instances of Class A nodes and both C1 and C2 to manipulate a graph consisting of instances of Class B nodes. The binary search tree interface package presented in the previous section was, in fact, generated by PGRAPHITE from the GDL description shown in Figure 4a.

3.4 The Role of the Preprocessor

As discussed in Section 3.2, we can directly store any type of data in an Ada object, except for references to other objects. A certain amount of information about instances of a potentially persistent type must be available to the persistence mechanism at run time, so that it can identify such references and transform them to a form suitable for secondary storage. In the setting of our prototype, this amounts to the identification of the origins of directed graph edges within a graph object, since such edges are represented in memory as non-persistent references, which cannot themselves be preserved. Unfortunately, Ada does not provide such information to application programs, and this fact makes Ada's features insufficient for our purposes.

Our response is to capture the necessary type information as part of the preprocessing performed by PGRAPHITE. Because PGRAPHITE "sees" the type definitions before Ada does, it has the opportunity to determine statically which components of the graph objects described by those type definitions are references to other objects. In the case of our binary search tree definition in Figure 4a, for example, the preprocessor has little difficulty determining that the values of attributes LeftChild and RightChild are references to other nodes. PGRAPHITE uses this information to generate run-time data structures that can be queried by the other parts of the persistence mechanism. The current implementation of PGRAPHITE represents this information as a static table that contains the type of each node attribute. Code to check the type information stored in this table is generated as part of the interface package input/output operations, and where the attribute represents a reference to another object, code to retrieve and store a persistent reference to the other object is included. In contrast, notice that attribute Identifier is a string, not a reference to another object, so it does not require any special processing.

Although PGRAPHITE only currently supports the automatic generation of a persistence mechanism for graph objects, the preprocessor approach could clearly be extended to provide persistence for other types. In fact, PGRAPHITE already supports all of Ada's predefined types (e.g., Integer and Boolean) and most of its type constructors, including arrays and records. A developer can therefore

define virtually any Ada type in GDL, and can use those types in the definition of graph object types. PGRAPHITE can make instances of such types persist as long as they are contained within graph nodes of some type that has been defined in GDL and passed through the PGRAPHITE preprocessor.

3.5 Persistence of Heterogeneously Typed Objects

As described so far, PGRAPHITE provides a means by which objects of any single class of node kinds can be made persistent. Typically, however, programmers will want to define graph structures that include references to nodes defined in other classes. More generally, they will want to have nodes refer to persistent objects of arbitrary types and to have objects of arbitrary types refer to persistent nodes. The question then becomes, how can we generalize our persistence mechanism, and in particular the preprocessor, to allow its use with arbitrary types?

The solution that we have adopted involves the definition of a protocol through which interface packages for arbitrary types, including PGRAPHITE-mediated graph types, can communicate necessary information to the persistence mechanism. As described above, each PGRAPHITE-generated interface package has the type information necessary to manipulate its own persistent objects in a decentralized fashion. That is, each interface package is responsible for providing a persistence mechanism for instances of the object types it defines. The interface to that mechanism is defined in the protocol to be the Get PID and Get NPR operations; we make no requirements on the underlying mechanism, though we do currently specify fixed signatures for these operations. To make any heterogeneous object (i.e., one that contains references to objects of other types) persist, then, the interface package that defines the object type must be able to identify those parts of the object that represent references to other objects, and ask the interface package that defines the other object type for persistent references to those objects before storing the object that is to persist. This action has the implicit effect of indicating to the persistence mechanism that the referenced object is to persist as well.

To demonstrate how a heterogeneous object would be processed under this decentralized approach, let us consider the binary search tree example again. Binary search trees are commonly used by compilers as a quick way to look up the definitions of identifiers scanned in the input program. Each binary search tree node would therefore have an additional attribute, i.e., a reference to a subgraph within an abstract syntax graph that represents the parsed declaration of the identifier. Figure 4b shows a GDL definition of such a binary search tree. The binary search tree is therefore a heterogeneous object type, as it contains references to objects of other types (namely, abstract syntax graph nodes).

Using the interface packages for binary search trees and abstract syntax graphs, the user defines the structure in Figure 5. At some point, he/she decides that the root of the binary search tree should persist. The user creates or opens a repository, begins a session, retrieves a PID for the

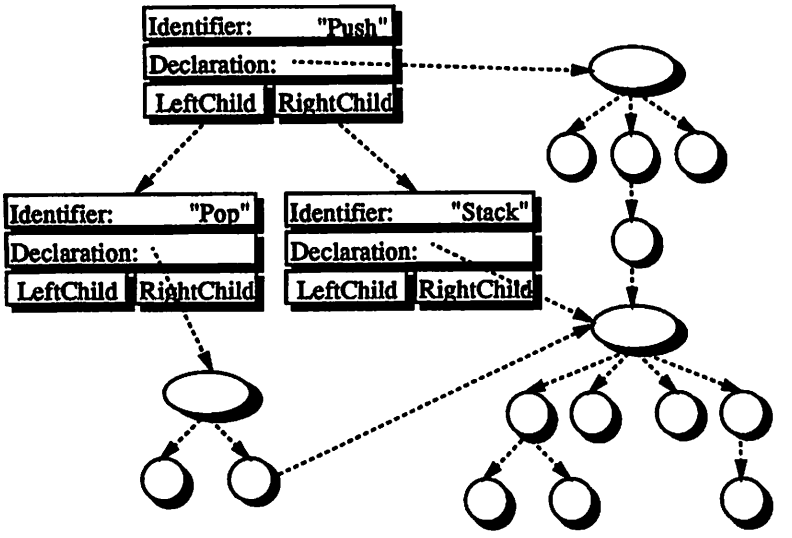


Figure 5: A Binary Search Tree with References to an Abstract Syntax Graph.

root, and ends the session. Because our model of persistence includes a decentralized approach to storage management, both the binary search tree and the abstract syntax graph interface packages must be notified that the session is over, so that they may each process their persistent objects.

Let us look at what happens when the binary search tree interface package is notified that the session has ended. This interface package has maintained internally a list of persistent nodes, and each time a GetPID operation is invoked, the node for which a persistent reference was requested is added to this list (provided that it was not already in the list). At the end of the session in our example, the binary search tree interface package finds that it has only one persistent node to process, namely, the root of the graph. It looks up the types of each node attribute, and discovers that the value of the attribute Identifier is a simple string, the values of the attributes LeftChild and RightChild are references to other binary search tree nodes, and the attribute Declaration represents a reference to an object of type Abstract_Syntax_Graph, defined in an interface package named Abstract_Syntax_Graph_Definitions. Strings require no further processing. However, the interface package knows that it must request persistent references to the values of the attributes LeftChild and RightChild, because the preprocessor was able to determine this fact, and so it invokes the GetPID operation on those attributes.

To process the value of the attribute Declaration, however, the preprocessor needs to know the name of the type of persistent references to, and the name of the Get PID operation defined on, abstract syntax graph nodes. The GDL import statement (the *with clause* shown in Figure 4b) includes optional syntax to convey this information to the PGRAPHITE preprocessor. Notice from this figure that the persistent reference type for abstract syntax graphs is called AST_PID, and that the Get PID and Get NPR operations are called Get_AST_PID and Get_AST_NPR, respectively.

This information would have been extracted by the preprocessor and included in the node attribute type information that was provided to the interface package for use at run time.

The end-of-session processing of heterogeneous objects is therefore straightforward. In precisely the same manner that it processes references to other objects of the same class, the interface package retrieves persistent references to other classes of objects. Requesting a persistent reference to an abstract syntax graph node has the effect of making the node, and all nodes reachable from it, persist. When the abstract syntax graph interface package does its part of the decentralized end-of-session processing, it will find that persistent references have been requested for all of the objects shown in Figure 5, and it will process these objects as described previously. The problem of making heterogeneously typed objects persist is therefore solved, as long as all of the object types to which the heterogeneous objects refer support the protocol by providing a persistent reference type and Get PID and Get NPR operations. Note again that the implementation of the persistence mechanism is completely transparent to the user; once the preprocessor is given the information it needs about externally defined types, it can automatically generate the end-of-session processing code.

3.6 Coordinating the Decentralized Processing

One problem that arises with the decentralized approach to persistent object management is that there is a need to coordinate the individual processing carried out by the different interface packages. For instance, consider what would have happened in the example of the previous section if the abstract syntax graph interface package had completed its end-of-session processing before the binary search tree interface package had completed its own processing. In our example, none of the abstract syntax graph nodes were made persistent explicitly — all of the abstract syntax graph nodes that persist do so because they were reachable from the root of the binary search tree, which was made persistent explicitly. If each interface package is only notified once that the session has ended, and if the order in which the interface packages are notified is not correct, objects that become persistent implicitly because of the end-of-session processing of other interface packages may not be processed, since they might not yet have been tagged as persistent when their interface package did its processing. The alternative is to notify each interface package that the session has ended multiple times, until all interface packages report that they have processed all of their persistent objects. Clearly, this solution has the potential to be considerably more time consuming than the former. We have therefore chosen the first strategy, with the understanding that some component of the system must have a sufficiently "global" view of the system so that it can perform the end-of-session notifications in the appropriate order.

Since interface packages have only a very localized view of the world, they clearly cannot themselves be responsible for coordinating the decentralized processing. Therefore, we allow repository managers to coordinate the persistent-store abstraction operations that each interface package performs. For every collection of object types that are to be stored in the same repository, there must

be a repository manager that "knows" about the interdependencies between the interface packages that define each of the object types. A repository manager for repositories that could contain the structure shown in Figure 5, for example, would know that binary search tree nodes can refer to, and therefore depend on, abstract syntax graph nodes. In carrying out the EndSession operation, then, the repository manager would first notify the binary search tree interface package that the session had ended, and would then notify the abstract syntax graph interface package.

To support the automatic generation of repository managers, we have implemented another tool, called RepoManGen. RepoManGen takes as input a list of interface packages, along with a list of the interface packages on which each depends, and the names of the operations provided by each interface package to implement its part of the decentralized persistent-store abstraction. It uses this information to determine the order in which the interface packages should be notified that a session has ended, and encodes this information in the repository manager package that it generates. It should be noted that our implementation of repository managers is such that the specification parts of all repository managers are the same; information about specific interface packages is encoded in the body part alone. This means that adding or deleting interface packages to/from those managed by any repository manager does not require any recoding of clients, and necessitates only a minor recompilation.

3.7 An Architecture for Experimentation

One of our stated goals was that the implementation of the persistence mechanism facilitate relatively easy experimentation with a number of underlying support systems, such as storage managers, concurrency controllers, transaction managers, and the like. Clearly, if changing underlying support systems necessitates recoding of existing applications, developers will be reluctant to incorporate new technology. Even paying the lesser price of large-scale recompilations is often unacceptable to developers. In this section, we discuss some of the aspects of our PGraphite implementation that have helped us to realize this goal and describe some related experiences.

The implementation of PGraphite interface packages is based on a layered architecture that minimizes the impact of changing the underlying support systems. While interface packages are responsible for properly utilizing the protocol described above, they are not directly responsible for physically storing or retrieving persistent objects, nor for managing repositories or sessions on the repositories. Instead, we have produced a general-purpose interface to underlying support systems. This interface defines operations that we feel are primitive capabilities of any storage management, transaction management, or concurrency control system, and is based in large part on the Mneme system [5]. The peculiarities of particular instances of those systems are hidden from the application programs by interface packages and from interface packages by the support-system interface.

We have found to date that the layered architecture has, in fact, provided us with the freedom to experiment. One way in which we have benefited from this is the ease with which we were able to

change our approach to persistent object retrieval. The first implementation of PGRAPHITE did *monolithic* retrieval of objects from stable storage; that is, all persistent objects were read, whether or not they were accessed. Since this approach was extremely inefficient, and since most applications do not access all persistent objects, we decided to incorporate a *demand-driven* approach to object retrieval, where objects are only read when information in them is specifically requested. The approach was designed with the goal of minimizing the number of storage accesses, which are obviously quite costly. Objects are not physically retrieved from stable storage until an application has a need for information stored in the object. Therefore, if an object is never accessed, it is not retrieved from stable storage.

As an example of this demand-driven approach, consider a user request to retrieve the root of the graph depicted in Figure 5. The user would invoke the GetNPR operation, supplying the appropriate PID. The value of the attribute Identifier would be retrieved, along with the PIDs of the attributes RightChild, LeftChild, and Declaration. However, the values of the attributes LeftChild, RightChild, and Declaration would not be retrieved from stable storage until information in them is requested (i.e., by a GetAttribute operation). Instead, we store what we call *node surrogates* as the values of these attributes. Node surrogates contain the PID of the object that is referenced, and an NPR to the actual object. If the object has not been "faulted" in, this reference is null.

This demand-driven approach to object retrieval has proven to be far more efficient than the monolithic approach. Despite the change to the internal representations of objects, however, only the implementation parts of interface packages had to be recompiled. No PGRAPHITE clients were otherwise affected, because this so-called "object faulting" is invisible to the application; from the perspective of the application, all objects are equally available, which is in keeping with our requirement that the persistence mechanism be transparent. The only visible effect was one of performance, not functionality, and we achieved this effect with no recoding and minimal recompilation. We are currently working on an enhancement to this scheme in which objects that are likely to be referenced together are clustered together for storage and retrieval. Once again, we can experiment without requiring *any* changes to application programs and only minor changes to interface packages.

In an effort to gain even more flexibility and better performance in the implementation, we have recently completed a port of PGRAPHITE from a storage management scheme based on operating-system files to one based on the primitive objects of Mneme [5]. We are using Mneme as a low-level, general-purpose interface to storage managers. We found that to reimplement PGRAPHITE interface packages to communicate with Mneme we had to change three internal operations, with a total of about fifty lines of code and less than an hour of programmer effort. Again, the changes were not visible to the user, and did not necessitate either recoding or recompilation of clients. This experience has given us more confidence that our layered architecture is, in fact, a reasonable approach, and it is our intention to experiment with other underlying storage, transaction, and concurrency control systems to determine the strengths and weaknesses of each. With this information, we will be able to decide which systems to use with different data types to maximize performance.

4 Conclusions and Future Work

A robust version of PGRAPHITE was completed in February 1988 and is currently being used by a number of our colleagues in a variety of applications, from program generation and analysis tools operating on shared representations to interactive graphical user interfaces. In fact, PGRAPHITE has been used in its own implementation.

Our experience with the development and use of PGRAPHITE has convinced us that a reasonably flexible, convenient, and efficient persistence mechanism can indeed be provided in a strongly typed, statically type-checked language. Moreover, it can be provided without making changes to the language and, therefore, without incurring the costs and complexity of "mucking" with a language implementation nor facing the dreaded constraints of non-portability.

One of the lessons that we have learned from experimentation with our prototype is that although our definition of persistence in terms of reachability is sometimes appropriate, it is often the case that an application will want to specify "bounds" on what actually persists. That is, beyond a certain node in a graph, or beyond a certain attribute of a node, the persistence mechanism should not continue making reachable objects persist. The problem in defining such a capability is that, in specifying the bounds of persistence, we do not want to violate our requirements that persistence be transparent and orthogonal. We have begun to examine the idea of specifying those edges that *might not* persist as potentially persistent relationships. This would allow the developer to specify in the GDL description a distinction between those properties of objects (e.g., node attributes) that must persist if the object persists, and those that it may not make sense to save (e.g., data that must be recomputed). Such an approach has the benefit of allowing the user to decide at run time what should persist and what should not. Clearly, this approach also satisfies our requirement that persistence be orthogonal from other properties of an object's type definition.

The only serious problem that we encountered in using Ada as the basis for our prototype was the lack of run-time type information. We were forced to capture and encode that information for ourselves through a preprocessor. One of the authors, A. Wolf, is investigating the implementation of our model of persistence in C++, which is also a strongly-typed, statically type-checked language, but with a somewhat different type model from that of Ada. Unlike Ada, C++ provides a certain amount of run-time type information, at least implicitly, through its dynamic binding mechanism. Initial results have indicated that it is substantially easier to incorporate our model into C++ than into Ada. This experience has given us further confidence that our tack is an appropriate one.

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. Finally, we would like to thank Richard Hudson, Penelope Sibun, David Stemple, and Joyce Vann for reading and commenting on drafts of this paper, and Alexander Wise for his invaluable help in producing figures.

REFERENCES

[1] Antonio Albano, Luca Cardelli, and Renzo Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[2] 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.

[3] Malcolm P. Atkinson and O. Peter Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[4] L. Cardelli. Amber. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1984.

[5] J. Eliot B. Moss and Steven Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Advances in Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science, pages 298–316. Springer-Verlag, September 1988.

[6] J.W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[7] J.M. Smith, S. Fox, and T. Landers. *ADAPLEX: Rationale and Reference Manual.* Computer Corporation of America, 1983.

[8] 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.

[9] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. PGRAPHITE: An Experiment in Persistent Typed Object Management. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, November 1988.

[10] Stanley B. Zdonik and Peter Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proc. Ninteenth Annual Hawaii International Conference on System Sciences*, pages 378–387, 1986.