Extending and Limiting PGraphite-style Persistence

Peri L. Tarr
Jack C. Wileden
Lori A. Clarke

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

Abstract

We have been working on both extending and limiting the approach to persistence embodied in our PGRAPHITE system. The extensions include implementing automated support for persistent objects of classes other than directed graphs, notably relationships and relations, and porting our system to storage managers other than Ada Direct_IO, notably Mneme, ObServer II, and the Exodus Storage Manager. The work on restrictions has focused on defining and implementing principled ways to specify the "limits" of the reachability-based persistence model employed by PGRAPHITE. The approach that we have taken relies on using potentially persistent relationships and relations to specify the "boundaries" that (de)limit persistence in complex connected structures. In this paper we describe both of these aspects of our recent work, our implementations of them and our experience with them.

This work was supported in part by the National Science Foundatation (CCR–87–04478) with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104).

1 Introduction

The PGRAPHITE system [3, 5] is a prototype realization of a particular approach to including persistence in modern programming languages. By "modern", we mean languages that are strongly typed and that provide rich support for the definition and use of abstract data types — Ada is our standard exemplar.

Given such a language, our approach to persistence is to augment the set of operations provided by any and all types in a program to include operations that can make individual instances of the types persist. From the programmer's perspective, this introduces two new abstractions, one for persistent objects and one for persistent stores, that can be smoothly integrated with the other abstractions used in the program. The result is an orthogonal, transparent persistence mechanism that permits late binding on persistence decisions. Most other approaches force a much earlier decision concerning persistence, e.g., at type-definition or instance creation time.

While the approach is fully general (as we have demonstrated by applying it manually and as has been demonstrated by the Persi system [6], which automates the approach for most of C++), our original PGRAPHITE prototype only provided automated support for the approach for a limited set of types. Because PGRAPHITE was developed as part of our work on object management support for software development environments (in the Arcadia project [4]), that set of types was abstract directed graphs, which are among the most frequently used types in such environments. This particular set of types forced us to address the interesting question of "extent" of persistence — if a given graph node is made persistent, what other objects, if any, should become persistent as a result?

Our experience with PGRAPHITE over the last couple of years has indicated that our approach to persistence and to determining the extent of persistence is appropriate for many environment applications. It has also led us to develop automated support for creating additional abstract types that embody our approach to persistence, to explore alternative definitions for the extent of persistence, and to develop an interface definition that modularizes our implementation architecture. In the remainder of the paper we will briefly review the PGRAPHITE approach and its prototype implementation, and then describe our recent work in each of these areas.

2 Review of PGraphite

The PGRAPHITE approach to providing persistence, and the prototype PGRAPHITE processor that produces Ada interface packages that implement the approach for arbitrary abstract directed graph types, have been described in detail elsewhere ([5], [3]). Here we briefly review two central features of the PGRAPHITE approach — PGRAPHITE's name space model and its reachability-based definition of the extent of persistence — then provide an example to illustrate the approach.

2.1 Side-by-side Name Spaces

Every language provides some way of referring to instances of types. But in most languages, this mechanism is somewhat restrictive. In particular, the validity of such references typically is 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 language. 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. This is usually accomplished by modifying the run-time system or by custom-building a name space on top of the one provided by the programming language.

Our model of persistence addresses this issue instead by using *side-by-side* name spaces and defining a mapping between references in the two spaces. One of the name spaces is made up of the "normal" references to objects provided by the run-time system. We refer to these as *non-persistent* references (NPRs) because they 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). When application programs manipulate objects, they use NPRs to refer to those objects, whether or not the objects persist. If a program wants to obtain a reference to an object that will be valid at some, perhaps indeterminate, time in the future, it must request a PID for the object. This has the appropriate side-effect of making the object persistent, if it is not already.

2.2 Reachability-based Persistence

As noted in the introduction, when "specializing" our model of persistence for directed graph objects, we had to choose some semantics for what happens when a node that becomes persistent has attribute values that represent references to other nodes. Since we believe that the meaning or value of a node includes the values of that node's attributes, we employ a *reachability-based* model of persistence for our graph objects. That is, if a node has attributes that represent references to other nodes, those other nodes will persist. This means that explicitly making a node persist will result in making all objects reachable from that node persist implicitly. Since node attributes can be of any type, we generalize the model by stating that the value of any composite object, of which nodes are an example, is derived from the values of its components, so that any object to which a composite object refers will persist.

2.3 A Graph Example

To demonstrate how PGRAPHITE's side-by-side name space and persistence model work for directed graph abstract data types, we will define a simple binary tree ADT and show how a developer would make instances of that ADT persistent.

Consider a simple binary tree object type. Each node in this kind of 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 and operations to create nodes and to set or retrieve the values stored in the nodes (i.e., the node attribute values).

Figure 1 shows a possible definition of a binary tree in GDL (**G**raph **D**escription **L**anguage), the Ada-like input language of PGRAPHITE. The GDL specification describes one node kind, Data_Node, which has attributes called Data (for the string value that each node will contain), Left_Child, and Right_Child. This specification is given as input to the PGRAPHITE preprocessor, which produces an Ada package (called an *interface package*) that defines the binary tree ADT.

One section of the specification part of the PGRAPHITE-generated interface package for the binary tree ADT described in Figure 1 appears in Figure 2. Using the types and operations defined in the interface package, a program could create a binary tree such as the one shown in Figure 3.

At some point, the program might decide that the tree should persist. The only reference to the tree that the program has is the value of the variable Root. Root represents a "normal," run-time system defined, non-persistent reference to the tree, however, so it cannot be preserved. Therefore, the program can use the GetPID operation defined by the PGRAPHITE-generated interface package to retrieve a persistent reference to the root of the tree, thus implicitly causing the root to become persistent. The persistent reference returned by GetPID is guaranteed to identify the root of this tree uniquely throughout its lifetime (unlike

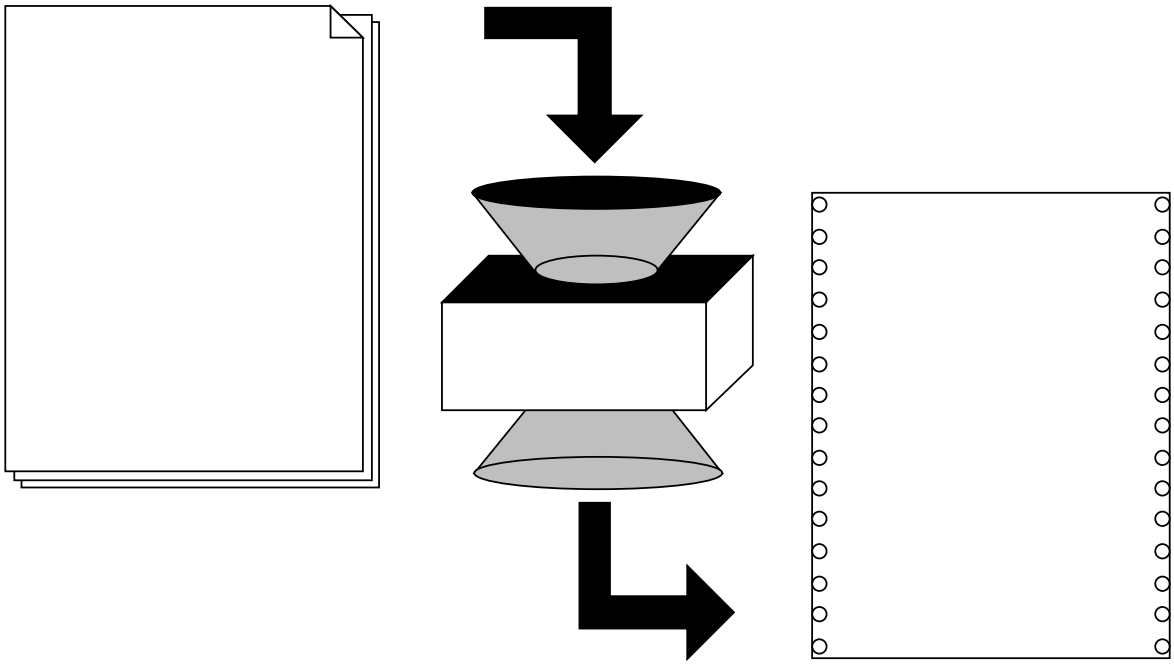

Figure 1: Using PGraphite To Generate A Potentially Persistent Binary Tree ADT.

the non-persistent reference Root, which lasts only as long as the program continues executing), and so the program can save that reference and use it during some later execution to retrieve the root of the tree, or it can pass the reference to any other programs that will need to manipulate the object. Retrieving a persistent object from stable storage is actually nothing more than mapping a persistent reference to a non-persistent reference. Therefore, if a program has a persistent reference to an object that it wants to manipulate, it need only use the GetNPR operation to map the persistent reference to a non-persistent reference. If the persistent object to which the persistent reference is assigned is already in memory, GetNPR will return a non-persistent reference to the in-memory object, which the program can then manipulate normally. If the object is not in memory, GetNPR will retrieve it from stable storage and return a non-persistent reference to it.

As discussed in Section 2.2, we believe that the value of a node includes the values of its attributes, and if an attribute value happens to be a reference to another node, then that other node should persist as well. Therefore, in the course of saving Root in stable storage, when the interface package finds the non-persistent references to other nodes in the root node, it simply uses the GetPID operation to retrieve persistent references to the root node's left and right children, and saves those persistent references with the root node on the stable storage device. This, of course, has the effect of making the children persist. The process continues recursively, and bottoms out when both children are null. Thus, asking for a persistent reference to the root of the tree is equivalent to making the entire tree persist.

```
package Binary_Trees is

-- Non-persistent reference type:
    type Binary_Tree_Node is private;
    Null_Binary_Tree_Node : constant Binary_Tree_Node;

-- Persistent reference type:
    type PID is private;
    NullPID : constant PID;

...

-- Operations to define and manipulate nodes:

    function Create ( TheNodeKind : NodeKindName ) return Binary_Tree_Node;

    procedure PutAttribute ( TheNode : Binary_Tree_Node;
                             TheAttribute : AttributeName;
                             TheValue : String );
    function GetAttribute ( TheNode : Binary_Tree_Node;
                            TheAttribute : AttributeName )
                            return String;

    procedure PutAttribute ( TheNode : Binary_Tree_Node;
                             TheAttribute : AttributeName;
                             TheValue : Binary_Tree_Node );
    function GetAttribute ( TheNode : Binary_Tree_Node;
                            TheAttribute : AttributeName )
                            return Binary_Tree_Node;

...

-- Persistent object abstraction:
    procedure GetNPR ( TheNode : out Binary_Tree_Node;
                       ThePID : in PID );
    procedure GetPID ( ThePID : out PID;
                       TheNode : in Binary_Tree_Node );

 ...

end Binary_Trees;
```

Figure 2: Part Of The PGraphite-generated Interface Package For The Binary Tree Definition.

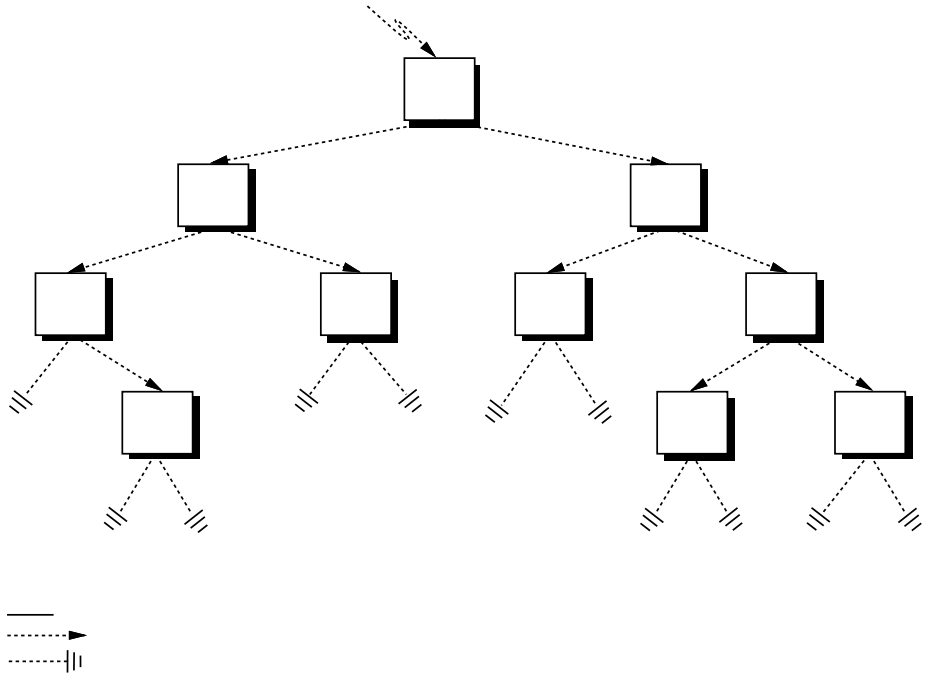

Figure 3: An Instance Of The Binary Tree ADT.

3 Extending PGraphite With Relations and Relationships

The model of persistence embodied by PGRAPHITE has proven to be both useful and very flexible in terms of meeting the needs of our users. In particular, graphs are appropriately used to describe relatively regular, anticipated patterns of connections among entities. That is, the type definition of a graph node indicates exactly what connections one *expects* to find at every such node (where "null" is, in fact, a legitimate node attribute value). We have also found, however, that graphs are less appropriate for describing more irregular, or perhaps unanticipated, patterns of connections. In our binary tree example, for instance, every program that manipulates binary trees expects to find connections to the data stored in the node and to the node's children. It is therefore always correct to set or retrieve the values of any of these attributes. On the other hand, different programs might or might not expect to find such information as execution counts, analysis data, or display information such as how a particular node is depicted (if at all), and if they do expect to find any of this information, they might or might not expect every node to have it.

In experimenting with PGRAPHITE and with the APPL/A system [2], we became convinced that relation and relationship are suitable for representing these kinds of irregular or unanticipated connections. Therefore, we have extended PGRAPHITE to automate the generation of persistent object management support for these types as well. By *relationship*, we are referring conceptually to a (possibly n-ary) connection between entities. For example, a relationship between a source code module and one or more executables created from it might exist. We use the term *relation* to refer to an unordered collection of relationships. A relation type, in our current implementation, is defined on instances of only one type of relationship.

Our model of relations and relationships has the following salient features:

OPERATIONS TO MANIPULATE A RELATIONSHIP TUPLE	
Create	creates a new tuple of a given kind
Get Field	retrieves the value of a tuple field of a given type
Put Field	sets the value of a tuple field of a given type
OPERATIONS TO MANIPULATE A RELATION	
Create	creates a collection of a given type
Insert	inserts a tuple into a relation
Remove	removes a tuple from a relation
Select Tuples	retrieves a (collection of) tuple(s) from a relation
Union	unions two relations
Exclude	removes the intersection of two relations
Iterate	steps through the tuples in a relation

Table 1: The relation/relationship abstraction operations.

- **Equal status for all**

  The traditional relational database model of relations defines relationships to be second-class entities, while relations are first-class entities. That is, relationships are defined only in the context of a specific relation — the same relationship instance cannot be shared by multiple relations, and all manipulation of relationships must be done through the relation.

  In our model, both relations and relationships are first-class objects. Instances of relationships can be created that do not belong to a relation; similarly, relationships can be part of multiple relations or can be shared by other objects. Relationships can connect *any* objects, including other relationships or relations. Table 1 shows the operations defined on both data types.

- **Persistence model**

  We have applied the PGRAPHITE model of persistence to relation and relationship types. Since both relations and relationships are first-class entities, this means that users can request PIDs for instances of either type of object. Because both relations and relationships are composite objects, we apply the reachability-based model to these objects as well. Thus, all relationships in a persistent relation will persist, as will all entities connected by a persistent relationship.[1]

  We note here that since relations and relationships are just abstract data types, we can define graph nodes with relations or relationships as attributes, and can define relationships that connect graphs with other entities. We believe that this further supports our claim that our persistence model (a) is orthogonal, and (b) extends to any arbitrary abstract data type.

- **Constraints**

[1] We have not yet decided on any specific semantics for what happens if objects connected by persistent relationships are destroyed. Several possiblities exist, ranging from stating that if any of the endpoints of a relationship cease to exist, the relationship will be destroyed, to just leaving the relationship in place with no value for the destroyed endpoint. We are currently exploring these options.

A *constraint* on a relation is a condition that must hold true for the relation to be considered to be in a consistent state. The user can statically define constraints and specify constraint violation recovery procedures. Constraint enforcement and relaxation can be dynamically controlled.

- **Triggers**

  Triggers are operations that are invoked when some event occurs. Triggers are generally used to make relationships active instead of passive — for example, a user can define a trigger that compiles and links source code when the source code is modified so that the executable code to which it is related will be up-to-date. We provide a trigger specification mechanism as part of the set of extensions to PGRAPHITE.

We note here that to simplify the process of experimenting with the new relation/relationship abstraction, we ended up defining a separate tool (which we call R&R) to generate implementations of potentially persistent relation and relationship ADTs. The structure of R&R parallels that of PGRAPHITE: it consists of a preprocessor that accepts type definitions in an Ada-like notation and produces corresponding Ada interface packages that implement the specified (relation and relationship) object types, augmented by the PGRAPHITE-style persistence mechanism.[2] The following example illustrates its structure and use.

3.1 Using The Relation/Relationship Abstraction: An Example

To illustrate how this model of relations and relationships can be used, we present an example that is very typical for software development environments. As noted earlier, directed graphs are extremely common structures in software development environments. In fact, many programs written to support the software development process work almost exclusively with graphs. Graphical[3] user interfaces to such programs often must be able to depict nodes of graphs in various ways to show human users of the tool what the tool is doing.

One of the pieces of information a graphical user interface needs is the location of the depiction of any given node on the display device (usually in terms of its X and Y coordinates), if the node is currently displayed. A user interface must therefore be able to determine whether or not any given node is currently displayed, and if so, it must be able to find out the coordinates at which it is displayed (note that any given object may be displayed at multiple locations, depending on the program and on the user interface).

We might therefore use RIDL (**R**elation/relationship **I**nterface **D**escription **L**anguage, the input language to R&R) to define the relation type shown in Figure 4. This relation type will allow a user interface for the binary tree ADT that we produced with PGRAPHITE in Figure 1 to maintain relationships between binary tree nodes and the coordinates at which the nodes are displayed. R&R takes this definition and produces an interface package, complete with operations to create and manipulate instances of relation and relationship types, as shown in Figure 5. Note that although the operations to manipulate relations and relationships are somewhat different from the operations to manipulate graphs (as would be expected), the GetPID and GetNPR operations for translating between the side-by-side name spaces are the same as the ones defined on graph types, and so instances of the new relation and relationship type can be made persistent or retrieved from stable storage in the same way that graph instances are.

---

[2] After we have completed our experimentation, the PGRAPHITE and R&R preprocessors will be combined.

[3] Here "graphical" refers to "using graphics and graphics display devices," not "related to graphs."

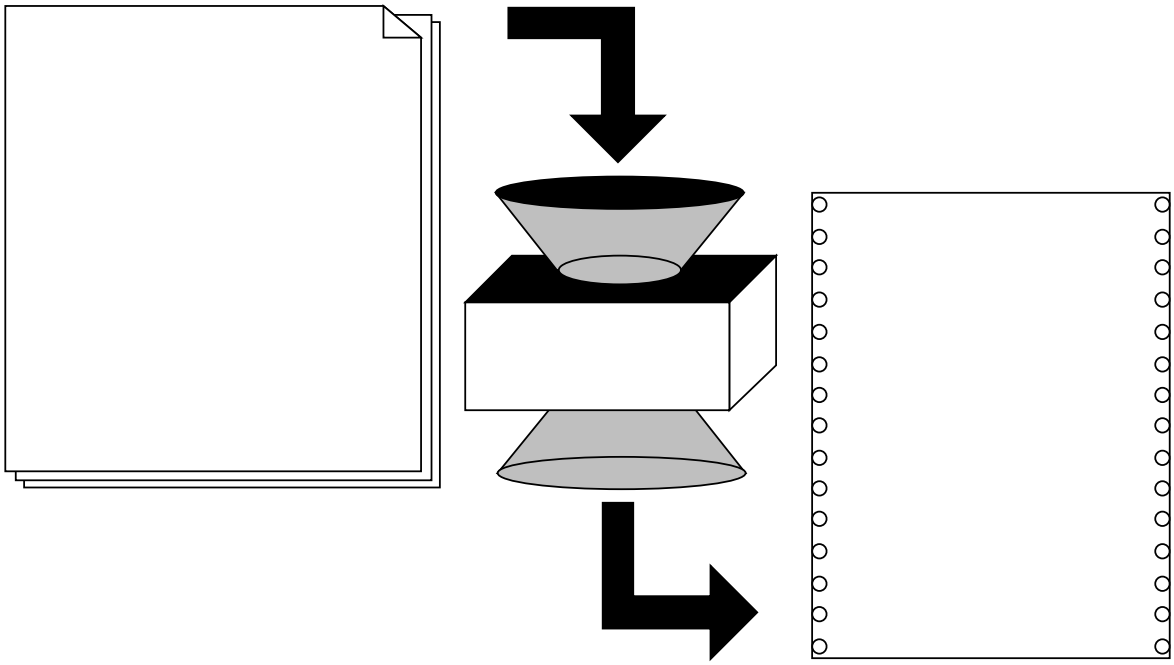

Figure 4: Using R&R To Generate A Potentially Persistent Display Information Relation ADT.

A user interface for the binary tree ADT might be asked to display the tree shown in Figure 3. In a small display window, it might only be able to display the root node and its two children. Therefore, the user interface will only create Display_Coordinates relationships for the root and its children, as shown in Figure 6, and place that limited set of relationship instances into a relation (where the variable Display_Info represents a non-persistent reference to the relation).

There are several points to note here. First, the display information may or may not be made to persist at some point, and whether or not it becomes persistent will depend entirely on the user interface utility, *not* on whether or not the graph persists. Second, all of the display information can be made persistent by making the relation in which it is stored persistent; thus, using the GetPID operation on the relation referenced by Display_Info in Figure 6 will have the expected effect of making all of the relationships in the relation persistent as well (since the value of a relation includes the values of all of the tuples in that relation). Finally, because relationships are first-class objects, it is not necessary to make *all* of the display information persistent — the user interface utility might decide that it only needs to save the display information for the root of the tree, and it can do so simply by asking for a persistent reference to that particular tuple instead of asking for a persistent reference to Display_Info.

With these points in mind, we can now present our approach to limiting the extent of persistence.

```
with Binary_Trees;
package Display_Info_Manager is

-- Non-persistent reference types:
    type Display_Coordinates is private;
    Null_Display_Coordinates : constant Display_Coordinates;
    type Display_Coordinates_Relation is private;
    Null_Display_Coordinates_Relation : constant Display_Coordinates_Relation;

-- Persistent reference type:
    type PID is private;
    NullPID : constant PID;
...
-- Operations to define and manipulate relationships:

    function Create return Display_Coordinates;

    procedure PutField ( TheRelationship : Display_Coordinates; TheField : FieldName;
                         TheValue : Binary_Trees.Binary_Tree_Node );
    function GetField ( TheRelationship : Display_Coordinates; TheField : FieldName )
                      return Binary_Trees.Binary_Tree_Node;

    procedure PutField ( TheRelationship : Display_Coordinates; TheField : FieldName;
                         TheValue : Integer );
    function GetField ( TheRelationship : Display_Coordinates; TheField : FieldName )
                      return Integer;
...
-- Operations to define and manipulate relations:

    function Create  return Display_Coordinates_Relation;
    procedure Insert ( TheRelation : Display_Coordinates_Relation; TheTuple : Display_Coordinates );

    function Select_Tuples ( TheRelation : Display_Coordinates_Relation; TheFieldName : FieldName;
                             Binary_Tree_Node_Value : Binary_Trees.Binary_Tree_Node )
                             return Display_Coordinates_Relation;
    function Select_Tuples ( TheRelation : Display_Coordinates_Relation; TheFieldName : FieldName;
                             Integer_Value : Binary_Trees.Binary_Tree_Node )
                             return Display_Coordinates_Relation;
...
-- Persistent object abstraction:
    procedure GetNPR ( TheRelationship : out Display_Coordinates; ThePID : in PID );
    procedure GetPID ( ThePID : out PID; TheRelationship : in Display_Coordinates );
    procedure GetNPR ( TheRelation : out Display_Coordinates_Relation; ThePID : in PID );
    procedure GetPID ( ThePID : out PID; TheRelation : in Display_Coordinates_Relation );
 ...
end Display_Info_Manager;
```

Figure 5: Part Of The R&R-generated Interface Package For The Display Information Relation Definition.

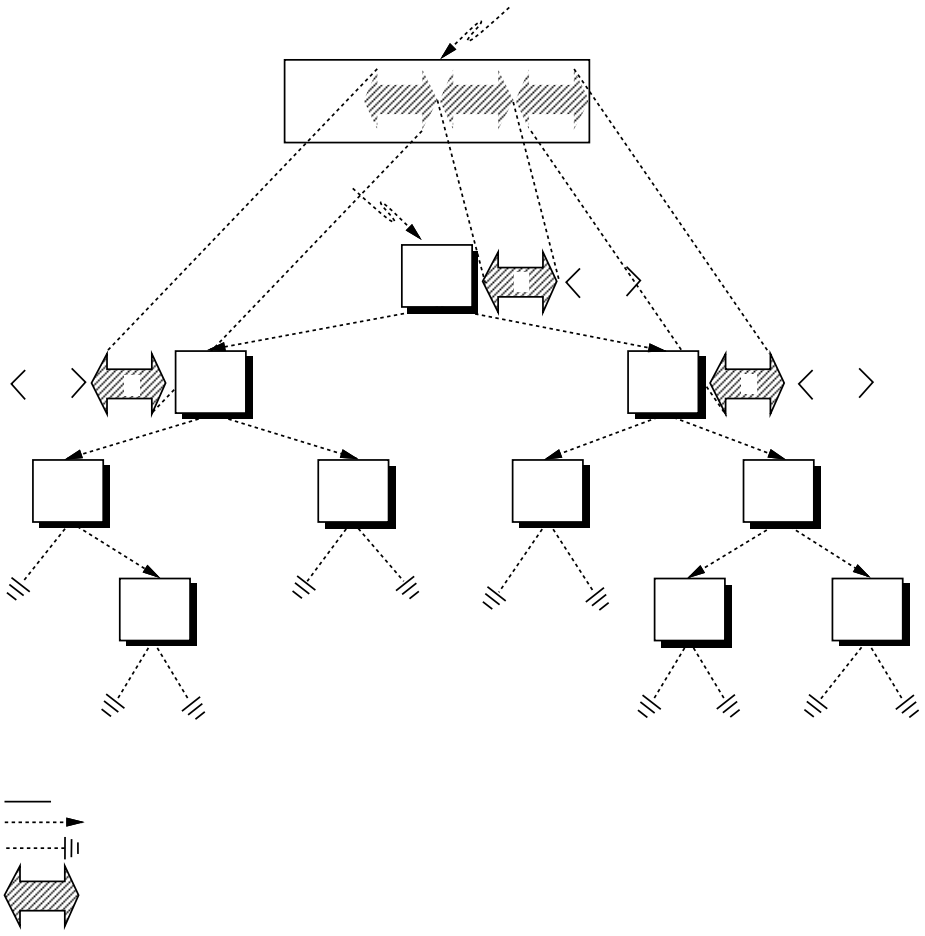

Figure 6: An Instance Of The Binary Tree ADT With Display Position Information.

4 An Approach to Limiting Persistence

One of the lessons that we have learned from experimentation with our PGRAPHITE prototype is that although a reachability-based definition of persistence is often appropriate, applications sometimes need to specify "bounds" on what actually persists. That is, certain information that could be viewed as "part of," or at least "associated with," a graph node should not necessarily become persistent just because that node does. The user should have control over the specification of these boundaries. One obvious way of providing such a capability is to force the user to differentiate between persistent and non-persistent parts of objects when defining the object's type (as has been done, for example, in the Ergo system [1]) by explicitly designating some components as "persistent" and others as "non-persistent." The problem with this solution is that it violates the orthogonality property that we desire for persistent object management systems by introducing distinctions based directly and solely on persistence properties into type definitions. It also means that the same object instance will look different after being retrieved from stable storage than it did before it was stored, since some of its parts may no longer exist. What we want instead is a principled way of controlling the bounds of persistence that preserves orthogonality and transparency. In particular, we want all persistent object instances to look the same, independent of whether or how often they have been saved or retrieved from stable storage.

The approach that we have developed is based on the observation that objects often have two kinds of parts: core parts and associated parts. *Core parts* are those components of an object that must always be present for the object to have any meaning. For example, any node in a binary tree must, by definition, have data and must have (possibly null) left and right children. *Associated parts* are pieces of information that some instances of a type have while others do not. As shown by our example in Section 3.1, a node in a graph that is being depicted on a display device might have associated with it the coordinates on the display at which the node appears, but no such information exists for nodes that are not displayed. Associated information is not less important than core data — the difference, for example, is that our user interface utility does not expect every node to have position information, and nodes that don't have it are still meaningful.

This distinction provides a basis for principled introduction of "bounds" or "limits" in our persistence mechanism. We take the position that core attributes must persist if the object to which they belong persists, and so should be subject to a reachability-based persistence model. Associated parts, on the other hand, need not persist. The user interface utility, for example, might or might not need to preserve any or all of the position information it creates, depending perhaps on whether or not the user asked to have some parts (or all) of the display saved for use at a later time. Therefore, associated parts should be *potentially* persistent — that is, they can be made persistent explicitly, but they will not become persistent automatically simply because the object with which they are associated persists.

This leaves us with a very natural way of letting the user specify the bounds of persistence that does not violate orthogonality or transparency. The user simply defines associated parts using relationships, while core attributes are represented as "normal" node attributes. Since relationships are potentially persistent entities, the persistence of associated parts is thus under explicit program control. Such control can be exercised at the level of individual associations (by making particular relationship instances persist) or at the level of collections of associations (by making a relation that contains a set of relationship instances persist).

Acknowledgements

We would like to thank Alex Wolf and Stan Sutton for all the helpful comments they have provided us over the years.

References

[1] Peter Lee, Frank Pfenning, Gene Rollins, and Dana Scott. The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments. In *Proceedings of SIGSOFT '88: Third Symposium Software Development Environments*, pages 25–34, November 1988.

[2] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, August 1990.

[3] Peri L. Tarr, Jack C. Wileden, and Alexander L. Wolf. A Different Tack To Providing Persistence In A Language. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.

[4] 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 Enviroment Architecture. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, November 1988.

[5] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An Experiment in Persistent Typed Object Management. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, November 1988.

[6] Alexander L. Wolf. Abstraction Mechanisms and Persistence. In *Proceedings of the Fourth Workshop on Persistent Object Systems*, September 1990. (To appear.).