Name Management and Object Technology for Advanced Software

Alan Kaplan
Jack C. Wileden
{kaplan,wileden}@cs.umass.edu

CMPSCI Technical Report 93–83
November 1993

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

Appeared in Proceedings of the International Symposium on Object Technologies for Advanced Software (Lectures Notes in Computer Science 742) Kanazawa, Japan, November 1993

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under grant MDA972-91-J-1009. The content does not necessarily reflect the position or policy of the U.S. Government and no official endorsement should be inferred.

Abstract

Name management is so fundamental to every aspect of computing that it is frequently overlooked or taken for granted. Our research is aimed at developing both *models* to improve understanding and *mechanisms* to improve practical application of name management approaches in various computing domains. One domain that seems to have particularly strong connections to name management is object technology for advanced software. Object technology has already proven very useful in our investigation of name management models and mechanisms. We also see great potential for beneficial application of improved name management mechanisms to object technology for advanced software. In this paper, we first outline our overall approach to research on name management and discuss some specific name management concerns arising in object technology for advanced software. We then illustrate the application of object technology in our efforts to construct name management models and mechanisms. Finally we give an example of how enhanced name management mechanisms might be incorporated into a representative instance of object technology for advanced software.

1 Introduction

Name management is so fundamental to every aspect of computing that it is frequently overlooked or taken for granted. By *name management*, we mean how a computing system allows names to be established for objects, permits objects to be accessed using names, and controls the availability and meaning of names at any point in time.

Names are used, and hence name management mechanisms are needed, for many purposes in all kinds of computing systems. Traditional programming languages rely on names for structuring programs as well as for managing data flow and control flow. Different languages use different scoping rules, and occasionally other mechanisms, to manage names and where and how they are used within a program. In operating systems, names are employed by users and processes to access commands and files. Again, various mechanisms, such as the environment variables, search paths and hierarchical directory structure found in Unix,[1] are used to manage the availability and meaning of names. Networks and distributed systems rely on names to support sharing and communication of information and resources among users and processes. Name servers are one mechanism used in managing the availability and meaning of names in these systems. Even in relational database management systems, where associative retrieval is the primary means of access, names are used to identify and organize databases, relations and attributes.

Despite, or perhaps because of, its ubiquitous role in computing, name management has received relatively little attention from computing researchers. We believe, however, that as systems become larger and more complex, underlying name management mechanisms will increasingly become focal points with respect to the ability to develop, maintain, integrate and port systems. Moreover, as domains such as programming languages, operating systems and

[1] Unix is a registered trademark of Unix System Laboratories, Inc.

database management systems continue to converge in coming years (a trend particularly notable in, and probably accelerated by, object technology), the incompatibility of their respective approaches to name management will become increasingly apparent and problematic. We are, therefore, actively pursuing a broad-based investigation of name management. Our research is aimed at developing both *models* to improve understanding and *mechanisms* to improve practical application of name management approaches in various computing domains.

One domain that seems to have particularly strong connections to name management is object technology for advanced software. In fact, we first encountered many of the issues and problems that have motivated our investigation of name management while working on, or with, object-oriented programming languages, object-oriented databases and software object bases. At the same time, object technology has already proven very useful in our investigation of name management models and mechanisms. In addition, we see great potential for beneficial application of improved name management mechanisms to object technology for advanced software.

In this paper we examine these various connections between name management and object technology for advanced software. We lay the groundwork, in Section 2, by briefly outlining our overall approach to research on name management and describing some preliminary models that we have developed. Then, in Section 3, we discuss some specific name management concerns arising in object technology for advanced software. In Section 4 we illustrate the application of object technology in our efforts to construct name management models and mechanisms. Finally, in Section 5, we give an example of how enhanced name management mechanisms might be incorporated into a representative instance of object technology for advanced software. We conclude with a summary and an indication of ongoing and future directions for our investigation of name management and object technology for advanced software.

2 Name Management Basics

Our research on name management has two primary facets. One facet is the development of *models* that can serve as a basis for enumerating, explicating or evaluating various approaches to various aspects of name management. The long-range goal of this facet is to provide a foundation for fundamental understanding of name management and its role in all domains of computing. The other facet of our research is the definition and implementation of, and experimentation with, prototype *mechanisms* for particular aspects of name management. Here the long-term goal is to provide a comprehensive set of powerful, flexible, uniform and broadly applicable mechanisms that will ease the construction and maintenance of (especially large and complicated) computing systems. These two facets are closely interrelated, as we will demonstrate in Section 4, since modeling both guides and is influenced by experimentation with mechanisms. The initial steps in our investigation of name management were directed toward modeling, however. In the remainder of this section we describe some models that have served as a starting point for ongoing iterative development of both models and mechanisms, and in particular have influenced our subsequent work on name management and object technology.

A first step in modeling is to establish basic terminology. We consider the following to be primitive, fundamental concepts related to name management:

object: An item of interest in a given setting. In a computer program, for instance, variables, functions, procedures, statements, etc. could all be examples of objects.

name: An identifier used to reference, access or manipulate an object.

binding: In its simplest, most basic form a (name,object) pair. Given the availability of binding (A, B), it is possible to use name A to reference, access or manipulate object B.

binding space: A collection of bindings.

context: A collection of bindings that is currently available for use in referencing, accessing or manipulating objects. A context may consist of, or be formed from, one or more binding spaces, or parts of binding spaces.

resolution: An operation that, given a name and a context, returns the object(s) bound to the name.

Given this basic terminology, we would like to have a means of characterizing and differentiating distinct approaches to name management. Informally, a particular approach can be described in terms of a particular set of choices regarding such things as: what kinds of objects can be named, the form of names, how bindings may be established, how binding spaces may be created and manipulated, and how contexts may be formed and manipulated. In an effort to formalize this view, we are developing a reference model, or classification framework, for name management that we call the NamingSpace reference model. In its current form, the NamingSpace reference model characterizes the universe of approaches to name management as a five dimensional space. Here we outline this classification framework by listing and briefly describing the five dimensions. For each of the five dimensions, we indicate some of the questions addressed in that dimension and identify a few representative points corresponding to one or more of the questions.

The Object Dimension: A primary question in this dimension is: What kinds of objects can be named? Representative points in the object dimension that might correspond to answers to this question include:

- **Transient, persistent or both**: A typical programming language would only allow naming of transient objects, while an operating system might only allow naming of persistent objects (such as files) and an object-oriented database or a database programming language (e.g., Napier [DCBM89]) might allow both.
- **Names, bindings, binding spaces, contexts**: Different choices are made in different name management approaches. For example, traditional programming languages allow none of these to be named, while in Unix names, binding spaces and contexts can all be named, but bindings cannot.

The Name Dimension: Among the questions in this dimension are: Are names first-class? What form can names take? Representative of points corresponding to the latter question is:

Simple identifier ↔ Hierarchical ↔ Multi-level hierarchical: Simple variable names in a typical programming language generally take the form of simple identifiers. Qualified variable names (e.g., record fields or object methods) in a programming language often take a hierarchical form. Internet path names, with their multiple different separators, are an example of multi-level hierarchical names.

The Binding Dimension: Among the questions in this dimension are: Are bindings first-class? How are bindings established? When are bindings established? What information is included in bindings? Representative points corresponding to the latter question include:

Simple (name, object) pairs: The most common answer – typical in most programming languages, operating systems, etc.

Support for typed objects/overloading/polymorphism: In effect, bindings are (name, type, object) triples [LB88]. Applies, for example, to certain classes of objects in Ada.

Support for type and mutability differentiation: The Napier language, for instance, views bindings as quadruples of the form (name, type, mutability, object) [MABD90].

Arbitrary subsets of object properties: In the limit, this supports associative access (e.g., [Bow90]).

The Binding Space Dimension: Among the questions in this dimension are: How are binding spaces created and manipulated? How are binding spaces related to one another? Representative of points corresponding to the latter question is:

Flat space $\leftrightarrow$ Hierarchy/tree $\leftrightarrow$ Directed graph: Examples of all of these can be found in Unix.

The Context Dimension: Some of the basic questions in this dimension are: How can contexts be formed? How can contexts be manipulated? What determines which context(s) will be used in interpreting a given name? Representative answers to the first of these questions include:

A single binding space: A Unix working directory, for example, can be a context.

Nested scope: Many programming languages, as well as the combination of directories represented by a Unix search path, provide examples of context formation based on nesting.

Arbitrary combination of (subsets of) binding spaces: Two examples of this approach to context formation are PIC/Ada [WCW85, WCW89] and Napier environments [Dea89].

The NamingSpace reference model is still evolving as we seek to capture the range of name management options more completely and accurately. Nevertheless, the model has already served as a basis for some experimentation with a variety of name management mechanisms. That experimentation, in turn, is leading to further insights about the range of options and hence will contribute to the evolution of the NamingSpace reference model. We illustrate this interplay between model and mechanism in Section 4.

Another class of useful models provides rigorous foundations for defining the semantics of approaches to, or components of, name management. Along these lines, we have been exploring some models based on set theory, using sets of bindings to model binding spaces and various set theoretic operations as basic context formation operators. Detailed treatment of these models is beyond the scope of the current paper, though their influence will be seen in some of the mechanisms described in Section 4.

A few other researchers have proposed models that are related, to some extent, to our name management modeling work. In the earliest such effort that we are aware of, Fraser developed a formalism, based on a simple extension of Church's λ notation [Chu41], that can be used to describe context and manipulations on context [Fra71]. The model includes methods for concatenating and composing contexts, and the implication of the formalism is that it could provide a rigorous semantics for the naming aspects of programming languages and file systems. In [CP89], Comer and Peterson present a formal model of name resolution in distributed systems. This model is very specific to distributed systems and, like the Fraser model, is directed primarily toward providing a formal semantics for certain aspects of name management. Not only are both of these models focused on specific domains, but, in terms of the NamingSpace reference model, both address only a subset of the dimensions, and even so they only cover a subset of the points that can be identified in those dimensions. The

classification model presented by Morrison, Atkinson, Brown and Dearle in [MABD90] is more broadly focused, having been developed to categorize the various binding mechanisms found in languages, including database programming languages, and operating systems. Their model is still somewhat more narrowly focused than the NamingSpace reference model, however, presenting only four dimensions (two of which are in fact subsumed by the binding dimension of the NamingSpace model) and identifying only two points on each dimension.

3 Name Management and Object Technology

As we noted earlier, name management and object technology for advanced software seem to have a particularly strong connection. On reflection this is not too surprising, since in object-oriented computation the identification of objects and operations by their names is at the heart of almost every computational step. As a result of their strong connection, however, we find some specific name management concerns arising in object technology for advanced software. Thus, in object-oriented programming languages the importance of names in identifying classes, object instances, and operations is immediately apparent, while the various proposed approaches to resolving name clashes in (especially multiple) inheritance mechanisms (see, e.g., [Knu88]) exemplify name management concerns that are specific to object-oriented computing. The addition of persistence, yielding object-oriented databases or database programming languages, places an even higher premium on name management, since more names, with longer durations, become relevant to a program, which leads to a higher probability of name clashes and an increased need for name management. Software object bases, which for our purposes here have much in common with object-oriented databases, place similarly heavy demands on name management.

Another indication of the strong connection is the fact that we first encountered many of the issues and problems that have motivated our investigation of name management while working on, or with, object technology for advanced software. In the remainder of this section we briefly survey some examples of name management issues that arise in object-oriented programming languages, object-oriented databases and software object bases, highlighting those that have arisen in our own experience. This brief survey elaborates on the nature of the connection between name management and object technology for advanced software, indicates some of the problems that we and others have encountered in existing approaches to name management, provides a starting point for populating the NamingSpace classification framework with example instances of particular choices in particular dimensions, and motivates our ongoing research in this area, which is discussed in subsequent sections.

3.1 Object-Oriented Programming Languages

Names are used in many ways in traditional programming languages. The name management mechanisms employed, however, have tended to cover a rather limited set of points along the various dimensions of the NamingSpace classification framework. Languages have always had their own idiosyncratic syntactic restrictions on how names may be formed. Bindings are usually defined via declarations of some kind, while binding spaces typically correspond to

program units, such as procedures, functions or blocks. In the many languages that use static, nested scoping to control visibility, context is defined by iteratively considering the most deeply nested to the least deeply nested binding space, starting from the current program counter location, and augmenting the context with all bindings whose name components are not found in any of the bindings already included in the context. The name resolution function simply returns the object with the given name relative to this context.[2] We, and many others, have often encountered problems resulting from the limits that this approach to context formation imposes on the ability to manage control flow and data flow in programs [CWW80]. Some recent languages, such as Ada and Modula, have provided additional mechanisms for defining binding spaces (e.g., the Ada package) and for forming contexts (e.g., the Ada `with` and `use` constructs). Even these provide relatively limited flexibility, though, so various proposals have been advanced to support yet additional control over context formation (e.g., [WCW89]).

Most object-oriented programming languages are based on one or more of the traditional languages, and hence inherit some or all of (the shortcomings associated with) their name management approaches. Object-oriented programming languages introduce some additional name management issues, however. One issue is the impact of inheritance, especially multiple inheritance, on binding space and context definition. It is worth noting that the standard approach to name resolution in a single inheritance structure is essentially the same as the nested scope mechanism found in traditional languages, with successive subclass definitions corresponding to successively nested blocks (binding spaces) and context formation defined by iterative consideration of successive superclasses. In a multiple inheritance mechanism, though, some additional semantics for handling name conflicts is typically added. In C++, for instance, a class is permitted to inherit more than one binding (member) with the same name from other binding spaces as long as that name is never actually referenced. Eiffel, on the other hand, does not allow a class to inherit multiple bindings with the same name.[3] Finally, CLOS relies on the specification order of inherited binding spaces to resolve name conflicts.

Another issue concerns the use of names, bindings, binding spaces and context in the construction of programs. While object-oriented programming languages clearly facilitate the description and construction of complex entities in programs, sophisticated capabilities are still lacking for the description and construction of complex programs. Shortcomings in current approaches to name management are one aspect of the overall problem. We have repeatedly found name management shortcomings to be the cause of problems in building large systems, most recently while attempting to install a large C++ system imported from another laboratory.

In C++, programs are typically built by compiling separate modules and linking the modules together to form a single main program. To facilitate separate compilation, header files (whose names, by convention, end with `.h`) containing class definitions are included into source files using the `#include` directive. One problem with this approach is that it is difficult to flexibly and arbitrarily specify context for header files. In particular, most systems typically provide a single context formation operation that we call *UnionOverride* (which Meyer calls

[2] Some languages allow names to be overloaded by type.

[3] To remove a name clash, the `rename` clause can be used in the heir class.

```
#include <foo.h>
#include <bar.h>
main () {
   Foo f;
   Bar b;
   ...}
```

Figure 1: **C++ Code Fragment**

"overriding union" in [Mey88]). The *UnionOverride* operation forms a new context given two directories (or binding spaces) in exactly the same way that nested scope forms a context from two nested blocks.[4] That is, the resulting context consists of all the bindings in the first directory and all the bindings in the second directory whose name components do not exist in any of the bindings in the first directory. In general, therefore, the order of the given directories is significant.

As a (very simplified) example of the problems that we have encountered, consider the C++ source code fragment in Figure 1, which includes class definitions contained in `foo.h` and `bar.h`. Suppose that there are two directories named `DIR1` and `DIR2`, each containing files named `foo.h` and `bar.h`. Now suppose a software developer wants to form a context that uses the file `foo.h` contained in directory `DIR1` and the file `bar.h` contained in directory `DIR2`. The *UnionOverride* operation clearly does not allow the desired context to be formed. In the C++ compilation command:

```
CC -IDIR1 -IDIR2 main.C
```

the `-IDIR1 -IDIR2` specification results in the `foo.h` and `bar.h` from `DIR1`. Switching `DIR1` and `DIR2` results, of course, in the `foo.h` and `bar.h` from `DIR2`. To create the desired context, the software developer is faced with two alternatives. The first choice involves changing the references in the source code (for example, changing `#include <foo.h>` to `#include <DIR1/foo.h>`). The second choice involves changing the file and directory structure to form the required context. Clearly neither of these solutions is desirable since modifying source code or modifying the file and directory structure, especially in large systems, is an error-prone, tedious and expensive process.

3.2 Object-Oriented Databases and Software Object Bases

From the perspective of name management issues, object-oriented databases and software object bases have a great deal in common. Indeed, vendors of object-oriented databases often cite software object bases as one of their intended applications, and some software object bases (e.g., Triton [Hei92]) have been implemented by customizing an object-oriented database. We

[4] In other words, iterative application of the *UnionOverride* operation directly corresponds to the standard notion of nested scope in programming languages.

therefore address name management issues related to object-oriented databases and software object bases jointly here.

In conventional object-oriented programming languages, the name management mechanisms for transient and persistent objects are quite separate. For transient objects (e.g., classes, objects), name management is dictated by the syntax and semantics of the language. Persistent objects are typically implemented using files; therefore, name management for persistent objects is provided by the underlying file system. Because distinct name management mechanisms are provided depending on the persistence property of an object, programmers are forced to use different, often conflicting, name management mechanisms. Such a separation is confusing and prone to cause errors in the development of advanced software.

Recent developments in object-oriented databases have attempted to make persistence an orthogonal property of types, thus obviating the need for files. While this approach addresses the impedance mismatch problem by providing a single, unified type model, there is little evidence of powerful, flexible and uniform approaches to name management in such systems. For example, many commercially available systems, such as GemStone [BOS91], ONTOS [ONT93], ObjectStore [LLOW91] and O_2 [Deu91], provide C++-based type models for both transient and persistent objects; however, these systems tend to provide rather limited support for name management. For instance, in GemStone and ONTOS, binding spaces may be organized hierarchically, but context formation is restricted to *UnionOverride* semantics. In ObjectStore, there are two kinds of names — names for databases (binding spaces) and names for objects. Database names are external to the system and are managed by the underlying file system, while object names are handled by a separate name management mechanism. Moreover, because database names cannot be bound in other databases, ObjectStore is restricted to a single level hierarchy of binding spaces.

The major alternative to basing a software object base on an object-oriented database is to build it as an extension to a (possibly not object-oriented) programming language. Generally such an extension involves adding persistence to the programming language. Software object bases of this kind, by integrating programming language and database (and/or operating system) capabilities, give rise to interesting name management problems that are essentially similar, however, to those arising in object-oriented databases. In fact, our interest in the general topic of name management grew out of earlier work on PGraphite [WWFT88, TWW89] and R&R [TWC90], two complementary prototype systems that together implemented (potentially) persistent relation, relationship and directed graph types as extensions to Ada. The goal of this work was to provide a software object base for use in Arcadia software environments [Kad92]. Experimental use of these systems highlighted a number of name management problems, arising primarily from the need to simultaneously manage naming of both transient and persistent objects. Among these, the two most fundamental were related to:

1. the lack of integration between names for transient (programming language-internal) and persistent (programming language-external) data objects, and
2. the incommensurate, and inadequate, mechanisms for controlling exactly which names (of both transient and persistent objects) were available for use (at any given point)

within a program.

Having noted these problems while experimenting with PGraphite and R&R, we soon discovered that similar problems existed in most other software object bases as well as in most object-oriented databases.[5]

Although these problems are prevalent in present generation systems, the desirability of improved approaches to name management in both object-oriented databases and software object bases seems to be widely recognized. For instance, various proposals for architectures for object-oriented database systems acknowledge the need for name management services, though only a few details are provided [OMG92],[WBT92]. Similarly, emerging standards for interfaces to software object bases [Tho89] and reference architectures for software engineering environments [ECM91] also allude to name management services as necessary components while offering few specifics. Thus the prospects for future developments in the area of name management and object technology for advanced software seem quite promising.

4 Object Technology Applied to Name Management

Object technology has proven very useful in our investigation of name management models and mechanisms. We have found the concepts of object orientation to be quite natural for formulating models of specific approaches to name management. In particular, they have helped us to create concrete descriptions of various components of an approach and how those components are interrelated. In addition, the fact that an object-oriented description, and especially the interrelationships among its components, can be so easily altered has facilitated our exploration of related alternative approaches. This has been particularly valuable for our efforts at exploring and evolving the NamingSpace classification framework. At the same time, because an object-oriented description can be so straightforwardly transformed into code, then combined with an experimental harness and executed, we have been able to easily experiment with running instances of the various approaches. Such experimentation with prototype mechanisms provides feedback and suggests additional approaches, which can in turn be described and then, almost as soon as the description is completed, experimentally evaluated. Thus object technology is contributing to our definition, evaluation and evolution of both models and mechanisms, while facilitating the crucial interplay between model building and mechanism development.

In this section we present some examples of our use of object technology in investigating name management models and mechanisms. These examples serve to illustrate some interesting aspects of some alternative name management approaches, or points in the NamingSpace, as well as the value of object technology when applied to name management.

[5] Others have made similar observations. For instance, in their list of requirements for database programming languages [AB87], Atkinson and Buneman appear to refer to both of these problems (requirements (4) and (9)).

4.1 Modeling and Implementing a Basic Name Management Approach

We begin by considering a basic approach to name management, corresponding to some of the most straightforward and elementary choices that can be made in the various dimensions of the NamingSpace. As a first step in studying this approach, we produce a simple graphical depiction of it, as shown in the diagram of Figure 2. The graphical notation that we employ is

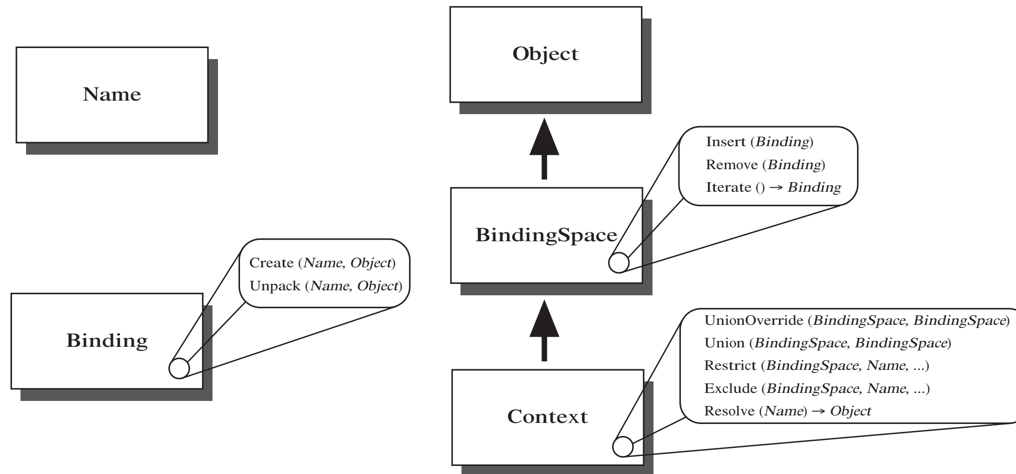

Figure 2: Model of a Basic Name Management Approach

an informal representation of an object-oriented structure, in which classes are represented as labeled rectangles, their associated operations are given as lists of operation signatures, and subclass relationships are represented by directed arcs from subclass to superclass. Some of the details of this notation, notably the way that operations are described, are influenced by the fact that we eventually implement our models in C++. Some ramifications of this will become apparent shortly.

The graphical depiction expresses many of the properties of this particular approach relatively clearly. Some other properties are not so evident from the diagram, however. Following the organization of the NamingSpace classification framework, the properties of the basic approach, and how (and how well) they are depicted in the diagram, are as follows:

The Object Dimension: The basic approach depicted in Figure 2 does not distinguish persistent and transient objects, so this approach does not restrict naming based on persistence. The diagram does, however, indicate that binding spaces and contexts can be named, while names and bindings cannot. This information is derived from the signature of the **Create** operation of the **Binding** class and the presence (or absence) of subtype arcs leading to class **Object**.

The Name Dimension: The diagram for this basic approach gives no information about the **Name** class beyond the fact that it exists. Thus, at the current level of abstraction, this approach takes no position about any questions in this dimension.

The Binding Dimension: The basic approach depicted in Figure 2 uses simple (name,object) pairs as bindings, as indicated by the signatures of the operations of class **Binding**. The **Create** operation creates a binding given instances of **Name** and **Object**, while the **Unpack** operation returns the **Name** and **Object** instances contained in a binding.

The BindingSpace Dimension: The diagram for this basic approach indicates, via the operations (and their signatures) of class **BindingSpace**, that binding spaces are composed of instances of class **Binding**, and that operations exist for inserting, removing and iterating over bindings in a binding space. The restriction that no two bindings in a binding space may have the same name component, which is intended to apply in this approach, is not evident from the diagram, however. (A richer notation that supported some rudimentary indication of operation semantics, such as pre- and post-conditions or even exceptions, could correct this shortcoming.) Since a binding space is a nameable object, binding spaces can be organized in a flexible manner – e.g., a binding space may contain multiple binding spaces, thus producing a hierarchical organization, and a particular binding space can be contained in (and hence shared by) multiple binding spaces.

The Context Dimension: In the basic approach depicted in Figure 2, contexts are formed from binding spaces or, given that **Context** is a subclass of **BindingSpace**, other contexts. Name resolution is an operation specific to contexts in this basic approach, so the context to be used in resolving a given name will be explicitly indicated. The context formation and manipulation operations are:

UnionOverride: Forms a new context given two binding spaces. The resulting context consists of all the bindings in the first binding space and all the bindings in the second binding space whose name components do not exist in any of the bindings in the first binding space.

Union: Used to combine two binding spaces. Since binding spaces must be unique with respect to the name components in the bindings, the operation fails when the binding spaces contain bindings whose name components are the same, but whose objects are not identical.

Restrict: Forms a new context by extracting from a given binding space the bindings whose name components match a specified set of names. The resulting context is a subset of the original binding space.

Exclude: Forms a new context by excluding from a given binding space the bindings whose name components match a specified set of names. The resulting context is a subset of the original binding space.

Of course, the diagram does not indicate any of the semantic details of these operations, although the names and signatures of the operations may suggest the intended inter-

pretations in most cases. (Again, a richer notation that supported some rudimentary indication of operation semantics could correct this shortcoming.)

Our choice of the particular set of context formation operations included in this basic name management approach is, in part, a response to problems of the kind illustrated by the example corresponding to Figure 1. To assess how successfully this set of operations overcomes those problems, which arise in name management approaches where *UnionOverride* is the only context formation operation, we could consider some possible ways in which they might be used to form a desired context for header files in C++, as discussed in the earlier example. A set of possible context specifications to be used in the C++ compilation command, in place of the options discussed earlier, appears in Figure 3. In the figure, subscripted lower case

(1) $c_1.UnionOverride(DIR1, DIR2)$

(2) $c_1.UnionOverride(DIR2, DIR1)$

(3) $c_3.UnionOverride(c_1.Restrict(DIR1, foo), c_2.Restrict(DIR2, bar))$

(4) $c_3.Union(c_1.Restrict(DIR1, foo), c_2.Restrict(DIR2, bar))$

(5) $c_2.UnionOverride(c_1.Restrict(DIR1, foo), DIR2)$

(6) $c_2.UnionOverride(c_1.Restrict(DIR2, bar), DIR1)$

(7) $c_2.UnionOverride(c_1.Exclude(DIR2, foo), DIR1)$

(8) $c_2.UnionOverride(c_1.Exclude(DIR1, bar), DIR2)$

(9) $c_1.Union(DIR1, DIR2)$

Figure 3: Some Possible Context Specifications

letters denote instances of **Context**, uppercase identifiers denote instances of **BindingSpace**, lowercase identifiers denote instances of **Name**, binding spaces $DIR1$ and $DIR2$ represent the directories `DIR1` and `DIR2` and names foo and bar represent the files `foo.h` and `bar.h`. Thus, for instance, possibility (1) of Figure 3 corresponds exactly to the context specification contained in the C++ compilation command:

```
CC -IDIR1 -IDIR2 main.C
```

Of course, since they employ only *UnionOverride*, the contexts represented by possibilities (1) and (2) of Figure 3 both fail to satisfy the software developer's requirements as discussed in the earlier example. The remaining possibilities all accomplish the desired result, however, with the choice among them depending upon some further assumptions about the situation. For instance, suppose that the software developer wishes to form a context containing *only* the foo from $DIR1$ and the bar from $DIR2$. Using the *Restrict* operation in addition to *UnionOverride*, this can be accomplished using possibility (3). Alternatively, the same result

can be achieved using *Restrict* and *Union*, as in possibility (4). Suppose, however, that in addition to having the preferred bindings for foo and bar, the software developer wants (or is willing) to have a context containing all of the other bindings in *one* of the directories. Then, depending upon *which* of the other directories is wanted, either possibility (5) or (6) will also produce an acceptable context. If instead the software developer wants (or is willing) to have a context containing *all* of the other (non-conflicting) bindings in *both* of the directories, along with the preferred bindings for foo and bar, using the *Exclude* operation as in possibilities (7) or (8) will accomplish the goal. Finally, suppose that the software developer does not know precisely what bindings are in $DIR1$ and $DIR2$,[6] but believes that the desired context should contain all the bindings in both[7] and wants to know if the resulting context is ambiguous. The *UnionOverride* operation is not sufficient, since the bindings in one binding space may override the bindings in the other binding space, but a context specification using the *Union* operation, like possibility (9), will suffice. Of course, given the hypotheses of our example, possibility (9) would raise an exception, since $DIR1$ and $DIR2$ each contain different bindings with the same name. This is precisely what the software developer wants, however, since it provides a notification that there is an ambiguity in the context specification.

4.2 Experimentation: Implementation and Variations

As the preceding discussion demonstrates, the graphical depiction of this basic approach to name management provides a significant amount of information and permits us to begin assessing its properties, and thereby positioning it in the NamingSpace. In order to study it further, we can quite straightforwardly translate from the graphical representation into an implementation in an object-oriented programming language. As noted earlier, we have been using C++ as our implementation language. The C++ implementation is, in fact, a more detailed representation of the approach than was the depiction in our simple graphical notation. It can, in particular, capture the semantic details of the operations associated with the various classes, which can only be inferred from the names and signatures appearing in the listings of operations in the graphical depiction. Of course, in the C++ implementation these details are buried in the coding of the specific methods. While this still does not facilitate reasoning about properties of a modeled approach as much as a more declarative representation might, it is certainly better than not capturing the information at all.

Naturally, the most important thing about being able to easily translate from model to implementation is that it permits us to experiment quite directly with any approach that we might choose to model. Initially we have carried out this experimentation by combining the C++ class definitions derived from a model of a name management approach with a simple testing harness, also implemented in C++, and then executing some simple scenarios that exercise the name management capabilities. In Section 5 we discuss steps directed toward providing a larger and more realistic testbed in which to experiment with implementations of various name management approaches. Even the simple experimentation using the testing

---

[6] In a realistic case, this might occur because the binding spaces are very large or change frequently.

[7] Or possibly more realistically all but some specified subset, which could be described using *Exclude*.

harness can yield some useful insights and information, however. It allows us, for instance, to empirically investigate the use of context formation operations by actually running experiments corresponding to possibilities like those enumerated in Figure 3.

Experimenting with the implementation of the basic approach described above reveals some interesting characteristics and suggests several possible variations. One striking characteristic of this approach is the central role that the **Binding** plays; naming is accomplished by explicitly creating a binding and then explicitly inserting that binding into a binding space, for example. Another characteristic, which may be considered more serious, concerns the name resolution function. Since the **Resolve** function returns any possible nameable object, the function's return type is **Object**. In using the C++ implementation, we found that this means the programmer must cast the result of the **Resolve** function into the appropriate type. Figure 4 illustrates the resolution of an instance of **Person** in some context. Based on these

```
Name n;
Person *p;
Context some_context;
...
p = (Person *) some_context.Resolve (n);
```

Figure 4: Casting the Result of the Resolve Function

two observed characteristics, we might designate the basic approach described in Figure 2 as a *type-weak, binding-centered* approach to name management.

A possible alternative worthy of some consideration might be called a *type-preserving, nameable-object-centered* approach. A modification to the model of Figure 2 that yields such an approach is shown in Figure 5, where the small box within the rectangle representing a class stands for a generic parameter and the dashed arrows mean "is an instantiation of".

The modification centers around the definition of a new class, i.e., **NameableObject** and the removal of the **Binding** class. **NameableObject** is a generic (or parameterized) class that inherits from the **Object** class. A C++ definition of such a generic class is shown in Figure 6. This class defines three operations:

AssignName: Assigns a name to an (instance of) **NameableObject** in a binding space.

RemoveName: Removes a name from an (instance of) **NameableObject** in a binding space.

Resolve: Returns an object of the parameterized type with the given name in the given context.

To use this class, an entity's class definition inherits from a **NameableObject** class parameterized by the entity class itself. Figure 7 shows an example of such a class definition in the C++ implementation of this model.

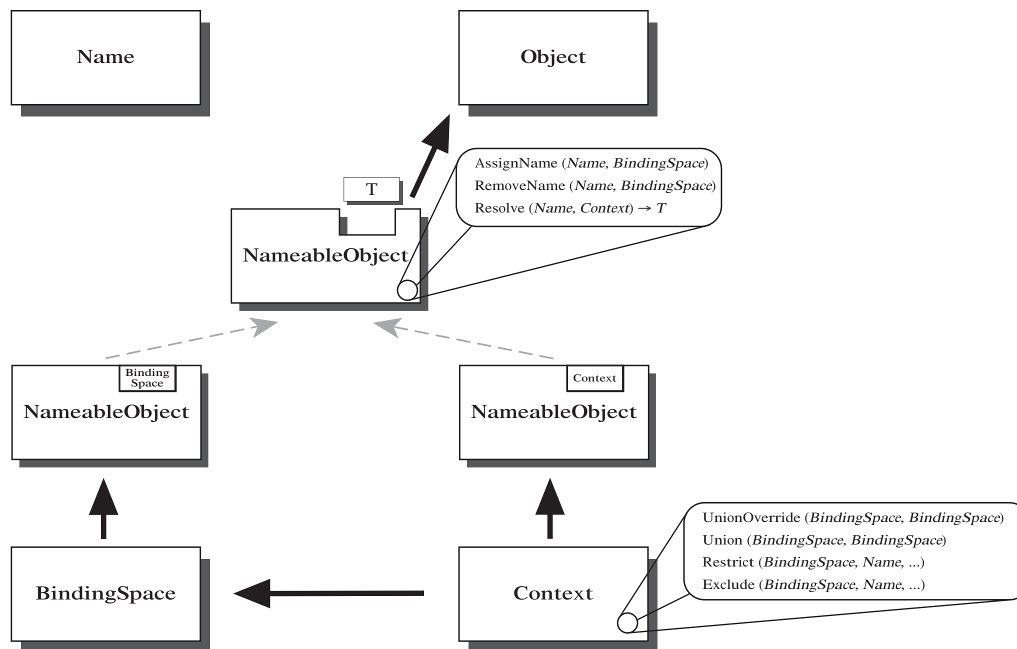

Figure 5: Model of a Type-Preserving, Nameable-Object-Centered Approach

The use of parameterized classes and inheritance in this model leads to every subclass of **NameableObject** having the appropriate name assignment, name removal and name resolution operations associated with it. As before, we can use both the model and the corresponding C++ implementation to assess properties of this approach and to position it in the NamingSpace. The major benefit of the approach is that the name resolution function returns the proper type and precludes the need for type casting. Another notable characteristic, of course, is that bindings are not explicitly manipulated in this approach; instead naming is accomplished using the **AssignName** operation associated with every nameable object. On the other hand, having the **Resolve** operation associated with the object, instead of with the

```
template <class T> class NameableObject : public
Object {
 public:
    int AssignName (Name& n, BindingSpace& b);
    int RemoveName (Name& n, BindingSpace& b);
    static T* Resolve (Name& n, Context& c);
};
```

Figure 6: NameableObject Class

class Person : **public** NameableObject<Person> {. . .}

Figure 7: Using the NameableObject Class

context, may seem somewhat unnatural.

Any number of other alternative approaches, based on major or minor modifications to the models, and corresponding C++ implementations, described here could be envisioned. One might, for example, want to consider how and whether names or bindings, or both, might be made nameable objects, what alternative context formation operations might be useful, or how name management approaches and persistence should interact. We have, in fact, already begun to investigate some of these questions. Based on our successful experience with it to date, we believe that the use of object technology, as illustrated in this section, will continue to prove valuable in exploring these and a wide variety of other issues related to name management.

5 Name Management Applied to Object Technology

Although most of our work in the area of name management and object technology for advanced software has focused on using object technology to study name management, we have also begun exploring the potential for beneficial application of improved name management mechanisms to object-oriented programming languages, object-oriented databases and software object bases. In this section, we give a brief outline of our experience incorporating an enhanced name management mechanism, implemented in C++, into a representative instance of object technology for advanced software, namely Open OODB, an open, extensible object-oriented database management system [WBT92]. Open OODB is designed to be easily tailored for a wide variety of advanced software applications such as software development environments, computer-aided design and manufacturing, and hypermedia systems. One goal of Open OODB is to allow researchers to experiment with, improve upon and refine various approaches to a broad range of object services such as persistence, name management, transaction, indexing and data dictionary services. Thus, Open OODB provides an excellent testbed in which to experiment with implementations of various name management approaches. We have currently installed an Alpha release of Open OODB on a SUN SPARCstation 10 running Unix 4.1.3. The release uses the EXODUS Storage Manager 2.2 [CDS89] as its underlying object store.

The persistence service in Open OODB provides an interface that allows applications to make instances of arbitrarily complex C++ classes persist. The present persistence mechanism used in our particular instantiation of Open OODB is based on a reachability model of persistence. When an object is made to persist, all objects reachable from the persistent object are also made to persist. The use of this mechanism, along with the object-oriented capabilities provided by C++, facilitates integration of, and hence experimentation with, our prototype name management mechanisms and Open OODB.

The first step towards integrating a particular name management mechanism with Open OODB involves the use of the Open OODB persistence extending tools, which are used to extend C++ classes with the necessary persistence capabilities. The tools are initially applied to the constituent classes of the name management mechanism. Next, a root binding space must be created and made persistent. A function that returns this binding space is made available to all applications that are using the prototype name management mechanism. Applications must then modify class definitions accordingly so that instances of a class may be nameable. For example, to use the mechanism described in Section 4.1 classes are required to inherit from the **Object** class, while the mechanism described in Section 4.2 requires classes to inherit from the **NameableObject<T>** class (where **T** represents the name of the heir class). Finally, the Open OODB persistence extending tools must be applied once again, this time to the modified class.

The integration of our name management mechanism into Open OODB results in a name-based persistence mechanism (i.e., an alternative persistence model based on name assignment and manipulation). Instances of any class may use the name management mechanism to assign names to objects, access objects using names, organize named objects (or, more precisely, (name,object) bindings) into binding spaces, and construct contexts in flexible and arbitrary ways. Since persistence is based on reachability, assigning a name to an object in a binding space that is (directly or indirectly) reachable from the persistent root binding space results in the binding, the name and the object being made persistent.

The results of this experiment have highlighted the advantages both of an object-oriented database architecture like that of Open OODB and also of our object technology implementations of advanced naming capabilities. Open OODB's capabilities for extending C++ classes with persistence, combined with our C++ implementations, facilitates the integration of a variety of name management mechanisms in a richer and more realistic testbed for experimentation. It is relatively straightforward to change the name management mechanism in this system, simply by substituting one of our C++-implemented approaches for another. Therefore, we will now be able to experiment with how improved name management capabilities affect an object-oriented database and also conduct much more realistic and extensive evaluations of alternative name management approaches than could be performed using only the simple testing harness employed in our initial experimentation.

Although this first attempt at integrating our name management work with a realistic instance of object technology for advanced software has been remarkably successful and worthwhile, there are several improvements and further steps that we would like to pursue. First, we would like to unify the name management capabilities provided by our mechanism with those in the C++ language that provides the application programming interface to Open OODB. While the name management mechanism that we have integrated into the system can be used to name both transient and persistent objects, it is still quite separate from the name management mechanism (applicable to transient objects only, of course) provided in C++. Another useful enhancement would be to support type checking in conjunction with the name management services that our mechanism provides. The data dictionary service in Open OODB provides information about types, but our current prototype does not utilize that information. We plan to rectify this in future versions. Further experimentation with

the present and future versions of this prototype integration of our name management mechanisms and Open OODB promise to yield additional insights and suggest other aspects to address.

6 Conclusions and Future Directions

Name management is fundamental to every aspect of computing, and it seems to have especially strong connections to object technology for advanced software. In particular, object technology has been instrumental in the two complementary facets of our research on name management, and it also appears that improved name management mechanisms could greatly benefit object-oriented programming languages, object-oriented databases and software object bases.

In this paper we have outlined our overall approach to research on name management, described some preliminary models that we have developed as a starting point for that research, discussed some specific name management issues arising in object technology for advanced software and given examples both of contributions that object technology has made to name management research and of the benefits that enhanced name management capabilities can make to object technology for advanced software. In the former category we have illustrated contributions to both facets of our research agenda – the use of object-oriented *modeling* in investigating various alternative approaches to name management as well as the value of object-oriented programming in implementing prototype *mechanisms* that can be used for empirically assessing modeled alternatives. In the latter category we have described how our extended name management mechanisms can be integrated into a particular object-oriented database system, resulting in a novel persistence control method as well as richer and more powerful name management features.

We believe that the work described in this paper provides a foundation for increased understanding and improved practical utilization of name management, especially as it relates to object technology for advanced software. Our models and our exploration strategy, based on object technology, offer a basis for enumerating, explicating and evaluating various approaches to various aspects of name management. Building on these, it will be possible to select or develop better and more appropriate name management approaches for use in current and future systems, especially object-oriented programming languages, object-oriented databases and software object bases. Our own future directions include expanding the scope of our investigation of name management, to encompass such things as its role in interoperability (e.g.,[WWRT91]), while continuing to develop and enhance both models and mechanisms for name management. With more and better models, coupled with more and better implementations, we hope to contribute to further progress in name management and object technology for advanced software.

Acknowledgements

The authors wish to thank Dr. Philip Johnson for his comments on an earlier draft of this paper.

REFERENCES

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[Bow90] C. Mic Bowman. *Univers: The Construction of an Internet-Wide Descriptive Naming System*. PhD thesis, The University of Arizona, Tucson, Arizona, August 1990.

[CDS89] M. Carey, D. DeWitt, and E. Shekita. Storage management for objects in EXODUS. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, 1989.

[Chu41] Alonzo Church. *The Calculi of the Lambda-Conversion*. Princeton University Press, Princeton, 1941.

[CP89] Douglas E. Comer and Larry L. Peterson. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, May 1989.

[CWW80] Lori Clarke, Jack C. Wileden, and Alexander L. Wolf. Nesting in Ada programs is for the birds. In *Proceedings of an ACM-SIGPLAN Symposium on the Ada Programming Language*, pages 139–145, 1980. Appeared as *SIGPLAN Notices* 15(11).

[DCBM89] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88–A database programming language? In *Second International Workshop on Database Programming Languages*, pages 213–229, June 1989.

[Dea89] Alan Dearle. Environments: A flexible binding mechanism to support system evolution. In *22nd Hawaii International Conference on System Sciences*, pages 46–55, Hawaii, U.S.A., January 1989.

[Deu91] O. Deux, et al. The O_2 system. *Communications of the ACM*, 34(10):34–49, October 1991.

[ECM91] ECMA. Reference model for frameworks of software engineering environments. Technical Report ECMA TR/55, 2nd Edition, ECMA/NIST, December 1991. (NIST Special Publication 500-201).

[Fra71] A.G. Fraser. On the meaning of names in programming systems. *Communications of the ACM*, 14(6):409–416, June 1971.

[Hei92] Dennis Heimbigner. Experiences with an object-manager for a process-centered environment. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, B.C., August 1992.

[Kad92] R. Kadia. Issues encountered in building a flexible software development environment: Lessons from the Arcadia project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 169–180, Tyson's Corner, VA, Dec 1992.

[Knu88] Jorgen Lindskov Knudsen. Name collision in multiple classification hierarchies. In *Proceedings of the European Conference on Object Oriented Computing (Lecture Notes In Computer Science 322)*, pages 93–109, Oslo, Norway, August 1988.

[LB88] B. Lampson and R. Burstall. Pebble, a kernel language for modules and abstract data types. *Information and Computation*, 76(2/3):278–346, Feb/Mar 1988.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[MABD90] R. Morrison, M.P. Atkinson, A.L. Brown, and A. Dearle. On the classification of binding mechanisms. *Information Processing Letters*, 34(1):51–55, February 1990.

[Mey88] Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, 1988.

[OMG92] OMG. Object services architecture, revision 6.0. OMG Document 92.8.4, Object Management Group, Framingham, MA, August 1992.

[ONT93] ONTOS, Inc. Technical overview. Burlington, MA, May 1993.

[Tho89] Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6:15–23, November 1989.

[TWC90] Peri L. Tarr, Jack C. Wileden, and Lori A. Clarke. Extending and limiting PGRAPHITE-style persistence. In *Implementing Persistent Object Bases : Principles and Practice / The Fourth International Workshop on Persistent Object Systems*, pages 74–86, Aug 1990.

[TWW89] Peri L. Tarr, Jack C. Wileden, and Alexander L. Wolf. A different tack to providing persistence in a language. In *Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.

[WBT92] David L. Wells, Jose A. Blakely, and Craig W. Thompson. Architecture of an open object-oriented management system. *Computer*, 25(10):74–82, October 1992.

[WCW85] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Ada-based support for programming-in-the-large. *IEEE Software*, 2(2):58–71, March 1985.

[WCW89] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. The AdaPIC tool set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

[WWFT88] 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 of Software Development Environments*, pages 130–142, September 1988.

[WWRT91] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.