NAME MANAGEMENT IN CONVERGENT COMPUTING SYSTEMS: MODELS, MECHANISMS AND APPLICATIONS

Alan Kaplan
kaplan@cs.umass.edu

CMPSCI Technical Report 96-60
May 1996

Convergent Computing Systems Laboratory
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This research is based upon work sponsored by the Advanced Research Projects Agency under grant MDA972-91-J-1009, by Texas Instruments, Inc. under Sponsored Research Agreement SRA-2837024, and by Intermetrics, Inc. funded by the Air Force Materiel Command, Phillips Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F29601-95-C-0003. The views and conclusions contained in this document are those of the author. They should not be interpreted as representing official positions or policies of the U.S. Government, Texas Instruments, or Intermetrics, and no official endorsement should be inferred.

NAME MANAGEMENT IN CONVERGENT COMPUTING SYSTEMS: MODELS, MECHANISMS AND APPLICATIONS

A Dissertation Presented

by

ALAN KAPLAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 1996

Department of Computer Science

NAME MANAGEMENT IN CONVERGENT COMPUTING SYSTEMS: MODELS, MECHANISMS AND APPLICATIONS

A Dissertation Presented

by

ALAN KAPLAN

Approved as to style and content by:

Jack C. Wileden, Chair

J. Eliot B. Moss, Member

David W. Stemple, Member

Barbara H. Partee, Member

David W. Stemple, Department Chair
Department of Computer Science

ACKNOWLEDGEMENTS

"They's a whole lot of new things I aim to be seeing!"
Martin Arrowsmith's great grandmother.
Sinclair Lewis, *Arrowsmith*

The pursuit of a PhD is one of the most remarkable journeys a person can take. Although in the end only a single author appears on the title page, there are a number of individuals and organizations that have helped make this long journey possible.

There are several organizations that provided valuable resources – both financial and technological – that facilitated this work. I am forever indebted to the Department of Computer Science and the University of Massachusetts. I am also grateful to the Advanced Research Projects Agency, Texas Instruments, Inc. and Intermetrics, Inc. for their support of this work.

I thank the members of my thesis committee – Eliot Moss, Dave Stemple and Barbara Partee – for their participation and assistance in my dissertation. Their keen insight into various name management problems and issues played an important role in this research.

I am certainly grateful to the various office and support staff within the Department of Computer Science. No matter how overloaded, they somehow managed to maintain the infrastructure required for graduate school- and research-related activities. In particular, I thank Ronnie Boss, Terrie Kellogg, Glenn Loud, Sharon Mallory, Rob Rice, Gary Rehorka, and Michele Roberts for their assistance over the years.

During my graduate career, I have been fortunate to work along side and meet a number of incredibly talented and unique individuals. In no particular order, I thank

Joe Fialli, Sue Mathisen, Keith Decker, Alan "Bart" Garvey, Tony Hosking, Carla Brodley, Bob Whitehair, Kathy McAdoo-Whitehair, Marty Humphrey, Dave Hart, Gary Wallace, Jody Daniels, Brendon Cahoon, Claire Cardie, David Skalak, Bob Cook, Dan Neiman, Kevin Gallagher, Joe McCarthy, Glenn Wong, Maureen Kocot, Carol Barr, Janet Maziotti and Phylis Clapis for their help and advice throughout the tribulations of this dissertation. I am particularly grateful to Jack Wileden, Andrea Leibson, their children, Alex and Lydia, the Westbrooks – Westy and Teri, their children, Brian and Josh and "Uncle" Jim, Zack Rubinstein and Alice Julier (and Zoe too), and Dave Hildum. They organized dozens of cookouts, camping trips, dinners, basketball games, softball games, trips to auctions, and numerous other social gatherings. Although I may never get used to camping, these events provided an important, if not crucial, outlet from the various stressful times of graduate school.

Various non-UMASS friends also deserve a special note of thanks. Their generosity and words of encouragement were a tremendous help during my graduate years. I thank Alan and Vicki Kirschenbaum, Fred and Ethel Kirschenabum, Judi and Serle Levin, Doug and Nancy Braziller, Paul and Jennifer Sabbah, Robert and Robin Blatt, Mark and Jill Siegelstein, Victor Levin, Steven Speiss, Bruce "Max" Bernstein, Lori Gunti, Jed Simmons, Bruce Almquist, Scott Sheppard, Skip and Amy Mueller, and Bill and Mary Schmutz.

It is hard to imagine how I would have completed my dissertation without the assistance and guidance of my friend and mentor, Jack Wileden. His ability to listen to my ideas (no matter how vague and unfocused), his confidence in my capabilities and his unique perspective all contributed to this dissertation. I am extremely grateful for his patience and support throughout my graduate career.

Special thanks go to my in-laws, Ann and Chick Lussier, my brother-in-law, Bill Lussier and my Uncle Irving and Aunt Judy Bernstein. I also thank my sister,

Randy Kaplan, my brother-in-law. Drew Zantop, my nephews, Jake and Max, my niece, Sami, and my brother, Gary Kaplan for their love and support.

I am forever grateful to my parents, Lou and Fran Kaplan. They made many sacrifices – both financial and emotional – throughout the years, which gave me the opportunity to pursue my PhD.

Finally, I am eternally grateful to my wife Ruthann (who I affectionately call Ruby) and my daughter Anna Rose. They are truly my source of inspiration. Their love, faith, confidence, and patience got me through the most difficult of times. I could not have completed this journey without them.

ABSTRACT

NAME MANAGEMENT IN CONVERGENT COMPUTING SYSTEMS:
MODELS, MECHANISMS AND APPLICATIONS

MAY 1996

ALAN KAPLAN

B.S., DUKE UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Jack C. Wileden

Name management — 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 — is a major challenge for developers and users of any complex software system. Experience indicates that name management is particularly problematic in *convergent computing systems*. These are systems that combine elements from several disparate computing domains into a single, synergistic whole; object-oriented databases, world-wide-web applications and image understanding environments are all examples of this increasingly important class of systems.

Despite its importance, historically name management has received relatively little attention from researchers in computer science. As a result, computing systems typically rely on *ad hoc* and primitive approaches to name management, making systems confusing to use and difficult to maintain.

The results of this dissertation include the development of novel modeling techniques and mechanisms with both theoretical and practical implications for name

management in convergent computing systems. In particular, some broadly-applicable models are defined to help reason about existing approaches to name management, especially their context formation and control capabilities. Example applications of the models illustrate how they can contribute to improving our overall understanding of the use of names, as well as some of the difficulties often encountered, in various approaches. Using the models as a basis, prototype mechanisms are then developed, implemented and evaluated for a representative instance of a convergent computing system. Specifically, a consistent and uniform name management mechanism is defined and demonstrated for both C++ and CLOS application programming interfaces to an object-oriented database (OODB) system. The resulting mechanisms represent a substantial improvement in the area of name management in this domain of convergent computing systems. Finally, based on these models and mechanisms, a new, seamless approach to achieving *polylingual* interoperability is defined and demonstrated. This approach permits applications to uniformly process data objects from an OODB, despite the fact that those objects were originally defined, created and persistently stored using the capabilities provided by several distinct programming languages.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The use and manipulation of names is among the most basic elements of communication and comprehension. Whether it be in naturally occurring systems, such as human discourse, or artificially constructed ones, such as computing systems, names, along with the various mechanisms used to govern their meanings, are essential to understanding and sharing information. Although a great deal of attention has been directed toward developing descriptive models of human naming (e.g.,[21, 40, 46, 75, 103, 114, 119]), relatively little work has been done on this topic in computer science. As Milner observed in his 1991 Turing Award lecture, naming has "been treated as a second-class citizen by theories of computing" [88]. Thus, a prime objective of the work presented in this dissertation is to provide a more thorough and rigorous understanding of naming in computing. In particular, our work concentrates on the issues that arise when multiple domains (or systems), each with its own approach to name management, are combined or integrated into a single, cohesive unit.

This introductory chapter begins with an overview of name management and its general role in computing in Section 1.1. We then, in Section 1.2, describe our perspective on *convergence*, a growing trend in computing, focusing on name management issues in convergent systems. A running example highlighting some common name management-related problems is given in each of these sections. Section 1.3 provides an overview of our research approach. The chapter concludes with an outline for the remainder of this dissertation.

1.1 Name Management

Name management — how a computing system allows names to be established for objects,[1] permits objects to be accessed using names, and controls the availability and meaning of names at any point in time — is fundamental to almost all aspects of computing. Names are used, and hence name management mechanisms are needed, for many purposes in all kinds of computing domains (although the specifics of name management tend to vary within and across these different domains). For example, 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 and organize commands and files. Again, various mechanisms, such as the environment variables, search paths and hierarchical directory structure found in UNIX,[2] 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. A *name server* is 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, with simple scope rules and qualified names controlling the use of names.

The primary purpose of a name is to identify some entity or resource, such as a file or mail box, while a secondary purpose of a name is to promote sharing of an entity or resource. For example, a (procedure) name permits the logic represented

[1] The term *object*, as used throughout this dissertation, is used to connote a *nameable object*, i.e., an object that may participate in a name management mechanism. Thus, in a computing system there may in fact be some objects that are nameable, while others are not. For brevity, however, we use the term object instead of nameable object.

[2] UNIX is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

by a subroutine to be shared among different programs. In computing systems two distinct approaches to forming names have generally been employed, yielding different approaches to, and hence raising different research issues with respect to, name management. In *identifier-based* approaches to name formation, names are arbitrary strings of symbols defined over some alphabet and formed using a collection of well-defined operators. Examples of identifier-based naming include variable names in programs and file names in operating systems. In an identifier-based approach, a name, even a mnemonic one, is arbitrary in the sense that it has no necessary connection to any intrinsic property of an entity. Thus, a name may stand for an entity, yet makes no assertions whatsoever about its referent [26]. In contrast, under *description-based* approaches to naming, a name can be viewed as a collection of properties or attributes associated with, or representative of, an entity (typically specified as sets of attribute-value pairs). For example, rather than referring to a particular printer's identifier-based name, a user wishing to print a document might issue a description-based name such as "the printer whose location is closest to my office" or "the printer whose job queue is the least congested." In comparison to identifier-based names, description-based names have very strong connections to the entities they denote. By their very nature, any change in an entity's state may result in a corresponding name change under description-based approaches, while under identifier-based approaches, an entity's state can be logically viewed as being orthogonal to its name(s). Relocating a printer, for instance, can have a dramatic effect on subsequent interpretations of its description-based name. Finally, each approach to name formation lends itself to distinct access methods – identifier-based approaches support navigational access to entities, while description-based approaches provide associative access to entities.

This dissertation focuses on identifier-based approaches to name management. Our primary motivation for this decision is the predominance of identifier-based ap-

proaches in computing. It also reflects our belief that such approaches are more general and broadly applicable, and that satisfactory solutions are lacking in this specific area, as will be illustrated throughout the remainder of this dissertation. Nevertheless, many of the ideas developed here are also potentially applicable to description-based approaches. Hence, unless otherwise noted, in the remainder of this dissertation we use the term *name* to refer to an *identifier-based name*.

Another important dimension with respect to names is the observation that names occur at various *levels* in computing systems [144]. In programming languages, for example, variable names, virtual addresses and physical addresses are all instances of names that exist at different levels in a running program. Similarly, file names, file descriptors and disk addresses as found in traditional file systems are also examples of names occurring at different levels. Comer and Peterson [32] term a specific level in an approach to name management as a *name space*. In this dissertation, we focus on the problems associated with a specific name space in a computing system.[3] Other research has focused on the interaction between different name spaces (e.g., [32, 144]).

Although frequently overlooked or taken for granted, in part because it is so pervasive in all domains of computing, name management has been an enduring problem. At the 1983 Symposium on Programming Language Issues in Software Systems, for example, a panel was held to discuss significant research issues in programming languages for the (then-) upcoming decade [113]. Among the many topics considered was name management. The consensus at the 1983 workshop was that (then-) current approaches to name management were confusing and inadequate, and that this area was an important research direction for future software systems. Although the intervening decade (or more) has witnessed several important advances, for the most part these have resulted from the recognition of shortcomings inherent in individual approaches, followed by development of various features, some of which have now

[3] Although the term *name space management* might be a more appropriate categorization for the work described here, for the sake the brevity, we use the term *name management*.

gained wide acceptance in many computing domains. Examples include hierarchical directory structures[4] found in today's file systems, and constructs supporting modularity and object-orientation found in modern programming languages. The provision of such features clearly reduces the overall complexities associated with using and manipulating names, especially in large systems. For example, hierarchical directory structures facilitate sharing and organizational capabilities in file systems, while modularity and object-orientation constructs promote re-use and ease maintenance activities in software development.

Despite certain improvements, such as those mentioned above, existing approaches to name management are still frequently complex and error-prone. Users and applications continue to rely on names as a fundamental access method, yet the underlying substrates supporting name management facilities are still often difficult to reason about and understand. Many problems are the result of an inability to easily specify or discover the intended meanings of names. It is all too often the case that a name may have no meaning at all, or the name may already have a meaning, but not the one that was anticipated. In other situations, the meaning of a name might be ambiguous. In other words, there might be several available meanings for a particular name, but there exists no means for selecting which meaning should be applied. The difficulties arising from these kinds of naming problems are often extremely subtle, hard to locate and hard to repair. Therefore, as systems become larger and more complex, underlying name management mechanisms increasingly become focal points with respect to the ability to develop, maintain, integrate and port software systems.

[4]Hierarchical directory structures pre-date 1983, although even in modern systems they are not universally employed. Object-oriented databases, for example, have generally used flat naming structures.

1.1.1 Some Example Problems

As an illustration of some common name management-related difficulties, consider the facilities provided by UNIX-based file systems, one of the most popular foundations upon which many computing systems have been, and continue to be, developed. As noted above, UNIX file systems provide users and applications with a hierarchical structure for sharing and organizing data, where compound names[5] are used to represent the hierarchical structure. Supplementing this component of UNIX's name management mechanism is the *environment variable* construct that, among other things, can be used to specify a sequence of directory names. When a file name (in a program, a script, a document, etc.) is interpreted, each directory, in the order specified in the appropriate environment variable, is searched until the required file name is located. One obvious benefit of this approach is that the use of long compound names can be avoided. Environment variables also facilitate the utilization of a particular system (e.g., source code, scripts, text documents, etc.) in different settings and/or by different users. By changing the value of a particular environment variable, a system can be installed in, or ported to, different environments with minimal changes to the original system.

For example, consider the approach employed by LaTeX,[6] a widely-used and extremely popular text formatting tool, often used in UNIX-based operating environments. Figure 1.1 shows a hypothetical, and somewhat simplified, UNIX directory structure containing a collection of LaTeX document and macro files, where the directory named **projects** (which is located in the directory named /usr) contains two LaTeX documents, each of which refer to names of LaTeX macro files, and the di-

[5] A compound or qualified name is one in which each "simple name" is separated by a delimiter. In the UNIX file system, the delimiter is a "/" and names are interpreted from left-to-right, where each simple name, except for possibly the last one in the sequence, corresponds to a directory.

[6] Although the example is in terms of LaTeX, we contend that the problems illustrated here are characteristic of almost all UNIX-based tools and utilities (e.g., compilers, linkers, editors, etc.).

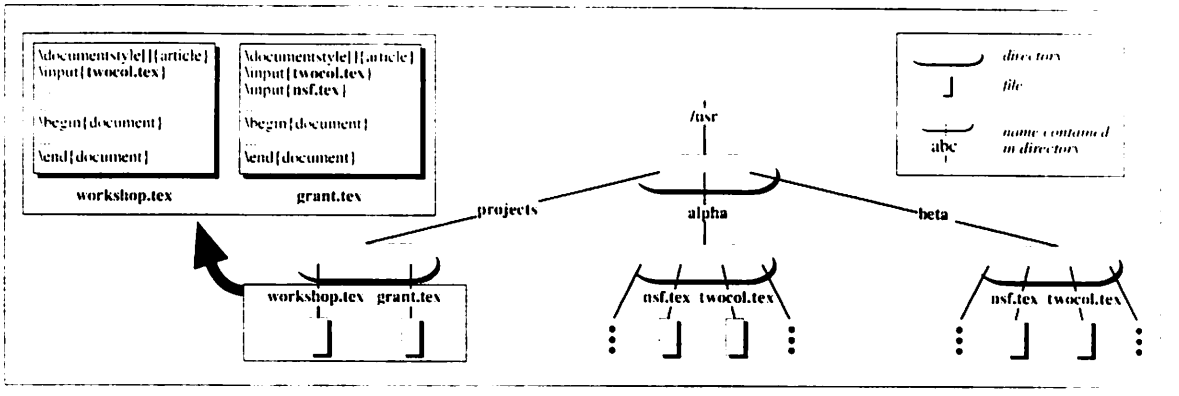


Figure 1.1 Name Management and LaTeX

rectories named alpha and beta contain some actual macro files.[7] LaTeX also uses an environment variable called TEXINPUTS to control the interpretation of names of macro files referenced in documents. In Figure 1.1, the document named workshop.tex refers to a macro file named twocol.tex. To connect that reference to the file named twocol.tex located in the directory named alpha, the value of the TEXINPUTS variable could be set as

TEXINPUTS=/usr/alpha

When LaTeX was invoked on the document named workshop.tex, it would encounter the name twocol.tex in the document file, and then, using the information stored in TEXINPUTS, locate the file with that name in the /usr/alpha directory. To change the interpretation of a file name, a user or application could simply modify the value of the appropriate environment variable. Hence, in this example, changing TEXINPUTS to

TEXINPUTS=/usr/beta

would result in using the file named twocol.tex located in the /usr/beta directory.

[7]In a real-world setting, these directories might correspond to different versions, libraries, or vendors of macro files.

Although use of environment variables is common practice in UNIX-based applications such as LaTeX, the approach is limited. Continuing with the example above, suppose a user now wished to process the document stored in the file named grant.tex, which refers to two macro files, i.e., twocol.tex and nsf.tex. Furthermore, suppose the user wished to use the file named nsf.tex contained in the directory named alpha and the file named twocol.tex contained in the directory named beta. Neither of the specifications[8]

TEXINPUTS=/usr/alpha:/usr/beta TEXINPUTS=/usr/beta:/usr/alpha

yields the desired results since the name spaces of the two directories overlap with one another. In particular, since environment variables provide little in the way of fine-grained control, the first of the two TEXINPUTS specifications above results in the inclusion of both macro files from the /usr/alpha directory, while the second specification yields the set of files from /usr/beta. One common solution to this problem involves creating a new directory with either copies of, or links to, the desired macro files and then replacing the directories listed in an environment variable with this newly created directory. Another possibility involves changing the references in the documents themselves by using absolute (or at least, more specific) names to refer to the intended macro files (e.g., changing the reference nsf.tex to /usr/alpha/nsf.tex in grant.tex). Clearly neither of these solutions is desirable, since copying and linking files, or changing source code, especially in real, non-trivial situations, is an error-prone, tedious and expensive process.

This simple example illustrates several other common shortcomings of typical approaches to name management. The use of a single construct to control the interpretation of names means that, at any given point in time, a specific name is restricted to having the same meaning across all the files in which it appears. Continuing with

[8]A colon (:) is used to separate or delimit multiple directories appearing in an environment variable specification.

the example shown in Figure 1.1, each document uses the value of the TEXINPUTS variable for determining the meanings of macro file names, even though the name twocol.tex is intended to have a different meaning in each of the documents in which it appears. As a result, users not only are forced to manually change the value of the TEXINPUTS variable on a case-by-case basis, but any changes made to this variable may have unintended consequences for other documents. A related problem concerns the implicit connection between a document and the mechanism used to control the interpretation of names used in that document. In LaTeX, the information indicating the association between a particular document and a specific setting of the TEXINPUTS variable can only be maintained by manual methods (e.g., by including such information in a comment in a document).

Finally, this example also demonstrates the lack of automated support for discovery of potential name management problems. While the value of an environment variable can be easily examined (by invoking appropriate UNIX shell commands or system calls), there are no direct means for computing the actual files that will be used or are required by a particular invocation of LaTeX on a particular document. Users are instead reduced to manual methods, such as combing through the various messages produced by LaTeX, to determine which files were actually used in a particular invocation. Furthermore, the approach does not provide any automatic methods for determining whether the value of an environment variable is well-formed. For instance, without manual intervention, an environment variable containing references to non-existent directories could easily go undetected. As a result, a tool may, depending on the extant directories included in an environment variable specification, interpret names in an unanticipated manner.

The problems illustrated by this simple example, and numerous others like it, motivate much of the research presented in this dissertation. Current approaches to name management in individual domains are often inadequate and inflexible. Despite

the critical role names play, users, software developers and applications typically rely on implicit conventions and manual methods for managing names in computing systems. As a result, name management-related problems and errors can be difficult to detect and correct, and thus can greatly diminish the overall value and utility of individual computing systems, domains or paradigms. As we describe in the next section, attempts to assemble diverse elements, each with its own approach to name management, into a convergent computing system face even worse difficulties.

1.2 Convergent Computing Systems

A *convergent computing system* is any system in which two or more distinct computing domains or paradigms are combined into what is intended to be a synergistic and seamless whole. One time-honored, simple, but widespread example of a convergent computing system is a programming language along with primitives (typically in the form of subroutines) for accessing a file system. A distributed system consisting of multiple, perhaps autonomous and heterogeneous, computers may also be viewed as an instance of a convergent computing system, while an application running on a range of distinct hardware platforms, or communicating with a number of different kinds of input/output devices, or utilizing information found in a set of distinct databases, or incorporating components written in several different programming languages, is yet another example. Other prominent examples of convergent computing systems range from persistent object systems (including object-oriented databases and database programming languages), which merge the domains of programming languages, databases and operating systems, to software development environments, which integrate various tools and techniques, to the much-heralded information superhighway, which is expected to meld components from virtually every domain of computing. The growing trend of convergence and recognition of its importance is further evidenced by current industrial efforts aimed at developing and promoting *de*

facto standards facilitating integration and cooperation of heterogeneous components in large and complex systems (e.g. [28, 44, 87, 100, 101]).

The engineering and operation of convergent computing systems poses a number of important challenges to computer science. Indeed, the viability and usability of many of the example convergent computing systems highlighted above are a direct result of research activities that have been, and continue to be, directedly aimed at developing new technologies toward overcoming barriers and synergistically combining capabilities of disparate computing domains, paradigms and systems. For example, work in the area of persistence has demonstrated that the capabilities of programming languages and databases can be transparently and efficiently combined (e.g., [5, 148]). Research in interoperability has continued to provide new techniques allowing heterogeneous systems to communicate and cooperate with one another (e.g., [150, 157, 158]). Most recently, advances in networking and distributed technology have yielded a very powerful and flexible foundation facilitating the convergence of the global Internet, namely the World Wide Web [10]. Another key research direction with important implications for convergent computing systems, and the focus of this dissertation, is name management.

Due to the fundamental role of name management, problems that arise during the development, use and maintenance of convergent computing systems can often be traced to complexities or shortcomings in their name management mechanisms. As demonstrated in the examples presented in Section 1.1, individual computing systems in general define specialized, often idiosyncratic and *ad hoc*, approaches to name management. Since they are not based on any fundamental understanding of name management, these approaches are frequently complex and error-prone. In the realm of convergent computing, name management raises additional important issues. Because no unifying models or mechanisms for name management currently exist, efforts to integrate diverse systems into a convergent one are often impeded

by the inconsistency, incompatibility and incompleteness of the various approaches to name management employed in the individual systems. As a result, developers and users of convergent computing systems are forced to manage and reason about multiple name management mechanisms, each with distinct and varying capabilities.

1.2.1 Some Example Problems

As an indication of some of the name management-related problems that arise in convergent computing systems, we build upon the LaTeX example described in Section 1.1, by combining its application (and its specific approach to name management) with an extremely simple use of *Make* [43] (and its specific approach to name management), a well-known and widely used software configuration utility. Make is a particularly interesting instance of a convergent computing system, combining the domain of dependency analysis and rules, which are the fundamentals of the Make paradigm, with the domains of various other tools and utilities, such as text formatters and file systems. While this example represents only a relatively limited degree of convergence, since the components (Make and LaTeX) are products of the same environment (i.e., the UNIX community) and are written in the same language (i.e., in C, more or less), it offers striking evidence of how even a seemingly benign convergent computing system can give rise to complicated name management problems.

Figure 1.2 depicts the contents of a simple Makefile (based on the Free Software Foundation's GNU Make [126]) that directs the application of LaTeX, as shown in the associated UNIX subdirectory and file structure. Although Make is a versatile and powerful tool with many features (see [126] for details), in our example it is used, as it most often is, to automatically determine which pieces of a large document or program need to be re-processed, and then issue commands to re-process them.

Consider, for instance, the fifth line in our simple Makefile, which defines a temporal relationship between the formatted document grant.dvi (the *target* in Make terminology) and the pieces found in the files grant.tex, nsf.tex and twocol.tex (these

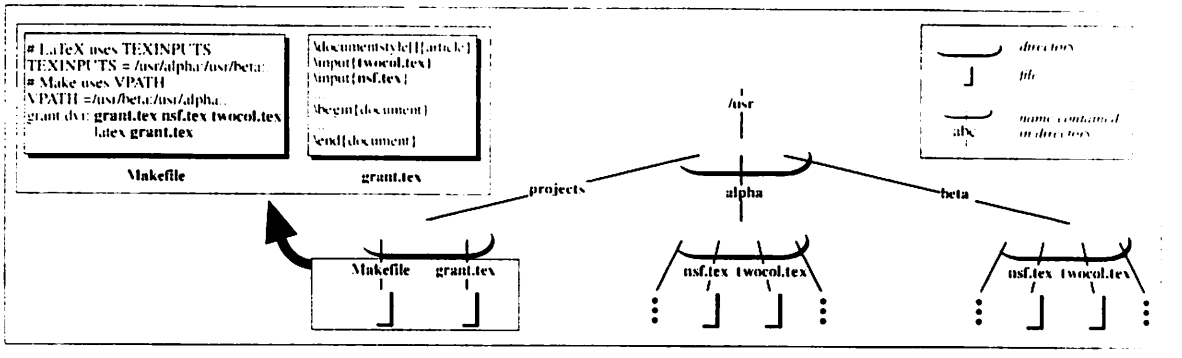


Figure 1.2 LaTeX, Make and UNIX

three items are called *dependencies* in Make terminology). The next line then states that if the time of last modification of any of the dependencies is more recent than that of the target (i.e., if the specified temporal relationship does not hold), the LaTeX formatting tool should be invoked on the grant.tex file. If the text formatting process is successful, a new output file named grant.dvi is created. It turns out that interpretation of the occurrence of the names nsf.tex and twocol.tex on the fifth line is controlled by Make's own name management mechanism based on the current contents of the VPATH variable (appearing on the fourth line of the Makefile), while the occurrence of those same names in the document is interpreted in a way, defined by LaTeX (i.e., via the current value of the TEXINPUTS variable), that might be completely different. Thus, in general, it is entirely possible that Make will use one set of files in determining whether a relationship holds and a partially or completely different set of files in performing the text formatting that is intended to re-establish the relationship in the case that it does not hold. Moreover, during a particular execution of Make, various tools may create newly named files or rename existing files, resulting in perhaps different and sometimes unanticipated interpretations of names occurring in a Makefile. The potential for serious and subtle difficulties is clear. Having the ability to specify that the desired interpretations of distinct occurrences of a given name are related, coupled with an analysis capability able to assess the consistency of

the interpretations, could help to avoid some very subtle name management-related problems.

In general, Make illustrates how the inconsistencies and incompatibilities among the name management mechanisms used by different components can impede system integration. That is, while some of the name management problems in Make arise from the inadequacies of the mechanisms of the underlying components, others can be traced to incompatibilities among the various components' name management mechanisms. The root cause of the latter kind of problems is that several, often conflicting and sometimes related, mechanisms may be controlling the interpretation of names. In particular, even in our extremely simple example, different mechanisms are associated with the interpretation of names during Make's dependency and rule analysis phase and during the execution of each UNIX command and tool. As a result, it is often not clear what interpretation will be used for a given occurrence of a name.

In summary, convergence exacerbates name management problems and difficulties. Attempts to combine diverse components are faced with the complicated task of overcoming the barriers imposed by often conflicting approaches to name management. As illustrated by the examples considered in this and the preceding section, the lack of a systematic approach to name management based on a fundamental understanding of name management principles leads to systems that are cumbersome to use and prone to error. As the apparent trend toward convergence of various computing domains continues in the coming years, the incompatibility of various individual approaches to name management will become increasingly apparent and problematic. Thus, given its ubiquity and importance, new approaches and techniques supporting name management in convergent computing systems must be developed. This dissertation represents a significant step in this direction.

1.3 Research Approach

The research described in this dissertation contributes to a broad-based investigation of name management, whose primary objective is to provide a well-defined foundation for name management in convergent computing systems. Given the kinds of problems described in the previous sections, we believe that research aimed at overcoming the limitations and shortcomings of current approaches needs to proceed along the following three dimensions:

- The definition and investigation of *models* that can serve as a basis for enumerating, explicating or evaluating various approaches to various aspects of name management.
- The definition and implementation of, and experimentation with, prototype *mechanisms* for name management in convergent computing systems.
- The development of *applications* enabled by these mechanisms, particularly applications that can facilitate other aspects of convergent computing systems.

The remainder of this section discusses these dimensions in greater detail and summarizes the progress made in each dimension in this dissertation.

1.3.1 Models

The long-range goal of our model-building efforts is to provide a foundation for fundamental understanding of name management and its role in all domains of computing. In this dissertation, we propose a conceptual, object-oriented framework for name management called PICCOLO. In addition to defining a collection of fundamental name management concepts, the framework provides insights into, and serves as a basis for informal reasoning about, name management mechanisms and the problems that arise in their use in convergent computing systems. We have also demonstrated how the semantics of the PICCOLO framework's concepts can be formalized

using *evolving algebras*, an operational semantics based on first-order logic. Such a formalization permits more rigorous reasoning about name management approaches modeled using PICCOLO and serves as a foundation for formal, and potentially automated, analysis of name management-related properties. In this dissertation, the utility of these models is demonstrated through their application in the evaluation and analysis of an approach to name management in a representative instance of a convergent computing system.

1.3.2 Mechanisms

The long-term goal of our mechanism-building activities is to provide a comprehensive set of powerful, flexible, uniform and broadly applicable name management mechanisms that will ease the construction and maintenance of (especially large and complicated) convergent computing systems. As part of our dissertation research we have, using our models as a basis, designed and implemented prototype name management mechanisms, and incorporated them into a representative instance of a convergent computing system, specifically the TI/Arpa Open Object-Oriented Database (Open OODB) [145]. With respect to existing approaches, the prototype mechanisms demonstrate a more uniform and flexible approach to name management in this class of convergent systems. In addition, we have shown how the underlying architecture is easily adaptable by applying the approach to independently-developed class libraries. We have also evaluated the ease with which our mechanisms can be incorporated into Open OODB and provided some preliminary performance data assessing the impact of our extended mechanisms.

1.3.3 Applications

The long-range goal of our application-building activities is to utilize our improved approaches to name management as leverage for other technologies in convergent computing systems. Since name management is central to almost all aspects of computing,

we believe that our extended name management mechanisms can be used as a means for enabling additional capabilities. Naturally, this also serves to further demonstrate both the importance of appropriate name management models and mechanisms, and the utility of our own name management approaches. In particular, with object-oriented databases, it is possible to have multiple application programming interfaces to the same database, through which objects created using different programming languages (and their respective type models) may all be stored in the same database. As a byproduct of our name management models and mechanisms, we have developed an approach to interoperability that provides uniform and transparent access to such objects. More specifically, this application of our extended mechanisms provides a uniform name management interface to the underlying database. It furthermore permits objects, whose types have been deemed to be compatible, to be manipulated through a single abstract interface, thus hiding the actual implementation language of objects. In other words, a C++ program views and manipulates database objects as if they were all implemented in C++, even though they may, in fact, be implemented in different languages. Here again, we have evaluated the usefulness of our mechanism, in particular, comparing our approach to current state-of-the-art approaches.

1.4 Outline of the Dissertation

Chapter 2 reviews previous work related to research in name management, beginning with a discussion of existing models of name management, both formal and informal, and continuing with a presentation of relevant name management mechanisms in convergent computing systems. The chapter concludes with a review of existing approaches to multi-language interoperability.

Chapters 3 and 4 present the PICCOLO framework. Chapter 3 lays the foundation for the framework. Several concrete examples demonstrating the framework's fundamental features are provided. Chapter 4 develops some extensions to the ba-

sic framework. These extensions focus on the details of name interpretation in name management. The extended framework is used to analyze several approaches to name management.

Chapter 5 addresses a formalization of the PICCOLO framework. As noted earlier, we show how the framework's semantics can be formalized using evolving algebras, an operational semantics based on first-order logic. The chapter first provides a brief introduction to evolving algebras. The chapter then demonstrates how precise semantics can be formulated in evolving algebras for various features of a PICCOLO model. Using this as a basis, some formal properties and analyses of name management approaches are given. As in Chapters 3 and 4, the chapter also contains concrete examples demonstrating the utility of the formal model.

Chapter 6 describes the prototype name management mechanisms we have developed as extensions to the TI/Arpa Open-Object Oriented Database (Open OODB). The chapter begins by giving an overview of Open OODB. The principal components of our extended name management mechanisms are then described. The chapter concludes by assessing the utility of the mechanisms, showing how the mechanisms overcome the limitations characteristic of current approaches, and examining the overhead (in terms of performance) of using the mechanisms.

Chapter 7 presents an approach to interoperability for programs developed in multiple languages that is enabled by our extended name management mechanisms. The chapter describes the POLYSPIN architecture and the POLYSPINNER toolset, which, as byproducts of the name management mechanisms developed in Chapter 6, facilitate certain aspects of multi-language interoperability in object-oriented databases. As in Chapter 6, the chapter includes an evaluation of the approach.

The dissertation concludes in Chapter 8 with an assessment of the current status of our work, a discussion of our contributions and a summary of ongoing and future

efforts aimed at further improvements for name management in convergent computing systems.

CHAPTER 2

RELATED WORK

There exist very few models devoted specifically to name management. Those that are tend to be too narrowly focused for the purposes of modeling name management in convergent computing systems. Existing name management mechanisms are often built in *ad hoc* ways and thus have proven to be inadequate for use in large and complex systems, especially convergent computing systems. Finally, existing approaches to multi-language interoperability (our particular application domain for name management) generally require interoperability decisions to be made *a priori*, are restricted to primitive language types or require the use of external type systems. This chapter reviews some of the more relevant models, mechanisms and interoperability approaches.

We note that due to the ubiquitous nature of names in computing, it is not surprising that there has been a great variety of tangentially related work involving the use of names. Much of this work, however, is not directly related to our interest in name management for convergent computing systems. For example, many formalisms for describing different kinds of computing systems make use of name-related concepts (e.g., [47, 92, 125]). Other models address how humans create names in computing systems (e.g., [26, 120]). In addition, various conventions, guidelines and frameworks have been proposed for devising more meaningful or mnemonic names for variables, procedures, etc. in programs, e.g., [70, 99, 123, 139] and user interfaces, e.g., [68]. While such work is interesting and important, it provides little or no insight into

how names are established, organized, controlled, and manipulated within computing systems, especially when multiple computing domains converge.

2.1 Models of Name Management

Given the pervasiveness of name management in computing, various existing formalisms often incorporate or address some aspect of name management. Early work in formal programming language semantics (e.g., [79, 80, 129, 130]) includes concepts and constructs for describing name manipulation in programs. The denotational approach [118] is based on the idea that each construct in a programming language grammar can be defined in terms of mathematical entities that it denotes (e.g., declaring a variable or locating a procedure in a given scope).

Other relevant models in the programming language domain tend to focus on other specific aspects of name management-related issues in programming languages. TEMPO [67], whose primary purpose is to serve as a teaching language, can be used to explain the different times names are assigned to programming language entities. Models of visibility control and module interconnection [72, 77, 109, 111, 154] can be used to characterize how names are managed and controlled among a program's constituent modules. Knudsen [73] has developed a formalism for understanding and specifying name collision semantics in inheritance hierarchies in object-oriented programming languages. Visual notations have been developed for understanding the meaning of names in both Algol-based languages and applicative languages [65, 140]. Like denotational models, all of these models have the obvious shortcoming of being specific to programming languages. In addition, their lack of generality due to their specific focuses means that no single one is a viable candidate as a basis for extending to name management issues in convergent computing systems.

In addition to the work done in the area of programming languages, a significant amount of research related to name management has been done in the distributed

systems domain [55, 122, 144], with perhaps the most comprehensive and general of these models due to Comer and Peterson [32]. The main emphasis in this domain seems to have been on models of name resolution in distributed environments, with a primary focus on the lower-level implementation issues, rather than more general, foundational concerns. Other models related to name management in distributed systems have examined performance, reliability and security issues (e.g., [29, 78, 121, 137, 141]). These issues, while important for implementations of name management mechanisms, are orthogonal to our interests in understanding the details of how names are managed by users and programs in convergent computing systems and hence are outside the scope of the research described in this dissertation.

Although the models discussed above address several fundamental name management issues, they are limited to particular aspects of name management in particular computing domains. As noted earlier, we are interested in models that provide a fundamental understanding of name management for convergent computing systems. In the earliest such effort that we are aware of, Fraser developed a formalism, based on a simple extension of Church's λ notation [30], that can be used to describe context and manipulations on context [45]. His model can be used to describe some aspects of context formation in programming languages and file systems, though only a few details are offered. A function-based model, due to Younger and Bennett [156], also attempts to define some general aspects of context manipulation with respect to file systems and distributed systems, though it is too incomplete for consideration as a comprehensive model of name management for convergent computing systems.

Another important and interesting formal model of name management was developed by Radia [108]. His model focuses on understanding how contexts are selected in distributed, autonomous file systems. In UNIX, for example, when a tool is invoked on a file, the initial context may be associated with the tool, the file or a combination of the two. The significant result of this work is a fairly general model that appears to

characterize a variety of context selection mechanisms in various computing domains, including operating systems, file systems and programming languages. It does not, however, appear to address how contexts are actually formed or specified.

Other efforts directed toward understanding general aspects of name management in computing systems can be characterized as being less rigorous and less formal. For example, Saltzer's somewhat simple, though insightful, classification scheme [115] provides a collection of definitions for a range of aspects of name management. Based on this organization, Saltzer describes the name management mechanisms common to memory addressing architectures and file systems. Similarly, a taxonomy for characterizing a broad range of aspects of name management in modern distributed systems in described in [155]. Finally, the classification framework presented by Morrison, Atkinson, Brown and Dearle in [93] is more narrowly focused, having been developed to categorize the various binding mechanisms found in languages (including database programming languages), operating systems and file systems.

2.2 Name Management Mechanisms

As noted earlier, names are used in many ways in traditional programming languages. Languages have always had their own idiosyncratic restrictions on how names may be formed. Early versions of FORTRAN and BASIC, for example, used syntactic properties of variable names for determining type information. More generally, programming languages have offered a variety of alternative approaches with respect to name management. Algol introduced the notion of static nested block structure for controlling the visibility of names in a program, while early dialects of Lisp used dynamic scoping to control the visibility of names. Some more modern languages, such as Ada [39] and Modula [151], have provided additional mechanisms for defining collections of names (e.g., the Ada package) and for controlling the meaning and availability of names (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 the management of names (e.g., PIC [153], Modula-3 [23]), while recent object-oriented programming languages, such as C++ [41], Eiffel [86] and CLOS [69], introduce some additional name management-related mechanisms (i.e., single and multiple inheritance schemes).

Despite the plethora of sophisticated, and often intricate, name management capabilities they provide, modern programming languages restrict the application of these capabilities to language-internal entities, such as variables and procedures. Name management for external entities, such as data files and program libraries, is provided by distinct and disparate mechanisms, typically by a file system. In general, these mechanisms center around a hierarchical organization of directories and files, as well as a collection of commands, tools and subroutines for populating, navigating, querying and manipulating the file system. *Search rules* in Multics [34], *environment variables* in UNIX, *transparent viewpaths* in 3-D [74] and *union directories* in Plan 9 [105] are examples of constructs that can be used to control the availability and meaning of names. These mechanisms have relied, for the most part, on the idea of searching a sequence of directories when locating a name. Such approaches, as illustrated in Chapter 1, can be restrictive and cumbersome.

File systems also illustrate some interesting approaches toward providing unified support for name management. The basic idea behind these approaches is that the name management component of the file system is used as a standard interface to a variety of system objects. In Multics, for example, the file system is used to name and access memory segments as well as files. In a similar fashion, the UNIX file system is used to manage names for a limited collection of hardware devices in addition to files.[1] The name management mechanism provided by the file system in Plan 9 extends the UNIX approach to its logical extreme, since all resources, includ-

[1] Not all hardware devices are named via the file system. For example, the names of printers are stored in a file (i.e., /etc/printcap) instead of being named as files.

ing hardware devices, are uniformly accessed through the file system. The obvious benefit of such approaches is that they attempt to provide a uniform interface to a diverse collection of system resources. In doing so, however, they restrict the name management capabilities to those provided by typical file systems.

Name management is of particular importance in distributed systems since it provides a primary means for sharing and communication among users and processes and is a key component in providing transparency [106]. In internets, for example, a variety of name management mechanisms have been implemented for naming and locating objects, such as users, hosts and services. Like file systems, these mechanisms have relied, to a greater or lesser degree, on hierarchical schemes for naming and organizing objects. Prominent examples include Grapevine [14, 117], Clearinghouse [102], Lampson's global name service design [78], the (widely-used) Domain Name System (DNS) [90, 91], and the X.500 protocol [62]. In most of these systems, however, there are no explicit means for controlling the meanings of names. Instead, this information tends to be encoded in names themselves.

The emerging domain of persistent object systems, including database programming languages, persistent programming languages and object-oriented databases, illustrates many of our concerns regarding name management for convergent computing systems. While research in this area has made remarkable advances in unifying various programming language and database (and/or operating system) capabilities, a review of related work reveals only partial solutions for name management. PGraphite [136, 148], R&R [135], E [112], ObjectStore [76] O_2 [38] and Open OODB [145] are prominent examples of such systems that provide little or no support for name management for persistent objects.

A notable exception to this characterization is the approach found in Napier [94]. Napier introduces a primitive type called an *environment* [35], which is used to organize the persistent store. The resulting mechanism is somewhat similar to a program-

ming language/file system combination, except that unlike directories, environments are tightly integrated with the Napier language run-time system. Napier clearly represents a significant contribution toward unifying name management in persistent object systems. From our perspective of name management, however, Napier does not appear to provide any flexible or general mechanisms for forming and controlling the interpretation for names of persistent objects. In addition, as we will illustrate in Chapter 4, Napier's approach to context specification can be extremely confusing to programmers and can lead to some serious programming errors.

2.3 Name Management-Enabled Applications

As noted earlier, a computing system's approach to name management often provides some useful leverage enabling other capabilities and features in the system. For example, syntax-directed editors (e.g., [142]) attempt to automatically interpret the meanings of names in a program being edited in order to ensure correct syntax. The Sun RPC Protocol Compiler [133], a support tool that aids software developers in the construction of RPC-based applications, uses the names of data files to drive the creation of the various code files it produces. Similarly, the SPARC C++ template instantiation mechanism relies on a similar technique. More recently, the World Wide Web's hyper-text transfer protocol (http) uses the underlying file system of a host's server as means of accessing and organizing hyper-links. In this dissertation, we have focused our attention on using our improved name management mechanisms to enable and facilitate transparent interoperability in convergent computing systems. In particular, we are interested in developing techniques supporting interoperability among applications developed using multiple programming languages.

Interoperability has been an enduring problem in computing. While the adoption of standards (e.g., ASCII, POSIX) has helped alleviate many such problems, the dynamic and flexible nature of computing makes it undesirable, if not impossible, to

create standards for every possible interoperating scenario. Thus, various approaches to the interoperability problem have appeared over the years. For the most part, these approaches have been based on establishing the compatibility of types at the representation level [150].

Perhaps the most straightforward approach to interoperability for multi-language applications is to use the native foreign function interface (FFI) features provided as part of most programming language environments, i.e., the programming language itself, the compiler, the operating system, etc. (e.g., [31, 83, 134]). In these approaches, programmers are provided with a variety of subroutines and libraries that permit the exchange of primitive data types (e.g., a C++ integer with a Lisp integer) between different languages (or different dialects of the same language). Although FFI-based approaches might be an important underlying mechanism for other interoperability approaches, they suffer a variety of shortcomings. One obvious drawback is that they fail to support more complex data types, such as user-defined abstract data types. In addition, FFI-based approaches provide almost no automated support for creating interoperating applications. Finally, such approaches are fairly limited with respect to flexibility and extensibility. Any change to an interoperating data type (e.g., changing an integer to a real) can have a dramatic effect on an interoperating application.

Approaches based on a remote procedure call (RPC) mechanism [13] suffer shortcomings similar to those of the FFI-based approaches. RPC approaches effectively abstract the underlying hardware and communication protocols between (possibly different) computers. Thus, an application can transparently invoke a procedure that is actually located in a different address space on a different machine. Applying RPC approaches is quite similar to applying FFI approaches, although some RPC mechanisms include tools that automatically generate the code for marshaling and unmarshaling arguments and communicating with the underlying network. NCS [2] and HRPC [12] are limited to interoperation between the same language, while more

recent approaches, such as ASN.1 [61], Q [84], MatchMaker [66], MLP [56] and Horus [48], allow the exchange of information between different languages. Like FFI-based approaches, RPC is limited to primitive type interoperation.

Another common approach to interoperability involves the use of a single data type model. This approach has been popularized by traditional databases, where all shared persistent data is represented using a relational data model. This results in maintaining multiple type definitions for the same interoperating entities - one in the language of the application and one in the schema of the database. Although extended relational databases [27] provide more complex data models and thus reduce the semantic gap between interoperating data types, they retain the same problem of requiring the maintenance of multiple type definitions.

Approaches to multiple database integration, i.e., supporting an application that needs access to multiple databases, have primarily involved creating a unified perspective on the multiple databases through some means of reconciling differences in the schemas, or in some cases the internal representations, used in the different databases. Because the multiple database integration problem, in its most general form, requires determination of semantic equivalence among different data type definitions, no single satisfactory solution to this interoperability problem exists. Nevertheless, a number of useful interoperability approaches have been proposed [19], such as various kinds of multidatabases [57] or federated databases [81]. Other related work that addresses essentially the same set of concerns from a perspective influenced by the AI view of information (or knowledge) and its processing has proposed such notions as mediators [146] or technology for knowledge sharing [97] to address the schema, model and internal-representation reconciliation facets of the multiple database integration problem.

Most recently, a set of interoperability approaches typified by the Object Management Group Common Object Request Broker Architecture (OMG CORBA), such as

CORBA [100], ODMG [28] and ILU [64], and event- or message-based software buses, such as Polylith [107] and Field [110], have received a great deal of attention. In these approaches, application developers describe interoperating types using a standard notation. With CORBA, for instance, the various kinds of objects would be described using IDL notation (ODL in the case of ODMG, ISL in the case of ILU, etc.). From these descriptions, application-specific type descriptions are generated, typically by automated tools (e.g., IDL to C++, IDL to Ada, etc.). While this approach is more flexible than FFI- and RPC-based approaches, and less error-prone and tedious then the common model approach, it is still unsatisfactory since it still requires the use of an external type system (e.g., an IDL). In addition, the generation tools can produce language-specific types that are non-intuitive to the application developer.

2.4 Summary

In this chapter we have reviewed the most relevant work with respect to name management models and mechanisms. We have also reviewed related work in the research area of interoperability. Although name management is crucial in almost all computing systems, there do not appear to exist any general or comprehensive models that allow for meaningful and rigorous descriptions of name management. At the same time, existing mechanisms, even the most modern ones, frequently have various shortcomings, which can make computing systems difficult to understand and operate. Traditional approaches to interoperability have tended to suffer a variety of shortcomings, most of which result from inadequate levels of abstraction. In the forthcoming chapters, we outline a systematic approach that improves upon previous work done in each of these three areas.

CHAPTER 3

THE PICCOLO FRAMEWORK

As we have demonstrated in Chapter 1, individual computing systems frequently employ distinct, specialized, often elaborate and idiosyncratic, approaches to name management. This can lead to computing systems that are difficult to use. Moreover, correct usage can be hard to achieve and even harder to confirm. Convergent computing systems present additional difficulties since their underlying components typically rely on different, often incompatible, name management mechanisms. We believe that this situation calls for improved support for reasoning about existing name management mechanisms, and also for improved name management mechanisms for use in individual and convergent computing systems.

This chapter introduces a general and flexible framework, called PICCOLO,[1] for describing different aspects of name management across a wide spectrum of computing domains. The PICCOLO framework, therefore, may serve as a basis for improved understanding of existing approaches as well as improved mechanisms for next-generation computing systems. As indicated in Chapter 2, despite the prominent role of name management in computing, there have been few, if any, comparably general and flexible frameworks. This is partly due to the ubiquitous nature of names — since almost any action in a computational system is based on a name and its meaning, name management tends to be intertwined with other components, capabilities and mechanisms in computing systems, often in subtle and complex ways. Thus, factoring out the underlying principles of name management appears to have been a difficult, if

[1] For historical reasons, the model's name is an acronym for Precise Interface and Context Control in Object Libraries and Objects.

not frequently overlooked, area of research. Although there have been several models involving or related to the use of names (e.g., [32, 45, 65, 93, 115, 128, 154]), such models have generally been specific to individual, or limited sets of, computation domains. PICCOLO generalizes various concepts found in many of these models and extends them with additional features to produce a unified perspective on a more complete set of name management issues across a much broader range of computational systems.

The next section provides some motivation for the overall structure of the PICCOLO framework. Section 3.2 then introduces the basic features of PICCOLO. Using these features as a basis, some interesting and useful name management-related properties are described in Section 3.3. The applicability of the framework is then demonstrated in Section 3.4 by using it to describe, as well as gain a better understanding of, distinct, yet representative approaches to name management. The chapter concludes with a general discussion of the benefits of the PICCOLO framework. Building on the basic foundations developed in this chapter, Chapter 4 develops some extensions to the PICCOLO framework, in particular focusing on the specifics of context specification and formation.

3.1 Motivating Piccolo

The PICCOLO framework provides a rigorous basis for detailed descriptions of name management. Although PICCOLO is a general and comprehensive model, it can be characterized as less abstract than other possible, perhaps more intuitive, modeling alternatives. Before delving into the specifics of PICCOLO, we briefly explore some alternative modeling strategies and show how they are inadequate, or awkward at best, in capturing many crucial details of name management. The purpose of this exercise is not to dismiss these and other potentially related modeling techniques, but rather to motivate and illustrate the level of abstraction utilized in our development

of PICCOLO and thus give some indication of where PICCOLO lies in the spectrum of modeling alternatives.

As an initial approach to describing name management in computing systems, we consider a simple, highly abstract model based on sets and functions. Specifically, we let *Name* be a set of names and *Object* be a set of (nameable) objects. Using these sets, we define the following functions.

$$Assign: \quad Name \times Object \rightarrow \emptyset$$

$$Resolve: \quad Name \rightarrow Object$$

As implied by the function names, *Assign* is a function supporting the assignment of names to objects, while *Resolve* is a function supporting name resolution.

Although such a model is appealing due to its overall simplicity, it turns out to be too abstract to serve as a viable approach for modeling any non-trivial approach to name management. Its most obvious shortcoming is that all the interesting and relevant name management details are completely hidden within the function definitions. For example, attempting to describe a file system directory mechanism, such as that in UNIX, or nested scoping rules in a programming language, such as Pascal, using this model would yield little or no insight into problems such as those discussed in Chapter 1.

One possible extension to this simple model might be to add the notion of *context*. Context is important in both human discourse and computing systems, and thus is an obvious candidate concept to represent in a model of name management. In human discourse, for example, context gives humans a frame of reference for some particular activity (e.g., participating in a conversation, reading a technical document, etc.). Similarly, establishing context in a computing system gives users the ability to specify how names should be interpreted with respect to some task (e.g., compiling a program, editing a file, etc.) Thus, we let *Context* be the set of valid contexts in

a computing system. We then modify the *Assign* and *Resolve* functions to include this new construct.

$$Assign: \quad Name \times Object \times Context \rightarrow \emptyset$$

$$Resolve: \quad Name \times Context \rightarrow Object$$

This minor alteration to the initial model is quite useful in terms of describing some of the more subtle details of name management. For example, by representing directories as contexts, we can begin to describe such things as name management in UNIX. Although this model increases our ability to describe certain aspects of name management at a finer level of granularity, it is still not expressive enough to describe many other features found in real systems. For example, this proposed formalism cannot account for the difference between directories and environment variables in the UNIX file system. More generally, such a model cannot accommodate different approaches to context specification and formation, nor how contexts are used and composed over time.

The PICCOLO framework is slightly less abstract than these and other similar models, yet general enough so that it be used to reason about and compare different approaches to name management. The model accomplishes this by essentially introducing some (still fairly abstract) structure on context. By doing so, it exposes and makes explicit the side-effecting taking place in the abstract functions presented earlier. In particular, the model allows for reasonably detailed accounts for aspects of name definition and name resolution. In the remaining sections of this chapter, we describe the basic features and properties of the PICCOLO framework. We also demonstrate its value by applying the framework to several existing approaches to name management in various settings.

3.2 Basic Features

PICCOLO is an object-oriented framework that is able to express and accommodate different approaches to name management in a relatively concise, easy to use manner. This section lays the foundation for the framework, using the concepts of *names*, *objects*, *bindings*, *binding spaces* and *contexts*. Although these, or closely related, terms have been used elsewhere in the literature, one of the distinguishing features of PICCOLO is the completeness and generality of its treatment of the way in which names are given meanings and how the meanings of names are interpreted. In particular, PICCOLO makes an explicit distinction between concepts explaining how names are assigned meanings – via *binding spaces*, which are primarily a means for creating and organizing collections of bindings, typically for user convenience – and how the meanings of names are interpreted – via *contexts*, which a system uses in interpreting names during its operation. Previous work in the area of name management has tended to commingle and confuse these concepts, while PICCOLO carefully separates and distinguishes them. As we will demonstrate later, making this distinction significantly facilitates modeling of many approaches to name management.

In this section, we describe each of the basic concepts of the framework in terms of an abstract data type (or, using object-oriented terminology, a class) and its associated abstract interface (or operations). In addition, the following notational conventions are observed:

- Terms in **sans serif** denote classes.
- Terms in ***slanted sans serif*** denote operations.
- Lower case Greek symbols indicate instances of classes. Unless otherwise noted, the following convention is used: η indicates an instance of the **name** class; o indicates an instance of the **object** class; β indicates an instance of the **binding**

class; σ indicates an instance of the bindingspace class; κ indicates instances of the context class;

- Braces ({}) denote sets.
- The result type of an operation is indicated by a double right arrow ($\Rightarrow$) symbol.
- A subclass relationship between two classes is indicated by a $\sqsubset$ symbol, i.e., foo $\sqsubset$ bar means that class foo is a subclass of class bar.
- The framework assumes the existence of various primitive classes (e.g., boolean) and their associated operations (e.g., operators $\neg$, $\wedge$, $\vee$).

In the remainder of this section, we present the basic features of the PICCOLO framework by describing and discussing each of the primary classes that make up the framework. Figure 3.1 gives a high level view of the prominent classes that make up PICCOLO, along with some of the relationships among them, where classes are represented as labeled rectangles and relationships between classes are represented by labeled directed arcs.

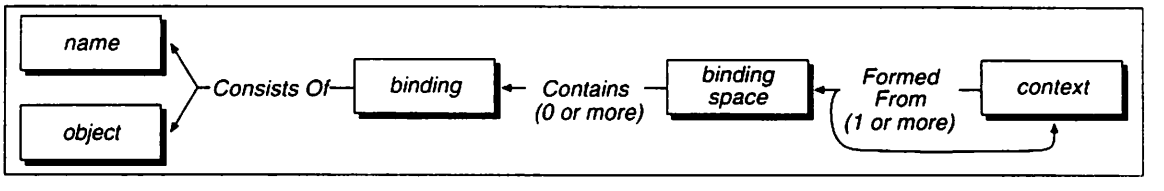


Figure 3.1 Basic Features of Piccolo

3.2.1 Names

Names are used to denote objects. In PICCOLO, a *name* is defined as a finite string of symbols formed from a given alphabet, Σ.[2] (The term *identifier* is used

[2]Names need not be restricted to character strings, linear sequences or one-dimensional structures. Graphics, icons, etc., could also be considered other kinds of names, but here we will model them, without loss of generality, as strings since an appropriate mapping from Σ^* to icons (or graphics, etc.) can always be defined.

interchangeably with the term *name* in PICCOLO.) As discussed in Chapter 1, from the perspective of PICCOLO, names have no necessary connection to the objects they denote. Therefore, changes to objects do not, as a side-effect, result in corresponding name changes.

3.2.1.1 Operations

Approaches to name syntax, formation and parsing have well developed foundations in formal language theory. In a PICCOLO model for a given name management mechanism, the relevant properties of names can be represented by properties of a name-formation grammar. The available operators for the name class will correspond to, and be defined for, these properties. Thus, for example, a simple model of names is described by the grammar (using standard regular expression notation) given in Figure 3.2(a). These names are simply arbitrary length strings over a given alphabet of symbols, Σ. For names of this kind, the only operations defined are *form-name* (Σ^+) $\Rightarrow$ name and equality (=). In many practical settings, a name management mechanism may impose a length bound on names. In this case, a richer description of valid names is required. For example, the grammar (again using regular expression notation) given in Figure 3.2(b) describes names whose maximum length is 5 (over the same alphabet, Σ).

Another common kind of model for names admits compound names and can be described using a grammar like that of Figure 3.2(c). In this model, a name consists of one or more strings over a given alphabet, Σ, separated by pre-designated symbols called *delimiters* (represented by the set Σ_d), where $\Sigma \cap \Sigma_d = \emptyset$. In addition to name formation and equality operations, this model supports a set of parsing operations, including *first* (name) $\Rightarrow$ name, *rest* (name) $\Rightarrow$ name, and *simple* (name) $\Rightarrow$ boolean.

$\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9\}$		
Name ::= Σ^+	Name ::= $\Sigma^1 \mid \Sigma^2 \mid \Sigma^3 \mid \Sigma^4 \mid \Sigma^5$	Name ::= simple | simple delimiter Name simple ::= Σ^+ delimiter ::= •
age, part123	*a, ab, abc, abcd, abcde*	*person, person•age, person•addr•zip4*
(a)	(b)	(c)

Figure 3.2 Example Name-Formation Grammars

3.2.1.2 Observations

A name-formation grammar can be viewed as a parameter to PICCOLO. In other words, the kinds of names supported by a particular approach can be described by a name-formation grammar. Once defined, the grammar (along with any necessary auxiliary parsing operations associated with the grammar) can simply be inserted into the framework. Clearly, the kinds of names supported by a particular approach are interdependent with other properties associated with the approach. Different properties related to binding spaces, for example, influence the kinds of names that can be expressed and *vice versa*. We explore the importance of such properties in greater detail in Sections 3.3.2 and 3.3.3, respectively.

3.2.2 Objects

An *object* is defined in PICCOLO as any entity of interest in a given setting that can do one or more of the following things:

- be assigned a name;
- refer to other objects using names;
- manipulate the names of other objects.

3.2.2.1 Operations

identical (object, object) $\Rightarrow$ boolean computes whether two objects are identical, returning true if they are, and false otherwise. As a notational convenience, a binary operator $\doteq$ can be used in place of the *identical* notation:

$$\forall o_i, o_j \in \text{object}.\ \textit{identical}\ (o_i, o_j) \equiv o_i \doteq o_j$$

type-of (object) $\Rightarrow$ class returns the class of an object. For our purposes in this dissertation, the classes of interest will be those of the PICCOLO framework, i.e., name, object, binding, bindingspace, and context.

3.2.2.2 Observations

The *identical* operation essentially encapsulates the notion of *identity* [71] in a name management mechanism. In computing systems, object identity is used to distinguish objects from one another. In some sense, identity acts a *meta-name* for an object, i.e., it is a special kind of name used to refer to and access an object independent of any other names (e.g., user-defined names) that may be assigned to it. Many systems associate a unique, internal identifier to objects (e.g., L-values in programming languages, file handles in file systems, and IP addresses in internets). In other systems, value-based equality might be used to determine object identity.

Since PICCOLO is an object-oriented framework, we can indicate that a particular class (of objects) in a computing system is supported by a name management mechanism by specifying an inheritance relationship between the (super-) class object and that class. As will become evident in our presentation of the other basic PICCOLO concepts, the class object is a parameter to various name management-related operations. In the UNIX file system, for example, files and directories are supported by UNIX's name management mechanism. In PICCOLO, this would be represented by indicating the relationships file $\sqsubset$ object and directory $\sqsubset$ object. Conversely, printers do not participate in UNIX's file system name management mechanism, so for our purposes, in PICCOLO printer $\not\sqsubset$ object.

3.2.3 Bindings

PICCOLO defines a *binding* as an association between a name and an object. Operations on bindings include binding creation, extraction, renaming and comparison.

3.2.3.1 Operations

bind (name, object) $\Rightarrow$ binding creates a binding between a name and an object.

name-of (binding) $\Rightarrow$ name returns the name component in a given binding.

object-of (binding) $\Rightarrow$ object returns the object component in a given binding.

rebind (name, binding) $\Rightarrow$ binding creates a new binding by using the object component in the given binding and the given name. Note that in a minimal framework, *rebind* can be defined in terms of the *bind* and *object-of* operations. It is provided, however, as a notational convenience.

name-collision (binding, binding) $\Rightarrow$ boolean determines whether two bindings have equal name components, but different object components.

3.2.3.2 Observations

PICCOLO models bindings as name–object pairs. In practical settings, however, a binding is typically implemented as a name–identity pair, where identity is an implementation-dependent shorthand (or meta-name) for an object. A variable binding in a (running) program, for example, consists of an (identifier, virtual address) pair, while in a file system, a directory entry consists of a (file name, file inode) pair. Identity is a useful construct since it provides a means of representing, locating and accessing abstract objects in a computing system. It also provides a convenient method for allowing the same object to participate in more than one binding at the same time. Thus, although in PICCOLO we model bindings as name–object pairs, we recognize that objects are generally represented by their identity in bindings as they are implemented in actual computing systems.

It is also worth noting that approaches to name management often augment bindings with auxiliary information to support other capabilities in a computing system. For example, bindings that consist of (name, type, object) triples [77] can provide support for typed objects, overloading or polymorphism, while the Napier programming language views bindings as quadruples of the form (name, type, mutability, object) [93], where the mutability value indicates whether or not the object may be modified. Similarly, file systems sometimes include time-related (e.g., time created, time last modified) and/or security (e.g., read/write privileges) information in bindings. PICCOLO is compatible with these extended notions of binding, but since such auxiliary information is generally not relevant to name management, we can usually restrict ourselves to the simple view of bindings presented here.

3.2.4 Binding Spaces and Contexts

Two closely related, yet distinct, name management concepts are *binding spaces* and *contexts*. A *binding space* in PICCOLO is defined as a collection of bindings that

serve as *definitions* for names. PICCOLO defines a *context* as a *virtual* collection of bindings that is 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, or other contexts.

From a purely abstract perspective, the notions of binding space and context are very similar. It is not surprising, therefore, that other models have simply neglected to distinguish these concepts and instead have chosen to model them both as a single concept. It is our contention, however, that distinguishing these concepts provides much more useful and precise descriptions of name management.

To help understand the difference between these concepts, consider name management in the UNIX file system and the Ada programming language. (We provide more detailed descriptions of the name management approaches found in these two systems in Sections 3.4.1 and 3.4.2, respectively.) In UNIX, binding spaces correspond to directories, while contexts correspond to the evaluation of environment variables and hence may comprise bindings from several different distinctive binding spaces. In Ada, binding spaces correspond to packages and contexts correspond to the result of evaluating the combination of the various with and use clauses utilized by packages. In both these examples, contexts are established by applying certain operations to extant binding spaces. Thus, contexts are best viewed as separate, distinct entities from binding spaces.

PICCOLO defines the following kinds of operations for binding spaces and contexts:

- creating binding spaces and contexts;
- inserting and modifying bindings in binding spaces;
- removing bindings from binding spaces;
- constructing contexts from existing binding spaces and contexts;
- resolving names in contexts.

3.2.4.1 Binding Space Operations

create-bindingspace () ⇒ bindingspace. forms a new, empty bindingspace.

define (binding, bindingspace) ⇒ bindingspace inserts a binding into a binding space.

undefine (name, bindingspace) ⇒ bindingspace removes all the bindings whose name component is equal to the specified name from a binding space. If no such bindings exist then the binding space returned by *undefine* is identical to the given one.

redefine (binding, bindingspace) ⇒ bindingspace inserts a binding into a binding space. In contrast to *define*, any pre-existing bindings in the given binding space whose name component is equal to the name component of the given binding are replaced by the new binding.

rename (name_1, name_2, bindingspace) ⇒ bindingspace replaces the name component of all the bindings in a binding space whose name component is equal to name_1 with name_2. If no such bindings exist, then the binding space returned by *rename* is identical to the given one.[3]

3.2.4.2 Observations on Binding Spaces

Note that the definitions of binding space operations leave certain properties unspecified. For instance, the definition of the *define* operation does not stipulate whether the inserted binding's name or object components may be identical to those of other bindings already in the binding space nor what will happen if they are. Leaving such properties unspecified allows the framework to be fairly general and flexible.

[3]Both *redefine* and *rename* are provided as conveniences and are not strictly necessary in a minimal model.

In practice, many name management approaches do, in fact, address these issues and we explore this question in greater detail in Section 3.3.1.

In PICCOLO, bindings and binding spaces have an important relationship to one another — in order for a name to have a meaning, it must be bound to an object in some existing binding space. For generality, in PICCOLO, a binding must first be created between the name and the object, and then the binding itself must be inserted into a binding space. In practice, this two-step process is often combined into a single step. In programming languages, for example, declaring a variable must be done within the confines of a procedure or module, while in file systems, creating a named file must be done with respect to some directory.

Binding spaces represent an important organizational unit provided to users by a name management mechanism. In particular, the ability to name binding spaces (in other words, if bindingspace $\sqsubset$ object) allows users to structure collections of named objects in useful and meaningful ways. Organization-related properties of binding spaces are examined in more detail in Section 3.3.2.

Finally, it is important to realize that while the above operations manipulate bindings in binding spaces, their application has no direct effect on the objects contained in the bindings. For example, removing a binding from a binding space via an application of *undefine* does not, as a side-effect, cause the destruction of the object contained in the removed binding. This is true even if the removed binding is the last binding to that object, although in that case the object may be rendered inaccessible.

3.2.4.3 Context Operations

create-context () $\Rightarrow$ context, forms a new, empty context.

infuse (bindingspace) $\Rightarrow$ context forms a context from a single binding space, where the resulting context consists of all the bindings in the given binding space.

resolve (name, context) ⇒ object returns an object bound to a name in a context. If there does not exist a binding in the context whose name component matches the given name, then a null object, ν, is returned (where ν is a distinguished instance of object).

The above are the minimally required operations to allow for name resolution (using contexts) in PICCOLO. They do not, however, capture many of the properties of contexts and binding spaces found in most name management approaches. Contexts get formed in various ways in different computing systems. Enumerated below are some standard, static context formation operations that go beyond the single binding space operation shown above. Although these operations are sufficient to explain name management in many systems, further refinements needed for describing more complex kinds of context formation are presented in Chapter 4.

union-override (context, bindingspace) ⇒ context, forms a context from a context and a binding space.[4] The resulting context consists of all the bindings in the given context plus all the bindings in the binding space whose name components do not exist in any of the bindings in the given context.

union (context, bindingspace) ⇒ context forms a context from a context and a binding space. The resulting context consists of all the bindings contained in the given context and binding space.

restrict (context, {name}) ⇒ context forms a context consisting of those bindings in the given context whose name components are contained in a specified set of names.

exclude (context, {name}) ⇒ context forms a context consisting of those bindings in the given context whose name components are not contained in a specified set of names.

[4]Meyer refers to *union-override* as "overriding union" in [85].

prefix (context, name) ⇒ context forms a new context by prepending the given name to the name component of each binding contained in a context.

3.2.4.4 Observations on Context

Notions of context are often implicit and vague in descriptions of approaches to name management, although their use and manipulation are crucial. Producing a clearer understanding of context and its use is a focal point of much of our work in name management. The operations above provide only some initial descriptions of context that are sufficient for understanding and reasoning about existing name management mechanisms. Later, in Chapter 4, we return to this important aspect of name management and consider some possible generalizations.

Also note that name resolution in PICCOLO is defined as a function rather than a relation, i.e., *resolve* can return only a single object. This reflects our belief that name resolution is a navigational process, i.e., that accessing a name with respect to some context should produce only a single result. Returning multiple objects, on the other hand, amounts to performing queries. This does not mean that a context may not contain two or more bindings with the same name component. In the event that a context has more than one object bound to the same name, then PICCOLO treats *resolve* as a non-deterministic operation. (In fact, many name management mechanisms treat such situations as exceptional, typically returning a null value as a result.)

3.3 Basic Properties of Name Management

To obtain a more thorough understanding of name management as well as to help distinguish among different approaches, the basic features presented in Section 3.2 are used in this section to develop some useful and interesting name management properties. In particular, properties addressing

- name uniqueness and aliasing,
- binding space organization, and
- compound names and name resolution

are developed. Name uniqueness and aliasing address the impact of pre-existing bindings on subsequent binding space and context operations. Binding space organization properties deal with the various ways that binding spaces are used to organize name spaces. Finally, properties related to compound names and name resolution examine the interrelationship of name-formation grammars and binding space organization, and its impact on context formation and name resolution.

3.3.1 Name Uniqueness and Aliasing

In many approaches to name management, an existing binding can have important implications for subsequent invocations of binding space and context operations. In particular, two important and related properties are *name uniqueness* and *aliasing*. Name uniqueness ensures that each name in a specific binding space or context is bound to only a single object. The same name may be bound to different objects only if the bindings are contained in distinct binding spaces or contexts. Aliasing, on the other hand, permits the same object to participate in multiple bindings. The following definitions of these properties apply to both binding spaces and contexts. For simplicity, we temporarily introduce the term *naming space* to include both binding spaces and contexts in these definitions.

Definition 3.1 (Name Uniqueness)

A naming space, Ψ, is name unique if and only if

$\forall \beta_i, \beta_j \in \Psi, i \neq j, \ \neg \textsf{name-collision}\ (\beta_i, \beta_j)$.

Definition 3.2 (Aliasing)

Two bindings, β_i and β_j, are aliases if and only if either

1. *there exists a naming space, Ψ, such that $\beta_i, \beta_j \in \Psi, i \neq j$, where*

   name-of $(\beta_i) \neq$ *name-of* (β_j) *and*

   object-of $(\beta_i) \doteq$ *object-of* (β_j), *or*

2. *there exist distinct naming spaces, Ψ_i, Ψ_j, such that $\exists \beta_i \in \Psi_i, \beta_j \in \Psi_j$ where*

   object-of $(\beta_i) \doteq$ *object-of* (β_j).

Most name management approaches support name uniqueness and aliasing properties for binding spaces and contexts. An obvious benefit of name uniqueness is that it avoids ambiguity (within binding spaces and contexts), while aliasing offers a degree of flexibility and autonomy to users of a name management mechanism. Some approaches employ variations on these properties. For example, binding spaces (i.e., packages) in Ada must be name unique, while contexts can contain ambiguous bindings. We discuss this particularly interesting aspect of Ada in greater detail in Section 3.4.2.2. In UNIX, binding spaces and contexts are guaranteed by the name management mechanism to be name unique. UNIX also provides a limited form of binding aliasing. Specifically, file objects may participate in aliases as long as the bindings do not span physical file devices in UNIX.

Name uniqueness and aliasing also have important implications for the binding space operations given in Section 3.2.4. Clearly if an approach does not support aliasing, then the *define* and *redefine* operations must be restricted so that aliases cannot be created, while if an approach supports name uniqueness, then the *define* and *rename* operations must prevent name collisions within a binding space.

These properties also have similar implications for the context operations. If name uniqueness within contexts is a property of a given approach, then the context formation operations must be guaranteed to return name unique contexts. In some

situations, name uniqueness for contexts falls out naturally from the combination of properties on binding spaces and the supported context formation operations. For example, in UNIX, since bindings spaces are name unique, and *union-override* is the context formation operation, contexts must always be name unique.

3.3.2 Binding Space Organization

Another important class of name management properties addresses organizational aspects of binding spaces.

Definition 3.3 (Flat or Structured Binding Space Organizations)
A binding space organization is structured if and only if
bindingspace $\sqsubset$ *object: otherwise, a binding space organization is flat.*

Examples of flat binding space organizations can be found in many early *file* systems and even in some more modern computing systems, such as object-oriented databases [82]. Mechanisms based on this property are simple to implement, relatively straightforward to understand and easy to use, especially for small systems. The primary disadvantage of such approaches, however, is they do not scale very well. Once a name has been assigned to an object, it cannot be used for a different object unless the original binding is removed or overridden (e.g., via *undefine* or *redefine*, respectively).

Name management mechanisms supporting structured binding spaces, on the other hand, provide much more flexible and natural organizational methods. Such organizations allow objects to be decomposed into logically-related, separate collections without impinging on the naming requirements of other objects. Most modern file systems, for example, are based on structured binding spaces, where binding spaces correspond to directories. Similarly, languages supporting programming-in-the-large are representative of such approaches, where binding spaces correspond to modules or packages. Although structured binding spaces offer a much richer approach to data

organization than the alternative flat organization, the existence of multiple binding spaces requires richer and more robust approaches for the specification, formation and use of contexts.

Within structured binding space organizations we can make further distinctions along the following three dimensions: *single rooted* or *multiple rooted*, *hierarchical* or *networked*, and *acyclic* or *cyclic*. To define these properties, it is convenient (although not strictly necessary) to represent various features of PICCOLO in terms of a graph called a *name space graph*.

Definition 3.4 (Name Space Graphs)

A name space graph, $nsg = (O_\sigma, O, N, B)$, is a directed graph where

O_σ is a finite set of non-terminal nodes representing binding spaces;[5]

O is a finite set of terminal nodes representing named objects (other than binding spaces);

N is a finite set of names;

B is a finite set of ordered triples of nodes and names (o_σ, o, η) denoting the relationship that o is bound to the name η in binding space o_σ, where $o_\sigma \in O_\sigma, o \in (O \cup O_\sigma), \eta \in N$.

The ordered triples determine labeled edges in the graph, i.e., the triple (o_σ, o, η) induces the directed edge $[o_\sigma, o]$ labeled η. In name management mechanisms that support aliasing it is possible to have multiple, labeled edges exiting a single node and entering another node as long as the associated labels are distinct, i.e., (o_σ, o, η_1) and $(o_\sigma, o, \eta_2) \in B$, where $\eta_1 \neq \eta_2$. In addition, we define the *in-degree* of a node $o_\sigma \in O_\sigma$ for a name space graph as the number of edges $[o'_\sigma, o_\sigma]$.

[5] In practice, a binding space could be empty, i.e, containing no bindings. Technically, in terms of the name space graph, the node representing such a binding space would be a terminal node. For modeling purposes, however, we assume that binding spaces are non-empty.

To consider properties of binding space organizations, we sometimes refer to a subgraph of the name space graph called a *binding space graph*.

Definition 3.5 (Binding Space Graphs)

A binding space graph, $bsg = (O_\sigma, O, N, B)$, is a name space graph where $O = \emptyset$.

Figure 3.3 shows several example binding space graphs, where circles represent binding Given this graph representation, we give the following definitions for binding space organization properties: space nodes and labeled directed arcs represent bindings.

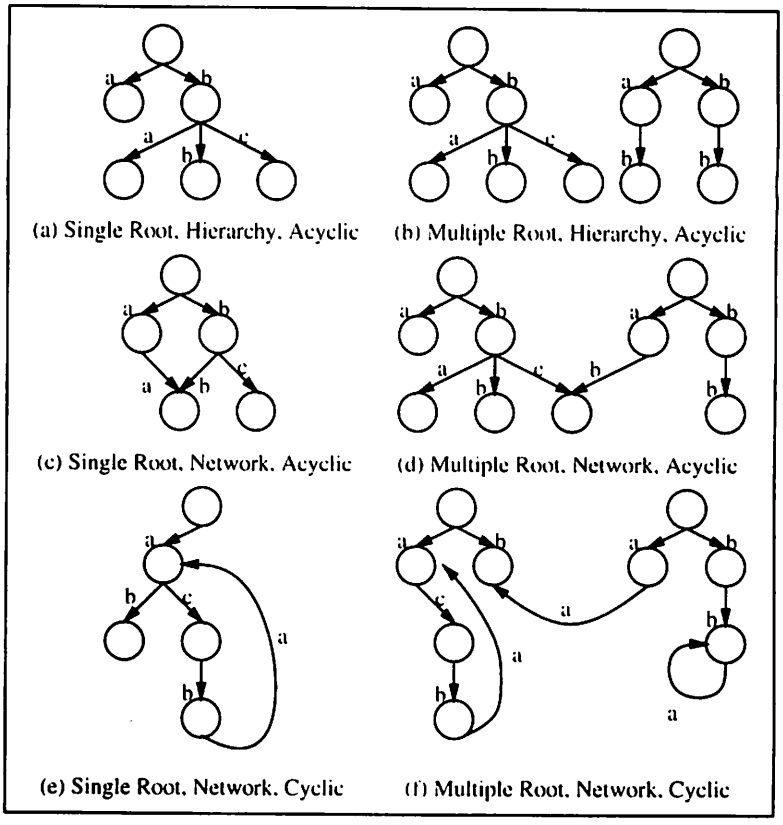


Figure 3.3 Examples of Binding Space Organizations

Definition 3.6 (Single or Multiple Rooted Binding Space Organizations)

Given a binding space graph, $bsg = (O_\sigma, O, N, B)$, let $Root = \{o_\sigma | in\text{-}degree(o_\sigma) = 0\}$.

A binding space organization is called single rooted if and only if $|Root| = 1$. A binding space organization is called multiple rooted if and only if $|Root| > 1$.

Definition 3.7 (Hierarchical or Networked Binding Space Organizations)

Given a binding space graph, bsg $= (O_\sigma, O, N, B)$*:*

A binding space organization is a hierarchy if and only if

$\forall o_\sigma \in O_\sigma$, *in-degree*$(o_\sigma) \leq 1$

Conversely, a binding space organization is a network if and only if

$\exists o_\sigma \in O_\sigma$, *in-degree*$(o_\sigma) > 1$

Definition 3.8 (Acyclic or Cyclic Binding Space Organizations)

Let a path from node o_{σ_1} *to node* o_{σ_k} *in a binding space graph, bsg, be a list of nodes* $[o_{\sigma_1}, o_{\sigma_2}, \ldots, o_{\sigma_k}]$ *such that* $[o_{\sigma_i}, o_{\sigma_{i+1}}]$ *is a labeled edge in bsg for* $i \in [1, \ldots, k-1]$.

A binding space organization is cyclic if and only if there is a path $[o_{\sigma_1}, o_{\sigma_2}, \ldots, o_{\sigma_k}]$ *such that* $o_{\sigma_1} \doteq o_{\sigma_k}$. *Otherwise, a binding space organization is acyclic.*

It is important to realize that these properties are, in general, independent of one another. As shown in Figure 3.3 for example, a binding space organization can be single rooted, networked and acyclic or it can be multiple rooted, hierarchical and acyclic. Of course, a cyclic binding space organization is, by definition, networked.

This particular collection of properties is significant primarily for two reasons. First, they play an important role in a user's view and use of a name management mechanism. If an approach supports networked binding space organizations, for example, then a binding space can potentially have more than one name bound to it. Second, as we will show in the next section, these properties also have important ramifications for other properties of name management approaches, in particular, ways in which compound names are formed and resolved.

3.3.3 Compound Names and Name Resolution

Name management mechanisms based on structured binding space organizations often employ compound names, as described in Section 3.2.1, as a notational shorthand for traversing or navigating the organization.

Figure 3.4 shows three example name space graphs. (In the figure, each node is

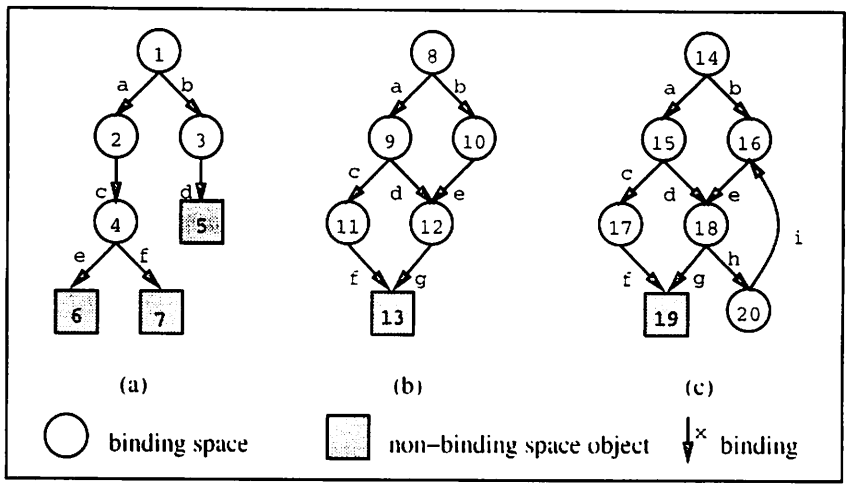


Figure 3.4 Examples of Name Space Graphs

labeled with a number as a simple indication of an object's identity.) Assuming the value for a delimiter in this approach is a period (.), then the names

a.c, a.c.e, a.c.f, b.d, c.e., c.f

are examples of compound names relative to the name space graph in Figure 3.4(a).

Continuing with the examples in Figure 3.4, notice the binding space graph (corresponding to its name space graph) in Figure 3.4(c) is cyclic, while the binding space graphs in Figures 3.4(a) and (b) are both acyclic. One interesting property of cyclic binding space organizations is that they can generate *infinite* compound name sets. For example, the names

b.e.g, b.e.h.i.e.g, b.e.h.i.e.h.i.e.g, b.e.h.i.e.h.i.e.h.i.e.g, ...

all refer to object 19 in Figure 3.4(c). By continually following the path of binding spaces named e, h and i, new compound names for object 19 can be created *ad infinitum*. The binding space graphs corresponding to Figures 3.4(a) and (b), in comparison, are acyclic, and can only generate finite name sets.

The ability of a name management mechanism to generate infinite compound name sets has important implications for modeling context formation and name resolution. More specifically, if an approach can generate only finite compound name sets, then a context formation operation can enumerate all possible names, possibly using a delimiter in the process of forming these names,[6] and as a result, produce a simple mapping between names and objects. Hence, name resolution can be described as a simple mapping from names and contexts to objects. Such an extensional model is attractive due to its conceptual simplicity. Unfortunately, some real computing system's name management mechanisms can be cyclic, so this appealing approach is insufficient as a general model.

Given an acyclic binding space organization, an extensional model of name resolution permits a slight simplification to the PICCOLO context formation operations. For example, we can refine the *infuse* context formation operation so that it exhaustively traverses all the named objects reachable from the given binding space and, based on a particular traversal, generates the appropriate compound names. Applying *infuse* to the root binding space in Figure 3.4(b), for instance, produces the following context (where subscripts indicate object identity)

$$\begin{aligned} \kappa &= \mathit{infuse}(\sigma_{root}) \\ &= \{(\mathsf{a}, o_9), (\mathsf{a.c}, o_{11}), (\mathsf{a.d}, o_{12}), (\mathsf{a.c.f}, o_{13}), \\ &\quad (\mathsf{a.d.g}, o_{13}), (\mathsf{b}, o_{10}), (\mathsf{b.e}, o_{12}), (\mathsf{b.e.g}, o_{13})\} \end{aligned}$$

[6]Used in this way, delimiters act as a notational convenience, primarily for users.

A subsequent invocation of *resolve* on the name a.c.f in this context

$$\textit{resolve}\ (\text{a.c.f.}\ \kappa)$$

returns object o_{13}.

Approaches supporting infinite compound name sets, in comparison, cannot be modeled in this fashion. In such approaches, the context resulting from an application of a context formation operation can contain only the (simple) names found in the bindings. Furthermore, delimiters must be treated as additional operators during name resolution, indicating that binding spaces must be dynamically traversed during the resolution process. (To help explain the details of name resolution, we employ some simple algorithmic control constructs, including exceptions, similar to those found in Ada.)

```
resolve-dynamic (η, κ) ≡

η_first = first (η)
η_rest = rest (η)
while not (empty (restof (η_rest))) loop
  o = resolve-dynamic (η_first, κ)
  if type-of (o) = bindingspace then
    σ = o
  else
    raise exception /* expected binding space */
  endif
  κ = infuse (σ)
  η_first = first (η_rest)
  η_rest = rest (η_rest)
endloop
return (object-of (β) where β = (η_first, o) ∈ κ)
```

If a name is simple, then name resolution amounts to a simple name lookup in the given context. If the name is compound, then the resolution operation must traverse the sequence of binding spaces (corresponding to each simple name in the compound name) in order to locate the desired object. Thus, at each step in the traversal, a new context must be formed and a new resolution must be performed.

For example, applying the *infuse* context formation operation to the root binding space in Figure 3.4(c) results in the following context:

$$\kappa = \textit{infuse}\ (\sigma_{root}) = \{\ (\mathsf{a}, o_{15}),\ (\mathsf{b}, o_{16})\}$$

Resolving the name a.d.h.i.e.g in this context

$$\textit{resolve}\ (\mathsf{a.d.h.i.e.g},\ \kappa)$$

actually requires a series of resolutions, until object o_{19} is finally returned.

3.4 Describing Name Management Mechanisms

In this section, we demonstrate how the framework can be applied to two distinct and representative approaches to name management. The objective of this section is not only to illustrate the framework's applicability, but to show how the framework can be used to gain better insight into, and compare different, name management approaches. We draw our examples from the domains of file systems and programming languages. Specifically, we first examine name management in UNIX (i.e., the UNIX file system) due to its widespread use in the development and utilization of many modern computing systems. Next, we look at name management in a relatively sophisticated programming language, namely Ada. Ada is an interesting case study, since unlike its many of its predecessors, it provides some flexible and powerful methods for manipulating names of language entities.

3.4.1 UNIX

In the UNIX file system, names are case sensitive character strings, where the the symbol "/" acts a delimiter in compound names. Nameable objects in UNIX include files and directories (represented by the classes file and directory, respectively), where binding spaces correspond to directories and files correspond to other nameable objects. Object identity is provided by *inodes*, which are unique numbers associated

with files and directories. In terms of PICCOLO, we describe the files and directories as follows.

$$\mathsf{file} \sqsubset \mathsf{object} \text{ and } \mathsf{directory} \sqsubset \mathsf{bindingspace} \sqsubset \mathsf{object}$$

Binding spaces are name unique in UNIX. In addition, all binding spaces contain at least two bindings — the name "." bound to the directory itself and the name ".." bound to the binding space's containing binding space. (The directory denoted by the name ".." is sometimes called a *parent* directory. Directories contained in bindings in a parent directory are sometimes called *child* directories.) A restricted form of aliasing is supported in UNIX. File objects may have aliases, while the only aliases permitted for binding spaces are the aforementioned "." and ".." bindings. Thus, a child binding space is permitted only one parent binding space.

In terms of PICCOLO, the binding space organization provided in traditional (non-distributed) UNIX is single rooted, networked and cyclic, while various distributed versions of the UNIX file system, such as Sun's Network File System (NFS) [132] and the Andrew File System [58], can be characterized as multiple rooted binding space organizations. The root binding space, denoted by σ_{root}, contains only a single binding, the name "/" bound to a child binding space.[7] In addition, associated with each active UNIX process is a specially designated binding space, called the *current working directory* in UNIX jargon. This binding space, denoted by σ_{cwd}, is used as the default for various file system operations. Figure 3.5 shows an example fragment of a UNIX name space graph.

3.4.1.1 Context and Context Formation

A popular misconception associated with UNIX is that the current working binding space serves as the current context for many UNIX commands. In fact, as illustrated by modeling with PICCOLO, the current working binding space is only part

[7] This child binding space is often designated the root directory in UNIX terminology and should not be confused with the root binding space in PICCOLO.

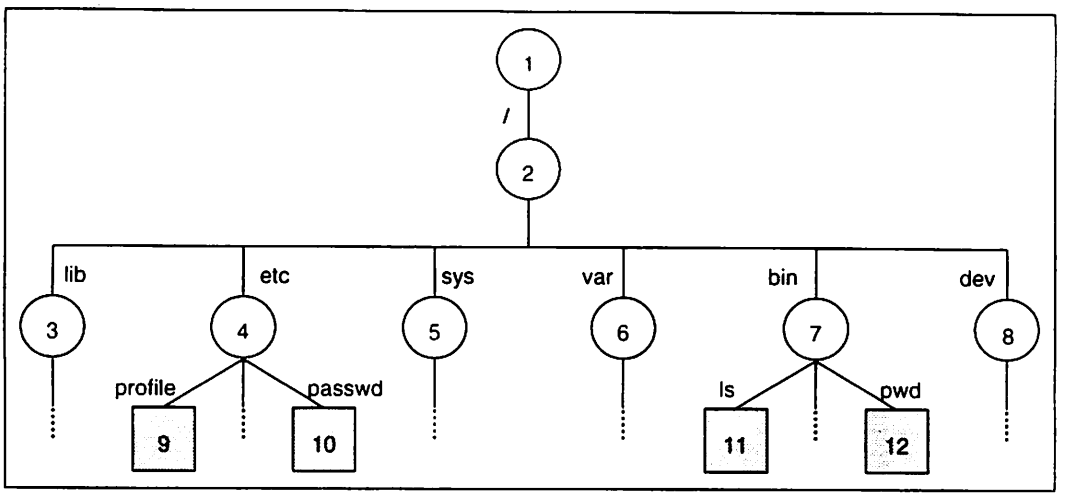


Figure 3.5 UNIX Name Space Graph

of the context formation equation. The other component of a current context in UNIX is the root binding space. For example, consider the cd command, whose primary purpose is to support navigation of a UNIX file system. In PICCOLO, we can describe its operation as follows (again making use of some simple control constructs).

```
cd η ≡

κ =union-override (σ_root, σ_cwd)
o = resolve-dynamic (η, κ)
if type-of (o) = bindingspace then
  σ_cwd = o
else
  raise exception /* expected binding space */
endif
```

As illustrated by this simple modeling exercise, context is actually formed by applying the *union-override* to the root binding space and the current working binding space. The resulting context, therefore, allows cd to resolve names either of the form cd $/\eta_1/\eta_2/\ldots$, cd $\eta_1/\eta_2/\ldots$ or cd η.

Another widely-used approach to context formation in UNIX involves the use of *environment variables*. An environment variable is a construct used to specify an

ordered sequence of directories. When a command or tool resolves a name, the sequence of directories is searched in the order specified in the appropriate environment variable. Recall from Chapter 1 that the TEXINPUTS environment variable is used to specify the context for macro files referenced in LaTeX documents. For example, we can describe the effect of setting TEXINPUTS as follows.

$$\text{TEXINPUTS= .:/usr/alpha} \equiv$$

$$\kappa = \textit{union-override}\ (\sigma_{root}, \sigma_{cwd}, \sigma_{/usr/alpha})$$

This and the prior example highlight some interesting shortcomings of name management in UNIX. First, the name "/" appears in every context in UNIX, even though there is no explicit command specifying that it do so. Moreover, there is no way of preventing the "/" from participating in a context.

Second, while there are many ways in which directories, or parts of directories, could be combined to form contexts, UNIX relies primarily on a single context formation operation, namely *union-override*. This single operation does not permit flexible and arbitrary definitions of context. Any kind of context formation based on the *restrict* or *exclude* operations, for example, must be performed manually (e.g., by creating redundant directories with copies or links to the desired objects). In fact, our choice of the particular set of context formation operations included in the basic PICCOLO framework is, in part, a response to problems of the kind illustrated by this example.

3.4.1.2 Reasoning About UNIX Commands

Describing name management in terms of PICCOLO also provides a much clearer and more precise understanding of various UNIX file system commands. For example, Figure 3.6 shows an abridged[8] version of the "man page"[9] for the mv command, which describes how mv is used to rename files and directories. Although it might seem that

[8]From Sun Release 4.1. For the sake of brevity, instructions detailing various low-level options for mv have been elided.

[9]*Man pages*, short for manual pages, are a standard on-line documentation tool in UNIX.

renaming should be a relatively straightforward task in a file system. examination of the man page reveals that the informal explanation given for the mv command is imprecise, incomplete and vague. It is not surprising, therefore, that in an analysis of UNIX command usage, the mv command was rated among the highest in terms of errors resulting from its use [53]. Using PICCOLO we can capture and describe

MV(1) MV(1)

NAME

mv – move or rename files

SYNOPSIS

mv *filename1 filename2*
mv *directory1 directory2*
mv *filename directory*

DESCRIPTION

mv moves files and directories around in the file system. A side effect of **mv** is to rename a file or directory. The three major forms of **mv** are shown in the synopsis above.

The first form of **mv** moves (changes the name of) *filename1* to *filename2*. If *filename2* already exists, it is removed before *filename1* is moved.

The second form of **mv** moves (changes the name of) *directory1* to *directory2*, *only* if *directory2* does not already exist — if it does, the third form applies.

The third form of **mv** moves one or more *filenames* (may also be directories) with their original names, into the last *directory* in the list.

mv refuses to move a file or directory onto itself.

Figure 3.6 UNIX Man Page for mv

the actual semantics of the mv command. So as not to detract from the analysis, the description below assumes the parameters to mv are non-compound names. Although the framework can certainly accommodate compound names, accounting for them, in this particular example, requires some additional parsing and type checking that slightly detracts from the central focus of the analysis.

```
mv η_from η_to ≡

κ = union-override (o_root, o_cwd)
o_from = resolve-dynamic (η_from, κ)
if o_from ≐ ν then
  raise exception /* name not bound */
else if type-of (o_from) = bindingspace then
  o_to = resolve-dynamic (η_to, κ)
  if o_to ≐ ν then
    σ_cwd = rename (η_from, η_to, σ_cwd)
  else if type-of (o_to) = bindingspace then
    β_mv = bind (η_from, o_from)
    σ_mv = o_to
    σ_mv = define (β_mv, σ_mv)
    σ_cwd = undefine (η_from, σ_cwd)
  else
    raise exception /* error */
  endif
```

[*continued from left hand column*]

```
else if type-of (o_from) = file then
  o_to = resolve-dynamic (η_to, κ)
  if o_to ≐ ν or type-of (o_to) = file then
    σ_cwd = rename (η_from, η_to, σ_cwd)
  else
    κ_tmp = infuse (o_to)
    o_tmp = resolve-dynamic (η_from, κ_tmp)
    if type-of (o_tmp) = bindingspace then
      raise exception /* error */
    else
      β_cp = bind (η_from, o_from)
      σ_to = o_to
      σ_to = redefine (β_cp, σ_to)
      σ_cwd = undefine (η_from, σ_cwd)
    endif
endif
```

Using PICCOLO, we can develop a carefully detailed description that exposes the inherent complexity of this seemingly benign command. Based on this exercise, it becomes evident that a user (or script) must be aware of a variety of different scenarios when using the mv command. Figure 3.7 uses name space graphs to illustrate these different scenarios. For each scenario, we assume

- a user is invoking the command mv a b, where σ_{cwd} is the binding space labeled "1" in the figure, and
- the figure to the left of the arrow represents the file system before invoking the command, and the figure to the right of the arrow represents the state of the file system after invoking the command. Scenarios that do not contain arrows mean that no name-related changes result from invoking the command.

As should be apparent from both the PICCOLO-based description and the name space graphs shown in Figure 3.7, the mv command is not only overly complicated, and hence difficult to comprehend for both novice and experienced users, but also has significant potential for causing serious problems. Use of PICCOLO in this manner has

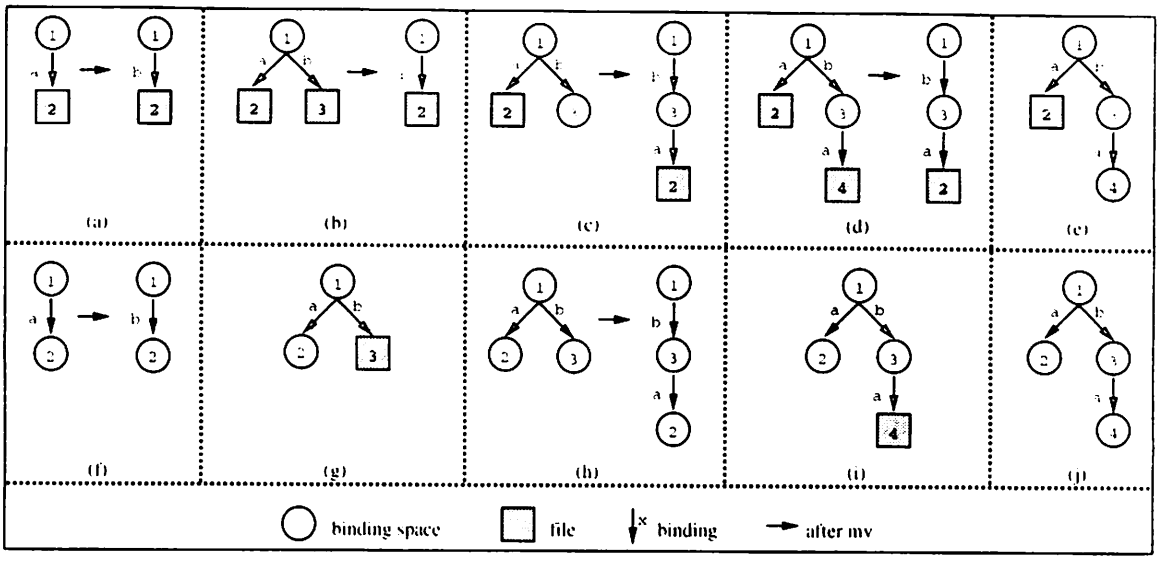


Figure 3.7 Effects of mv Command

the potential of suggesting improvements for commands, such as mv, to developers of new computing systems.

3.4.2 Ada

File systems are fairly dynamic naming structures, since users, programs, processes, etc. manipulate and modify named objects on a continual basis. Programs, in comparison, produce relatively static naming structures. Once a program (source) has been developed, its corresponding naming structure remains unchanged (except for eventual updates due to maintenance or enhancements). Furthermore, most programming languages are lexically scoped. This means that the meaning of names can be determined by their lexical position in source code.

Despite such differences, the PICCOLO framework can be used not only to gain a better understanding of name management in programming languages, but also to point out some interesting similarities and differences in file systems and programming languages. In this section, we apply PICCOLO to the Ada programming language (although we contend that the reasoning generalizes to other languages).

3.4.2.1 Overview

In Ada, names are alphanumeric character strings, sometimes separated by a delimiter (“.”). Name management-capable objects in Ada include packages, procedures, constants, variables, and types, where binding spaces correspond to packages. Since binding spaces are nameable objects, the binding space organization is hierarchially structured. In addition, a single Ada library can be viewed as a single-rooted organization, while the presence of multiple Ada libraries can be viewed as a multiple rooted organization. Finally, Ada's binding space organization (within a single Ada library) is hierarchical and acyclic. Unlike UNIX, therefore, the delimiter “.” does not act as an operator during name resolution. Figure 3.8 shows an example fragment of an Ada name space graph.

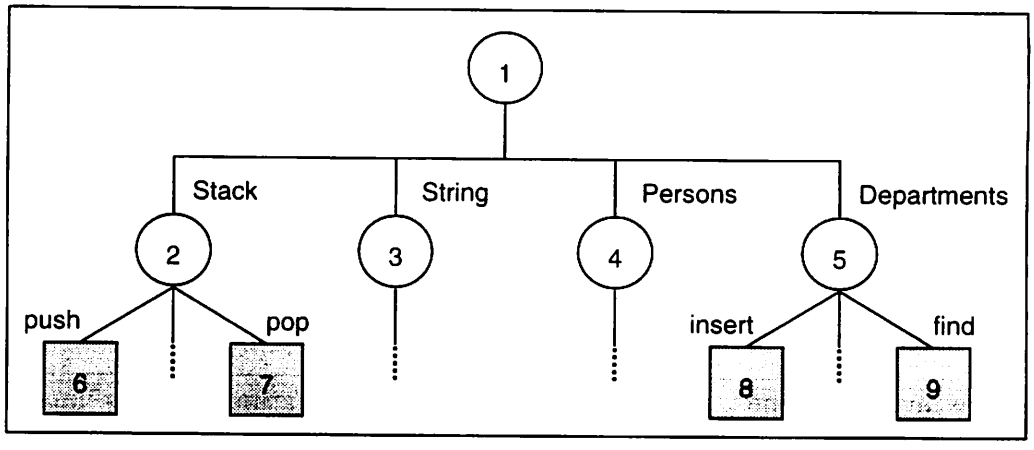


Figure 3.8 Ada Name Space Graph

3.4.2.2 Context and Context Formation

An interesting point of comparison between UNIX and Ada concerns context and context formation. While UNIX is limited to *union-override* semantics, Ada employs a variety of context formation operations. Here we use PICCOLO to describe some of these operations.

A primary purpose of a with clause in Ada is to import services defined in some other package, procedure or function. A with clause may be attached to a different package, procedure or function. For example, in the following Ada code fragments

```
package A is    with A;
I : Integer;    package B is
...             ...
```

the with clause attached to package B gives access to all the (visible) entities defined in package A. Using PICCOLO, this creates a context

$$\kappa = \textit{prefix}\ (\textit{infuse}\ (\sigma_A),\ \text{"A."})$$

where the binding space σ_A corresponds to package A. The resulting context, κ, consists of all the bindings defined in the A package, with their name components prefixed with the string "A." (e.g., $\{(\text{A.I}, o_I), \ldots\}$).

Complementing Ada's with clause is the use clause, which augments a context created by a with clause with the names of all the entities defined in a given package. Given the following Ada code fragments

```
package A is    with A;
I : Integer;    use A;
...             package B is
                ...
```

we can describe the context constructed for package B as

$$\kappa_1 = \textit{prefix}\ (\textit{infuse}\ (\sigma_A),\ \text{"A."})$$
$$\kappa_2 = \textit{infuse}\ (\sigma_A)$$
$$\kappa = \textit{union}\ (\kappa_1, \kappa_2)$$

Notice that Ada uses *union* semantics when forming contexts based on import clauses, as opposed to *union-override*.[10] Utilizing *union* semantics means that the order of the with clauses has no consequences for the resulting context. One interesting

[10]In fact, Ada, like many other programming languages, also uses *union-override* semantics when forming contexts in certain situations, i.e., lexical scoping rules correspond directly to *union-override* semantics.

aspect of context formation in Ada is that it does not enforce name uniqueness and hence admits the possibility of forming ambiguous contexts. Consider the following Ada code fragments:

```
package A is        package B is        with A, B;
I : Integer;        I : Integer;        use A, B;
...                 ...                 package C is
                                        J : Integer := I;
                                        ...
```

Using PICCOLO, we can model the context constructed for package C as

$$\begin{aligned}
\kappa_1 &= \textit{prefix}\ (\textit{infuse}\ (\sigma_{\mathsf{A}}),\ \text{"A."}) \\
\kappa_2 &= \textit{infuse}\ (\sigma_{\mathsf{A}}) \\
\kappa_{\mathsf{A}} &= \textit{union}\ (\kappa_1, \kappa_2) \\
\kappa_3 &= \textit{prefix}\ (\textit{infuse}\ (\sigma_{\mathsf{B}}),\ \text{"B."}) \\
\kappa_4 &= \textit{infuse}\ (\sigma_{\mathsf{B}}) \\
\kappa_{\mathsf{B}} &= \textit{union}\ (\kappa_3, \kappa_4) \\
\kappa &= \textit{union}\ (\kappa_{\mathsf{A}}, \kappa_{\mathsf{B}})
\end{aligned}$$

The resulting context can then be viewed as

$$\kappa = \{(\mathsf{A.I}, o_{\mathsf{A,I}}),\ (\mathsf{I},\ o_{\mathsf{A,I}}),\ (\mathsf{B.I},\ o_{\mathsf{B,I}}),\ (\mathsf{I},\ o_{\mathsf{B,I}})\}).$$

Notice that the resulting context contains a *name-collision* involving the name I. An Ada compiler will report this ambiguity only when the resolution involving the ambiguous binding is performed. Thus, in the example above, compiling package C results in an error. If the package did not include the reference to the entity named I, then the (potential) problem would not be reported and the compilation would have continued processing the source. Casting name management in terms of PICCOLO, therefore, suggests a possible enhancement to Ada compilers, that is, that they should, in the very least, report the potential for invalid contexts.

Finally, the Ada **renames** clause offers the possibility of augmenting contexts with bindings whose name components were not originally assigned to objects in an existing binding space. For example, in the following code fragments

```
package A is                    with A;
I : Integer;                    package B is
...                             package AA renames A;
                                J : Integer := AA.I;
                                ...
```

the renames clause augments the context for package B with bindings whose name components contain the prefix "AA.". Using PICCOLO, we can describe this approach to context formation as

$$\kappa_1 = \textit{prefix}\ (\textit{infuse}\ (\sigma_{\mathsf{A}}),\ \text{"A."})$$
$$\kappa_2 = \textit{prefix}\ (\textit{infuse}\ (\sigma_{\mathsf{A}}),\ \text{"AA."})$$
$$\kappa = \textit{union}\ (\kappa_1, \kappa_2)$$

As a result, package B can use both the names A.I and AA.I to refer to the integer object defined with the name I in package A.

3.5 Summary

Name management plays a critical role in almost all computing systems. Although numerous mechanisms have been developed over the years, there has been little or no work done in attempting to understand the fundamental principles and properties underlying these various approaches. In this chapter, we have developed the basic features of a general model of name management called PICCOLO. The basic features are presented in terms of an object-oriented framework, where each feature is represented as a class and its interface. We have also offered two kinds of evidence for the value of the PICCOLO model. First, using PICCOLO's features, we have defined several distinguishing properties related to name management. Second, we have shown how the basic concepts and properties provided by PICCOLO can be used to uniformly describe and compare various aspects of name management approaches found in disparate computing domains.

The basic PICCOLO framework offers several benefits to both users and designers of name management mechanisms. The basic concepts establish a comprehensive and

standard collection of name management-related terminology that is applicable to a broad spectrum of computing systems. This facilitates comparisons among different approaches, and thus provides a useful basis for assessing similarities and clarifying differences among various approaches. Utilization of these fundamental concepts also allows for more precise descriptions of name management mechanisms. Thus, one potential application of the framework might include augmenting natural language explanations of name management capabilities with PICCOLO-style annotations. In addition to serving as a useful reference for users, its use in this manner can also help designers avoid omissions and ambiguities, which are typical of informal descriptions. We provide additional techniques supporting this capability in Chapters 4 and 5.

An important feature of the framework is its explicit distinction between the concepts of binding space and context. As noted earlier, other name management models tend to commingle these concepts. It is our belief that distinguishing these concepts facilitates more rigorous and detailed descriptions of existing approaches to name management in computing. At the same time, it is clear that this distinction can sometimes become blurred. In the very abstract, binding spaces and contexts are similar in the sense that they both contain, either directly or indirectly, collections of bindings. What sets them apart is the set of operations provided for each concept. Hence, from a user's perspective these concepts are different, although their implementations may be similar.

In the next chapter, we build on the concepts defined in the basic framework, in particular, describing extensions to PICCOLO that focus primarily on various aspects regarding context, i.e., how and when contexts are formed, how contexts are associated with objects and how contexts are selected for use.

CHAPTER 4

UNDERSTANDING CONTEXT

The basic PICCOLO framework, presented in Chapter 3, represents an important step toward establishing a proper foundation for name management. While providing standard and general terminology for name management, and thus facilitating useful comparisons and descriptions, application of the framework can also help in revealing various subtleties, idiosyncrasies and anomalies inherent in existing approaches to name management.

One of the most crucial areas in the operation of almost any computing system is related to context. Just as humans rely on well-formed context(s) to conduct meaningful conversations, computing systems require consistent and coherent contexts in order to operate properly. For example, almost any programmer who has attempted to port or install a system developed by some other individual or external organization has been faced with the arduous task of determining a compatible context for that system in its new environment. All too often, such efforts are delayed, if not abandoned, due to unforeseen, or sometimes unsolvable, problems associated with establishing an appropriate context. In the end, context-related problems can be quite counter-productive and expensive to overcome.

Although the initial formulation of context included in the basic PICCOLO framework is extremely useful in characterizing and understanding various approaches, it is not sufficient for describing certain more complex facets of context manipulation. In particular, the basic version of the framework does not account for

- the possibility that individual components may have formed contexts independent of one another yet require a consistent interpretation of names, and
- the possibility that an object may require several contexts, each with perhaps a different associated method of context formation.

In this chapter, we focus on the specifics of context manipulation in convergent computing systems. We begin the chapter by considering some scenarios in which context-related problems complicate the use of some representative convergent computing systems. With these scenarios as motivation, we then, in Section 4.2, present a small set of extensions to the PICCOLO framework aimed at describing these, and other related, problems. Section 4.3 shows how this extended version of the framework provides insights into, and a basis for informal reasoning about, name management mechanisms and the problems (in particular, context-related problems) that arise in their use. Section 4.4 summarizes and assesses the work presented in this chapter. Chapter 5 builds on these extensions, by showing how the framework's semantics can be formalized and demonstrating its potential as a basis for formal reasoning and automated analysis tools.

4.1 Context-Related Problems

We now turn our attention to two scenarios in which context-related problems complicate the use of some representative convergent computing systems. In general, each scenario illustrates how inadequate approaches to context management can lead to surprising, somewhat non-intuitive, interpretations of names. In Section 4.1.1, we examine context-related problems in Make, while in Section 4.1.2 we study the role of context in Napier, a state-of-the-art persistent programming language.

4.1.1 Make

Our first scenario is based on a variation on the Make example presented in Chapter 1. Consider the Makefile show in Figure 4.1. The (C++) source code for a hypothetical tool is stored in a file named tool.C and the corresponding executable is

```
# Makefile for compiling a tool

# Specify context for Make
VPATH = bin:src

# Compile tool.C if it's
# more recent than tool
tool: tool.C
        cd src; CC tool.C -o tool
```

Figure 4.1 Example Makefile

represented by a file named tool. When the Makefile is executed, Make will examine the modification times of each of these files. If the source code file's time-stamp is more recent than the executable file's time-stamp, then the C++ compiler (indicated by the command CC) is invoked on the source code file, resulting in a new executable file. Using PICCOLO, the creation of this new executable can be described in terms of *redefine* semantics, i.e.,

$$\sigma_{src} = \textit{redefine}\ (\textit{bind}\ (\eta_{tool}, o_{tool}), \sigma_{src})$$

where σ_{src} corresponds to the src directory. Normally, experienced users of Make would expect that any subsequent invocations of the Makefile without any intervening modifications to the tool's source code would not result in a re-compilation (assuming the initial compilation was successful).

An application developer trying to reason about this Makefile might believe that all names used in the Makefile are interpreted in a single, uniform context. In fact, however, there can be several, often conflicting and sometimes related, contexts active

simultaneously. An interesting aspect of context management in Make is that separate contexts exist for each of the following kinds of names: targets and dependencies, commands and tools to execute, arguments given to these commands and tools, directories to traverse, directories containing included Makefiles, directories containing libraries, etc. In general, the syntactic category to which a particular name belongs determines which context is to be used in interpreting that name. In contrast, in most programming languages the relevant context is determined entirely by the position of a name within the program: in effect there is exactly one context "active" in any given region of the program. Thus, while a given name will have exactly the same meaning throughout a given region of a program, in a Makefile different occurrences of the same name on the same or adjacent lines may have different meanings! For example, in the Makefile shown in Figure 4.1, the VPATH environment variable specification forms a context for targets and dependencies, while the src directory implicitly serves as the context for arguments to the C++ compiler.

An additional complicating factor is introduced by the state of the file system. Consider the Makefile in Figure 4.1 in the setting of each of the three subdirectory structures in Figure 4.2. In each of these settings, the binding of the name tool is

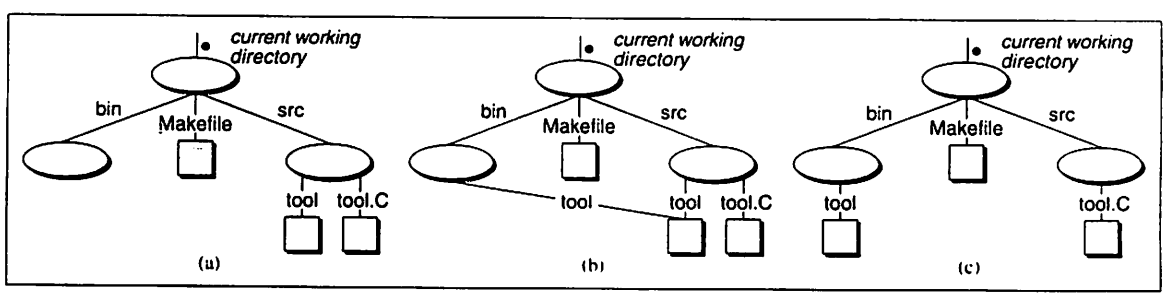


Figure 4.2 Three Example Subdirectory Structures

varied slightly. In Figure 4.2(a), the name tool is bound to an executable in the src directory, while in Figure 4.2(b), the name tool is bound to the same executable in both the src and bin directories. (In UNIX terminology, a *hard link* exists for the file

named tool.) Finally, in Figure 4.2(c), the name tool is bound to an executable in the bin directory. It turns out that the two relevant contexts (i.e., those specified by the VPATH and src directory, respectively) for the example Makefile of Figure 4.1, given the file system states shown in either Figure 4.2(a) or 4.2(b), are consistent while the contexts resulting from the file system state as shown in Figure 4.2(c) are inconsistent. Specifically, the name tool appearing as a target on the second to last line of the Makefile has an interpretation that is different from that of the same name appearing in the last line. Assuming the source code file is more recent than the executable file, invoking the Makefile will produce a new executable file in the src directory. Since the VPATH specification results in a context that does not include this new binding, it will not be visible to the dependency analysis component of Make. Thus, subsequent invocations of the Makefile will result in unnecessary re-compilations of the tool source code, even though the source code may have already been successfully compiled.

4.1.2 Napier

Our next scenario is based on Napier, a persistent programming language that combines many of the capabilities traditionally found in programming languages, database systems and operating systems. A distinguishing feature of Napier is its provision of orthogonal and transparent persistence. This means that persistence is treated as an additional, independent property of types and that programs manipulate both transient and persistent objects in a uniform manner [149]. Details regarding Napier can be found in [94].

An interesting property of Napier's approach to name management is that it uses a single name management mechanism for both transient and persistent objects. In other words, all objects in Napier are nameable, including objects of primitive types (e.g., integers, floats), complex types (e.g., abstract data types) and procedures. As noted earlier, bindings in Napier are (name, object, type, mutability)

quadruples. Included in Napier's type system is the type *environment* (denoted by the env construct), which corresponds to a binding space in the terminology of the basic PICCOLO framework. Name uniqueness and aliasing properties are associated with environments and the binding space (environment) organization in Napier is a single rooted,[1] networked and cyclic organization.

Before delving into the specifics of context in Napier, we provide PICCOLO-based descriptions of Napier's central name management-related constructs. Table 4.1 explains each of these constructs by giving a simple example of its use and a corresponding PICCOLO description. To summarize, the in-let and drop-from constructs

Table 4.1 Name Management Operations in Napier

Construct	Example	PICCOLO
in-let	in myEnv let myInt := 0	β = *bind* $(\eta_{myInt}, o_{integer})$ σ_{myEnv} = *define* (β, σ_{myEnv})
drop-from	drop myInt from myEnv	σ_{myEnv} =*undefine* $(\eta_{myInt}, \sigma_{myEnv})$
use-with	use myEnv with myInt : int in myInt := 5	κ = *restrict* (*infuse* (σ_{myEnv}), η_{myInt})

are used to create and remove bindings (in/from binding spaces), respectively, while the use-with construct is used to construct contexts. In addition, use-with constructs can be nested, corresponding to *union-override* semantics in PICCOLO.

Figure 4.3 depicts four (fragments of) Napier procedures that access and manipulate a particular binding located in the persistent store. Specifically, the procedures in Figures 4.3(a) and 4.3(b) each form a context containing a binding to the object named counter (which is located in the root binding space), and then print and increment its value. The procedure in Figure 4.3(c) removes the binding from the root binding space and then creates a new binding (containing a new object with the same

[1] The root binding space in Napier programs can be accessed by invoking a specially designated procedure named PS.

name). The procedure in Figure 4.3(d) simply invokes, in sequence, the increment1 and increment2 procedures shown in Figures 4.3(a) and 4.3(b).

```
! increment value of
! persistent counter
let increment1 := proc ()
  use root with counter : int in
  begin
    printInt (counter)
    counter := counter + 1
  end
```

(a)

```
! increment value of
! persistent counter
let increment2 :=
  use root with counter : int in
  proc ()
  begin
    printInt (counter)
    counter := counter + 1
  end
```

(b)

```
! remove old counter
! add new counter
let dropadd := proc ()
  begin
    drop counter from root
    in root
      let counter := 0
  end
```

(c)

```
! invoke the
! procedures
let callprocs := proc ()
  begin
    increment1 ()
    increment2 ()
  end
```

(d)

Figure 4.3 Napier Procedures

A programmer attempting to understand the two increment procedures might initially observe no difference in their behavior. For example, suppose the value for the integer object bound to the name counter is initially set to 10. Invoking the procedure callprocs (declared in Figure 4.3(d)) results in printing the values

10 11

Suppose now that the dropadd procedure (declared in Figure 4.3(c)) is invoked, perhaps by a different user and unbeknownst to the programmer. This time invoking the procedure callprocs results in printing the values

0 12

Here, the programmer will most likely become confused since he/she has not made any modifications to the two increment procedures, yet the increment1 procedure appears to be using a new value for the counter variable, while the increment2 procedure continues to use the old value.

Although consistent with Napier's (informal) language definition [35], such behavior can be difficult to reason about from a name management perspective. The difference in behavior is due to the placement of the use-with construct in each of the increment procedures. In the procedure shown in Figure 4.3(a)), the use-with

construct occurs in the body of the procedure. In contrast, in the procedure shown in Figure 4.3(b) the use-with construct occurs in the procedure declaration. In the former case, the context for the procedure is formed each time the procedure is invoked, while in the latter case, the procedure's context is formed only once, specifically, when the procedure is created (more accurately, when the assignment statement associated with the procedure definition is first executed). Thus, depending on how other procedures manipulate bindings in shared binding spaces (in this scenario, the persistent store represents a shared binding space), it is possible that different procedures may appear to have contexts that are consistent with one another, but are, in fact, referring to totally different objects.

We recognize that part of the problem in this scenario is due the aesthetics of the language, i.e., that the subtle placement of a language construct, in this case the with-use construct, can have dramatic effects on the semantics of a program. However, programmers typically rely on source code as a means of understanding, maintaining and debugging programs. As shown above, examining source code is clearly not sufficient for understanding contextual relationships between the two increment procedures.

This simple exercise also points out some interesting aspects related to Napier's approach to context as compared to approaches used by traditional file-based programming languages. In particular, in traditional programming languages context for data files is always created during the execution of a program.[2] Napier, on the other hand, provides two separate times for forming context for persistent objects, each resulting in different interpretations for the same name.

[2]In traditional programming language terms, names for files are controlled by dynamic scoping rules, while names for transient objects are controlled by static scoping rules.

4.1.3 Summary

The problems identified in these scenarios are representative of name management difficulties that can complicate the engineering of even relatively simple systems. We have encountered exactly these kinds of problems in practice, for example in real Makefiles, developed by outside organizations, that were only slightly more complicated than the simple example presented here. These experiences have strengthened our belief that significant improvements in name management capabilities will be required if large-scale convergent computing systems are to be successfully built and maintained. One such improvement involves developing a richer understanding of context. In the next section, we present some extensions to PICCOLO that provide the necessary structure for reasoning about context in convergent computing systems.

4.2 Extended Features of Piccolo

The basic PICCOLO framework, as presented in Chapter 3, is sufficient for describing and reasoning about several aspects of different approaches to name management. As shown in the scenarios above, however, the control and management of context in convergent computing systems can be quite intricate and complex. Although our initial formulation of context has proven to be quite useful, we believe that it requires some additional structure to account for many of the context-related features present in convergent systems. In this section, we develop a small collection of extended features for the PICCOLO framework aimed at satisfying this objective. The primary concepts underlying these extended features are *epochs*, *context formation templates* (or CFTs), *context formation processes* (or CFPs) and *extended closures*. Epochs designate discrete times at which name management activities occur. CFTs are directives capturing the contextual specifications for an entity, while CFPs are used to model the process used to form a context. An extended closure associates this information with an object.

We explain each of these concepts in greater detail in the remainder of this section. We note that this presentation is slightly less formal than the explication for the basic PICCOLO concepts given in the previous chapter. Partly in response to this, we show more formal definitions of these concepts in the next chapter. In this chapter, our primary objective is to introduce these concepts and demonstrate their applicability and utility at a fairly high level of abstraction.

In the following section, we demonstrate the utility of these features, using them to reason about context in convergent computing systems, in particular, applying them to the Make and Napier scenarios presented in Section 4.1.

Epochs One shortcoming of most existing models of name management is the typical assumption that all name management activities occur at one particular time during the lifetime of a system. This makes it difficult to model many complex situations, so the PICCOLO framework provides a means for explicitly representing distinct times at which name management activities may occur. In PICCOLO, an *epoch* denotes a particular time period during which name management activities may take place. The set of all epochs is modeled by an enumeration such as $\mathsf{epoch} = \{\epsilon_1, \epsilon_2, \ldots\}$. For example, in a typical programming environment, a representative set of epochs might include *language definition*, *language implementation*, *compilation*, *linking*, *loading*, and *run-time*. For each one of these epochs, programmers make important assumptions about contexts. A context for language keywords, for instance, is established at the language definition epoch, while a context for program libraries is established at the link epoch.

Context Formation Templates A *context formation template* (or CFT) is a collection of directives governing the formation of contexts. The definition for the actual contents of a CFT is intentionally general, allowing for a broad spectrum of CFT directives. In modeling a particular approach, we will need to refine the

CFT concept, specifying the actual information that will be included in CFTs. In a typical approach to name management we can further distinguish between two kinds of CFTs:

Requestor: A CFT *requestor* (or CFTR) describes the context requirements for an object. In particular, it specifies a collection of names referenced by the object. It can additionally include directives indicating such things as preferred sources of definitions (i.e., binding spaces) for the referenced names and preferred times (i.e., epochs) for context formation or modification steps.

Consider, for example, the LaTeX document processing tool. This tool relies on not one, but several environment variables for controlling the meanings of various names. The TEXINPUTS variable is used to specify a context for macro files, while the TEXFONTS variable is used to specify a context for font files. Each of these variables can be viewed as corresponding to a portion of the CFTRs associated with a LaTeX document.

Provider: A CFT *provider* (or CFTP) describes the context definitions potentially available from an object. Since binding spaces in PICCOLO serve as collections of bindings, CFTPs are typically associated with binding space objects. Analogously to CFTRs, CFTPs can additionally specify such things as preferred targets (i.e., other objects) for these definitions and preferred times for context formation or modification steps.

In most traditional naming schemes, a binding space has almost no control over how its bindings are made available to other objects (although security capabilities sometimes operate in this capacity). One reason for including this particular refinement to CFTs is that PICCOLO is partly inspired by Wolf's Precise Interface Control framework [153]. Although PICCOLO is more general than PIC (which addresses a different problem domain).

PIC makes a useful distinction between requisition and provision of access to an object. In addition, more recent name management mechanisms have included provision information in forming context. Examples include **export** declarations in Modula-3 and **protected** and **friend** specifications in C++, both of whose semantics can be described using CFTPs.

Context Formation Processes A *context formation process* (or CFP) is a procedure that produces a context from an initial context and one or more CFTs. As with CFTs, the actual contents of CFPs are intentionally general and must be specialized when modeling any particular approach. A CFP might, for instance, be based on the context formation operations defined in Chapter 3. A simple CFP, for example, might correspond directly to *union-override*. Other types of CFPs might incorporate more sophisticated procedures for creating and analyzing well-formed contexts. The PIC-based analyses, for instance, can be viewed as a kind of CFP, enforcing access control constraints based on properties of request-provide specifications. Other possibilities include CFPs based on law-based systems [89], which essentially permit the inclusion of attributes in CFTPs, and subject-oriented programming techniques [54], which allow an object to have multiple, simultaneous interfaces.

Extended Closures Applying the PICCOLO framework involves defining a specific set of epochs appropriate to the host system's context formation and manipulation needs, specific kinds of directives that can appear in CFTs and one or more specific CFPs. It also implies a suitable generalization of the concept of closure, such as the following. In PICCOLO, an *extended closure* is an association between an object, an initial context, a set of context formation templates for directing the incremental formation of future contexts, and a set of context

formation processes that will be directed by those CFTs during incremental formation of future contexts.

Thus, given an object, we can directly obtain its associated context-related information. The use of extended closures assumes a simple model of computation. At each epoch change, an object's closure must be examined and a new context must be formed, e.g., if its CFTs indicate that a new context is required for the new epoch.

4.3 Application of the Extended Features of Piccolo

We envision two major categories of use for the extended features of the PICCOLO framework. First, they can serve as a basis for analyzing context control mechanisms and problems. Careful description of a particular name management mechanism in terms of PICCOLO's extended set of constructs permits rigorous reasoning about and comparison of various existing, proposed or possible approaches. We give examples of this use of the model below, where we use it in informal reasoning about some example name management questions, and also in the next chapter, where we base more formal analyses on a suitably formalized version of the PICCOLO framework.

Second, the PICCOLO model can serve as a foundation for defining and implementing name management mechanisms. By defining a mechanism using the model, we can avoid *ad hoc* solutions, reason about properties of a mechanism before implementing it, and achieve uniformity in the functioning of the mechanism. We explore this facet of PICCOLO in greater detail in Chapter 6.

As an illustration of the PICCOLO framework's potential value for analyzing mechanisms and problems, we return to the scenarios described in Sections 4.1.1 and 4.1.2.

4.3.1 Make

Consider the following informal reasoning about the interpretation of names in the simple Make example of Figure 4.1 and the subdirectory organization in Figure 4.2(c). Guided by the PICCOLO model, a developer would be more likely to attempt to understand Make's name management in terms of context formation epochs, name resolution epochs, extended closures and which constructs in a Makefile may have extended closures associated with them. Pursuing this reasoning leads to several insights. One insight is that there are two separate name resolution epochs associated with the actions of this simple Makefile, one with dependency analysis and the other with execution of the compile (CC) command. A related observation is that each name resolution epoch has a corresponding context formation epoch. An attempt to produce even an informal description of the extended closure information associated with each Makefile construct reveals that, although they use essentially the same CFP (based on the *union-override* of a series of binding spaces) and all the CFTPs are trivial (reflecting the simplicity of access controls in the UNIX file system), their respective CFTRs are quite different: the CFTR associated with the dependency analysis epoch is derived from the value of the VPATH variable, while the CFTR associated with the compilation is essentially a pointer to the src directory. Pursuing this reasoning a step further reveals that the interpretations of the two occurrences of the name tool.C in the example Makefile will be the same, which was presumably the developer's intent. On the other hand, similar reasoning also reveals that the interpretations of the two occurrences of the name tool will be different, with the first referring to the existing object named tool in the directory (binding space) named bin while the second will refer to a new object that will be created in the src directory as a side effect of the compilation. Since this would not succeed in rectifying the dependency failure that triggered the compilation, this is probably not what the developer intended. Applying even such informal reasoning, under the guidance of the concepts of the

PICCOLO model, can reveal the error, assist the developer in revising the Makefile to correct it, and then serve as a basis for confirming that the revised version does indeed realize the behavior intended by the developer.

We return to this example in Chapter 5, where we repeat the above analysis in more detail using a suitably formalized version of our PICCOLO model of name management in Make and UNIX.

4.3.2 Napier

We can pursue a similar line of reasoning to understand context formation in Napier. First, we define two epochs, $\epsilon_{\text{procedure-entry}}$, corresponding to the time a procedure is entered, and $\epsilon_{\text{procedure-create}}$, corresponding to the time a procedure is actually created. (A procedure is created in Napier when it is assigned a name. For example, the four procedures shown in Figure 4.3 are created when their respective assignment statements are executed.)

Next, we instantiate CFTs for Napier. In particular, a CFTR formulation for Napier includes the names that are required in a context, an epoch indicating when the context is formed, and a binding space designating which environment should be used to form the desired context. Since the names of objects are visible to all objects, CFTPs are not required in modeling context formation in Napier.

A CFP for Napier can then be formulated in terms of the basic PICCOLO context formation operations. Specifically, an appropriate CFP can be defined as:

$$\text{CFP}_{Napier}\text{: } \textit{restrict}\ (\textit{infuse}\ (\ \sigma_{\text{env}}),\ \eta_1, \ldots, \eta_n)$$

Given these general descriptions, we can now give specific descriptions of the contextual requirements for the increment procedures in Figure 4.3 by using the extended features of PICCOLO. These descriptions precisely illustrate how contexts are being formed and manipulated in Napier. Figure 4.4 shows the extended closures for each of the increment procedures, as well as the persistent store, throughout various stages

(i.e., epochs) in the scenario from Section 4.1.2. In stage 1, the persistent store contains a single binding — the name counter bound to an integer object (labeled with the object id A), whose initial value is 10. In addition, the extended closures for each of the increment procedures show that their respective contexts are initially empty.

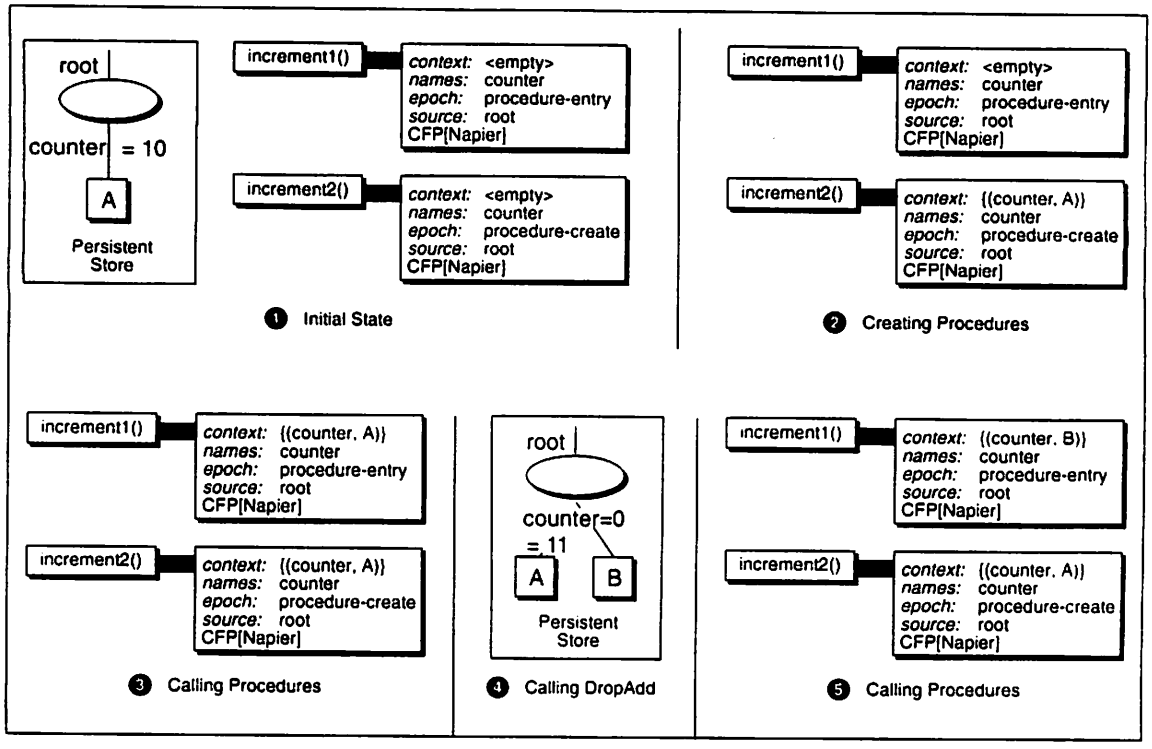


Figure 4.4 Reasoning About Context in Napier

They also indicate that they use the same CFP. In stage 2, the procedures are actually created. Notice that the context for the increment2 procedure now contains a binding, while the context for the increment1 procedure remains empty. The existence of a binding in the context for increment2 can be traced to its CFTR specification, which indicates that a context should be formed for the procedure at the time the procedure is created. In the third stage, the increment procedures are called. At this point, the context for increment1 is now augmented with the counter binding from the root binding space. We also see that the contexts used by the two procedures are consistent with one another, i.e., they both refer to identical objects using the

same names. In stage 4, the dropadd procedure is invoked. This results in a new object (labeled with the object id B) in the root binding space, whose initial value is 0. Note that the original object (i.e., object A) is not deleted from the persistent store. It is simply not bound to any name in the binding space organization for the persistent store. Finally, in stage 5, the increment procedures are called again. At this point, we see precisely why the procedures print different values for the integer named counter. Once formed, the context for the increment2 procedure remains fixed throughout each of the stages, while the context for the increment1 procedure is rebuilt each time increment1 is invoked by some client. Since the root binding space contains a new object with the same name, a different integer value is printed by the increment1 procedure.

4.4 Summary

In this chapter we have defined some extensions refining the PICCOLO framework to permit more detailed accounts of context-related aspects of name management. The extensions revolve around the development of several concepts, specifically epochs, CFTs, CFPs and extended closures. By instantiating these general concepts appropriately for a particular mechanism, we can obtain fine-grained descriptions of context. We have demonstrated the applicability and utility of these extensions by applying them to some representative name management mechanisms.

The extended features of the PICCOLO framework, together with the fundamental concepts defined in Chapter 3, offer some particular advantages over existing, related name management models. By isolating name management concepts, PICCOLO allows for a more detailed, insightful and intuitive understanding of name management. For example, in denotational models, the name management concepts we have identified tend to be dispersed among the various functions used to model a particular language. This is not too surprising since denotational semantics is used to study additional

aspects of programming languages. Such an approach does, however, result in more awkward and less natural descriptions of name management.

We believe that the flexibility and generality of PICCOLO allow it to capture important aspects of context management across a wide spectrum of computing domains in a uniform manner. Specification of different approaches to contextual requirements can easily be described using an appropriate formulation of CFTRs. Similarly, CFTPs can be used to describe those systems that support notions of provision control. CFPs can be realized to model almost any approach to context formation.

Another important benefit of PICCOLO is that it suggests potential tools for aiding the development and maintenance of large systems. In many approaches to software development, important contextual information is discarded once a program is built. For example, consider the following C++ compilation command:

```
CC -I/usr/local/include -I/usr/include program.C
```

The -I switches instruct the C++ compiler to build a context for header files ("*.h* files" in C++ terminology) using the *union-override* of the two directories. Once the executable corresponding to the source code is built, this valuable information is essentially thrown away. Maintenance programmers must either browse through scripts or reverse-engineer the executable in order to determine the context that was used in constructing the program. Capturing this information and allowing programmers to access this information in a uniform manner could greatly reduce the costs of developing and maintaining large and complex systems. Chapter 6 discusses one PICCOLO-based approach to providing information of this kind to users and programmers.

Although we believe that this chapter demonstrates that the PICCOLO framework contributes toward providing an improved understanding of name management, in particular, context management, there are several aspects of the framework that

deserve worth further investigation. We outline some possible future enhancements to PICCOLO in Chapter 8.

CHAPTER 5

TOWARD A FORMALIZATION OF PICCOLO MODELS

The basic concepts and extended features of the PICCOLO framework provide a useful basis for semi-formal descriptions and informal reasoning about various name management-related issues and problems. Formalizing the semantics of the PICCOLO framework's concepts makes them suitable as a basis for formal reasoning, and automated tools supporting such reasoning, about name management aspects of convergent systems. In this chapter, we explore some of the benefits accruing from a particular formalization of PICCOLO based on one particular approach to formal semantics, namely *evolving algebras*. This formalization is applied to a specific name management mechanism, leading to a more rigorous and carefully defined analysis than the relatively less formal ones illustrated in previous chapters.

Section 5.1 provides a brief introduction to evolving algebras. Section 5.2 then considers some aspects of an evolving algebra (EA) formalization of PICCOLO, including analysis opportunities arising from such a formalization. In particular, we draw on the Make/UNIX example of the preceding chapter to demonstrate this formalization. Section 5.3 evaluates the model and Section 5.4 concludes with a summary and assessment of these efforts.

5.1 Evolving Algebras

Evolving algebras were introduced by Gurevich as a framework for defining operational semantics [50]. The framework itself has evolved, its current definition being captured in [51]. Essentially, an evolving algebra models a system as a sequence of

states (or algebras), starting at some designated initial state and changing (or evolving) to subsequent states in a sequence of steps. The behavior of a system can be observed by examining a particular state or collection of states.

Evolving algebras have been used in defining formal semantics for several programming languages, including C++ [143], Modula-2 [52] and Prolog [18]; for other aspects of computational systems, such as data models for object-oriented databases [49]; and for particular algorithms, such as the Kermit communication protocol [59] and Lamport's bakery algorithm [17]. The formalism has also been used as a basis for proving properties of systems (e.g., [17, 59]) and has been implemented in the form of an interpreter [60] that can be used for investigating properties of EA descriptions of systems. Among the advantages of the EA approach is the ability to formulate semantic definitions at multiple levels of abstraction, so that different facets of a system can be described at their respective "natural abstraction levels" [51]. We have found this feature very useful in formalizing the semantics of PICCOLO concepts, as will be apparent in the examples of Section 5.2.

The EA formalism is based on multi-sorted first order logical structures with functions. The sorts are represented as *universes* of distinct atomic elements (or nullary functions). One important universe, for example, is *Boolean*, with elements *true* and *false*. A distinguished element *undef* is considered to belong to every universe, such that a function f is interpreted as being undefined at a tuple $\bar{a}$ if $f(\bar{a}) = undef$.

An operational semantics is defined by an abstract machine. An evolving algebra A is an abstract machine whose states consist of collections of elements from the universes included in A and functions defined on those universes. State transitions in A result from the addition or deletion of elements or modifications to the definitions of one or more functions, or both. The possible transitions of A are determined by the initial state of A and a finite collection of *transition rules*, defined recursively as follows:

- *Update rules* have the form:

  $$f(a_1, \ldots, a_n) := a_{n+1}$$

  where each a_i is a closed term (i.e., contains no free variables) and evaluates to an element of the appropriate universe. The meaning of the rule is that the value of function f on tuple $(a_1, \ldots, a_n)$ is changed to be a_{n+1} in the next state of the abstract machine.

- *Guarded rules* have the form:

  if a_0 **then** R_0
  elseif a_1 **then** R_1
  $\vdots$
  elseif a_n **then** R_n
  endif

  where each a_i is a closed term evaluating to an element of *Boolean* and each R_i is a transition rule. The meaning is that the rule R_i for the smallest i such that term a_i evaluates to *true* is executed. If none of the a_i evaluate to *true*, then no rules are fired.

- A *Rule block* is a collection of transition rules:

  $$R_1, \ldots, R_n$$

  $n \geq 1$, where all the rules in the block are interpreted as being executed simultaneously. Conflicts among the rules (i.e., updating the same function at the same step with different values) are not permitted.

- *Extension rules* have the form:

  extend U **by** $\epsilon_1, \ldots, \epsilon_n$ **with**
  $R_1, \ldots, R_m$
  endextend

  where U is a universe, $\epsilon_1, \ldots, \epsilon_n$ represent newly created elements of U, and R_i are transition rules, which (typically) use the newly created elements of the universe as values in their corresponding terms.

Given an initial state and a set of transition rules, a *run* of an evolving algebra, A, consists of a sequence of states, S_i, resulting from successive application of all transition rules defined in A. *Static functions* are functions whose domains never change during a run, while *dynamic functions* may be updated. In addition, *external functions* are provided as means of interacting with the outside environment (e.g., I/O), where their values are determined by some oracle.

This brief introduction to evolving algebras, while sufficient for our purposes in this dissertation, glosses over a number of technical points and omits several interesting aspects of the formalism. The interested reader is referred to [51] for a complete treatment.

5.2 An EA/Piccolo Formulation of Name Management in Make/UNIX

To help illustrate how we can apply evolving algebras to a PICCOLO-based model of name management[1] we return to the Make/UNIX example described in Chapter 4 (summarized in Figure 5.1).

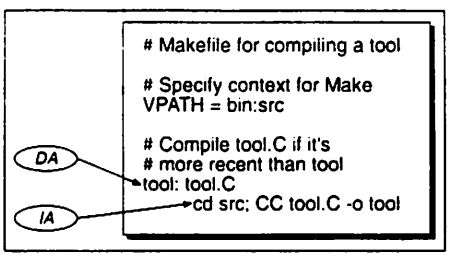


Figure 5.1 Example Makefile

Informally, the Make/UNIX algorithm itself is relatively straightforward. Make analyzes the target-dependency rules in a given Makefile. If the timestamp of a target is older than any of its dependencies (appearing on some line), then the actions

[1] We frequently use the term "EA/PICCOLO" to describe this technique.

(appearing on a different line or lines) associated with the rule are invoked. Thus, if the file named tool appearing on the line labeled "DA" in Figure 5.1 is older than the file named tool.C appearing on the same line, then the C++ compilation appearing on the line labeled "IA" is invoked.

In this section, we give a definition of an evolving algebra, $A_{Make/UNIX}$, based on the PICCOLO framework that gives a formal description of name management in the combination (or convergence) of Make and UNIX. Utilizing the EA formalism's support for multi-level descriptions, we first describe $A_{Make/UNIX}$ in terms of some high level abstractions and then refine these abstractions in order to provide more precise semantics for context formation and context consistency in Make and UNIX.

5.2.1 A High Level View of Name Management in Make/UNIX

To explain name management in Make/UNIX, we first define the following universes (in *italics*). These universes correspond directly to concepts defined in PICCOLO.

- *objects* that participate in the Make/UNIX name management mechanism. This universe can be further decomposed into the following (sub) universes:
  - *files* $\subset$ *objects* that are manipulated in a Makefile;
  - *lines* $\subset$ *objects* that are particular constructs in a Makefile (i.e., a line containing a target-dependency rule or an action);
  - *bindingspaces* $\subset$ *objects* that correspond to valid UNIX directories;
- *names* that correspond to valid Make/UNIX names;
- *contexts* that correspond to valid contexts;
- *epochs* that correspond to various epochs.

Next, we define several dynamic functions that determine the name management-related state of a Makefile:

CurrentLine → *lines*
CurrentEpoch → *epochs*
LookUp (*bindingspaces*, *names*) → *objects*
Resolve (*contexts*, *names*) → *objects*
TheObject (*lines*, *names*) → *objects*
ContextDefinedBy (*lines*, *epochs*) → *contexts*
Step → *Integer*
Cwd → *bindingspaces*
Src → *bindingspaces*
Bin → *bindingspaces*

CurrentLine, an external function, returns the currently active line (in the Makefile), while *CurrentEpoch* returns the current epoch for a name management operation. Since bindings are not first class, visible entities in Make/UNIX, they are modeled using *LookUp*, which defines a name for an object in a binding space, based on ***redefine*** semantics[2]. *Resolve*, an external function, returns the object bound to a name in a given context. If the name is not found, *undef* is returned. *TheObject* returns the actual object associated with the given name and line. As we will see in Section 5.3, this function facilitates the analysis of contextual consistency in Make/UNIX. *ContextDefinedBy*, an external function, returns the context associated with a Makefile construct at a particular epoch. In this initial, high level view of name management in Make/UNIX, *Step* acts as a synchronizer, ensuring that operations are performed in the proper sequence. (In Section 5.2.2, *Step* is replaced with a finer-grained sequencing function.) Finally, the (external) functions *Cwd*, *Bin* and *Src* store the values for the current working directory, the bin directory and the src directory, respectively.

Given these universes and functions, the basic operation of the name management mechanism in Make/UNIX is described by the following EA transition rule.

[2]In Chapter 6, we revisit the issue of the explicit provision of bindings in a name management mechanism.

```
if (Step = 1)
  /* first compute the context for the line */
  CurrentContext := ContextDefinedBy (CurrentLine, CurrentEpoch)
  Step := 2
else if (Step = 2)
  /* second get the objects */
  TheObject (CurrentLine, η_tool) := Resolve (CurrentContext, η_tool)
  TheObject (CurrentLine, η_tool.c) := Resolve (CurrentContext, η_tool.c)
  Step := 3
else
  /* perform the computation (rule, action, ...) */
  ...
end if
```

This transition rule represents, at a very high level, the various steps that must occur when a particular Makefile construct is processed. First, a context is selected for the current Makefile construct and epoch. Using this context, the names of the objects (in this example, the files) are then resolved. Note that one of the benefits of the EA formalism is that because of rule block concurrency it does not force us to impose an artificial sequencing for these resolutions. Assuming the resolution is successful, the appropriate computation on the objects is performed. From our perspective on name management, however, we can ignore the consequences of such computations, unless, of course, they have some effect on any facet of name management in Make/UNIX.

Through application of EA/PICCOLO, we can formally describe the fundamental concepts of the approach to name management found in Make/UNIX at a fairly high level of abstraction. Although discerning potential name management-related problems requires some refinement of this abstraction (as we show below), it should be evident that EA/PICCOLO uniformly captures various features of name management in Make/UNIX, even at this high level of abstraction, in a more precise manner than the informal use of PICCOLO for the same purpose presented in Chapter 4.

5.2.2 A First Refinement

In this section, we refine $A_{\text{Make/UNIX}}$, thereby moving to a lower level of abstraction, and provide more detailed EA/PICCOLO descriptions of context formation, name resolution and binding definition in $A_{\text{Make/UNIX}}$. First, we elaborate the definition for the universe representing epochs and the functions defined on this universe. Specifically, the elaborated universe *epochs* consist of seven different kinds of epochs, *epochs* $= \{\epsilon_{\text{da}_{\text{cf}}}, \epsilon_{\text{da}_{\text{nr}}}, \epsilon_{\text{ia}_{\text{db}}}, \epsilon_{\text{ia}_{\text{cf}}}, \epsilon_{\text{ia}_{\text{nr}}}, \epsilon_{\text{ccc}}, \epsilon_{\text{hc}}\}$, where $\epsilon_{\text{da}_{\text{cf}}}$ represents a *dependency analysis/context formation* epoch, $\epsilon_{\text{da}_{\text{nr}}}$ represents a *dependency analysis/name resolution* epoch, $\epsilon_{\text{ia}_{\text{db}}}$ represents an *invoke action/define binding* epoch, $\epsilon_{\text{ia}_{\text{cf}}}$ represents an *invoke action/context formation* epoch, $\epsilon_{\text{ia}_{\text{nr}}}$ represents an *invoke action/name resolution* epoch, ϵ_{ccc} represents a *context consistency check* epoch, and ϵ_{hc} represents a *halt condition* epoch. Note that the epochs ϵ_{ccc} and ϵ_{hc} are not actually part of Make's name management mechanism. We include them in our formulation, however, to indicate when appropriate analyses may be performed. (An example of such analyses is given in Section 5.3.) The static function *NextEpoch* (*epoch*) $\rightarrow$ *epoch* computes the successor to a given epoch. In particular, at this level of abstraction, the initial state, S_0, of $A_{\text{Make/UNIX}}$ contains the following definitions of *NextEpoch* and *CurrentEpoch*:

NextEpoch $(\epsilon_{\text{da}_{\text{cf}}}) := \epsilon_{\text{da}_{\text{nr}}}$	*NextEpoch* $(\epsilon_{\text{da}_{\text{nr}}}) := \epsilon_{\text{ccc}}$	*NextEpoch* $(\epsilon_{\text{ccc}}) := \epsilon_{\text{ia}_{\text{db}}}$
NextEpoch $(\epsilon_{\text{ia}_{\text{db}}}) := \epsilon_{\text{ia}_{\text{cf}}}$	*NextEpoch* $(\epsilon_{\text{ia}_{\text{cf}}}) := \epsilon_{\text{ia}_{\text{nr}}}$	*NextEpoch* $(\epsilon_{\text{ia}_{\text{nr}}}) := \epsilon_{\text{da}_{\text{cf}}}$
CurrentEpoch $:= \epsilon_{\text{da}_{\text{cf}}}$		

Next, we provide a formal definition of extended closure. Recall that the PICCOLO definition of extended closure "is an association between an object, an initial context, a set of context formation templates for directing the incremental formation of future contexts, and a set of context formation processes that will be directed by those CFTs during incremental formation of future contexts." In Make/UNIX, contexts (and directives for forming contexts) are directly correlated with the different constructs

in Make. Thus, to model extended closures,[3] we define the following universe and external function.

- *cfts* is a universe of context formation templates;
- *ContextDirective* (*lines*) $\rightarrow$ *cfts* returns the CFT for a particular line in a Makefile.

To complete this first refinement to our preliminary EA/PICCOLO description of name management in Make and UNIX, $A_{\text{Make/UNIX}}$ is augmented with a unary function, *ContextFor*(*lines*) $\rightarrow$ *contexts*, which returns a context needed for a particular line in a Makefile. The transition rule modeling the use and manipulation of names in the Makefile shown in Figure 5.1 is given below. Note that at this level of abstraction, the *Step* function is replaced with the *NextEpoch* function.

---

if $CurrentEpoch = \epsilon_{\text{da}_{\text{cf}}}$ **then**
  $ContextFor\ (CurrentLine) := ContextDefinedBy\ (\epsilon_{\text{da}_{\text{cf}}}, CurrentLine)$
  $CurrentEpoch := NextEpoch\ (CurrentEpoch)$
elseif $CurrentEpoch = \epsilon_{\text{da}_{\text{nr}}}$ **then**
  $TheObject\ (CurrentLine, \eta_{\text{tool}}) := Resolve\ (ContextFor\ (CurrentLine), \eta_{\text{tool}})$
  $TheObject\ (CurrentLine, \eta_{\text{tool.c}}) := Resolve\ (ContextFor\ (CurrentLine), \eta_{\text{tool.c}})$
  $CompleteResolution\ (\eta_{\text{tool}}, \eta_{\text{tool.c}})$
elseif $CurrentEpoch = \epsilon_{\text{ia}_{\text{db}}}$ **then**
  extend *files* **by** *newfile* **with**
    $LookUp(Src, \eta_{\text{tool}}) := newfile$
  endextend
  $CurrentEpoch := NextEpoch(CurrentEpoch)$
elseif $CurrentEpoch = \epsilon_{\text{ia}_{\text{cf}}}$ **then**
  $ContextFor(CurrentLine) := ContextDefinedBy(\epsilon_{\text{ia}_{\text{cf}}}, CurrentLine)$
  $CurrentEpoch := NextEpoch(CurrentEpoch)$
elseif $CurrentEpoch = \epsilon_{\text{ia}_{\text{nr}}}$ **then**
  $TheObject(CurrentLine, \eta_{\text{tool}}) := Resolve(ContextFor(CurrentLine), \eta_{\text{tool}})$
  $TheObject(CurrentLine, \eta_{\text{tool.c}}) := Resolve(ContextFor(CurrentLine), \eta_{\text{tool.c}})$
  $CompleteResolution(\eta_{\text{tool}}, \eta_{\text{tool.c}})$
endif

---

[3]Since Make/UNIX relies primarily on a single context formation process, namely *union-override*, we choose not to explicitly include a CFP in the extended closures for Make/UNIX.

This transition rule defines for each epoch (except for ϵ_{ccc} and ϵ_{hc}, which are discussed in the next section) how the state of $A_{\text{Make/UNIX}}$ is changed to reflect the effects of the various name management operations. For example, when the value of *CurrentEpoch* equals $\epsilon_{da_{cf}}$, a new context is formed as dictated by the function *ContextDefinedBy* (along with the values of *CurrentLine* and *CurrentEpoch*). Similarly, when the value of *CurrentEpoch* equals $\epsilon_{da_{nr}}$, the names η_{tool} and $\eta_{\text{tool.c}}$ are resolved using the context constructed in the previous epoch. At the end of each epoch, *CurrentEpoch* is updated via an invocation of the *NextEpoch* function. (The macro *CompleteResolution* includes an invocation of *NextEpoch*, as described in Section 5.2.3.) Also note the creation, more specifically, the redefinition, of a new binding during the $\epsilon_{ia_{dl}}$ epoch. This corresponds to the "-o tool" switch given to the C++ compiler. Thus, in this refined EA/PICCOLO formulation of Make/UNIX name management, we can see precisely how and when contexts are formed, names are resolved and bindings are defined.

5.2.3 A Second Refinement

To obtain a more detailed understanding of name management in Make/UNIX, the next step in applying EA/PICCOLO to our example is to further refine the various abstractions in $A_{\text{Make/UNIX}}$. In particular, the functions *ContextDirective*, *ContextDefinedBy* and *Resolve* were earlier defined as external functions. Recall that this means their values are determined by an entity or oracle outside the given abstract machine. In this section, we replace these external functions with definitions of internal functions and, as a result, provide more precise semantics for context manipulation in Make/UNIX.

First, we choose to represent CFTRs[4] for Make/UNIX as pairs of epochs and lists of binding spaces (*epochs, list[bindingspaces]*[5]). We then define a static function *ContextDirective* (*lines*) → (*epochs, list[bindingspaces]*) initialized by the execution of:

ContextDirective (*TD*) := ($\epsilon_{\mathrm{da}_{\mathrm{cf}}}$, *Append* (*Bin, Append* (*Src, nil*)))
ContextDirective (*IA*) := ($\epsilon_{\mathrm{ia}_{\mathrm{cf}}}$, *Append* (*Src, nil*))

The first update rule corresponds to a combination of the VPATH specification and the line labeled "TD" in the Makefile in Figure 5.1. The second update rule represents the context directives for the invocation of the C++ compiler in the line labeled "IA" in the Makefile. (*TD* and *IA* are simply external functions indicating the constructs corresponding to these lines, respectively, in the Makefile in Figure 5.1.) We also provide projection functions, *GetEpoch* (*cfts*) → *epochs* and *GetSources* (*cfts*) → *list[bindingspaces]*, that select an epoch and list of source binding spaces in *cfts*, respectively.

Next, we define *ContextDefinedBy*. In Make and UNIX, contexts are evaluated lazily – when a name is resolved in a context, each binding space in the list returned by the *ContextDirective* is searched until either an object with the required name is found or there are no binding spaces left to search. To model this lazy evaluation approach to context formation, the context returned by *ContextDefinedBy* is simply the list of binding spaces. The definition of *ContextDefinedBy* is given in terms of a parameterized macro.[6]

---

[4]The simplicity of UNIX access control makes CFTPs trivial in Make/UNIX, so for simplicity we omit modeling them in $A_{\mathrm{Make/UNIX}}$.

[5]A *list* is represented as a universe in $A_{\mathrm{Make/UNIX}}$. In addition, standard list operations, such as *Append, First, Rest, nil* (with their traditional meanings) are defined on elements in *list*.

[6]EA macros are simply a syntactic shorthand and, therefore, do not affect the formalism's computational expressiveness.

```
Macro theContext := ContextDefinedBy (theEpoch, theLine) ≡
if (theEpoch = GetEpoch (ContextDirective (theLine)) then
  theContext := GetSources (ContextDirective (theLine))
```

Next, we define *Resolve*, again making use of a parameterized macro. The parameters to the *Resolve* macro are a context, a name and the resolved object:

```
Macro theObject := Resolve(context, name) ≡
if ResolveStep(name) = SetupResolve then
  Sources(name) := context
  CurrentSource(name) := First (context)
  ResolveStep(name) := AttemptResolve
elseif ResolveStep(name) = AttemptResolve then
  if Sources(name) = nil then
    theObject := undef
    ResolveStep(name) := CompleteResolve
  elseif Sources(name) ≠ nil then
    if LookUp(CurrentSource(name), name) = undef then
      CurrentSource(name) := First(Rest(Sources(name)))
      Sources(name) := Rest(Sources (name))
    elseif LookUp(CurrentSource(name),name) ≠ undef then
      theObject := LookUp (CurrentSource(name), name)
      ResolveStep(name) := CompleteResolve
    endif
  endif
endif
```

As noted above, contexts are not actually formed until a name resolution is performed. The *Resolve* macro expands into a transition rule that describes this approach to context formation. To resolve a name (for an object), the body of the rule searches each binding space, as specified by the context, until either an object is found or there are no more binding spaces left to search. Some auxiliary functions are defined to facilitate this task. Specifically, *Sources* (*names*) → *contexts* stores the list of source binding spaces and *CurrentSource* (*names*) → *bindingspaces* stores the value of a current binding space to look for a definition of a given name. For a given name, the transition rule proceeds as follows. First, it sets up up the resolution for the name.

Next, it attempts to locate a definition for the name in the first binding space in the list of source binding spaces. If the name is defined, *Resolve* is updated with the definition for the name and the transition rule terminates. If the name is not defined, then the next binding space (in the list of sources) is searched. If there are no more binding spaces to search, and no definition for the name has been located, then *Resolve* is updated with the value *undef*, indicating the name can not be resolved in the given context.

Finally, the *CompleteResolution* macro is parameterized by the names being resolved:

Macro *CompleteResolution*($name_1$, $name_2$) $\equiv$
if *ResolveStep*($name_1$) = *CompleteResolve* $\wedge$
  ResolveStep($name_2$) = *CompleteResolve* **then**
 ResolveStep ($name_1$) := *SetupResolve*
 ResolveStep($name_2$) := *SetupResolve*
 CurrentEpoch := *NextEpoch*(*CurrentEpoch*)
endif

It expands into a transition rule that synchronizes completion of the activities performed by the transition rules produced by the *Resolve* macros (which are executed simultaneously), then reinitializes the *ResolveStep* functions, and updates the *CurrentEpoch* function appropriately by invoking *NextEpoch*.

We assert that the collection of EA universes, functions and transition rules presented here, whose formation was guided by the PICCOLO framework, provides a formal and detailed model of name management in Make/UNIX. In the next section, we attempt to substantiate this claim, using the EA/PICCOLO model, $A_{Make/UNIX}$, to investigate properties of name management in Make/UNIX.

5.3 Evaluating and Using the Formalization

We now consider the evaluation and utilization of an EA/PICCOLO model such as the formalization of Make/UNIX presented in the preceding section. Our approach to both evaluation and utilization involves the use of the Evolving Algebra Interpreter [60]. The Evolving Algebra Interpreter permits EA descriptions, such as the ones illustrated throughout Section 5.2, to be translated into a representation that can be interpreted by a computer program. Executing the interpreter permits a user to iterate over, as well as examine and query, the states of an EA model. Thus, a user can actually run a particular EA model and observe its behavior (e.g., the individual states of the algebra). Potential faults can also be injected into the model at arbitrary stages in a run, allowing a user to concretely observe how a certain model will react under various conditions. Based on these observations, a specific model can modified or enhanced (as needed), and re-run and re-observed in a systematic manner.

Our approach to evaluating the EA/PICCOLO model $A_{\text{Make/UNIX}}$, described in Section 5.3.1, involves checking the model itself, i.e., testing whether $A_{\text{Make/UNIX}}$ is an accurate model of Make/UNIX. Utilizing the model, described in Section 5.3.2, involves using $A_{\text{Make/UNIX}}$ to analyze some specific properties and help detect some specific problems related to name management in Make/UNIX.

5.3.1 Checking The Model

Our basic assertion is that the EA model, presented in Section 5.2, is an accurate representation of the actual operation of our Make/UNIX example. To check the validity of this assertion, we conducted a series of tests on the Makefile and its (proposed) model. In these tests, the Makefile remained static and the binding space organization in which it operated varied. Specifically, we performed the following steps:

Instrument the Makefile We instrumented the Makefile so that it printed the names and identities of the files (in UNIX terms, their i-nodes) that were used during each of the dependency analysis and invoke action epochs. For example, the line labeled "TD" in the Makefile in Figure 5.1 contains the name of a target, tool, and a dependency, tool.C. When their temporal relationship is examined, the name and i-node information is printed for each file. Note that obtaining the analogous information for $A_{Make/UNIX}$ is facilitated by the Evolving Algebra Interpreter, which provides the ability to observe and query a particular state and hence allows us to easily determine the names and identifiers of the objects used during any epoch.

Generate Test Cases We generated test cases representing instantiations of different alternative binding space organizations for the Makefile in Figure 5.1. Figure 5.2 shows the name space graphs of several test cases. Generating these test cases required the following:

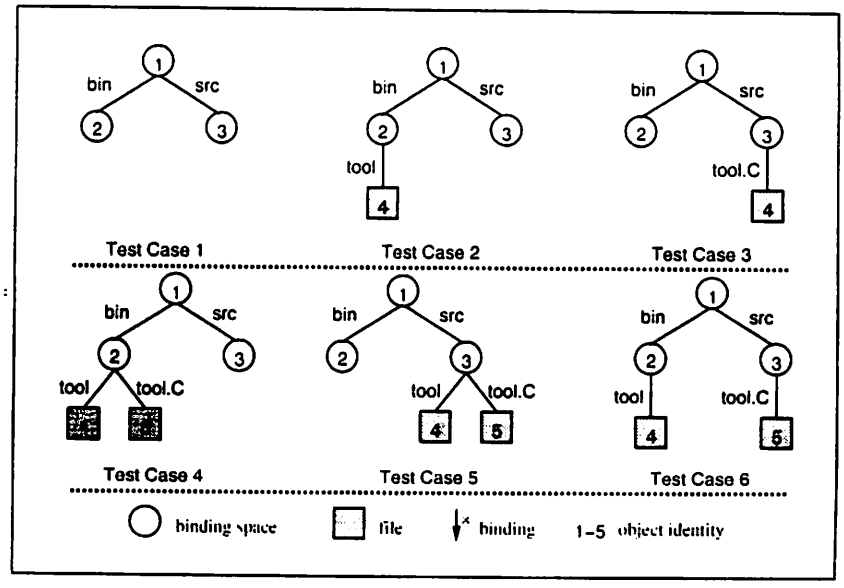


Figure 5.2 Sample Test Cases for Model

- Constructing an actual UNIX subdirectory structure representing a sample name space graph.
- Creating the EA description corresponding to this subdirectory structure. Note that the C++ compiler creates a new binding (i.e., by redefining a binding in the **src** directory). This event is already described in $\mathcal{A}_{\text{Make/UNIX}}$, specifically by the transition rule that executues when the condition $CurrentEpoch = \epsilon_{\text{ia}_{\text{db}}}$ holds.
- Creating a table mapping i-node numbers of UNIX files in a sample subdirectory structure to object identifiers in a corresponding EA description of the sample. Again, the C++ compilation produces a new (executable) file and the EA description represents this change. This information is added to the table when the actual file is created and assigned a name.

Execute Makefile and $\mathcal{A}_{\text{Make/UNIX}}$ in Parallel To test how well $\mathcal{A}_{\text{Make/UNIX}}$ models Make/UNIX, we invoked the Makefile and $\mathcal{A}_{\text{Make/UNIX}}$ in lockstep (i.e., by executing Make on the Makefile and applying the Evolving Algebra Interpreter to $\mathcal{A}_{\text{Make/UNIX}}$). We recorded the names and object identifiers used during each of the dependency-analysis and invoke-action stages in Make and in $\mathcal{A}_{\text{Make/UNIX}}$. We then checked whether the i-nodes and identifiers used in a test run of Make and $\mathcal{A}_{\text{Make/UNIX}}$ were consistent with the mapping table.

To illustrate this methodology, consider the UNIX subdirectory structure, EA transition rule and mapping table shown in Figure 5.3. As prescribed above, we now invoke Make and $\mathcal{A}_{\text{Make/UNIX}}$ in parallel, recording the file i-nodes used in Make and the object identifiers used in $\mathcal{A}_{\text{Make/UNIX}}$. Table 5.1 shows the values used by Make and $\mathcal{A}_{\text{Make/UNIX}}$ corresponding to the test case in Figure 5.3. As indicated in the column labeled "Mapping" in Table 5.1, the mapping table from Figure 5.3 confirms

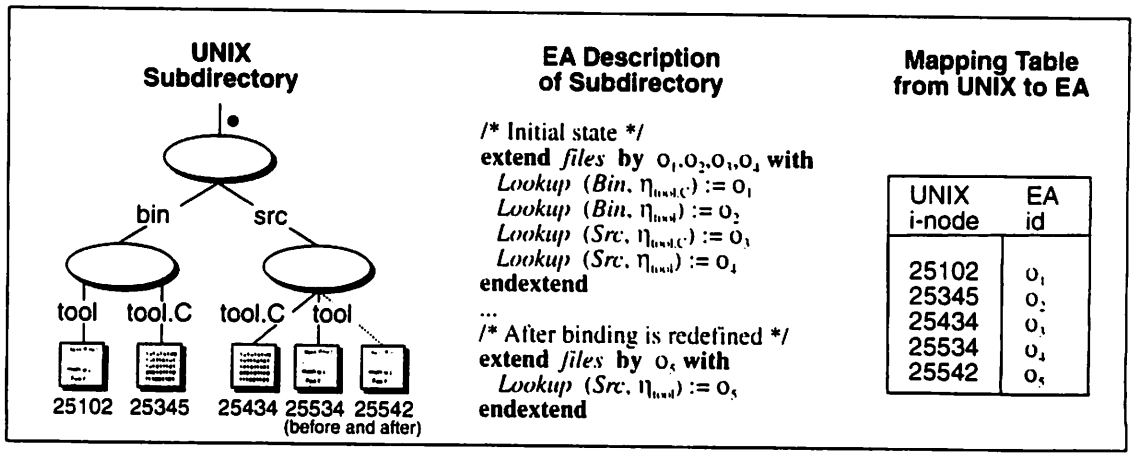


Figure 5.3 UNIX Subdirectory, EA Description and Mapping Table

Table 5.1 Comparing Make/UNIX and $A_{\text{Make/UNIX}}$

Epoch	Name	UNIX I-Node	EA Object	Mapping
Dependency Analysis	tool	25102	o_1	Yes
Dependency Analysis	tool.C	25345	o_2	Yes
Invoke Action	tool	25542	o_5	Yes
Invoke Action	tool.C	25434	o_3	Yes

that our EA/PICCOLO formulation of Make/UNIX is valid for this particular set of data points.

Such testing not only helps confirm the "accuracy" of the model, but also helps reveal interesting subtleties and anomalies in the name management mechanism being modeled. Carrying out the testing methodology described above, for example, led to an interesting discovery regarding Make's name management mechanism. Specifically, we generated a test case as illustrated by the name space graph in Figure 5.4. Based

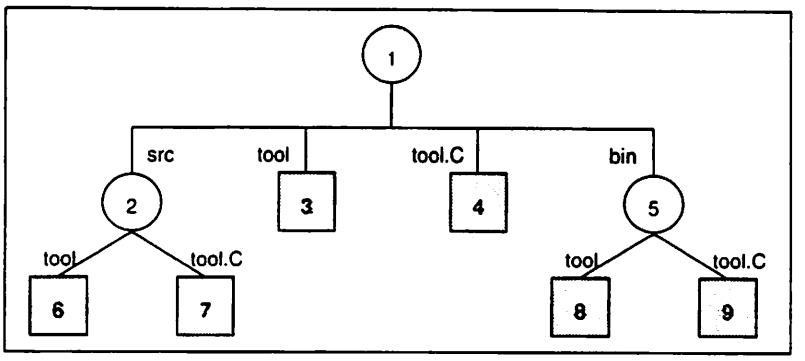


Figure 5.4 Test Case Revealing Anomaly in Model and Mechanism

on the VPATH specification

VPATH=bin:src

and its extended closure description, we expected Make and our model to locate these files during dependency analysis in the bin directory. Surprisingly, Make located these files in the current working direcory, while our EA/PICCOLO model located them in the bin directory. Upon investigating the cause for this discrepancy, we discovered that the current working directory is implicitly included as the first directory in all VPATH specifications. As described informally in the Gnu Make manual:

> The value of the 'make' variable 'VPATH' specifies a list of directories that 'make' should search. Most often, the directories are expected to contain dependency files that are not in the current directory; however, 'VPATH' specifies a search list that 'make' applies for all files, including files which are targets of rules. Thus, if a file that is listed as a target or dependency does not exist in

the current directory. 'make' searches the directories listed in 'VPATH' for a file with that name. If a file is found in one of them, that file becomes the dependency. Rules may then specify the names of source files in the dependencies as if they all existed in the current directory.

Based on this somewhat vague and ambiguous description of the VPATH capability we should have, in retrospect, included the current working directory in the CFT for the target dependency rule. In our initial EA formulation of Make we created a CFT for the dependency analysis that only used the bin and src binding spaces. Updating our model to reflect the actual behavior of the Make VPATH required the following, relatively straightforward, modification to $A_{\text{Make/UNIX}}$. Specifically, we changed the function update for *ContextDirective* from

---

$$\mathit{ContextDirective}\ (TD) := (\epsilon_{\mathrm{da}_{\mathrm{cf}}},\ \mathit{Append}\ (\mathit{Bin},\ \mathit{Append}\ (\mathit{Src},\ \mathit{nil})))$$

---

to

$$\mathit{ContextDirective}\ (TD) := (\epsilon_{\mathrm{da}_{\mathrm{cf}}},\ \mathit{Append}\ (\mathit{Cwd},\ \mathit{Append}\ (\mathit{Bin},\ \mathit{Append}\ (\mathit{Src},\ \mathit{nil}))))$$

After making this slight modification, we repeated our tests according to the methodology given above, and no further discrepancies were found. Although such techniques can only test for the presence of faults and not their absence,[7] based on this technique we concluded that $A_{\text{Make/UNIX}}$ is a valid model of our Make/UNIX example.

5.3.2 Checking Name Management Properties

Recall that the *epoch* universe in $A_{\text{Make/UNIX}}$ included two special-purpose epochs that do not have direct correlates in the Make/UNIX name management mechanism. These epochs were added in order to facilitate formal reasoning about our

[7]This observation is generally attributed to Edsger W. Dijkstra.

EA/PICCOLO model of name management in Make/UNIX, thus representing the other facet of our evaluation efforts. In particular, they are used to enable automated analyses of the model using the Evolving Algebra Interpreter. The epoch ϵ_{ccc} is used to indicate when a particular analysis should be performed, and the epoch ϵ_{hc} is used to interrupt execution of $A_{Make/UNIX}$ by the EA Interpreter when an analysis reports an inconsistency. For example, suppose we wish to test for the following property in our Make/UNIX example:

> *Assuming no changes to the source files, the set of files used during Make's dependency analysis should be consistent with the set used during the corresponding compilation in the Makefile.*

More specifically, in our example Makefile we want to determine whether the files bound to the names tool and tool.C during Make's dependency analysis are identical to the files bound to the same names during compilation. Since the compilation creates a new binding, we want to check this property after the dependency analysis, but before the subsequent compilation is performed.

Using EA/PICCOLO, we can formally represent this assertion by augmenting $A_{Make/UNIX}$ with a transition rule that will be executed when the current epoch is ϵ_{ccc}:

---

if *CurrentEpoch* $= \epsilon_{ccc}$ **then**
  if *ContextFor* (IA) = *undef* **then**
    CurrentEpoch := *NextEpoch* (*CurrentEpoch*)
  elseif *TheObject* $(TD, \eta_{\mathsf{tool}})$ = *TheObject*$(IA, \eta_{\mathsf{tool}}) \wedge$
    TheObject $(TD, \eta_{\mathsf{tool.C}})$ = *TheObject* $(IA, \eta_{\mathsf{tool.C}})$ **then**
    CurrentEpoch := *NextEpoch* (*CurrentEpoch*)
  else
    CurrentEpoch := ϵ_{hc}
  endif
endif

The rule first checks that a context for the invoke action epoch has indeed been defined. If it is defined, then the consistency of the two contexts can be computed

by comparing the results of the most recent resolutions of each name (which were preserved as values of the *TheObject* function) and determining whether or not those resolutions returned identical objects. If they did, then the contexts are consistent; otherwise, the epoch ϵ_{hc} is triggered and the EA Interpreter execution of $A_{Make/UNIX}$ is interrupted.

To demonstrate the utility of this analysis capability, consider Figure 5.5, which shows a UNIX subdirectory structure, its corresponding EA/PICCOLO description and a mapping table. When applied to $A_{Make/UNIX}$, the interpreter takes the following

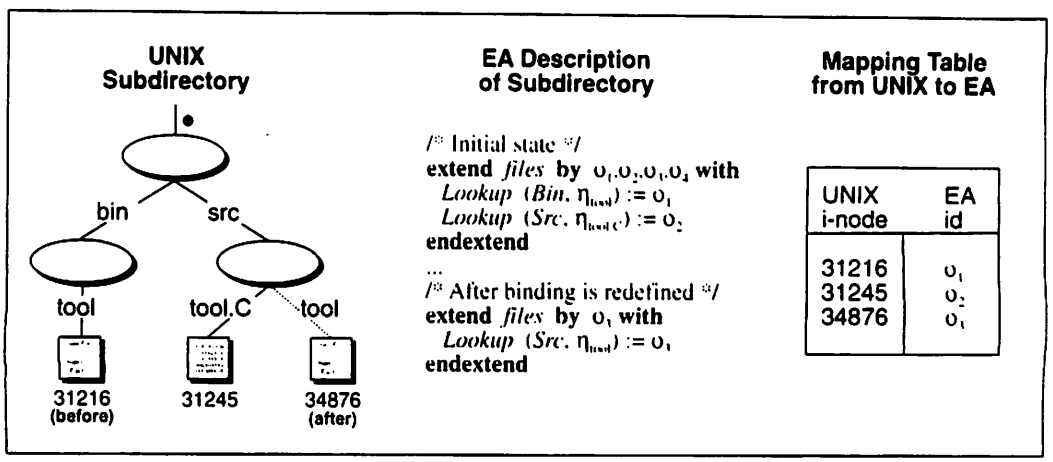


Figure 5.5 Checking Contextual Consistency

course of action:

1. Forms a context and resolves the names for the dependency analysis epoch of Make.

2. Performs the context consistency check. Since a context has not been formed yet for the invoke action epoch, the interpreter continues processing.

3. Creates a new binding for the file object named tool by updating the *Lookup* function appropriately.

4. Forms a context and resolves the names for the invoke action epoch of Make (i.e., the C++ compilation).

5. (Again) Forms a context and resolves the names for the dependency analysis epoch of Make.

6. Performs the context consistency check. This time a context has been formed for the invoke action epoch (from step 4 above). Furthermore, the interpreter detects a contextual inconsistency since the name tool is bound to two different objects in the two different contexts. More specifically, the C++ tool executable produced by the C++ compiler is not used by Make in checking the target dependency relationship. Instead, the tool.C file in the bin directory is always used during dependency analysis, even though the C++ compiler produces a new file named tool.C in the src directory. This it probably not what the developer intended, since it can lead to unnecessary compilations, which in a large complex system can be counterproductive and expensive. On detecting this problem, the halt condition epoch is fired and the EA interpretation terminates.

One limitation of this approach is that does not inform a user as to the actual cause of the inconsistency: it only reports the existence of a potential problem. The approach does, however, help pinpoint where the problem might be located. In this case, the name tool is bound to two different objects in the contexts employed by the two different Makefile constructs. The user can examine the model to see how the contexts are formed in each of these constructs, focusing on the ambiguous name. For the target dependency epoch, a context is formed using the union override of the current working directory, the bin directory and the src directory, while the context for the invoke action epoch is formed solely from the src directory. Furthermore, both the bin and src directories contain bindings, perhaps accidentally or intentionally, of the same name, tool, to different objects. Thus, it should be evident that Make is

using a binding for tool as defined in the bin directory, while the C++ tool is using a binding for tool as defined in the src directory.

Once alerted to this situation, a user could rectify the problem in one of several ways. For example, the tool file could be moved from the bin directory to the src directory. (In this case, the specification of the bin directory in the VPATH becomes superfluous.) Making such a change to the UNIX subdirectory structure requires a corresponding modification to the original extension rule:

---

extend *files* **by** o_2 **with**
 ...
 $LookUp(Src, \eta_{\mathsf{tool}}) := o_2$
 ...
endextend

As an alternative, the user could modify the Makefile to instruct the C++ compiler to define the new binding to tool in the bin directory instead of the src directory, i.e.,

```
cd src; CC tool.C -o ../bin/tool
```

Choosing this solution requires an associated change to $A_{\mathsf{Make/UNIX}}$ where the binding for tool is redefined:

---

elseif $CurrentEpoch = \epsilon_{\mathsf{ia_{db}}}$ **then**
 extend *files* **by** *newfile* **with**
  $LookUp(Bin, \eta_{\mathsf{tool}}) := newfile$
 endextend
 $CurrentEpoch := NextEpoch(CurrentEpoch)$

For either of these changes, the EA Interpreter reports no context inconsistencies since the names have the same meanings in the different epochs. A complete EA description of $A_{\mathsf{Make/UNIX}}$, which can be run by the Evolving Algebra Interpreter is given in the Appendix.

5.4 Summary

In this chapter, we have investigated the suitability of the evolving algebra formalism as a basis for formalizing the semantics of Piccolo-based models of name management. In particular, we have shown how the evolving algebras provide a potentially useful foundation for a formalization of the PICCOLO framework, resulting in the ability to formally describe and reason about name management in a representative example of a convergent computing system. More generally, we have demonstrated that such formulations can indeed serve as a basis for formal analysis, which can aid users and developers in the design, implementation and maintenance of name management mechanisms.

Although given in terms of a specific example, it is our contention that in general this approach offers a number of important and useful benefits. First, a user or designer attempting to understand the specifics of an existing or a proposed name management mechanism can utilize this technique to capture the semantics of an approach, and hence avoid the ambiguities inherent in informal, potentially ambiguous, descriptions. Second, the operational nature of evolving algebras makes the technique much more intuitive and accessible to practitioners than more abstract modeling approaches. In comparison, employing other modeling techniques, such as denotational approaches or algebraic specifications, often requires using baroque syntax and complex semantics. Thus, we believe the combination of PICCOLO and the evolving algebras formalism can lead to a more concrete understanding of an approach. Third, the fact that EA descriptions can be translated into a form that can be executed by a computer program is extremely useful. As we have demonstrated, it facilitates testing how well an abstract model of name management corresponds to a real system. It also allows users and designers the capability to experiment with a name management mechanism. Finally, it also suggests possible automated name management-related analysis capabilities.

There are clearly some weaknesses evident in the approach outlined in this chapter. As noted, we have only shown a formalization of a particular example for a particular mechanism. This activity, as outlined in this chapter, essentially relies on a manual translation from Make/UNIX into a specific EA/PICCOLO model. A much more general formalization of Make/UNIX, for example, would require an automatic translation from a Makefile into a general EA/PICCOLO model of name management for Make/UNIX. Given such a translator, we would be able test arbitrary Makefiles for contextual consistencies, instead of individual instances. Another shortcoming with the approach is that it shares all the weakness inherent in other testing-related techniques. Thus, one obvious extension to the approach includes the development of formal proof techniques for consistency properties in name management mechanisms. Such proof techniques can potentially lead to more meaningful information for describing the causes of an inconsistency.

Going further, one might seek a generally applicable formalization of PICCOLO semantics. Our conclusion, based on the investigation of formalization relative to a specific example reported in this chapter, however, is that this would be an ambitious undertaking, appropriately deferred to future work. This and other possible future directions concerning formalization of semantics for PICCOLO are discussed in Chapter 8.

CHAPTER 6

NAME MANAGEMENT MECHANISMS

Chapters 3–5 focused on the descriptive capabilities of the PICCOLO framework. In this chapter, we continue to demonstrate the framework's value, this time describing its role as a prescriptive tool to help guide the development of, gain experience with, and conduct experiments on, enhanced name management mechanisms in convergent computing systems. This naturally contributes to the overall evaluation of our models and concomitantly demonstrates the viability and utility of improved name management capabilities in convergent computing systems. We focus our attention on the domain of persistent object systems (POSs) which, as we discuss below, have historically provided weak support for name management. Improvements in name management for POSs should not only lead to richer and more robust approaches, but should also contribute to the acceptance and proliferation of this important technology.

Persistent object systems are a particularly interesting class of convergent computing systems. POSs promise to break down barriers between capabilities traditionally found in programming languages, database systems and operating systems and to combine all (or at least most) of the best features of each. In recent years, POS research has led to significant progress at overcoming barriers and synergistically combining capabilities related to various aspects of such systems, including type models (e.g., [33]), persistence mechanisms (e.g., [148]), optimization techniques (e.g., [96]) and concurrency control mechanisms (e.g., [127]).

A notable exception to this trend is the limited attention that has been given to name management for POSs. There have been a few instances of POSs offering some improvements in name management (e.g., [7, 6, 20, 42, 116]). By and large, however, the relatively weak name management mechanisms found in the ancestors of POSs, i.e., programming languages, database systems and operating systems, have tended to endure in POSs, being neither improved nor even effectively integrated. Without better name management, POSs will likely prove cumbersome to use and prone to error, will provide inadequate support for large-scale data storage, multi-user computing, code reuse and interoperation of independently developed object stores, and will therefore fail to realize their potential for beneficial employment in a variety of important application areas.

We believe that our work on name management for convergent computing systems has particular relevance for POSs, so in this chapter we describe our efforts toward providing improved name management approaches in this domain. The chapter is organized as follows. In Section 6.1, we given an overview of the TI/Arpa Open Object-Oriented Database [145], the specific persistent object system used in performing our research. In Section 6.2 we use the features of the PICCOLO framework outlined in Chapter 3 to explore some alternative designs for a basic name management extension to Open OODB. Section 6.3 then describes how a particular design is realized in Open OODB. Section 6.4 describes CONCH, a shell-style user interface to Open OODB. Building on the basic mechanism developed in Section 6.3, CONCH provides various features of the extended PICCOLO framework (as described in Chapter 4) to application developers. Section 6.5 summarizes our mechanism building efforts.

6.1 Open OODB: An Experimental Platform

In this section, we describe Open OODB. We first present a general overview of the system. We then describe some specific Open OODB services and features, focusing on those that we used in constructing name management mechanisms for Open OODB. We conclude with an assessment of the default name management services in Open OODB.

6.1.1 Overview of Open OODB

The Open OODB is an open, extensible, object-oriented database management system (OODB) [145]. An OODB essentially combines the data and control model capabilities of an object-oriented programming language with the data storage capabilities of a database management system. Open OODB is a research prototype, 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 the Open OODB prototype is to allow researchers to experiment with, improve upon and refine various approaches to a broad range of object services, including name management, persistence, transaction, indexing and data dictionary services. Thus, Open OODB provides an excellent testbed in which to experiment with implementations of various approaches to name management.

A high level architectural view of Open OODB is shown in Figure 6.1. Applications interact with Open OODB through *application programming interfaces* (or APIs), where each API defines various language-specific services for applications. For example, a C++ API provides Open OODB services for C++ application developers. Each API then interfaces with the Open OODB *kernel*, which provides general database services (e.g., name management, persistence, recovery, etc.). Beneath this

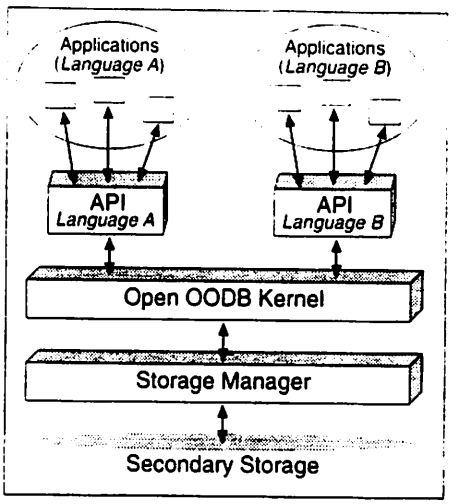


Figure 6.1 Architecture of Open OODB

kernel is a *storage manager*, whose primary responsibility is managing and organizing data on secondary storage devices.

Open OODB can be classified as a *persistent[X]* POS. This is the class of POSs that have been created by extending some existing programming language with support for persistence (and possibly some additional database and operating systems capabilities), but otherwise leaving the underlying language unchanged. The other major class of POSs, namely the *de novo* POSs, are those that have been created by defining new languages specifically designed to provide persistence and possibly some additional database and operating system capabilities. This latter class of POSs has sometimes been able to avoid various name management problems endemic in persistent[X] POSs, since they need not be upwardly compatible from a transient-data-only programming language and its transient-data-only name management mechanism. From our perspective of convergent computing systems, the persistent[X] class poses more interesting challenges. By focusing on Open OODB we limit our attention to that class in this chapter.

We have currently installed an Alpha release[1] of Open OODB on a Sun SPARCstation 10 running SunOS 4.1.4. This single-user, non-distributed version of Open OODB provides APIs for C++[2] and the Common Lisp Object System (CLOS).[3] The kernel itself is implemented in C++. This release uses the Exodus Storage Manager 2.2 [25] as its underlying object store.

6.1.2 Open OODB Services and Features

Open OODB provides a variety of services and features that facilitate the development of large and complex applications. Here we provide a brief overview of a selected set of these capabilities, focusing on those that play significant roles in the design and integration of, and experimentation with, name management in Open OODB. The interested reader is referred to [145] for a more complete description of Open OODB. Unless otherwise noted, these capabilities apply to both the C++ and CLOS APIs.

One important service provided by Open OODB is the *persistence* service, which defines an interface allowing applications to make instances of arbitrarily complex classes persist (or, in other words, outlive the lifetime of the program that created them). Persistence in Open OODB is orthogonal to type: thus, class definitions for persistent objects are identical to those for transient ones. The persistence mechanism used in Open OODB is based on a reachability model of persistence. When an object is made to persist, all objects in its transitive closure (i.e., reachable from the object) will also be made to persist. When an object is later retrieved from the persistent store, all the objects in its transitive closure will also (logically) be retrieved.[4] Thus, programmers are relieved of the burden of having to develop and maintain application-specific code supporting file or database I/O in applications. Finally, in Open OODB, an object may be made to persist at any point during the object's lifetime. In contrast,

[1] The Alpha release of Open OODB is being used by approximately 25 research laboratories.
[2] Release 3.0.1, SC 2.0.1.
[3] Lucid Common Lisp Version 4.1.
[4] Different implementation policies might dictate when and how objects are actually retrieved from the persistent store (e.g., [96]).

less general approaches frequently require persistence decisions to be made either at the time of class definition or object creation.

To gain a better understanding of the approach to persistence in Open OODB, consider Figure 6.2(a), which shows a C++ class definition for a Person class along with applications creating and retrieving a persistent instance of Person.[5] The class definition for Person shows two data members, one of a primitive type (i.e., int) and the other of a user-defined class (i.e., a pointer to Date). From an application developer's perspective, the class definition of Person is nearly identical to that of a non-persistent version of the same class. The only exception arises from Open OODB's requirement that a specially designated header file (i.e., OpenOODB.h) be included in the class definition. This file contains various definitions that make classes persistence-capable.

The application in Figure 6.2(b) shows how a programmer might use the persistence mechanism in Open OODB. First, the database is opened and a transaction is initiated on the database. To designate that an instance of a persistence-capable class should be saved, the persist operation is invoked on the object. When the transaction is committed, any object that was designated as persistent, and any other object (directly or indirectly) reachable from a designated object, is saved to the persistent store. In this example, making a Person object persist results in making the weight and dob data members persist as well.

The application in Figure 6.2(c) shows how a persistent object can be retrieved from the database. This is accomplished by invoking the fetch operation on an instance of the OODB class (a system-supplied object). Fetching the Person object saved in the previous application results in the automatic retrieval of the weight and dob fields. This application also illustrates another useful feature of the persistence mechanism in Open OODB, specifically the ability to obtain *persistent identifiers* (PIDs) for persistent objects and use these PIDs to retrieve persistent objects.[6]

[5]The CLOS version of this example is almost identical.
[6]Open OODB uses the term GIDs instead of PIDs.

```
// C++ Person class
#include <OpenOODB.h>
class Person {
public:
  int Age ();
  void BornOn (char*);
  ...
private:
  Date* dob;
  int weight;
  ...
};
```

(a)

```
// Create a persistent
// Person in the OODB
#include <OpenOODB.h>
#include <Person.h>
main () {
  // Open database
  OODB oodb(...);
  // Create a person
  Person me;
  me.BornOn ("9/23/61");
  ...
  // Save the person
  oodb->beginT ();
  me.persist ("alan");
  oodb->endT ();
  ...
};
```

(b)

```
// Fetch a persistent
// Person from the OODB
#include <OpenOODB.h>
#include <Person.h>
main () {
  // Open database
  OODB oodb(...);
  // Pointer to a person
  Person* anyone;
  ...
  // Get the person using name
  oodb->beginT ();
  anyone = (Person*)
              oodb.fetch ("alan");
  cout << "Age = "
       << anyone->Age();
  // Get the person using a GID
  // (GID = PID)
  id = oodb.ObjectGID("alan");
  anyone = (Person*) oodb.fetch (id);
  oodb->endT ();
  ...
};
```

(c)

Figure 6.2 Persistence in Open OODB

One of the most important features of this approach to persistence is that the Open OODB API and kernel understand how to manage pointers to data objects. For example, the Person class in Figure 6.2 contains a data member that is of type pointer to Date. When the persistence service encounters such a data type, the pointer is translated into an internal persistent identifier, which is used to identify and locate the persistent Date object. (Note that non-pointer data types require no special processing.) When the Person object is fetched from the persistent store, the persistent identifier for the Date data member is translated back to a C++ pointer.[7] The details of this translation process are generally hidden from the application programmers (although, as illustrated above, there are facilities for using persistent identifiers).

Open OODB provides other useful services and features to application developers. A *name management* service provides support for managing the names of persistent objects. We discuss this service, in particular its various shortcomings, in greater detail in Section 6.1.3. Open OODB also provides a *data dictionary* service, which provides a globally shared repository of data type information. Using this service, an application can dynamically access various type information for objects, e.g., the name of an object's class. As we will demonstrate throughout the remainder of this chapter, the use of these services, along with the object-oriented capabilities provided by the APIs, facilitate development and integration of, and hence experimentation with, our prototype name management mechanisms in Open OODB.

6.1.3 Name Management in Open OODB

In its plain, vanilla form, Open OODB provides relatively limited support for name management for persistent objects. As illustrated above, each Open OODB API defines a persist operation for (potentially persistent) classes. This operation takes a name as an argument and when invoked assigns the given name to the (newly persistent) object. The basic APIs also provide a fetch operation, which is used

[7] These kinds of translations are frequently termed *unswizzling* and *swizzling* [95].

to retrieve persistent objects. Figure 6.3 illustrates the usage of these operations in both C++ and CLOS API-based applications. A final important characteristic

```
// C++ Open OODB API                          ;; CLOS Open OODB API
Person me;                                    (setf me (make-instance 'Person))
// Assign name to persistent Person           ;; Assign name to persistent Person
me.persist ("alan");                          (persist OODB me : name "alan")
...                                           ...
Person* anyone;                               ;; Retrieve the Person named "alan"
// Retrieve the Person named "alan"           (setf anyone (get-object OODB "alan"))
anyone = (Person *) OODB→Fetch ("alan");      ...
```

Figure 6.3 Default Name Management Mechanism in Open OODB

of the default name management mechanism in Open OODB (and in OODBs in general) is that the persistent store is segregated according to the language used to define objects. For example, names for persistent C++ objects are not visible to CLOS objects and *vice versa*. Thus, in Figure 6.3, the two occurrences of the name alan refer to distinct objects. Moreover, not only is the name space segregated, but the data space is segregated as well. For example, persistent CLOS data can not be accessed by C++ applications, and *vice versa*. We explore this problem, and solutions to it, in Chapter 7.

In terms of PICCOLO, nameable objects in Open OODB are instances of (C++ and CLOS) classes. Names are formed from any sequence of characters that can be enclosed by a pair of matching quotes. Binding spaces support name uniqueness and name aliasing. The binding space organization in Open OODB is flat, i.e., there is only a single repository (i.e., per language) for named, persistent objects in Open OODB. The persist operation is restricted to *redefine* semantics, i.e., using an already assigned name as an argument to the persist operation results in overriding any previously existing binding that contains the same name. Context specification is limited to the *infuse* of the single binding space. In addition, there is only a single epoch for context

formation, i.e., at run-time. The existence of a single binding space and epoch results in extremely simple CFTs, CFPs and extended closure descriptions. In also, however, leads to very limited capabilities and flexibility; hence, the need for enhancements to Open OODB's name management service.

6.2 Designing a Basic Name Management Mechanism

As illustrated by Open OODB, existing approaches to name management in OODBs tend to be incommensurate with the rich type and computation models they support. Open OODB also illustrates another interesting complication for existing approaches. Specifically, the provision of multiple languages in a single OODB raises the possibility that (potentially) persistent objects may be defined and accessed by different languages. Together, these characteristics make OODBs obvious candidates for extended name management mechanisms.

A first step toward the development of a mechanism involves creating an appropriate design for it. Guided by the PICCOLO framework, we explore three alternative designs of name management mechanisms. We begin by outlining some overall requirements for an extended name management mechanism for Open OODB in Section 6.2.1. Section 6.2.2 then describes the details of each design. We conclude with an assessment of these designs in Section 6.2.3. In particular, we argue for the selection a particular design, whose implementation we elaborate in Section 6.3.

6.2.1 Requirements

The designs presented in this section are, more or less, equivalent to one another in terms of name management semantics. The flexibility inherent in object-oriented languages, however, results in various possible design alternatives, which in turn provide somewhat different interfaces as seen and used by application developers. Before delving into the specifics of these designs, including their various features and

tradeoffs, we first define a set of requirements that we would like each of the designs to meet. In particular, these requirements revolve around some general mechanism properties, some specific name management properties, and the choice of a design notation.

General mechanism properties The internals of Open OODB should remained unchanged. Moreover, we want to develop a modular name management mechanism that relies, to the greatest possible extent, on the general object-oriented database features provided by Open OODB. By proceeding in this manner, we hope to define mechanisms that can not only be potentially adapted by, or reused in, other OODBs, but also can be easily modified and/or extended in the future.

In keeping with principles of orthogonality and transparency [148], the mechanism should not discriminate between persistent and transient objects, although one primary use of the name management mechanism is intended to be in organizing the persistent repository.

Finally, a single, unified mechanism should be provided to both C++ and CLOS application developers (in the case of Open OODB.). One potential benefit of the resulting mechanism is that it should facilitate the integration of additional APIs with our name management mechanism (although appropriate API-specific interfaces would clearly need to be defined). Furthermore, a uniform name management interface should ease an application developer's ability to move among APIs. In Chapter 7, we show how a uniform approach to name management facilitates and enables transparent interoperability in OODBs.

Specific Name Management Properties For our purposes in designing a name management mechanism for Open OODB, we want to support the following name management properties:

- Nameable objects should be instances of classes, as defined by the underlying object-oriented API.
- Binding spaces and contexts should support, at the very least, all the associated operations as defined in PICCOLO.
- Name uniqueness and aliasing for both bindings spaces and contexts.
- A structured binding space organization that is single rooted, networked and (potentially) cyclic.
- Compound names using "/" as a delimiter. Note that the possibility of cyclic binding space organizations slightly complicates the resolution of compound names.

Our choice of name management properties was guided primarily by our goal of developing a general and flexible approach. We believe the properties outlined above meet these demands.

Design Notation We selected C++ as our design language. One factor influencing our choice is related to the ease of translating from design to implementation. This facilitates the rapid creation of executable prototypes, which can be tested and observed. Another factor is the relative popularity of C++ and CLOS. Although Open OODB supports both languages, we felt that using C++ would attract a wider audience, given its prevalence in both the research and industrial communities. Finally, the fact that C++ is so closely related to C, a common language for implementing foreign language interfaces, also contributed to our decision. (This importance of this latter factor will become more apparent in Chapter 7.)

6.2.2 The Designs

Given these requirements, we now present three alternative designs for a name management mechanism. Our designs are intentionally concise – for each design, we give the primary C++ class definitions, including the most relevant operations. Of course, in designing any mechanism, auxiliary classes and operations may be required (e.g., Boolean and Validity classes could be used as return codes from operations). For the sake of brevity, however, the definitions of such additional classes are omitted from our designs.

The first design, presented in Section 6.2.2.1, corresponds almost directly to the PICCOLO framework. In Section 6.2.2.2, we describe a slight variation on this first design, which attempts to couple some rudimentary type checking with name management services. The third design, presented in Section 6.2.2.3 represents a middle ground between the first and second designs.

6.2.2.1 A Binding Centered, Type Weak Approach

We begin by considering a design corresponding to a straightforward mapping of the PICCOLO framework into a mechanism. Figure 6.4 shows the relevant C++ classes and operations corresponding to such a design. Since this design is based on a direct mapping of PICCOLO, it should not be surprising that the classes shown in Figure 6.4 are very similar to those in Figure 3.1 (in Chapter 3).

Experimenting with a prototype of this design reveals some interesting characteristics and suggests several possible variations. A striking characteristic of this approach is the central role that the Binding class plays: naming is accomplished by explicitly creating a binding and then explicitly inserting that binding into a binding space, for example. Although useful for modeling purposes, making bindings explicit in a name management approach may seem unnatural and awkward to users. Furthermore, and more importantly, making bindings explicit entities in a name management approach

```
class NameableObject {
public:
  Boolean Identical (NameableObject&);
  TypeInfo Typeof ();
  ...
};
```

```
class Name {
public:
  Name (char*);
  Name (Name&);
  Name& Leftmost ();
  Name& Rest ();
  Boolean Compound ();
  ...
};
```

```
class Binding {
public:
  void Bind (Name&, NameableObject&);

  Name& UnbindName ();
  NameableObject& UnbindObject ();
  void Rebind (Name&);
  Boolean NameCollision (Binding&)
  ...
};
```

```
class BindingSpace : public
NameableObject {
public:
  Boolean Define (Binding&);
  Boolean Redefine (Binding&);
  Boolean Undefine (Name&);
  Boolean Rename (Name&, Name&);
  Names List ();
  ...
};
```

```
enum Validity {Valid, Invalid};
class Context {
public:
  NameableObject* Resolve (Name&);
  Validity Infuse (BindingSpace&);
  Validity Restrict (Name&, ...);
  Validity Exclude (Name&, ...);
  Validity UnionOverride (BindingSpace&);
  Validity Union (BindingSpace&);
  Validity Prefix (Name&);
  ...
};
```

Figure 6.4 A Binding-Centered, Type-Weak Approach

can lead to some potential complications. In particular, if bindings are first class objects (in the object-oriented sense of the term), then the name assignment operations defined by the BindingSpace class can be bypassed. Thus, an object's assigned name could be changed (i.e., by using Rebind) without going through its containing binding spaces. This can clearly lead to some problems in an approach where name uniqueness is to be guaranteed in binding spaces and contexts. One possible solution to this problem is removing Rebind from the (public) interface of the Binding class. While this eliminates the possibility of a binding being changed without going through the BindingSpace interface, as noted above, bindings still remain visible entities, which, as also noted, may seem non-intuitive to programmers.

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 (a pointer to) NameableObject. This means the programmer must cast the result of the Resolve function into the appropriate type. Figure 6.5 illustrates the resolution of an instance of Person in some context. Although type

```
Name someName;                                  // instance of Name
Person* somePerson;                            // pointer to a Person
Context someContext;                           // instance of Context
// Assume someContext has been formed, etc.
someContext = ...
// Type cast result of name resolution
somePerson = (Person *) someContext.Resolve (someName);
```

Figure 6.5 Casting the Result of the Resolve Function

casting is a common technique in languages such as C++ and CLOS, its use can lead to serious programming errors and anomalies.

Based on these latter two observed characteristics, we designate the basic approach described in Figure 6.4 as a *binding-centered, type-weak* approach to name management.

6.2.2.2 An Object-Centered, Type-Preserving Approach

Our next design attempts to overcome the shortcomings inherent in the initial design. This alternative might be said to represent an *object-centered, type-preserving* approach. A modification to the design of Figure 6.4 that yields such an approach is shown in Figure 6.6. The modification centers around the removal of the Binding class as a first class entity and a partitioning of the NameableObject class into two separate classes – a BaseNameableObject class and a NameableObject template class.

In this design NameableObject is a generic (or parameterized) class that inherits from the BaseNameableObject class. The NameableObject class includes several operations that were provided by the BindingSpace and Context classes in the binding-centered, type-weak design. Specifically, the name assignment and manipulation operations (defined in the first design) and the name resolution operation are now defined by this class. To use this class, an entity's class definition inherits from a NameableObject class parameterized by the entity class itself. Figure 6.7 shows an example of the C++ class definition.

The use of parameterized classes and inheritance in this design leads to every subclass of NameableObject having the appropriate name assignment, name removal and name resolution operations associated with it. In this design, each class provides the primary name management operations for its instances. 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 name definition operations associated with the NameableObject class. On the other hand, having the Resolve operation associated with the class of an object, instead of with the context, may seem somewhat unnatural and counterintuitive to programmers.

```
class BaseNameableObject {
public:
  Boolean Identical (BaseNameableObject&);
  TypeInfo Typeof ();
  ...
};
```

```
class Name {
public:
  Name (char*);
  Name (Name&);
  Name& Leftmost ();
  Name& Rest ();
  Boolean Compound
();
  ...
};
```

```
template <class Type>
class NameableObject : public BaseNameableObject {
public:
  Boolean Define (Name&, BindingSpace&);
  Boolean Redefine (Name&, BindingSpace&);
  Boolean Undefine (Name&, BindingSpace&);
  Boolean Rename (Name&, Name&, BindingSpace&);
  static Type& Resolve (Name&, Context&);
  ...
};
```

```
class BindingSpace :
  public NamableObject<BindingSpace> {
public:
  Names ListNames ();
  ...
};
```

```
enum Validity {Valid, Invalid};
class Context {
public:
  Validity Infuse (BindingSpace&);
  Validity UnionOverride (BindingSpace&);
  Validity Union (BindingSpace&);
  Validity Exclude (Name& name, ...);
  Validity Restrict (Name& name, ...);
  Validity Prefix (Name&);
  ...
};
```

Figure 6.6 An Object-Centered, Type-Preserving Approach

```
class Person : public NamedObject<Person> {
public:
  // public interface for Person
  ...
private:
  // private section
  ...
};
```

Figure 6.7 Using the NameableObject Class

6.2.2.3 A Binding Space/Context-Centered, Type Weak Approach

The first design we described is appealing from the standpoint of simplicity, while the second design provides some useful type preserving capabilities. Although desirable, these properties, in both cases, are achieved at the expense of making the name management interface somewhat non-intuitive. Thus, a final design that we will consider represents a compromise between the previous two designs. We characterize this third alternative as *binding space/context-centered, type-weak*. Figure 6.8 shows the C++ classes that make up this design.

Similar to the first design, the Resolve operation is defined by the Context class and returns a pointer of class NameableObject. Again, the programmer is required to specify the appropriate type cast when invoking this operation. In addition, the name assignment and manipulation operations are associated with the BindingSpace class. Like the second design, these operations take a Name and an NameableObject as parameters; hence, bindings are not first class entities. Thus, this design essentially makes a trade off between the type preserving properties of the second design and some of the simplicity properties of the first design, while retaining the second design's advantage of shielding users from explicit manipulations of bindings.

```
class NameableObject {
public:
  Boolean Identical (NameableObject&);
  TypeInfo Typeof ();
  ...
};

class Name {
public:
  Name (char*);
  Name (Name&);
  Name& Leftmost ();
  Name& Rest ();
  Boolean Compound ();
  ...
};

class BindingSpace : public
NameableObject {
public:
  Boolean Define (Name&,
NameableObject*);
  Boolean Redefine (Name&,
NameableObject*);
  Boolean Undefine (Name&);
  Boolean Rename (Name&, Name&);
  Names ListNames ();
  ...
};

enum Validity {Valid, Invalid};
class Context {
public:
  Object* Resolve (Name&);
  Validity Infuse (BindingSpace&);
  Validity Restrict (Name&, ...);
  Validity Exclude (Name&, ...);
  Validity UnionOverride (BindingSpace&);
  Validity Union (BindingSpace&);
  Validity Prefix (Name&);
  ...
};
```

Figure 6.8 A Binding Space/Context-Centered, Type-Weak Approach

6.2.3 Summary

In this section, we have presented three designs for a name management mechanism. We experimented with each of these designs to help select an appropriate choice on which to base an enhanced name management mechanism for Open OODB. Initially we carried out this experimentation by combining the C++ class definitions with a simple testing harness, also implemented in C++, and then executing some simple scenarios that exercised the various name management capabilities. This involved choosing an arbitrary collection of C++ classes and incorporating them into the test harness (i.e., by making them subclasses of the appropriate name management-enabling class). We then created instances of the classes, assigned names to them (in binding spaces), created various contexts, and resolved names in contexts. Once satisfied with a particular design, we could straightforwardly integrate it into Open OODB and perform further experimentation.

Although each of the designs is a reasonable candidate, we chose the third design (from Section 6.2.2.3) as the one to actually integrate with Open OODB. Our selection was based primarily on practical considerations. We were especially concerned with providing an interface that would seem most natural for application developers. Making bindings first class entities did not seem very intuitive, while associating all name manipulation operations with the objects themselves seemed cumbersome. Although the third design still suffers from the type-casting problem, we felt that of the three it was the most reasonable compromise among the three designs. The next section details how this particular design is actually integrated and used in Open OODB.

6.3 Integrating a Name Management Mechanism

One facet of implementing a name management mechanism for Open OODB involves fleshing out its class definitions (e.g., adding any needed auxiliary functions).

providing appropriate implementations for the corresponding class operations, and building the necessary underlying data structures. The other facet involves integrating the mechanism into Open OODB itself.

In the remainder of this section, we discuss several important implementation issues that must be addressed in order to integrate a particular design into Open OODB. Specifically, we show how a name management mechanism is bootstrapped. We then describe how arbitrary classes are incorporated, so that instances of such classes can be supported by the name management mechanism. Next, we describe some novel implementation techniques that we have developed. In particular, these techniques permit independently developed applications, perhaps written in different languages, to utilize the same persistent binding space organization.

6.3.1 Bootstrapping the Name Management Mechanism

As noted earlier, one of our objectives in developing a name management mechanism for Open OODB is that it should provide a structured, single rooted binding space organization for the persistent store. To achieve this, the following steps must be performed. Note that these steps need only be performed once, in all likelihood by a database administrator.

1. Apply the Open OODB persistence-extending tools, which are used to extend C++ classes with the necessary persistence capabilities, to all of the constituent classes of the name management mechanism. As a result, instances of the classes that make up a name management mechanism (i.e., names, objects, bindings, binding spaces, and contexts) are themselves persistence-capable.

2. Create a persistent root binding space. This is easily accomplished by executing a procedure that makes an instance of BindingSpace persist using the default persistence mechanism in Open OODB. A function that returns this binding space is then made available to all applications that are using the name man-

agement mechanism. Figure 6.9 shows the fragments of the CreateRoot and FetchRoot operations that support the required functionality.

```
// Returns a pointer to a
// new root binding space.
BindingSpace* CreateRoot () {
  // Create the binding space
  BindingSpace* Root = new BindingSpace;
  // Make it persist using
  // Open OODB default
  Root→persist ("___ROOT___");
  // Return the root
  return (Root);
}
```

```
// Fetches the root binding
// space from database.
BindingSpace* FetchRoot () {
  // Pointer to a binding space
  BindingSpace *Root;
  // Retrieve it using
  // Open OODB default
  Root = (BindingSpace *)
    OpenOODB→fetch ("___ROOT___");
  // Return the root
  return Root;
}
```

Figure 6.9 Bootstrapping the Name Management Mechanism

6.3.2 Incorporating a Class

Once the name management mechanism has been bootstrapped, any class that desires to participate in the mechanism must be modified to inherit from the name management-enabling class. This requires that classes inherit from the NameableObject class. Once a class is made to inherit from the appropriate name management-enabling class, the Open OODB persistence-extending tools must be applied once again, this time to the modified class. Note that objects that are not required to be named, but which may be required to persist (in other words, by being made reachable from other name management- and/or persistence-capable objects) must also have the Open OODB persistence-extending tools applied to them. In our running example in this section, the Date class in not nameable, but it is persistence-capable.

6.3.3 Implementation Considerations

Our prototype implementation of a name management mechanism takes advantage of several features of the Open OODB itself. In particular, orthogonal persis-

tence and persistence by reachability are two features that have proven to be very valuable in developing and integrating enhanced name management mechanisms in Open OODB. They allow the design of a name management mechanism to be developed without regard to persistence and facilitate the subsequent integration of that mechanism into Open OODB. The resulting mechanism provides improved name management capabilities and is accomplished without necessitating any significant changes to Open OODB itself.

Although these features have been extremely useful, there are significant costs incurred as a result of their use in our prototype implementations. Most of these costs arise because applications that utilize these capabilities make the tacit assumption that the object code[8] implementing the various classes of the persistent objects that they may access has been linked into the application *a priori*. For example, an application accessing an instance of the Person class expects the object code for both the Person class and the Date class to be linked in the address space of the application. While this assumption is perfectly appropriate for many applications, it has some problematic implications for a name management mechanism.

Specifically, a binding space organization can be viewed as a polymorphic data structure containing and organizing references to many types of objects (where objects are required to be subclasses of the NameableObject class). This data structure, because it contains pointers to every possible (nameable) object type, effectively spans the entire persistent store. The persistent store is intended to act as a shared repository of named data, which applications, developed independently of one another, can deposit objects into, and access objects from. The existence of the binding space data structure, however, has the effect of giving any application apparent access to all the types of objects stored in the repository. This in turn imposes the unrealistic, if not

[8]By object code, we mean the binary code created by a compiler.

completely infeasible, requirement that each application must link the object code for all the classes of objects contained in the repository.

The Open OODB persistence features also make the implicit assumption that all persistent objects used by an application were developed in the same language as the application. This means, for example, that when a C++ application indicates that an object should persist, Open OODB assumes that any pointers to, or within, the object are references to C++ objects. This assumption causes no difficulties as long as our name management extensions to Open OODB are implemented in the same language as the applications that use them. For instance, to support applications using the C++ API, bindings (in binding spaces) can be implemented as (Name, NameableObject*) pairs in a C++-specific name management mechanism. The persistence service (for C++) understands how to process these pointers and perform the required translations (from in-memory to secondary storage and *vice versa*). A CLOS-specific mechanism could be built similarly.

One of our major objectives, however, is that the name management mechanism should support objects defined in multiple languages (in the case of Open OODB, C++ and CLOS). In fact, in our prototypes we implemented the name management mechanism entirely in C++ and simply created the necessary name management interfaces in C++ and CLOS, where the CLOS interfaces are, more or less, merely call-outs to the appropriate C++ implementations.

Thus, a binding mechanism implemented as (Name, NameableObject*) pairs, while sufficient for C++ objects, is not adequate to support objects defined in other languages. When a transaction is committed, the Open OODB persistence service assumes that all references are to objects defined in a specific language. In particular, if the pointer is to a C++ object, then the C++ persistence mechanism knows hows to unswizzle it. When the persistent data structure being stored consists of (C++-implemented) bindings of names to (CLOS) persistent objects, this would lead to the

C++ persistence mechanism being applied to CLOS objects, with unpredictable (and meaningless) results.

To summarize, issues related to reachability and uniformity introduce complications in the development of a general and flexible name management mechanism. Thus, the question is how can we define a general binding construct in a name management mechanism so that:

- bindings can contain objects defined in different languages, and,
- the orthogonal persistence and persistence by reachability properties are still maintained.

To overcome these complications, we developed the notion of a *universal object representation* (or UOR). A UOR is a persistence-capable class that encapsulates language-specific information about objects, including the language used to define an object and its various internal identifiers (e.g., its transient and persistent memory addresses). Instead of maintaining bindings between names and pointers to the base class NameableObject, the name management mechanism creates bindings between names and UORs. Although the construct was initially invented to overcome the problems enumerated above, it turns out that UORs, with some minor extensions, also enable transparent interoperability in Open OODB. Therefore, we provide only a general description here, and defer a more detailed examination of UORs to Chapter 7.

When a name is assigned to an object, a UOR is created for the object. The UOR initially stores the language used to define the object and an in-memory address for the object. The UOR is then inserted into the binding (in the given binding space). Before a transaction is committed, the binding space organization for the persistent store is traversed and each UOR is examined. For each UOR in a binding, the native persist operation, i.e., native to the language used to define the object, is invoked. In our implementation, if the object is a C++ object, then the C++ API

persist operation is invoked: if the object is a CLOS object, then a call-out is made (from C++) to the CLOS API persist operation. In both cases, a *persistent identifier* is returned. This identifier, which uniquely identifies a persistent object, is then recorded in the UOR and the in-memory reference is erased. The persistence service then treats instances of the UOR class like any other persistence-capable object. UORs, however, are strictly self-contained objects. Since the in-memory address field is empty, the persistence service will not attempt to interpret that field as a data pointer and the persistent identifier is treated as sequence of bytes. Finally, when a name for an object is resolved, its binding is first examined (assuming, of course, that the resolution was successful). If the UOR does not contain an in-memory address, then the appropriate persistence service must be called in order to retrieve the object from the persistent store. Again, if the language of the object is C++, then the C++ persistence service is called. If the language of the object is CLOS, then the CLOS persistence service is called. In each case, the persistent identifier (recorded in the UOR) is passed to the appropriate persistence service and an in-memory reference is returned. This reference is recorded in the UOR and returned by the Resolve function.

6.3.4 Evaluating the Basic Mechanism

Guided by the PICCOLO framework, we have designed, developed and implemented a mechanism that is more general and flexible than approaches found in existing POSs, including those embodied by emerging standards (e.g., [28]). As observed by Loomis [82], existing OODBs use flat naming structures, and at the same time, as far as we are aware, none provide uniform support for objects defined in multiple languages. Thus, we contend that the approach described in this section represents a significant improvement in the area of name management in OODBs (and more generally in POSs).

Beyond this general assessment, we offer three more specific kinds of evidence regarding the utility of the mechanism described in this section. First, we note that

the an operational prototype of our name management mechanism has been included and distributed in the most recent release of Open OODB. Second, in Section 6.3.4.1, we give a more comprehensive qualitative assessment of our enhanced mechanism's capabilities, in particular, providing more detailed comparisons of it to the state-of-the-art. Finally, although performance was not a primary objective of this prototyping effort, we also include some preliminary performance data in Section 6.3.4.1 comparing our approach to the default mechanism in Open OODB.

6.3.4.1 Summary of Features

The name management mechanism described in this section provides several advantages over existing approaches in the OODB domain. A summary of the features leading to these advantages is given below.

Organization The structured binding space organization, around which the persistent stored is organized, clearly provides a substantial improvement over existing approaches in OODBs. It is (not surprisingly) quite similar to the organizational structures found in many file systems. Unlike file systems, however, our name management mechanism supports typed objects (as opposed to type-less byte streams in file systems). Similarly to file systems, multiple, independently developed programs can access the persistent store without requiring them to be aware of non-pertinent objects.

Context OODBs limited to flat binding space organizations typically have a rigid context formation policy, i.e., there is a single context corresponding to the single binding space. The combination of the structured binding space organization and the context formation operations in our mechanism allow applications to construct contexts in a flexible, incremental fashion. In Section 6.4, we introduce some even more powerful and general context-control mechanisms.

Multiple Languages Many modern OODBs support multiple languages or APIs where the persistent store is segregated by the language used to define the persistent objects. Our approach unifies the persistent store, allowing objects defined in different languages to co-exist. In addition, the name management interface is consistent across APIs. Thus, programmers can move between APIs, while the semantics of the underlying name management mechanism remains unchanged. As we will see in Chapter 7, this feature facilitates interoperability between languages in Open OODB .

Basic Abstractions Our enhanced name management mechanism was constructed using the basic abstractions provided by C++, CLOS and Open OODB. Building the mechanism is this fashion (as opposed to modifying the internals of the OODB) leads to several distinguishing features. Since the classes that make up the mechanism are treated like any other (C++ or CLOS) class, the name management mechanism can be extended or customized using standard object-orientation-based techniques. In addition, the mechanism can be used to create name spaces for purposes other than the persistent store. For example, an arbitrary object (transient or persistent) can create private or local name spaces by simply creating instances of the BindingSpace type. Furthermore, such a private name space (or some subset of it) can be moved into the persistent repository by simply assigning a name to the appropriate binding space in a persistent binding space.

Orthogonality For the most part we have focused on how the name management mechanism can be used to organize and access the persistent store. However, the mechanism can be used independent of the persistent store. Specifically, because naming is an orthogonal property, the name management mechanism can be used, for example, to create temporary naming structures for programs.

In other words, a program can create one or more binding space organizations for either transient or persistent objects or both. Binding space organizations can be merged or divided using the appropriate operations (i.e., Define, Undefine).

Named-based Persistence An additional benefit of incorporating our enhanced name management mechanism into Open OODB is that it gives rise to a novel name-based persistence mechanism (i.e., an alternative persistence model based on name assignment and manipulation). Since persistence is based on reachability in Open OODB, assigning a name to an object in a binding space that is (directly or indirectly) reachable from a (specially designated) persistent root binding space results in the binding, the name and the object (including the objects in its transitive closure) being made persistent. In the resulting, extended Open OODB, 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. A high-level illustration of our name-based persistence mechanism in Open OODB is given in Figure 6.10. The left hand portion of the figure shows the initial state of the binding space organization for the persistence store. In the middle of the figure, a C++ code fragment assigns the name alan to an instance of a Person class in a binding space. After assigning the name (and assuming the database transaction is successful), the state of the persistent store is shown in the right hand portion of Figure 6.10.

6.3.4.2 Assessing Costs

Although the various capabilities described above are clearly important and useful, an obvious set of questions that arise concern the various costs incurred by using our mechanism. As an initial response to these questions, we can consider the following issues:

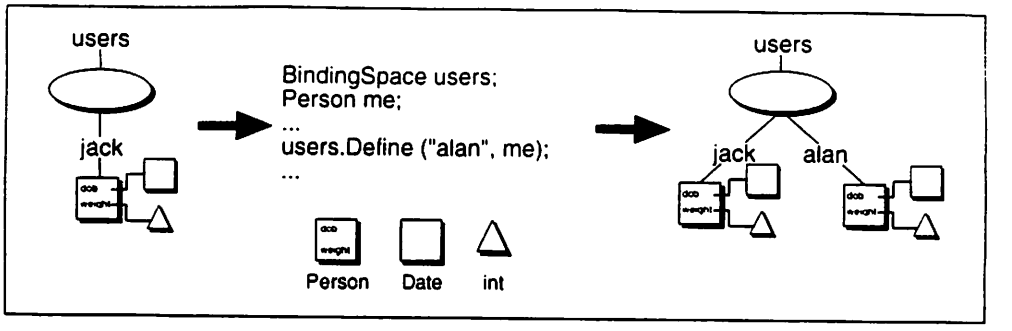


Figure 6.10 Name-based Persistence

- How easily can the mechanism be adopted?
- What is the performance overhead?

Obviously, any thorough assessment of ease of use for a computing system should be based on substantial feedback from users and programmers. We believe, however, that the features provided by our mechanism will be fairly familiar and natural to many potential users and hence relatively easy to adopt. As noted above, the structured binding space organization is similar to that found in file systems. In addition, incorporating classes into the mechanism is relatively straightforward. This is achieved by simply making the class inherit from the NameableObject class. We realize that one possibly shortcoming of this approach is that it implies that the source code for all classes is available. This might be infeasible in some situations, especially where classes are delivered in the form of object libraries. On the whole, however, we feel safe in claiming that the barriers, both conceptual and practical, to adopting our enhanced name management mechanism for Open OODB are quite modest.

To obtain a preliminary indication of performance overhead, we conducted a series of simple experiments comparing the response time of Open OODB's default name management mechanism to our enhanced one. To perform these experiments, we

first chose an arbitrary C++ class library from a publicly available repository.[9] In particular, we selected the AMOS (Active Media Object Store) Audio class library, which is part of a toolset that can be used to edit and manipulate audio data.[10] In its original form, the Amos Audio class library uses the native file system to load and save audio files.

Given this class library, we developed a simple program that created instances of the Amos Audio class library populated with sample audio data, assigned names to the instances, and then resolved names of the instances. The program is organized as follows:

Establish Data Objects The program creates a specified number of instances, n, of the Amos Audio class. It then populates each instance by randomly choosing from a set of 50 sample audio data files. The data files range in size from approximately 750 bytes to 9000 bytes, where the average size of a data file is approximately 3600 bytes. Next, the program generates n names, which are random sequences of characters ranging in length from 10 to 20 characters.

Assign Names to Objects A transaction is initiated on the database. The n names are then assigned to the n objects. The transaction is then completed.

Resolve Names for Objects A transaction is initiated on the database. In this step, $n/2$ names for objects are resolved, where the names are chosen randomly. The transaction is then completed.

For each of the experiments, the program was run on data sets ranging in size from 100 to 1000 instances, incrementing by steps of 100.[11] For each run of the program,

[9] Announcements of public domain C++ classes and libraries are frequently posted in various Usenet newsgroups and WWW services.

[10] The AMOS Audio class library is available from the German National Research Center for Computer Science (GMD), Integrated Publication and Information Systems Institute (IPSI), Dolivostrasse 15, D-64293 Darmstadt, Germany.

[11] We realize that the data sets may not be indicative of realistic settings. One limiting factor in running these experiments was that we were restricted in terms of local disk space.

the previous database was removed and a new, empty one was created. The first series of experiments was based on the Amos Audio library using the default name management mechanism in Open OODB. The second series of experiments used our enhanced name management mechanism. In this latter case, a single binding space was used for name assignment and name resolution to parallel the situation in the default mechanism.

The graphs in Figures 6.11, 6.12, and 6.13 compare response times for each approach. Figure 6.11 measures the total running time for the program (under each approach), while Figures 6.12 and 6.13 gives response times for name assignment and name resolution, respectively. As should be evident from this data, the enhanced name management mechanism incurs some additional, but not overwhelming, overhead. A combination of factors, however, probably contributes the observed performance degradation.

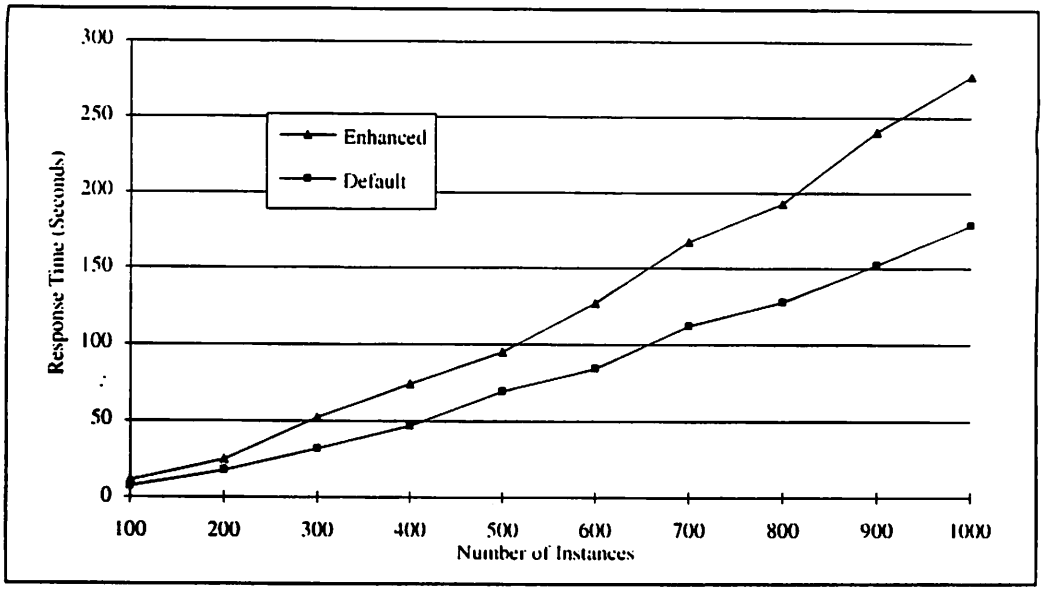


Figure 6.11 Total Response Time

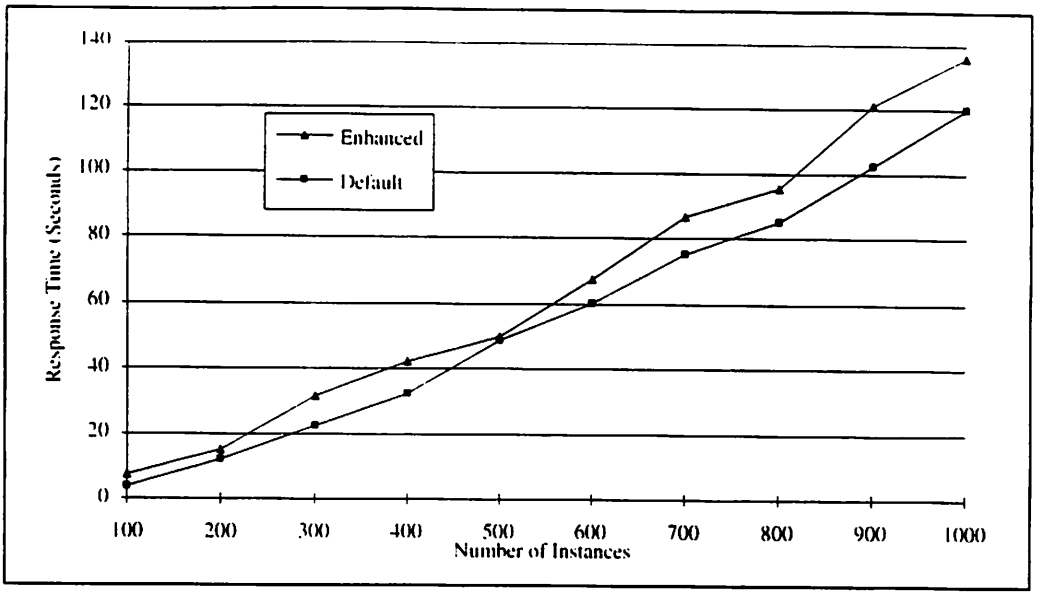


Figure 6.12 Name Assignment Response Time

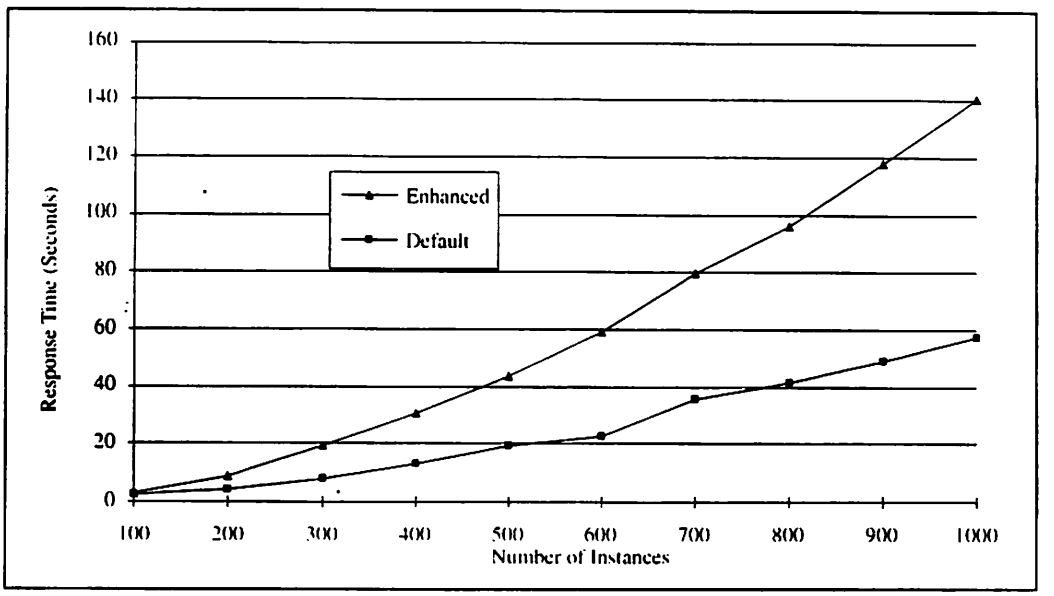


Figure 6.13 Name Resolution Response Time

- Recall that the UOR construct is used to represent objects defined in multiple languages. Although our experiments were conducted entirely in C++, the various translations and tests performed by UORs are still executed. In particular, when a name is assigned to an object, a UOR must be created for it. This means that all the language, type information, etc. must be stored in a UOR. When a name is resolved for an object, the UOR must be mapped into the actual object. Among other things, this requires objects to be retrieved (rather naively) from the persistent store on a per-object basis. Finally, when a transaction is committed, the binding space organization must be traversed so that the correct identifiers are computed and set in the UORs. In particular, if a PID has not been assigned in a UOR for an object, then the low-level persistence mechanism must be invoked.

- In hindsight our choice of data structures supporting binding spaces was not optimal. Specifically, we use binary trees to represent binding spaces, while the default mechanism in Open OODB uses hash tables. Furthermore, since the default mechanism is implemented as an internal component of Open OODB, it is well-tuned for the storage and retrieval of persistent objects. The data structure supporting the binding space organization, on the other hand, is simply treated as a single data object by Open OODB. Thus, when the root binding space is retrieved, the entire binding space data structure is (logically) retrieved. Note that the UORs contain raw PIDs; hence, the individual objects are not retrieved until the object is actually resolved. This partly explains why name resolution seems to incur more overhead than name assignment.

- Another notable source of performance overhead is the additional I/O. Since the binding space structure itself requires more space, some additional time is required for processing the data structure.

- Finally, as we noted earlier, our prototype implementation was not constructed with performance as a primary goal. In addition to less than optimal algorithms and data structures, some performance costs can be attributed to inefficiencies in an object-oriented programming paradigm.

We believe that the additional performance overheard is not too severe considering the capabilities provided by our enhanced name management mechanisms. Furthermore, we believe that it would be straightforward to make substantial improvements in its overall performance. For example, rather than fetching objects on a per-object basis, a more opportunistic policy might pre-fetch a collection of objects. We defer a discussion of these and other possible performance enhancement strategies to Chapter 8

6.4 Conch - A Context Controlling Shell

As noted in the introduction to this chapter, POSs tend to provide inadequate approaches to name management, especially with respect to controlling context. While orthogonal persistence should obviate the need for traditional persistence mechanisms, such as file systems and databases (at least from the programmer's perspective), users of POSs are typically still faced with using primitive name management mechanisms, based on those found in operating systems or database management systems, for controlling the meanings of names for persistent objects. The mechanism described in Section 6.3 produces a substantial improvement by adding the basic features of the PICCOLO name management model to the name management capabilities provided in Open OODB. The result, as seen in Section 6.3, was improved support for name space organization. Even this enhanced mechanism, however, does not provide as powerful and general mechanism for forming, manipulating, controlling and reasoning about contexts as a user might need in certain circumstances. For example, context formation in the mechanism described in Section 6.3 is restricted to run-time.

In this section, we describe the results of our work on further improving the context control facet of name management in POSs. We begin by sketching some representative context control problems arising in POSs that are not addressed by the enhanced mechanism presented in Section 6.3. We then describe a prototype realization of the framework in the form of CONCH, a shell-style user interface to Open OODB and show how it can be used to address the representative problems. The section concludes with an assessment of the current status of this work.

6.4.1 Scenario

As an illustration of the shortcomings of context control for POSs, consider the situation facing a hypothetical application developer wishing to build a system that accesses various electronic mail-related objects from a persistent store. Some name management problems that must be resolved include:

1. The developer wants to be able to organize objects and their names in the persistent store into logical, meaningful collections.

2. The developer wants to be able to flexibly form contexts giving particular meanings to names used in the application. The resulting contexts may not correspond exactly to any single collection in the set of collections from (1) above. Furthermore, the developer wants to be able to specify different contexts for the application without necessarily having to re-compile (or possibly even re-link) the application.

3. The developer wants to be able to reason about how contexts are formed and used by the application. For example, given an application, the developer should be able to determine what names the application refers to, whether objects corresponding to those names exist, whether the objects can be accessed and/or modified by others, and whether the names used have unambiguous meanings.

As an example of the first point, our hypothetical developer might wish to impose the conceptual organization depicted in Figure 6.14 on a relevant subset of objects in the persistent store. In this figure, persistent objects are represented by both

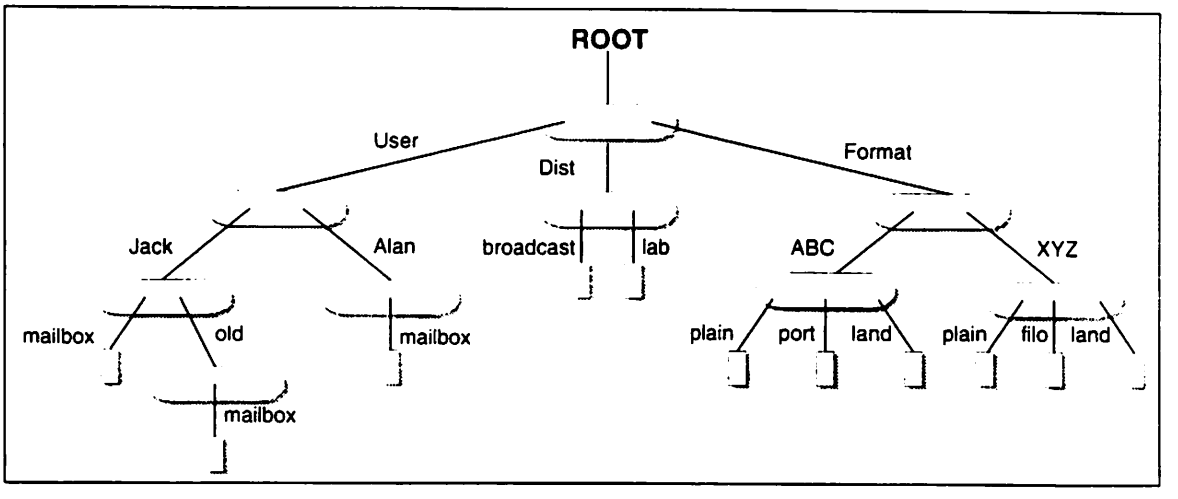


Figure 6.14 Conceptual Organization of the Persistent Store

ellipses and rectangles. Ellipses denote collections of named objects, where names are attached to arcs emanating from an ellipse to a corresponding object, represented by either a rectangle or another ellipse. The only exception to this is the name ROOT, which identifies the root collection of the structure and is not contained in any other collection. For the purposes of this example, assume that the objects named mailbox contain electronic mail messages, that the objects named broadcast and lab represent distribution lists, and that the collections named ABC and XYZ contain a variety of message-printing formats provided by two different vendors. Note that these collections happen to have common names for some objects, a problematic, though not uncommon, situation in the commercial arena [1]. In addition, we assume that multiple users and/or applications may be accessing and modifying the persistent store.

Given this logical organization of the persistent store, our developer's goal is an application that can manipulate a user's mailbox (e.g., compose, send, read, print and archive messages), making use of all the distribution lists and some subset of

the message formats. Correct execution of this application will depend upon the availability of an appropriate context, connecting names used by the application with specific objects in the persistent store. Thus the application developer, or user, needs some means of controlling context formation. For example, the developer should be able to specify what names are needed in the desired context, what objects are bound to those names, at what point in the application's lifetime name-object associations are formed, and whether names are associated with existing objects or copies of objects in the persistent store.[12] For instance, one possible set of context requirements (and clearly not the only set) for this hypothetical application might include the following:

1. It should be possible to use the application with *any* user's mailbox without either having to re-compile (or re-link) the application or having to interactively interrogate the user for the location of the mailbox. This means that the context for the application must include an association between the name mailbox and a specific object in the persistent store that is formed at the time when the mailbox object is actually accessed in the application (as opposed to at compile-time, link-time or load-time).

2. The application will need to access both the broadcast and the lab distribution lists, but the contents of these lists should be constant during any single execution of the application. Hence the context associating specific (and subsequently immutable) objects with those names should be formed at the time the application is loaded into the run-time system.

3. Finally, our developer is satisfied with the current versions of the message-printing formats named plain and port contained in the ABC collection, and the

[12]For example, linkers in traditional programming environments typically create copies of object code modules from libraries and make links to the copies, rather than to the library modules themselves, when constructing an executable program.

formats named filo and land contained in the XYZ collection. So, it would be appropriate to form a context associating those names with copies of those specific objects at compile-time. This means that, once compiled, the application will be shielded from any changes to the objects through the persistent store (e.g., name changes, message format changes).

One goal, then, for a context control mechanism is to give developers and users explicit control over context specification for persistent objects [4]. Another important goal is to give subsequent developers or maintainers access to this information in a uniform manner. In other words, instead of having the tedious and error-prone task of manually analyzing source code and/or configuration scripts, developers and maintainers should be able to query an object directly for its relevant context formation information. Similarly, both humans and automated tools should be able to reason about this context information, thus facilitating the detection of potential name management-related inconsistencies and errors.

Most existing POSs support few, if any, of these capabilities. In the next two sections, we outline a prototype realization of the extended PICCOLO framework with the potential to support all these capabilities and more in a uniform and powerful way.

6.4.2 Conch Specifics

The idea of a "shell" acting as an intermediary between applications and the underlying persistent store is by no means a new one. In traditional operating systems, such as UNIX, a variety of shells have been implemented to provide an interface between programs and the underlying (e.g., UNIX) file system. Similarly, relational database systems often provide an interactive SQL that allows users to interact with the database. In recent years, graphical browsers for POSs have begun to emerge [36, 37]. Neither previously existing shells nor graphical browsers,

however, provide sufficiently powerful and general context control capabilities for use with convergent computing systems like POSs. In this section, we report on CONCH, a prototype context controlling shell for POSs that we have implemented as a user interface for Open OODB. We then illustrate how CONCH can be applied to the scenario presented in Section 6.4.1.

CONCH is a realization of the extended features of the PICCOLO framework. It facilitates experimentation with that framework as well as with enhanced context control capabilities for POSs. More specifically, the present version of the CONCH prototype represents an experiment with a particular method of providing a particular set of name management capabilities and an attempt to unify these capabilities from a "shell" perspective.

The CONCH prototype addresses many of the shortcomings of, and has resulted in improved support for, name management in Open OODB. The current set of commands defined by the CONCH interface is listed in Table 6.1. (Since Open OODB ordinarily runs under UNIX, the CONCH command names and syntax are intentionally UNIX-like.) First, the commands lbs, abs, cbs, rtbs, rmb, and bind permit developers and users to easily traverse, modify and query the Open OODB persistent store. The

Table 6.1 Conch Commands

Command	Description
lbs	List contents of the *active* binding space
abs <name>	Binding space with name <name> becomes *active* binding space
cbs <name>	Create a new binding space
rtbs	ROOT binding space becomes active
rmb <name>	Remove binding for name <name>
bind <name1> <name2>	Bind name <name2> to object bound to name <name1>
cftr <name>	Create a CFTR with name <name>
cftp <name>	Create a CFTP with name <name>
cftq <name>	Query the CFT with name <name>
inclos <cft-name> <obj-name>	Insert CFT into object's closure
prclos <name>	Print contents of object's closure

lbs command also allows users to view the language and type of any object. Similarly to other shell-style approaches, CONCH supports the notion of an *active* (or current) binding space. The only way to create new bindings from the shell is through use of the bind command. Note that the use of UORs in bindings means that CONCH does not have to link the object code for all the different types represented in the persistent store. That is, new objects of new types can be inserted into the store and CONCH can manipulate bindings to such object without having to be re-compiled or re-linked.

Second, CONCH allows developers to precisely and explicitly express context formation requirements for applications accessing the persistent store by providing specific kinds of CFTRs, CFTPs, and CFPs. The commands cftr, cftp and cftq allow users to create and query CFTs. More specifically, a CFTR in CONCH consists of one or more *request* clauses, where a request clause consists of the following fields:

Names: the names whose corresponding bindings should be contained in the context.

Bind: an indication of whether the bindings should be to existing objects (REF) or to copies of objects (COPY).

Epoch: when the bindings in the context should be formed. (CONCH supports the epochs {COMPILE, LOAD, RUN}.)

Sources: the source binding spaces for the desired bindings.

Similarly to a CFTR, a CFTP in CONCH consists of one or more *provide* clauses, where a provide clause consists of analogous fields. Finally, CONCH defines a rudimentary CFP for forming contexts. In the current prototype, this CFP is automatically included in the closure for all objects, thus ensuring that all objects utilize the same method for context formation. The pseudo-code for the CFP employed by CONCH is shown in Figure 6.15.

```
Context CFP (Context currentContext,
             CFTR c,
             Epoch current)
begin
 Context newContext = currentContext
 foreach clause in c do
   if current = clause.Epoch then
     foreach name in clause.Names do
       object = find (name) in
             clause.Sources
       if not found then
         raise ContextError
       if clause.Value = copied then
         newContext.Insert (name, object.copy)
       else
         newContext.Insert (name, object)

 return new
end CFP
```

Figure 6.15 Default CFP

This simple CFP creates a new context by augmenting the contents of a given context with the bindings as directed by a CFTR and the current epoch. It also checks to ensure that the resulting context is valid and provides warning or error information in the event the checking should fail. Note that the CFP is not invoked by an explicit command in CONCH. Instead, in the prototype the CFP is invoked by various system-level tools, such as the compiler, linker and run-time system.

6.4.3 Illustrating Conch

To illustrate the use of CONCH, we return to the scenario outlined in Section 6.4.1, in which a developer is faced with the problem of specifying and constructing a context that associates appropriate objects in the persistent store with names used in a mail system application. The developer's task is complicated by the fact that the different bindings in the resulting context must satisfy different requirements. With CONCH, one way to solve the problem in the scenario is to use the cftr and inclos commands

to create the CFTR shown in Figure 6.16 and form a closure associating that CFTR and the application code.

```
request plain, port
copy
compile
sources Root.Formats.ABC;

request filo, land
copy
compile
sources Root.Formats.XYZ;

request broadcast, lab
copy
load
sources Root.Dists;

request mailbox
ref
run
sources *active*;
```

Figure 6.16 Example CFTR

The CFTR directs a context to be created as follows:

- At compile-time, the context must contain the names plain, port, filo and land, each bound to a copy of the appropriate persistent object. The preferred source for the objects named plain and port is the binding space named ABC, while the preferred source for the objects named filo and land is the binding space named XYZ.

- When the program is first loaded into the run-time system, the context must additionally contain the names broadcast and lab bound to copies of the appropriate persistent objects, whose preferred source is the Dists binding space.

- Each time the mailbox object is accessed, the context should be updated to contain a binding pairing the name **mailbox** with the then-current object associated with the name **mailbox** in the *active* binding space.

Once this CFTR has been created and inserted into the applications' closure, the next step is to compile, link and run the application. The entire process is depicted in Figure 6.17, where the arrows denote and identify epoch boundaries. The starting point is a closure consisting of the application source code, an (initially) empty context, the CFP from Figure 6.15, and the CFTR from Figure 6.16. Invocation of the compiler initiates the **compile** epoch. The CFP, CFTR and context are retrieved from the source code's closure and a new closure consisting of a binary executable, a (partially formed) context, the CFP and the CFTR is produced. Then, using the **cbs** command, the user sets the *active* binding space to be **Jack**.[13] In Figure 6.17, the shaded ellipse denotes the *active* binding space. Next the program is loaded into the run-time system. This signals initiation of the **load** epoch, so the CFTR directs the context to be augmented appropriately and a correspondingly updated closure is formed. Finally, execution of the application begins, initiating the **run** epoch, and the mailbox object is accessed. As directed by the CFTR, the context is now augmented with a binding pairing the name **mailbox** with the object associated with the name **mailbox** in the *active* binding space, i.e., the object named **mailbox** in the binding space named **Jack**.

Note that the closure mechanism in CONCH associates a CFP and a CFTR as well as a context with each object. Although not illustrated in this simple example, maintaining this information allows application developers and maintainers to easily determine the context formation requirements for an application using the **prclos** command.

[13]In fact, this step could be taken either earlier or later than this, so long as it has occurred before the beginning of the **run** epoch.

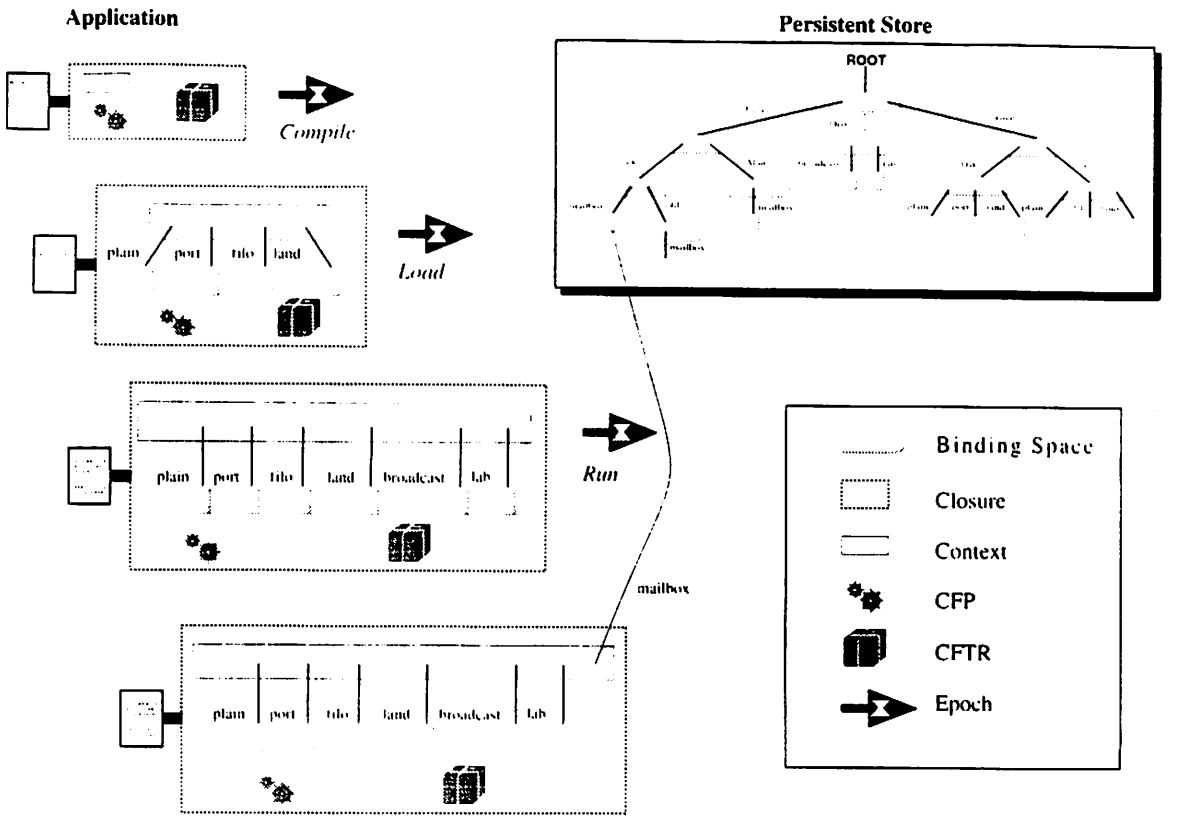


Figure 6.17 Context Formation Process

6.4.4 Evaluating Conch

In this section we have described CONCH, a prototype context-controlling shell implemented as a user interface to Open OODB. Initial experimentation with CONCH appears to confirm our expectation that a context control mechanism based on PICCOLO would provide more general, flexible and powerful context control than that available in existing POSs. In particular, the approach facilitates the sharing of context information among objects, while at the same time permitting individual objects to define their own context formation requirements. Furthermore, since context data is never discarded, application developers and maintainers can access this valuable information in a uniform and efficient manner. We therefore believe that the approach embodied in CONCH and PICCOLO can contribute to making POSs easier to use, less prone to error and better suited for use in a wide range of applications.

Since the existing context control features in persistent[X] POSs (such as the Open OODB) are so limited, CONCH's superiority over them is quite clear and perhaps not surprising. It our belief, however, that existing approaches to context control in *de novo* POSs are also not as powerful and general as the approach described in this section. While Napier [35, 94], for example, certainly allows for flexible organizations of persistent stores, developers must describe specific persistent store navigations in each application [3]. Moreover, Napier does not provide adequate means for querying objects regarding their contextual formation information. As shown in Chapter 4, this can be problematic in Napier programs that create bindings in closures local to a procedure, since there is no means for determining whether other Napier programs may be able to access the objects in those bindings. The approach taken by Farkas et al. [42] ameliorates these problems to some degree, but it is unclear how well a graphical browsing paradigm is suited for managing contexts in large and complex applications. We view our work on the PICCOLO framework and the CONCH shell as an approach that could complement these and other approaches in existing and future POSs. Indeed, in the future we hope to explore the incorporation of CONCH-like capabilities into a *de novo* POS such as Napier in order to further assess the generality and the efficacy of both the PICCOLO framework and the CONCH constructs.

In summary, even the limited evaluation reported here clearly demonstrates that CONCH has several advantages in terms of greater capabilities, flexibility and expressiveness when compared to existing context control features in POSs. Our experimentation with CONCH has, however, also pointed up a number of possible improvements and extensions that would make the approach even more beneficial. We return these issues in Chapter 8.

6.5 Summary

In this chapter we have outlined our overall approach for designing, developing and experimenting with extended name management capabilities for a particular class of convergent computing systems, namely OODBs. Using the PICCOLO framework as a guide, we extended Open OODB's name management services with more flexible and general capabilities. This collection of capabilities represents a substantial improvement in the area of name management in this domain of convergent computing systems. Although there is some performance overhead in our initial prototype, we believe that the capabilities provided justify the additional costs incurred. Furthermore, we have identified several ways in which these extra costs can be reduced. We have also demonstrated the benefits of a context control mechanism based on the extended features of PICCOLO. Specifically, the CONCH shell offers developers the possibility of specifying context at a fine level of granularity. In addition, developers can actually obtain important context-related information associated with programs. In the next chapter, we show how our enhanced name management mechanisms can lead to improvements in other capabilities in convergent computing systems.

CHAPTER 7

ENABLING POLYLINGUAL APPLICATIONS

Using our models of name management, we have designed and developed an improved approach to name management in a representative instance of a convergent computing system. The name management mechanism we integrated into the Open OODB represents a significant improvement over capabilities found in most modern OODBs. Our mechanism not only supports enhanced organizational and context-control features for persistent objects, but its approach to language uniformity represents an important step toward eliminating language barriers typically found in OODBs supporting multiple languages or APIs.

This latter point leads to the next topic in this dissertation, that is, demonstrating how improved name management mechanisms can help enable additional capabilities in convergent computing systems. As noted previously, a given OODB may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages, and an application may need to uniformly process those data objects. We call such an OODB *polylingual* and term the corresponding interoperability problem the *polylingual access* problem.

In this chapter we focus on the polylingual access problem for object-oriented databases and demonstrate how our enhanced name management mechanism leads to some novel approaches to overcoming this problem. We begin in Section 7.1 by outlining our general perspective on the interoperability problem. In Section 7.2, we discuss the polylingual access problem in OODBs, introducing an example that illustrates various interoperability and heterogeneity issues, identifying some important

facets of OODB interoperability problems, particularly the polylingual access problem, and assessing current approaches to OODB interoperability. In Section 7.3, we describe PolySPIN, an approach integrating persistence, interoperability and naming for polylingual object-oriented databases, and its current realization as extensions to the Open OODB. In addition, we show how PolySPIN can facilitate aspects of interoperability in polylingual object-oriented databases, returning to the example of Section 7.2 to illustrate PolySPIN's capabilities. Although PolySPIN provides an enabling substrate for interoperability in this class of systems, its direct use demands far too much involvement in low level details to be appealing to most software developers. Hence, in Section 7.4 we describe a prototype realization of a toolset providing automated support for applying PolySPIN. Section 7.5 then evaluates our approach, in particular comparing the PolySPIN approach to existing state-of-the art approaches. The chapter concludes with a summary and an assessment of our results to date.

7.1 Interoperability: Problems and Approaches

Over the years, as information systems applications have grown larger and more complex, various kinds of *heterogeneity* have appeared in those applications. As a result, individuals and organizations involved in developing, operating or maintaining such applications have increasingly been faced with *interoperability problems* – situations in which components that were implemented using different underlying models or languages must be combined into a single unified application. To aid in overcoming such problems, a range of *interoperability approaches* have been employed. As interoperability problems evolve, due in part to evolution of the underlying models and languages used in information systems applications, interoperability approaches must also evolve.

In applications developed using traditional database technology, there have been two primary sources of heterogeneity. One of these is the need or desire to code different components of an application in different programming languages. The other is the need or desire to make use of two or more different databases in a single application. These have given rise to two corresponding classes of interoperability problems, which we refer to as the *multilingual access* problem and the *multiple database integration* problem.

One of the important extensions to database technology that has appeared during the last decade has been the introduction of object-oriented databases (OODBs). By virtually eliminating impedance mismatch, object-oriented database technology can be viewed as a significant evolution of the underlying models and languages used in information systems applications and hence has many ramifications. Among these are new possibilities for heterogeneity and concomitant new interoperability problems, which necessitate the evolutionary development of new interoperability approaches. As illustrated in Chapter 6, a given object-oriented database may contain data objects originally defined, created and persistently stored using the capabilities provided by several distinct programming languages. We call such a database *polylingual*. This novel kind of heterogeneity induces new interoperability problems, such as the possibility that an application may need to uniformly process (including both queries on and updates to) the data objects in a polylingual OODB. We term this interoperability problem the *polylingual access* problem. Existing interoperability approaches provide little or no support for polylingual access, so new approaches must evolve to provide such support.

While many of today's commercial OODBs support multiple programming language interfaces (e.g., ObjectStore [76], GemStone [22]), none provide transparent polylingual access to persistent data. Instead, present day interoperability mechanisms generally rely on external data definition languages (such as ODMG's ODL [28]

or CORBA's IDL [100]), thus reintroducing impedance mismatch and forcing developers to anticipate heterogeneity in the applications, or depend upon direct use of such low-level constructs as the foreign language interface mechanisms provided in individual programming languages. In addition, many current approaches require that all the data in a polylingual database be stored using a single common representation, and thus force a substantial amount of data translation to precede their use. Others, while avoiding data translation through the use of so-called "wrapper" techniques, often support only a subset of the manipulations that would be available if the data were accessed from its native language.

7.2 The Polylingual Access Problem

As noted above, one of the interoperability problems found in applications based on traditional database technology is the multilingual access problem. This problem arises when two or more pieces of application software, written in two or more different languages, need to access the same database, as shown in Figure 7.1(a). Because traditional database technology has imposed a strong separation between the languages used for defining and manipulating information in a database and the languages in which applications are coded, application developers using traditional databases have been accustomed to dealing with two different type systems – the native type system of the language in which the application is written and the type system embodied in the DDL associated with the database - and two different sets of data manipulation constructs – those of the programming language and the DML, respectively. This dichotomous treatment of database-resident (persistent) data and program-resident (transient) data has been advantageous with respect to interoperability approaches aimed at addressing the multilingual access problem. In particular, it has meant that simply by defining language-specific bindings between each application programming language and the constructs of the database system's DDL and DML, an interop-

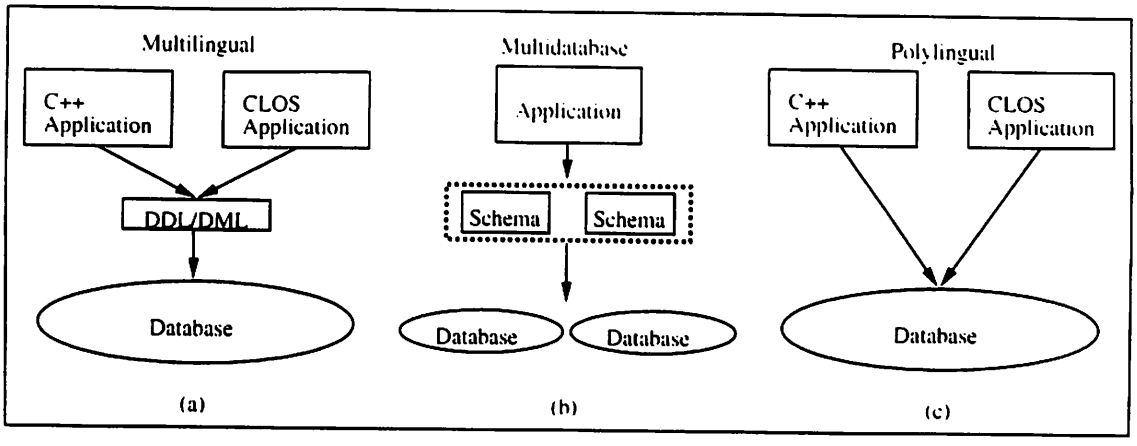


Figure 7.1 Interoperability Problems in Databases

erability approach that fit smoothly with the programming style already familiar to application developers could be achieved.

The other interoperability problem found in applications based on traditional database technology is the multiple database (or schema) integration problem, as shown in Figure 7.1(b). This problem arises when two or more different databases are needed by the same application. Approaches to addressing this interoperability problem have primarily involved creating a unified perspective on the multiple databases through some means of reconciling differences in the schemas, or in some cases the internal representations, used in the different databases. Because the multiple database integration problem, in its most general form, requires determination of semantic equivalence among different data type definitions, no single satisfactory solution to this interoperability problem exists. Nevertheless, a number of useful interoperability approaches have been proposed [19], such as various kinds of multidatabases [81] or federated databases [57]. Related work that addresses essentially the same set of concerns from a perspective influenced by the AI view of information (or knowledge) and its processing has proposed such notions as mediators [146] or enabling technology for knowledge sharing [97] to address the schema, model and

internal-representation reconciliation facets of the multiple database integration problem.

With the advent of OODBs (which have largely removed the dichotomous treatment of persistent and transient data), the traditional interoperability problems multilingual access and multiple database integration – remain, but become somewhat more complex. Moreover, object-oriented databases also introduce additional kinds of heterogeneity and concomitant new interoperability problems. In particular, for applications developed using traditional database technology, multilingual access to a single database involves accessing objects that might have been produced by multiple programs written in multiple languages, but which in fact are all defined in terms of a common type model (DDL) and can be manipulated by a common set of operations (DML). With OODBs, however, it is possible to have multiple application programming interfaces (APIs) to the same database, through which objects created using different programming languages (and their respective type models, rather than a common DDL) may all be stored in the same database. As noted earlier, the TI/Arpa Open Object-Oriented Database (Open OODB) for instance, provides both a C++ and a Common Lisp Object System (CLOS) API. Moreover, type definitions in an object-oriented model include a set of operations for manipulating instances of the types, so the OODB may well contain operations that are defined and implemented in different programming languages. We use the term *polylingual database* to connote this new kind of heterogeneity, in which a single database may contain objects that differ not only in the language used to define their data representations but also in the language used to implement the operations provided by their interface definitions.

Polylingual object-oriented databases, as shown in Figure 7.1(c), give rise to new interoperability problems. Specifically, in such a database it is possible that even

though two objects are of the same type, or equivalent or compatible types,[1] their implementations might be based on different programming languages, posing a serious interoperability problem for an application that wishes to access both of them. Ideally, the differences in their underlying implementation languages should be invisible to the application, which should be able to treat the objects uniformly. In type systems research, a similar goal of making differences in data types transparent to code that processes instances of the types is referred to as polymorphism (e.g., [24]). By analogy, we term this interoperability problem the *polylingual access* problem.

The remainder of this section is organized as follows. Section 7.2.1 presents a hypothetical interoperability scenario illustrating the polylingual access problem. Section 7.2.2 discusses some general interoperability issues arising in OODBs. Section 7.2.3 concludes with a discussion of existing approaches and how these approaches address the issues presented in the preceding section.

7.2.1 An Example

As a simple illustration of heterogeneity and interoperability problems in object-oriented databases, consider the following example:

> At Hypothetical University, two colleges have independently developed information systems applications, using object-oriented database technology, for managing personnel information regarding their students and faculty. Although both colleges have in fact utilized the same OODB, the Arts College has built their application on a CLOS API while the Sciences College has built theirs on a C++ API. Figure 7.2 shows a portion of the C++ schema used by the Sciences College, a portion of the CLOS schema used by the Arts College, and the OODB containing instances of the

[1] Although techniques for determining type equivalence or compatibility are a challenging research topic in their own right (e.g., [150, 157, 158]), for purposes of this dissertation we presume *such* determinations can be made, in at least some interesting cases. We discuss this issue further in Section 7.6.

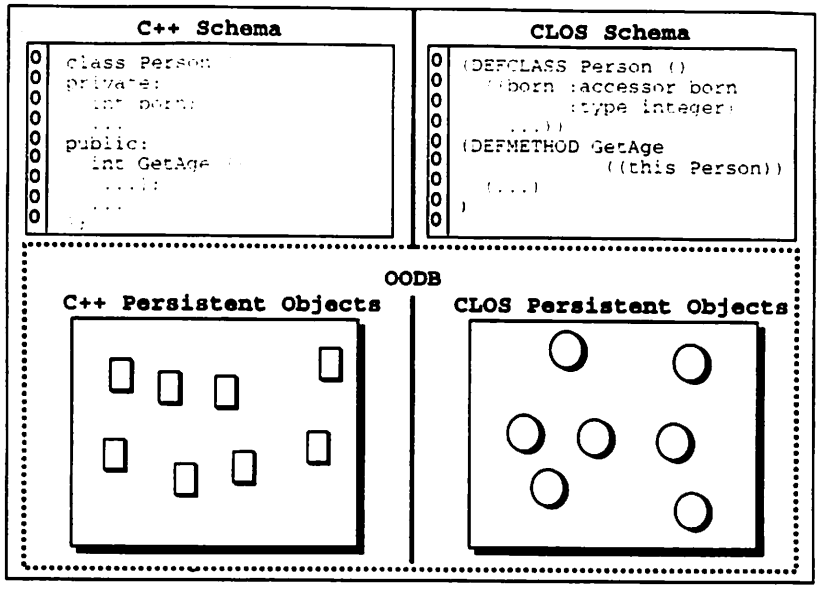


Figure 7.2 OODB Used By Hypothetical University

personnel data object from both colleges implemented in their respective languages.

The central administration at Hypo U would like to develop some applications making use of personnel information from both colleges. Naturally, they cannot hope to convince either college to translate its personnel information to a representation corresponding to the other's API. Nor can they expect to convince other colleges, when they develop their own personnel information systems in the future, not to use the API of their choosing (e.g., Ada 95 for the Engineering College, Object-Oriented COBOL for the Business College, etc.). Hence the administrators would like their application to be able to be oblivious to the implementation languages of individual persistent objects. They would also like to be able to employ either navigational access or associative access in processing the personnel information from the various colleges. An example of an OQL-style query (based on [138]) that might be part of a C++ application, in

this case seeking candidates for early retirement incentives, is shown in Figure 7.3. Note that the query should be able to be applied to *all* the personnel data resident the OODB, i.e., independent of the language used to create the persistent objects.

```
setPerson = ...; // constains Person objects
Set<Person*> result = NULL;
Query {
  result = SELECT *
         FROM Person* matched IN setPerson
         WHERE matched->GetAge() > 45;
}
```

Figure 7.3 OQL-style Query

Despite the fact that the two college's personnel information schemas are clearly equivalent, existing OODBs, even those such as the TI/Arpa Open OODB that provide multiple APIs, do not support the kind of polylingual access desired by the Hypo U administrators. Several aspects of current OODB technology stand in the way of polylingual access. We first briefly discuss interoperability goals and issues in general, then specifically consider the approaches to polylingual interoperability that are necessitated by existing OODB interoperability support. Later sections then indicate how these goals and issues are addressed in our POLYSPIN approach.

7.2.2 Interoperability Goals and Issues

Our work on interoperability is motivated by a primary concern for the impact of an interoperability approach on applications developers. As characterized in [150], among the most important objectives for any approach to interoperability are the following:

- Developers should have maximum freedom to define types of objects that their programs manipulate. In particular, they should always be able to use the type

systems provided by the language(s) in which they are designing and developing components of their application.

- Whether a data object is to be shared among an application's components should have minimal impact on the components' developers. In particular, making (or changing) a decision about whether, or with what other components, a data object may be shared should not affect the definition of, or interface to, the object.

Given these objectives, we have identified three sets of issues regarding interoperability in OODB-based applications. Briefly, these are:

Naming How are objects in the persistent store accessed by applications that wish to interoperate through sharing those objects? Current OODBs typically rely on distinct and often incompatible name management mechanisms for each of the programming languages or application programming interfaces (APIs) they support. This results in disjoint persistent stores segregated according to the language used to define the persistent objects and also leads to inconsistent semantics for the name management capabilities provided by the various language interfaces.

Timing When is the decision to share data objects among an application's components made? This question has a dramatic impact on the suitability of different approaches to interoperability. Three distinct timing scenarios for interoperability decisions can be characterized by the relationship between the relative times at which the sharing or shared components are developed and the decision to share them is made, as illustrated in Figure 7.4. The salient features of each scenario are:

Easiest case: The decision to share is made before any components are developed. In this case, a common (e.g., IDL) description of the shared

data objects can be created prior to development of the components that will share them, language-specific descriptions can be directly created by mapping from the common description, and hence determination of type compatibility is trivial.

Common case: The decision to share is made after one of the sharing components is developed but before any others are. In this case, a common (e.g., IDL) description of the shared data objects can be created by mapping from the language-specific description whose existence predates the sharing decision and then the remaining language-specific descriptions can be directly created by mapping from the common description, so determination of type compatibility is again trivial.

Megaprogramming: The decision to share is made after the sharing components are developed.[2] In this case, common (e.g., IDL) descriptions of the shared data objects can be created by mapping from each of the language-specific descriptions, but determination of type compatibility will then depend upon some kind of comparison of these synthesized descriptions and hence is nontrivial.

Typing How do developers determine whether the types of objects that they wish to share are of compatible types? As noted earlier, this question reduces to the schema integration problem for traditional database technology. For object-oriented database technology, most approaches to addressing this question have been based on use of a unifying type model [150], such as the ODMG ODL. While such approaches may suffice for the easiest and common interoperability scenarios, however, they are inadequate for the megaprogramming case. Since, as our example scenario suggests, that case is perhaps the most important and

[2]The concept of *megaprogramming* is discussed in [16, 147].

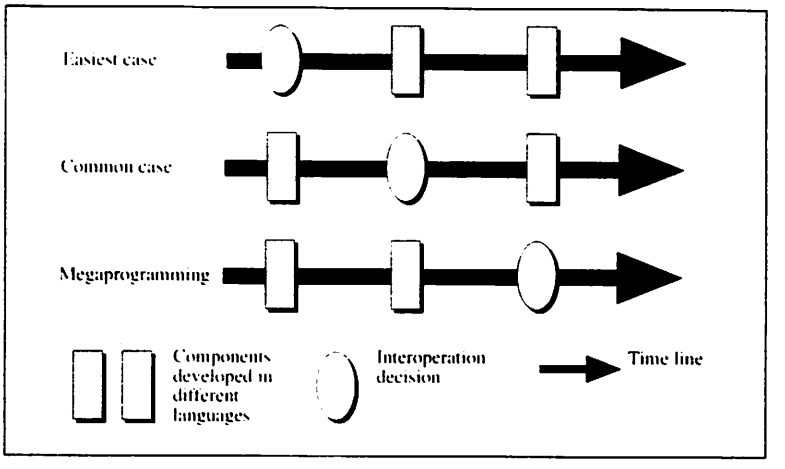


Figure 7.4 Interoperability Scenarios

offers the greatest potential rewards, we have focused our research efforts on attempting to handle it.

7.2.3 Existing Approaches

Having discussed interoperability issues in general and framed an example scenario with which to illustrate them, we now consider how existing approaches to supporting interoperability in OODB-based applications might address those issues, specifically by examining how an ODMG-based approach might be applied to the Hypo U situation.

The Object Data Management Group's approach to supporting multilingual applications is based on the use of a (more or less) language-independent unifying type model, expressed in the Object Definition Language (ODL). ODMG defines bindings between ODL and various programming languages, notably C++ and Smalltalk. The expectation is that, in concert with these bindings, ODL will provide a basis for interoperability in multilingual OODB applications.

Let us look in some detail at how this would apply to the Hypo U example. As mentioned in our discussion of timing issues related to interoperability, in the easiest

and common cases an approach utilizing a unifying type model could work reasonably well. The Hypo U situation, however, is a paradigmatic example of the megaprogramming case, since the type definitions for the objects that are to be used by the new application have already been formulated in both C++ and CLOS and persistent data instances implemented in both languages have also already been created and stored. How can an ODMG-style approach be made to apply in such an instance?

One possibility would be to proceed to formulate an ODL type definition for the Person type, then attempt to determine whether the ODL definition was consistent with both the C++ and CLOS type definitions for Persons. While automated support for transforming ODL definitions into corresponding type definitions in various programming languages exists, we are unaware of any automated support for going in the opposite direction. Even if such support existed, however, it would leave the (in general undecidable) problem of determining whether the *two* resulting ODL definitions corresponding to the original C++ and CLOS definitions of Person were, indeed, equivalent. Thus, as we noted earlier, the ODMG-style approach offers little assistance with the type compatibility aspect of interoperability problems in the megaprogramming case.

Assuming for the moment, however, that the type compatibility problem could be handled, or ignored, how would the ODMG-style approach be applied in the case of Hypo U? Unless the developers were so incredibly lucky as to have the C++ and CLOS interface definitions for Person that corresponded to the ODL definition be *identical* to those that had originally been defined and used by the respective college's programmers, the new application would not be able to directly access instances created and stored using those definitions. This would necessitate one of the following:

- Use of "wrapper" code at every point where existing instances were accessed, serving to bridge, or paper over, the discrepancies between the ODL-derived and the original interfaces. This approach essentially negates the usefulness of

the unifying model and forces the application to explicitly distinguish between instances created using C++ and CLOS. This clearly falls far short of neatly supporting polylingual application development.

- Transformation of all existing data object instances, either into forms that matched the ODL-derived definitions for their respective languages or into some common (e.g., all C++ or all CLOS) form. While this might solve the problem momentarily, it would not address the possibility that the existing applications might need to be used again in the future, forcing additional transformations on the instances that they might create and store during those future executions. The alternative of modifying the existing applications so that they would create objects according to the ODL-derived definitions might (or might not) be possible, but would likely be error-prone, expensive and in violation of our goal of minimizing the impact of interoperability on the developers of shared, or sharing, components. Hence, use of transformations also fails to make an ODMG-style approach to interoperability acceptable for polylingual programming situations.

Finally, it is worth noting that existing approaches generally require that developers make direct use of mechanisms for multilingual programming, such as foreign-function call-out capabilities, thus forcing explicit recognition of language boundaries and further falling short of supporting polylingual application development. For example, C++ programming environments often permit C++ functions to be wrapped in C-style declarations, thus allowing for C++ functions to be invoked by other C programs (or programs written in other languages whose foreign function interface is based on C). [3]

[3] Wrapping functions in C-style declarations is not absolutely required for interfacing C++ with other languages. It does, however, ease the development of interoperating applications since the implicit first argument in C++ class methods, along with the fact that C++ compilers typically "mangle" the names of functions, results in non-intuitive operation names and signatures.

Most of the shortcomings in existing approaches to interoperability result from inadequate levels of abstraction. By basing higher level abstractions for OODB data object access on a general and uniform approach to name management, PolySPIN provides a suitable framework for supporting polylingual access. In the remainder of this chapter, we outline this framework and demonstrate its value in overcoming the polylingual access problem.

7.3 The PolySPIN Architecture

PolySPIN is a generic, object-oriented architecture that unifies persistence and interoperability capabilities in OODBs from a name management-based perspective. As an outgrowth of our work on enhanced name management mechanisms, PolySPIN provides a uniform name management mechanism that not only offers application developers a library of useful abstractions for organizing and navigating object-oriented databases but, as a byproduct, offers an interoperability mechanism providing (language-transparent) polylingual access to persistent objects, thus allowing applications to manipulate objects as though they were all implemented in the language of the application.

In this section, we begin by briefly describing PolySPIN's approach to name management and persistence, which is more or less based on the name-based persistence mechanism introduced in Chapter 6. We then provide an overview of name management and interoperability in PolySPIN. The section concludes with a discussion of the internal features of PolySPIN. Throughout this section, we will refer to the scenario presented in Section 7.2.1 as a means of explicating various aspects of PolySPIN. We also note that the example described here has been implemented as extensions to Open OODB, using Sun C++ [134] and the Lucid Common Lisp Object System (CLOS) [83].

7.3.1 Name Management and Persistence in PolySPIN

The name management mechanism in PolySPIN is based on the name-based persistence mechanism developed in Chapter 6 where any object (including those in its transitive closure) bound to a name in a binding space reachable from a specially designated root binding space automatically persists. The approach is similar to that used in Napier [94], where *environments* correspond to binding spaces. The name management mechanism in PolySPIN is more general, however, since it supports objects defined in multiple languages.

Recall that to participate in this mechanism, an object's class definition must inherit from a common base class, designated the NameableObject class. By inheriting from this class, instances of the subclass can be, among other things, named and resolved using the operations supported by the various abstractions that make up the PolySPIN name management mechanism. For example, Figure 7.5 shows a (partial) C++ class definition for Person object, a code fragment showing how a name might be assigned to an instance of Person, and a portion of a persistent store organization based on this approach.

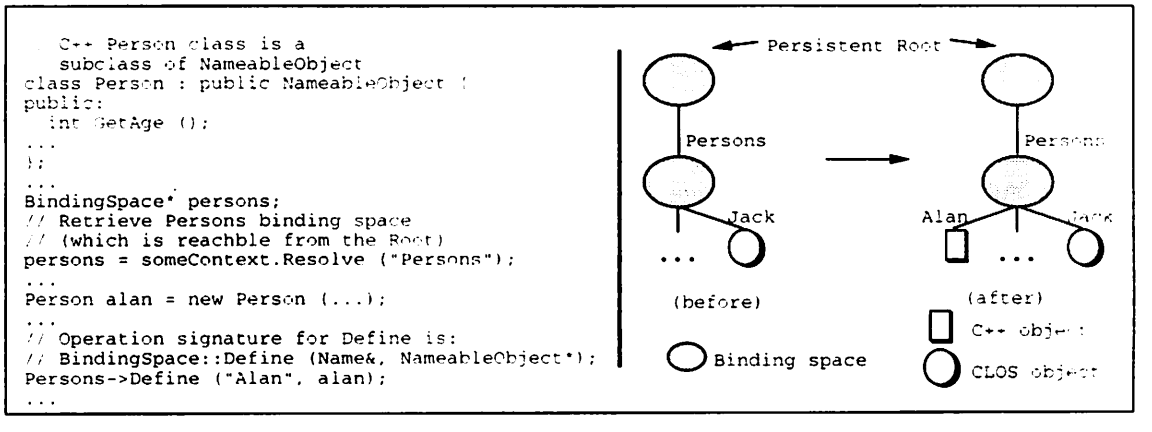


Figure 7.5 Using PolySPIN's Name Management Mechanism

7.3.2 Name Management and Interoperability

As suggested above, having a class inherit from NameableObject could, and frequently might, be done quite independently of any intention to make objects interoperate. Inheriting from NameableObject does, however, also enable the use of the interoperability capabilities of PolySPIN. First, having a uniform name management mechanism in place results in a language-independent method of establishing visibility paths to persistent objects (i.e., via their assigned names), regardless of the defining language of either the objects or the applications. Second, the name management mechanism serves as a useful place for capturing and recording language-specific information about objects, which can be used to support polylingual access. In particular, once an application has established an initial connection to a persistent object (via its name), the name management mechanism can provide the necessary information permitting the application to create a data path to the object. In other words, when resolving a name of some object (on behalf of some application), the name management mechanism can detect the defining language of the object and set up the necessary communication medium for manipulating the object. The features supporting this capability are hidden from application developers within the internals of the PolySPIN architecture, which we discuss in greater detail in Section 7.3.3.

Given this interoperability mechanism, what is needed is the ability to determine whether two class interfaces defined in different languages can indeed interoperate, and in the event they can, to instrument their implementations (including generating any necessary foreign function interface code) such that the interoperability features of PolySPIN can be employed. For example, Figure 7.6 shows the C++ and CLOS class definitions and implementations for the Person class, where the plain face type represents the original source code and the boldface type represents the code enabling polylingual access. The details of the various operations calls will be explained shortly, but the general idea is that a C++ application continues to use its original class

interface, while the implementation of the class is modified so that it invokes the appropriate language. Initially, we carried out this modification manually. In other words, the code shown in bold face in Figure 7.6 was generated and inserted into the class definitions by hand. As a step toward automating our approach, however, we have developed the PolySPINner toolset, which we describe in Section 7.4.

```
class Person : public NameableObject {
private:
  int born;
public:
  int GetAge ();
};
// GetAge member function
int Person::GetAge () {
  if (this->language == CLOS)
    return
      (__Callout_CLOS_Person_GetAge(this->tidForObject));
  else {
    int result;
    result = 1995 - born;
    return (result);
  }
}
// Foreign Function Interface Section
// Callout CLOS Person GetAge
extern "C" int __Callout_CLOS_Person_GetAge (TID this);
// Callout from CLOS into C++
extern "C" int __Callout_CPP_Person_GetAge (TID self )
{
  Person* object = (Person *) TidToCid (self);
  return (object->GetAge());
}
```

C++ Person Class

```
(defclass Person (NameableObject)
  ((born :accessor born
         :type Date
         :initform "MM/DD/YY")
  )
)
;; GetAge method
(defmethod GetAge ((this Person))
  (declare (return-values integer))
  (cond ( (EQUAL (language this) CLOS)
          (- Today (born this))
        ( (EQUAL (language this) C++)
          (__Callout_CPP_Person_GetAge (tid this)))
  )
)
;; Foreign Function Interface Section
;; Callout C++ Person GetAge
(DEF-ALIEN-ROUTINE ("__Callout_CPP_Person_GetAge"
                    __POLYSPIN_CPP_Person_GetAge)
  int  (self TID )
)
;; Callout from C++ into CLOS
(DEF-FOREIGN-CALLABLE
                (__Callout_CLOS_Person_GetAge
               (:language :c) (:return-type int))
               ( ( this TID) )
  (GetAge  (tid-to-cid this))
)
```

CLOS Person Class

Figure 7.6 Interoperating C++ and CLOS Classes

7.3.3 The Internal Features of PolySPIN

The fact that objects themselves may be implemented in different languages is completely hidden within PolySPIN's name management mechanism. To support this level of transparency in applications, PolySPIN utilizes the following components:

- A three-level object identifier hierarchy.
- A common base class encapsulating language-specific information for transient objects.

- A universal object representation encapsulating language-specific information for persistent objects.

As we illustrate in the remainder of this section, these abstractions, together with their interactions with one another, form a suitable foundation for providing transparent polylingual access.

7.3.3.1 The Object Identifier Hierarchy

A common solution to the interoperability problem involves converting between data representation formats. For example, to achieve interoperability in the scenario described earlier, it might be possible to simply translate C++ Person objects into CLOS Person objects (and *vice versa*). Unfortunately, even when hidden from users and applications, such techniques can be prone to error and computationally expensive, especially for large and complex objects.

An alternative approach involves utilizing object references (or L-values or native pointers) for identifying objects. This solution has the obvious benefits in terms of efficiency and maintainability; one drawback, however, is that different programming languages use distinct and incompatible object reference mechanisms. For example, native references to objects in C++ can not be interchanged with references to CLOS objects (and *vice versa*). Instead, a distinct mechanism must be used in a CLOS application to identify a C++ object. Languages supporting garbage collection, such as CLOS, present further complications since the value of an object reference may change over the course of a computation. Although transparent to CLOS applications, garbage collection may cause subsequent accesses by a C++ application using a native CLOS object identifier to result in invalid or dangling references. The addition of persistence yields yet another identifier mechanism that must be managed despite the fact that persistent identifiers are generally hidden from applications. More specifically, when an object is designated as being persistent, a persistent iden-

tifier is created for the object. When the object is retrieved from the database, the persistent identifier is used to locate the object. A reference (i.e., an L-value or a native pointer) must then be created for the object so that the application can access and manipulate the object.

As a step toward relieving application developers from managing separate object identifier mechanisms or building special-purpose ones, PolySPIN maintains a three-level identifier hierarchy, as shown in Figure 7.7. The hierarchy, in order of increasing lifetime, consists of:

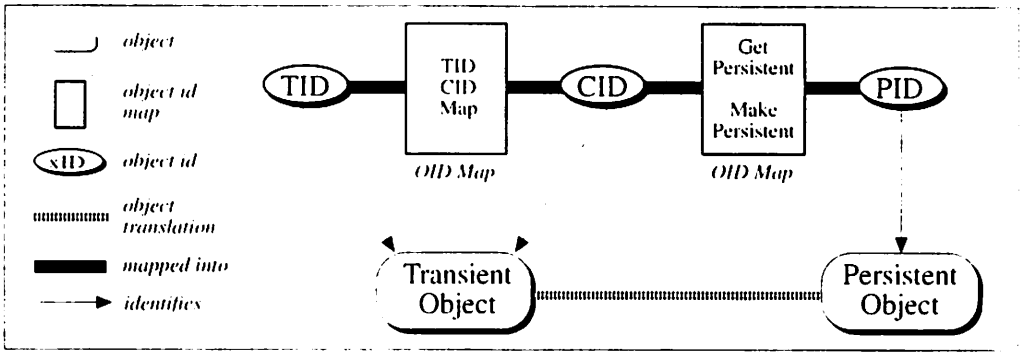


Figure 7.7 PolySPIN's Three Levels of Identifiers

- A *computation identifier* (or CID), which is an L-value or native pointer used by applications for identifying, accessing and manipulating objects defined in the same language. A CID for a particular object may change over the course of a computation, although such changes, for all intents and purposes, are invisible to programmers.

- A *transient identifier* (or TID), which is an active-computation-unique identifier for an object. Once assigned during some active computation, it is assumed that a TID does not change over the course of that computation's lifetime.

- A *persistent identifier* (or PID), which is a globally unique identifier for a persistent object. When a object is made persistent, a PID is assigned to the object. A PID is assumed to be immutable over the course of the object's lifetime.

For each language, PolySPIN also provides two-way TID↔CID and CID↔PID mapping mechanisms, where the former map can be implemented using a traditional hash table, and the latter map is often supplied by the default persistence mechanism provided by the OODB. PolySPIN's maintenance of these identifiers, along with functions for mapping between them, means that applications simply use CIDs to manipulate objects. Any required identifier translations are handled by PolySPIN. For example, when an application retrieves an object from the persistent store (i.e., via name resolution), a CID identifying the object is returned to the application. Similarly, when an object is made persistent, its CID is mapped into a corresponding PID. As we will show in the following sections, a CID always points to an object that, from the application's point of view, looks and behaves as if the object were defined in the same language as the application (although in reality the object may be implemented in a different language).

7.3.3.2 The NameableObject Class

As noted in Section 6.3.2, inheriting from the NameableObject class offers applications developers the ability to use PolySPIN's improved name management mechanism. At the same time, the NameableObject class encapsulates various language-specific information for an object including a defining language, a TID, and various type-related information. As shown in Figure 7.8, values for this information can be computed when an object (derived from NameableObject) is instantiated. In particular, the constructors for the C++ Person and NameableObject classes in Figure 7.8 illustrate how this information is computed and recorded. (The same information is computed in an analogous fashion for CLOS objects.)

```
// NameableObject class
class NameableObject {
private:
  LanguageId language;
  TID tidForObject;
  ClassId classInfo;
public:
// Constructor for NameableObject
  NameableObject() {
    language = C++;
    tidForObject = CidToTid (this);
    ...
}
// Constructor for Person class
Person::Person () {
// Implictly invokes NameableObject
// Then set type information
  classInfo = "Person";
  ...
}
```

Figure 7.8 The NameableObject Class

When an operation is invoked on an object, the data maintained by the NameableObject can be used to determine the actual implementation of an object. Since this is hidden from users, however, all instances of the class can be viewed and accessed through a single language interface, even though various instances may in fact be implemented in various languages. To help illustrate this technique, consider Figure 7.9, where the directed arcs indicate the sequence of events that occur when the GetAge operation for the C++ Person class is invoked. First, GetAge checks the value of the defining language for the object (1). If the object is implemented in C++, then the C++ implementation of the GetAge operation (2) is used. If, on the other hand, the object is implemented in CLOS, then the corresponding CLOS operation must be invoked (on the CLOS object). This involves making a call-out (3) to a CLOS function (in this example, using the foreign function interface mechanisms of C++ and CLOS, (4)) and passing the CLOS object's TID. On the CLOS side, the TID is first mapped into its CID value (5) and then the actual CLOS GetAge operation (6)

is invoked. Note that by storing an object's TID rather than its CID, the effects of garbage collection will be hidden from a polylingual application.

```
class Person : public NameableObject {
private:
  int born;
public:
  int GetAge ();
};
// GetAge member function
int Person::GetAge () {
  if (this->language == CLOS)
    return
      (__Callout_CLOS_Person_GetAge(this->tidForObject));
  else if (this->language == C++) {
    int result;
    result = 1995 - born;
    return (result);
  }
}
// Foreign Function Interface Section
// Callout CLOS Person GetAge
extern "C" int __Callout_CLOS_Person_GetAge (TID this);
// Callout from CLOS into C++
extern "C" int __Callout_CPP_Person_GetAge (TID self )
{
  Person* object = (Person *) TidToCid (self);
  return (object->GetAge());
}
```

```
(defclass Person (NameableObject)
  ((born :accessor born
         :type Date
         :initform "MM/DD/YY")
  )
)
;; GetAge method
(defmethod GetAge ((this Person))
  (declare (return-values integer))
  (cond ( (EQUAL (language this) CLOS)
          (- Today (born this))
        ( (EQUAL (language this) C++)
          (__Callout_CPP_Person_GetAge (tid this)))
  )
)
;; Foreign Function Interface Section
;; Callout C++ Person GetAge
(DEF-ALIEN-ROUTINE ("__Callout_CPP_Person_GetAge"
                    __POLYSPIN_CPP_Person_GetAge)
  int  (self TID )
)
;; Callout from C++ into CLOS
(DEF-FOREIGN-CALLABLE
                (__Callout_CLOS_Person_GetAge
                (:language :c) (:return-type int))
                ( ( this TID) )
  (GetAge  (tid-to-cid this))
)
```

Figure 7.9 Polylingual Access to CLOS from C++

7.3.3.3 The Universal Object Representation

PolySPIN unifies the persistent store by permitting the co-existence of objects implemented in different languages. Furthermore, access to the persistent store is provided by a name management mechanism that is uniformly available across multiple programming languages. To support polylingual access to persistent objects, PolySPIN introduces a level of indirection in bindings between names and (persistent) objects called a universal object representation or UOR. Recall that UORs were first introduced in Chapter 6 as a means of overcoming some limitations of transparent and reachability-based persistence mechanisms. In this section, we provide a more detailed description of UORs showing how they not only overcome these limitations but, as noted above, enable polylingual access in OODBs.

Like the NameableObject class, a UOR encapsulates various language-specific information about objects, including an object's PID and references to its specific

persistence routines. A UOR is created for an object when that object is assigned a name. In particular, the information stored by the NameableObject is transferred to the UOR. Consider Figure 7.10, which shows both a C++ and a CLOS application creating instances of the Person class. As described in Section 7.3.3.2, when the C++ object is instantiated (1), its NameableObject superclass is populated with language-specific information for the object (2). When the object is assigned the name Alan (3), a UOR is created(4). The UOR initially caches all the information from NameableObject. Finally, the binding of the name Alan and the UOR is inserted into the binding space (5). An analogous sequence of steps takes place for the CLOS object named Jack. As a result, the binding space contains bindings to objects defined in different languages.

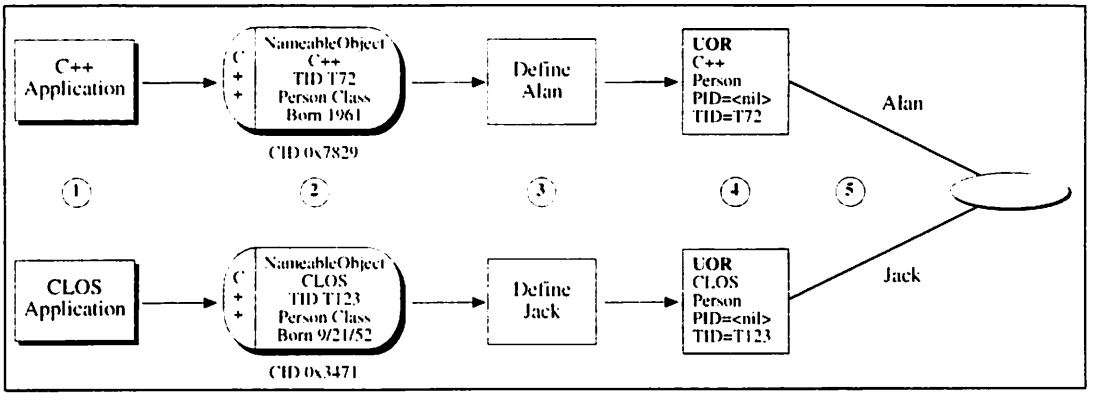


Figure 7.10 Representing Objects as UORs

If the object is later designated as being persistent (as described in Section 7.3.1), then a value for the object's PID is determined and is also set in the UOR. This is computed by invoking the appropriate persistence service (i.e., based on the actual language of the object) for the object represented in the UOR. The object's TID is mapped into its CID. The persistence service then maps the CID into a PID. Later, when the object is accessed (by resolving the name of the object), the name

management mechanism can use the information stored in the UOR to return an appropriate object to the application.

Using UORs, name resolution can be viewed as multi-step process. First, the UOR for a successfully resolved object is retrieved. Based on the information recorded in the UOR, the appropriate language persistence routine can be invoked. Once the actual object is retrieved from the persistent store, then, based on the language of the object and the calling application, the appropriate object can be returned to the application. For example, Figure 7.11 shows the various tasks PolySPIN performs on behalf of a C++ application accessing two instances of the Person class, where one object, bound to the name Alan, is implemented in C++ and the other, bound to

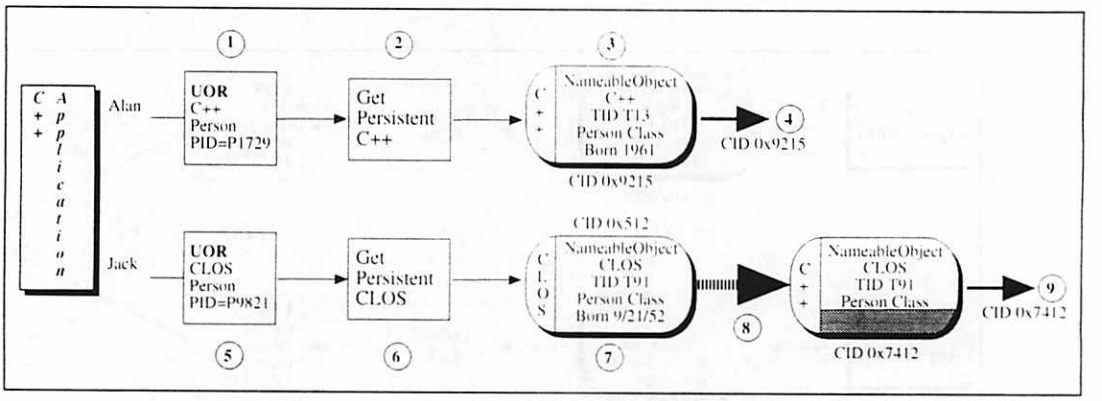


Figure 7.11 Accessing Objects Via UORs

the name Jack, is implemented in CLOS. When the object named Alan is accessed, PolySPIN examines the UOR for the object (1), determining the object's defining language and its PID. Based on the information, the object is retrieved from the store using the C++ persistence service (2) and a transient, C++ version is constructed (3). Since the languages of the application and the object are the same, the object (i.e., its CID) is simply returned to the application (4). When the object named Jack is accessed, PolySPIN again examines the UOR for the object (5), determining the object's defining language and its PID. Based on the information, the object

is retrieved from the store, this time using the CLOS persistence service (6), and a transient, CLOS version is constructed (7). Since the languages of the application and object are in this case different, PolySPIN creates a C++ Person object (8), which acts as a "surrogate" for its corresponding CLOS version. The surrogate's defining language is set to "CLOS" and its TID is set to the TID of the CLOS object. The rest of the data associated with the C++ Person surrogate is simply ignored (as indicated by the shaded portion of the surrogate object in the figure). Finally, the CID of the surrogate is returned to the application (9).

Subsequent accesses to both objects will (eventually) invoke the implementing object (as described above in Section 7.3.1). For example, the query shown in Figure 7.12 calls the GetAge operation for both objects. As shown in Figure 7.9, for the object named Alan the original C++ GetAge operation is called, while for the object named Jack the CLOS GetAge operation is invoked. Thus, the query is able to access and process both objects, despite the fact that one object is implemented in C++ and the other in CLOS.

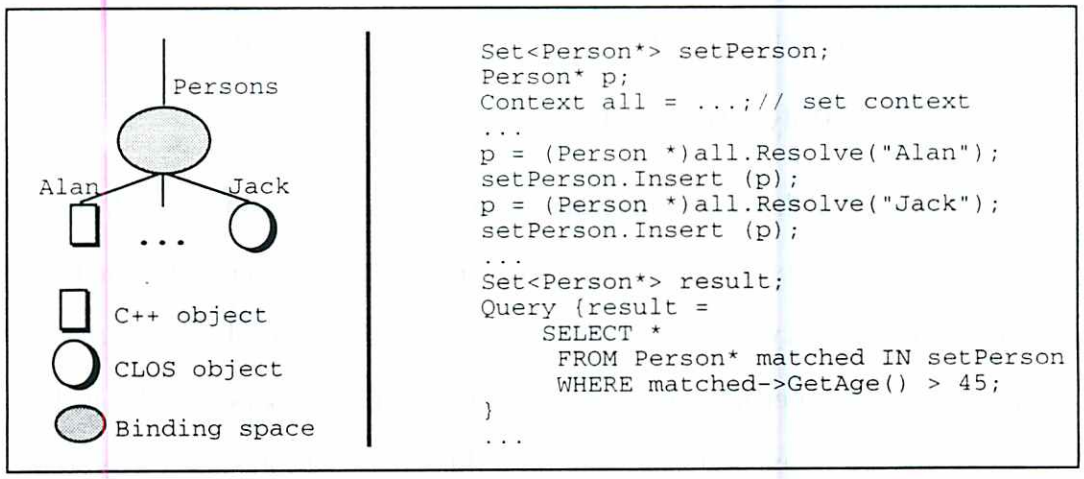


Figure 7.12 PolySPIN-based OODB

7.4 PolySPINner

Although PolySPIN hides many of the details associated with developing polylingual applications, its use initially required manual intervention in terms of modifying class implementations in order to utilize its capabilities. Hence we have begun to develop PolySPINner, which automates support for applying PolySPIN. The overall objective of PolySPINner is to provide transparent polylingual access to objects with minimal programmer intervention as well as minimal re-engineering of existing source code.

In this section, we describe an initial prototype of this toolset, which has been developed as an extension to Open OODB. The current prototype uses an *exact signature matching* rule [157] in determining the compatibility between C++ and CLOS classes. It also encapsulates the foreign function interface mechanism for both Sun C++ and Lucid CLOS, as well as the various internal features of PolySPIN. We first describe the details of the components that make up the prototype in Section 7.4.1. We then illustrate the toolset's application to our running example in Section 7.4.2, specifically showing how the code in Figure 7.6 is generated.

7.4.1 The PolySPINner Prototype

Figure 7.13 shows a conceptual organization of the toolset. The PolySPINner prototype consists of three major components: the language-specific parser tools, the class compatibility checker tool, and the PolySPINgen tool. To use the toolset, an application developer first invokes the language-specific parser tools on the source code for each class. Next, the class compatibility checker tool is used to assess the compatibility of the two classes. If the classes are deemed compatible, then the PolySPINgen tool can be used to generate code enabling polylingual access to instances of either class from applications written in either language.

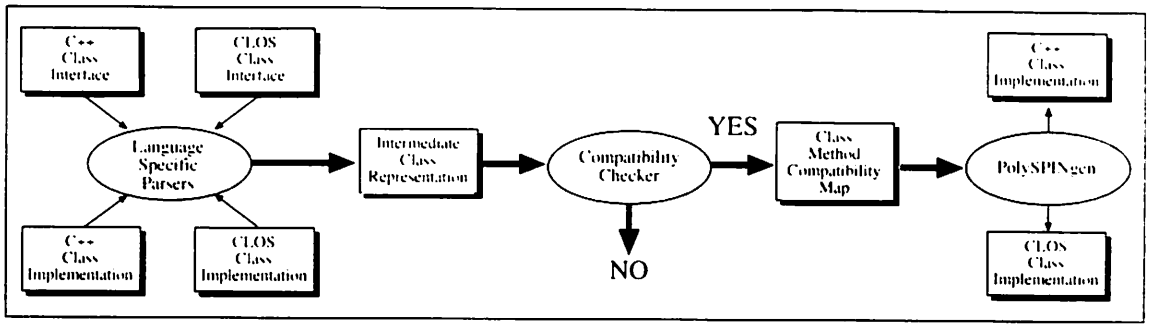


Figure 7.13 Conceptual Organization of PolySPINner

7.4.1.1 Language-Specific Parser Tools

POLYSPINNER provides parsers for both C++- and CLOS-defined classes, where each parser translates source code for a class (including both the class interface and implementation) into a common, intermediate class representation (or ICR). The ICR captures various class-specific features including the language and name of the class, along with the names and signatures for each method (or operation) defined on the class. The method signatures are represented by storing the name, type, and passing protocol for each formal parameter. In addition, since object-oriented languages such as C++ often use implicit object parameters, the ICR for a method includes a flag indicating whether or not the parameter is implicit in the signature. Finally, the parser tools generate *published names* for each method encountered in a class definition. These names will be used by other languages when the method is invoked externally, i.e., from another language's run-time system.

7.4.1.2 Compatibility Checker Tool

Based on the ICR, the Compatibility Checker assesses the compatibility of two classes. In the current prototype, the Compatibility Checker makes this assessment by performing the following matching rule:

Two classes are considered compatible if:

1. Each class has the same name.

2. The public interface of each class provides the same number of methods.

3. The signatures for each method are exact matches:

   (a) Their method names are the same.

   (b) Each method has the same number of parameters (making allowances for any implicit parameters).

   (c) The types of the parameters and return values are the same, and the parameters occur in the same order in each of the two methods.

If classes are assessed as being compatible, the Compatibility Checker produces a Class Method Compatibility Map (CMCM), where the CMCM represents the correspondence between matching methods in the compatible classes. Thus, given an method defined on one class, the CMCM can be used to determine its corresponding or matching method in the other class.

7.4.1.3 The PolySPINgen Tool

Recall that when an operation is invoked on an object, the object may be implemented in the same language as the calling application or in a completely different language. To support this possibility, code must be generated such that the appropriate code in the appropriate language is executed. Specifically, if the language of the calling application and the invoked object are the same, then the original sequence of code representing the object's operation must be executed. In the case that the languages are different, then a call-out to the operation's corresponding foreign operation, along with appropriate parameter marshaling, must be made.

The PolySPINgen tool incorporates the constructs provided by POLYSPIN, along with the constructs provided by the foreign function interface mechanisms defined by

each language. Then, using the information represented by the ICR and CMCM, this tool produces source code enabling polylingual access between compatible C++ and CLOS classes. More specifically, for each C++ member function and its corresponding CLOS method, the PolySPINgen tool performs the following tasks:

1. Generate *foreign function call-in code*: In order for CLOS to invoke a C++ member function, for example, a function must first be generated on the C++ side that accepts a call from CLOS, unmarshals the parameters, performs any TID-to-CID mappings, and then invokes the actual C++ member function. Parameters of atomic or basic types (e.g., integers, floats, etc.) are simply passed according to the atomic type bindings defined by the languages' foreign function mechanisms. Parameters of complex types (i.e., classes) are passed using the values of their TIDs. In the body of this generated function, TIDs are first translated to CIDs (using the TID↔CID mapping mechanism) and then the actual method is invoked.

2. Generate *foreign function call-out code*: This step creates an external declaration for the call-in function generated above. For example, in order for CLOS to call this generated C++ function, a declaration of the function must be appear on the CLOS side. The foreign function interface mechanisms in most languages provide the necessary capabilities for generating this declaration.

3. Generate *toggling logic*: The original body of each method must be modified so that the appropriate code is executed when an operation is invoked on an object. For example, if both the application and the object are implemented in CLOS, then the CLOS code for the operation must be executed. If the object is implemented in C++, on the other hand, then the C++ call-out function, declared in the previous step, must be invoked. To accomplish this, the source code of the body for each operation is augmented with some switching

or toggling logic. This code simply tests the value of the defining language (as maintained by the NameableObject class) and executes the appropriate code.

In the following section, we demonstrate an application of the prototype. This demonstration details the above steps.

7.4.2 Applying PolySPINner

To help illustrate how an application developer might use PolySPINner, we return to the scenario presented in Section 7.2.1. In this example, the OODB contains instances of a Person class, where some of the instances have been developed in C++ and others have been developed in CLOS. To take advantage of the naming facilities offered by PolySPIN, we further assume that the original class definitions for each class already inherit from the NameableObject class, as provided by each language. Thus, prior to any decision to interoperate, the objects resident in the OODB might be organized as shown in the left hand portion of Figure 7.12. In this scenario, the central administration at Hypo U wished to develop an application supporting queries of the kind shown in the right hand portion of Figure 7.12.[4] Specifically, the C++ OQL-style query shown here is embedded in a fragment accessing both the C++- and CLOS-defined objects and performing the desired query. Note that the implementation of each object is completely transparent to the C++ OQL-style query. That is, from the application's perspective, both objects are instances of the C++ Person class, even though one is obviously implemented as a CLOS object. To accomplish this, the application developer would take the following steps:

1. Apply PolySPINner to the interfaces and implementations of both the C++ Person and CLOS Person classes. Figure 7.14 shows the source code for these classes.

[4] Although a query is given in this example, an update could be applied to the objects in a similar manner.

```
class Person : public NameableObject {
private:
  int born;
public:
  int GetAge ();
};
// GetAge member function
int Person::GetAge () {
   int result;
   result = 1995 - born;
   return (result);
}
```

C++ Person Class

```
(defclass Person (NameableObject)
  ((born :accessor born
         :type Date
         :initform "MM/DD/YY")
  )
  )
;; GetAge method
(defmethod  GetAge ((this Person))
  (declare (return-values Integer))
           (- Today (born this))
)
```

CLOS Person Class

Figure 7.14 Original C++ and CLOS Classes

(a) PolySPINner first determines whether or not the two Person classes are compatible by comparing the class interfaces. Using an exact match rule, it should be clear that the C++ and CLOS class interfaces shown in Figure 7.14 are compatible with one another.

(b) Since the C++ and CLOS Person classes are deemed compatible, the tool next generates foreign function interface code corresponding to each of the operations associated with each of the classes. This permits calls from C++ to CLOS and *vice versa*. For example, foreign function interfaces corresponding to each of the GetAge operations provided by each of the classes must be generated.

(c) The tool also modifies the implementations of each of the operations defined by a class. The modifications essentially wrap each operation with switching logic that determines the actual language an object is implemented in and makes the call-out to the code generated in the previous step, if need be. For example, if the C++ application invokes the GetAge operation on what is in reality a CLOS object, the CLOS GetAge operation

must be invoked; otherwise the original C++ code implementing the C++ GetAge operation must be executed.

(d) The resulting generated code is shown in Figure 7.6. Although generating this code could be done manually, PolySPINner automates this process, thus shielding programmers from having to use low-level communication mechanisms such as foreign function interfaces.

2. Re-compile the modified class implementations and the generated source code.

3. Re-link the application.

As should be evident, neither the class interfaces nor the persistent data are modified by the PolySPINner tool. Only the class implementations must be re-compiled, along with the generated source code. In addition, the original application remains unchanged, although it must be re-linked to accommodate the changes made to the class implementations.

7.5 Evaluation

A prototype of PolySPIN has been built as a set of extensions to the TI/Arpa Open Object-Oriented Database. The prototype provides a unified name management mechanism for both a C++ and a CLOS API to the Open OODB, where applications developed in either language may transparently access either C++- or CLOS-defined objects. Moreover, if a particular C++ and CLOS class are known to be equivalent or compatible, instances of their respective classes may be transparently accessed by an application written in either language. A prototype of PolySPINner has also been implemented. The prototype automates the application of PolySPIN, thus freeing application developers from having to use and maintain the low-level details of PolySPIN and foreign function interface mechanisms. Although currently limited

to exact signature matching rules, the toolset also helps determine the compatibility of classes defined in different languages.

We have already used the PolySPIN approach to construct a polylingual application in Open OODB. In particular, since the PolySPIN prototype is itself a polylingual application, several components of the prototype have in fact been constructed using PolySPIN and PolySPINner. For example, the BindingSpace and Context classes (shown in Figure 6.8) are both implemented in C++. They are, however, accessed transparently by CLOS applications using PolySPIN technology.

Based on our experience with its use, we now offer a preliminary evaluation of PolySPIN. Our approach to evaluation involves comparing PolySPIN to current state-of-the-art techniques supporting polylingual programming. In particular, CORBA-based approaches have emerged as one popular solution to the interoperability problem. As noted earlier, these approaches require the use of a separate type model, often called an *interface description language* (or IDL) for describing interoperating objects in a system, where an IDL's type model typically represents a common denominator for the various language-specific type models (e.g., C++, CLOS, etc.) in a system. Given a particular IDL description to an object, translators can produce language-specific classes and communication logic. Application developers then provide the class implementations.

In the remainder of this section, we describe a particular CORBA-style approach, namely the Inter-Language Unification (or ILU) system,[5] an interoperability mechanism developed at Xerox PARC [64]. ILU serves as a useful point of comparison (and hence evaluation) for several reasons. As noted above, it is representative of the state of the art in approaches to interoperability. Although ILU is not used in conjunction with any OODB (in its current form), it is worth noting that the ODMG standard is heavily influenced by CORBA; hence, ILU is a reasonable approximation

[5]We used ILU version 1.8 [63] on a Sparc 10 running SunOS 4.1.x in performing this evaluation.

of an interoperability mechanism for OODBs. Finally, ILU is appealing since it is the only available system that we are aware of (other than POLYSPIN) that supports polylingual access between C++ and CLOS.

This section is organized as follows. In Section 7.5.1, we use the Hypo U Scenario from Section 7.2.1 as an aid in describing ILU. We then, in Section 7.5.2, compare POLYSPIN and ILU based on the objectives and issues outlined in Section 7.2.2.

7.5.1 An Overview of ILU

The ILU approach is based on a distributed client-server architecture using RPC as its underlying communication substrate, where servers essentially manage instances of classes and clients manipulate these instances. ILU uses a simple two-level binding space organization, where servers correspond to binding spaces. An implicit root binding space contains the named servers. Servers register names for objects in this binding space organization and clients access these objects by resolving names.

ILU uses an IDL called ISL (Interface Specification Language) for describing classes. Classes are specified by declaring a class identifier and associating a set of operations with the class. Note that ISL does not permit data members to be defined for classes. Figure 7.15 shows the ISL for the Person class in the Hypo U scenario.

```
INTERFACE Person;

TYPE ClassInterface = OBJECT
  METHODS
    GetAge () : INTEGER
  END;
```

Figure 7.15 ISL for Person Class

Creating an ILU-based polylingual system requires that the following steps be performed:

1. Create the language-specific class interfaces. This is accomplished by applying appropriate language-specific translators (supported by the ILU system) to the ISL class description. For example, Figure 7.16 shows a portion of the C++ code generated when the ISL-to-C++ translator is applied to the ISL in Figure 7.15.[6] The C++ class contains operations that correspond almost directly to the operations given in the ISL specification. (Although not shown, other auxiliary functions, generated by the translators, are also included with the class.) Thus, a GetAge operation appears in both in the ISL class specification in Figure 7.15 and the generated C++ class interface in Figure 7.16. Using the ISL-to-CLOS translator, a similar class interface would be generated for use by CLOS servers and clients.

```
class PersonInterface : public virtual iluObject {
public:
  PersonInterface();
  virtual ilu_Integer GetAge (PersonStatus *_status);
};
```

Figure 7.16 Generated C++ Class Interface for ISL Person

2. Construct class servers. An ILU server manages the instances of a particular class. Thus, a C++ Person server is responsible for C++ Person objects and a CLOS Person server is similarly responsible for CLOS Person objects. In the Hypo U scenario, for example, each server can be viewed as a separate personnel database. Constructing a server involves the following:

[6] The ILU translators produce additional server and client code which, other than needing to be compiled and linked into a client and server, respectively, can be ignored by a developer. The primary purpose of these components is to set up the necessary communication connections among clients and servers.

(a) A separate class, which is a subclass of the class generated in step 1, must be supplied by the programmer. This class contains the actual implementations of the various member functions, as well as any required data members. Figure 7.17, for example, shows (a portion of) the class PersonActual, which inherits from the (generated) PersonInterface class from Figure 7.16. Thus, the PersonActual class is used by the servers, while the PersonInterface is used by clients.

(b) A server program must be implemented. The server program essentially creates instances of the class implementation, assigns them names, and then waits for requests. A portion of the C++ server is shown in Figure 7.17. A corresponding CLOS server is shown in Figure 7.18.

3. Construct clients. Clients issue requests to servers for objects and then perform operations on those objects. With respect to our Hypo U scenario, a C++ client might correspond to the OQL-style query given in Figure 7.19 (although ILU does not have an OQL interface). A client interacts with a server through the interface generated in step 1. Thus, the client views an instance as if it were implemented in its language, even though it may turn out to be implemented in a different language.

4. Invoke the servers and the clients.

```
// Implementation class for Person
class PersonActual : public virtual PersonInterface {
public:
private:
  int age;
  virtual ilu_Integer GetAge (PersonISLStatus *_status) {
    return (age);
  }
};

// Server
main (int ac, char **av)
{
  // Create a server
  iluServer myServer ("Server", NULL);
  // Create a person and assign a name
  PersonInterface *alan;
  alan = new PersonActual (50, "alan", &myServer);
  alan→ILUPublish();
  // Run the server
  myServer.Run();
}
```

Figure 7.17 C++ Class and Server Implementation for ISL Person

7.5.2 Comparing PolySPIN and ILU

At a very general level, PolySPIN and ILU are quite similar. Both approaches rely on a unified name space as a means of hiding language properties from applications. Thus, the fact that an object may be implemented in a language that is different than the application's language is made transparent. An application simply resolves the name of an object and the underlying framework ensures that the correct language system is invoked on its behalf. In addition, both approaches support tools that automate the development of interoperating code. Thus, programmers are unaware, more or less, of the underlying transport and communication mechanisms (e.g., foreign function interface, RPC, etc.).

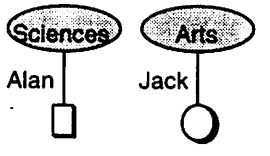
Despite these similarities, we believe that the PolySPIN approach offers a number of significant advantages over the ILU approach. One obvious shortcoming in ILU is related to its name management mechanism. Clearly, the general structured

```
;; Implementation class for CLOS Person
(defclass person-actual (person-interface)
  (
   (age :initarg :age
        :accessor :age)
   ))

(defmethod person-actual:get-age ((self person-actual))
  (values (age self))
  )

// Server for Arts College
(defun start-server ()
  (let* (
          ;;; Create a server for the Arts college
          (arts (make-instance 'ilu:kernel-server :id "Arts"))
          ;;; Create a person and assign a name
          (jack (make-instance 'person-actual :ilu-kernel-server arts
                                :ilu-instance-handle "Jack")))
    ;;; publish the binding
    (ilu:publish jack)
    ;;; run the server
    arts))
```

Figure 7.18 CLOS Class and Server Implementation for ISL Person



C++ object

CLOS object

Server

```
Set<Person*> setPerson;
PersonInterface* p;
PersonInterfaceStatus status;
...
p = (PersonInterface *)
        iluObject::Lookup ("Alan@Sciences");
setPerson.Insert (p);
p = (PersonInterface *)
        iluObject::Lookup ("Jack@Arts");
setPerson.Insert (p);
...
Set<Person*> result;
Query {result =
   SELECT *
    FROM PersonInterface* matched IN setPerson
    WHERE matched->GetAge(status) > 45;
}
...
```

Figure 7.19 ILU-based Client

binding space organization in PolySPIN is more robust and flexible than ILU's two-level binding space organization. In addition, servers in ILU are mono-lingual. Thus, although the language of the server is hidden from clients, clients might be able to exploit this property. There are no such restrictions in PolySPIN, since binding spaces can contain bindings to objects defined in multiple languages.

An obvious point of distinction concerns various typing issues. First, ILU requires the use of an external type model, i.e., the ISL, while PolySPIN allows programmers to use their native type models. As discussed in Section 7.1, IDLs (such as ISL) reintroduce impedance mismatch, thus forcing programmers to simultaneously use, reason about and maintain multiple type models. Second, the class implementation technique in ILU can be characterized as being cumbersome or awkward. The ISL translators generate a class interface; this is the interface that is seen and used by clients. The actual implementation of the class is represented by two class definitions - the interface class and the implementation class (which is a subclass of the interface class). This somewhat unusual approach to constructing classes is not only counterintuitive, but also can be difficult to maintain and debug. In PolySPIN, programmers use the usual class construction techniques. As a result, their is only single class definition per class per language. Third, by its very nature, ILU is limited to exact signature matching rules for types. Although our examples (and our current prototype) have similar limitations, there is no general restriction in our approach. For example, the compatibility checker component of PolySPINner could be relaxed so that it used an *intersection* matching rule instead of the current exact matching rule. Thus, in the Hypo U scenario, we may only want to consider Person classes that provide a GetAge operation in there interface; any other operations can be ignored.

Finally, we believe that PolySPIN offers a greater degree of flexibility in terms of timing (i.e., when interoperability decisions must be made) and impact of change.

Specifically, ILU seems more suitable for the easiest and common interoperability cases described in Section 7.2.2. When programmers begin with an ISL description, the generated classes interoperate by definition. To support the megaprogramming case, a programmer would need to create an ISL description from the existing classes and then, as described above, connect the existing classes to the generated classes. This would clearly require substantial changes be made to existing applications and classes. In the PolySPIN approach, existing classes can be made to interoperate by simply applying PolySPINner to them. Assuming the classes are compatible, PolySPINner instruments the class implementations, while leaving the existing class interface and application intact. We note that this technique is predicated on the fact that the existing classes already participate in PolySPIN's name management mechanism. If this is not the case, then the programmer would first need modify the definitions of the classes such that they inherited from the NameableObject class prior to applying PolySPINner. While the first case clearly supports the megaprogamming scenario, this latter case would require existing applications that depended on the class to be re-compiled. However, the modifications required by our approach are much less extensive that those involved in using ILU.

7.6 Summary

In this chapter, we have described a new class of interoperability problem for OODBs, namely the polylingual access problem. We have also described PolySPIN, an approach supporting persistence, interoperability and naming in OODBs, and we have shown how PolySPIN can be used to overcome the polylingual access problem in OODBs. We have described PolySPINner, a tool to help automate the use of PolySPIN and illustrated its capabilities using a simple, but representative, example of a polylingual OODB application. Finally, we have evaluated how our approach compares with present day approaches to similar interoperability problems.

We believe the work reported in this chapter represents an important extension to object-oriented database technology. While modern OODBs often provide multiple language interfaces, interoperating among the various languages can be a cumbersome and complex process, thus limiting their overall potential. PolySPIN provides transparent, polylingual access to objects (of compatible types), even though the objects may have been created using different programming languages. Thus, application developers are free to work in their native language without precluding the possibility of interoperating with foreign language objects or applications. The benefits of PolySPIN can be summarized as follows:

Name and Data Unification Based on the name management mechanism developed in Chapter 6, both C++ and CLOS objects are organized under a single, unified name management mechanism. In addition, an application can transparently access data independent of the language used to define that data.

Megaprogramming Interoperability decisions do not have to be made *a priori*. As long as class definitions participate in the name management mechanism, their implementations can later be modified to enable transparent interoperability between them. Thus applications, written in completely different languages can be made to interoperate at any point in time (assuming of course that their types are deemed compatible).

Automated Support The compatibility of types described in different languages can be determined automatically. Although limited to exact matching rules in the current prototype, the approach is general enough to accommodate more flexible matching rules (such as subset or intersection). We discuss this in greater detail in Chapter 8. In addition, PolySPINner relieves programmers of the burden of having to manually utilize low-level interoperability mechanisms, such as foreign function interfaces.

Impact of Change Only the implementations of operations are modified, while class interfaces remain unchanged. Applications do not require any modifications and thus do not have to be re-compiled (although they may need to be re-linked).

Native Type Model The PolySPIN approach does not require programmers to use a separate type model. Instead, programmers use their native type system.

CHAPTER 8

CONCLUSION

Name management is among the most basic foundations of computing, yet historically it has received relatively little attention from researchers in computer science. As a result, computing systems typically rely on *ad hoc* and primitive approaches to name management, making systems confusing to use and difficult to maintain. Name management is a particularly important and challenging problem in convergent computing systems because the different paradigms underlying a convergent computing system's individual components typically rely on different and often incompatible name management mechanisms. In this dissertation, we have attempted to demonstrate the importance of name management issues to both users and developers and to lay a foundation for improved treatment of those issues. Our primary objective has been to contribute to a better, more thorough understanding of this critical, and yet underappreciated, aspect of computing. The results of our research efforts include the development of novel modeling techniques, mechanisms and applications with both theoretical and practical implications for name management in convergent computing systems. In this chapter, we first summarize the contributions of this dissertation. We then outline several important directions for future research, and conclude with some brief closing remarks.

8.1 Contributions

The results of this dissertation have contributed to a more rigorous foundation for increased understanding and improved practical utilization of name management in

convergent computing systems. In broad terms, we defined a model of name management and applied it to several real-world systems, thus improving our understanding of the use of names, as well as some of the difficulties often encountered, in these systems. Use of the model also led to a more principled approach to developing extended name management capabilities in an extremely important domain of convergent computing systems, i.e., object-oriented database systems. We demonstrated the usefulness of our prototype mechanisms along several dimensions, including the provision of some preliminary performance data confirming the feasibility of our approach. Finally, the resulting mechanisms enable some very interesting and novel approaches to seamless and transparent interoperability. More specifically, the contributions of this dissertation include:

- A conceptual, unifying framework for name management in convergent computing systems, called PICCOLO. The PICCOLO framework generalizes upon previous work in this area and yields important insights into name management, along with the various problems that arise in the use of convergent computing systems.

  In particular, the basic component of the framework described in Chapter 3 provides a well-defined foundation for name management. In addition to establishing a standard and generally applicable terminology, perhaps its most interesting contribution is its distinction between the concepts of binding space and context. In contrast, other models have tended to combine these concepts into a single one. We have argued that making this distinction leads to more precise and explicit explanations of existing mechanisms, which not only leads to a much clearer understanding, but can also help reveal or uncover various anomalous behaviors.

  The extended features of the framework, presented in Chapter 4, focus on context specification and formation. The significant result of this facet of PICCOLO

is its ability to capture a broad range of different approaches to context specification and formation. In particular, epochs, context formation templates, context formation processes and extended closures have proven to be valuable features for describing and reasoning about this often complex aspect of name management.

- Demonstration of evolving algebras as a suitable formalization technique for the PICCOLO framework's semantics. Although given in terms of a specific example, we have argued that application of this kind of formalization permits rigorous and precise descriptions of existing name management approaches, and thus serves as a more solid foundation for formal reasoning about name management mechanisms and their use in convergent computing systems. In addition, its use serves as a foundation for potentially automated analyses of name management-related properties in convergent computing systems.

- Design, implementation and evaluation of well-defined, novel mechanisms providing improved support for name management in convergent computing systems. In particular, our focus has been on providing improved approaches to name management in persistent object systems, specifically object-oriented database systems. The prototype mechanisms, described in Chapter 6, are distinguished from previous efforts in several dimensions. They provide more uniform and flexible name management capabilities than those found in many of today's systems. Specifically, the basic name management mechanism we developed for the Open OODB allows multiple, independent programs, perhaps written in different languages, to use and access the persistent store though a single, consistent name management interface. We have been able to provide these extended capabilities by using basic OODB abstractions (as opposed to modifying the internals of the OODB). We have also shown that the resulting

architecture is easily adaptable with modest performance overhead. Finally, we have demonstrated the benefits of incorporating some of PICCOLO's extended context-control features into a shell-style user interface called CONCH.

- Development of enabling technology for polylingual interoperability in object-oriented databases. The approach, called POLYSPIN (Polylingual Support for Persistence, Interoperability and Naming) and described in Chapter 7, synthesizes and builds upon the efforts as the previous four chapters. Applications based on POLYSPIN can access and manipulate objects as if they were all implemented in the same language of the application. Furthermore, application developers are free to work in their native language without precluding the possibility of interoperating with foreign language objects or applications. Most existing interoperability approaches, both commercial and research prototypes, typically provide no such capabilities, while those that do require the use of external type models (e.g., CORBA, ILU, ODMG) or explicit data translations (e.g., translating C++ to CLOS, translation of C++ classes to a relational data model). Finally, we have developed automated support for the POLYSPIN technology. Specifically, the POLYSPINNER prototype helps assess the compatibility of classes defined in different languages. Based on this assessment, the tool automatically instruments the implementations of compatible classes with POLYSPIN logic.

8.2 Future Directions

Given the pervasiveness of name management in computing, there are a wide range of possible future directions, many of them with important connections to other areas in computer science. Continuing with the overall theme of this dissertation, this section outlines a representative collection of future work in each of the areas of models, mechanisms and applications.

8.2.1 Models

The initial success of the existing PICCOLO model as a basis for explaining, reasoning about, designing, implementing and analyzing name management mechanisms across a reasonably broad range of system classes, and particularly convergent computing systems, is quite encouraging. There are, however, several dimensions in the modeling facet of our research that are worth exploring.

8.2.1.1 Generality and Applicability of Piccolo

In an effort to further confirm, or find ways to extend, PICCOLO's generality and applicability, the model should be applied to additional classes of systems. There are a variety of systems that would serve as useful stress tests for PICCOLO. A reasonable starting point might be a detailed examination of UNIX, which could reveal additional anomalies and intricacies similar to those reported for the mv command in Chapter 3. The results of such a study could have important implications for the development of new operating systems command line shells. Additions to existing systems frequently result in name management mechanisms that are difficult to use and comprehend. Thus, applying PICCOLO to the child library unit mechanism of Ada 95 [] or the namespace [131] mechanism of C++ could serve as a useful reference to programmers, both novice and experienced. Similarly, the appearance of new computing systems with intricate and complex approaches to name management seems to proceed unabated. Some prominent examples include World Wide Web http servers [11] and the Java programming language []. Both of these systems would serve as excellent experiments since they are representative of convergent computing systems, the former combining features of the Internet and file systems and the latter melding features of programming languages, operating systems and the Internet. In addition, applying PICCOLO to these critically important domains might yield some important insights and clarifications to both users and developers, which in turn,

might lead to improvements on the systems' underlying name management mechanisms. Finally, one notable shortcoming in our presentation of our modeling efforts in this dissertation is that we do not fully exercise all the capabilities in the PICCOLO framework. Specifically, the CFTP directive was not illustrated in any of our examples, where the CFTP is intended to describe provision directives in context formation. Such directives are frequently found in many modern programming languages. Thus, it would be useful to apply PICCOLO to languages such as PIC/Ada, Modula-3, C++ and Java.

8.2.1.2 Extensions to Piccolo

There are a variety of ways that PICCOLO itself might be extended to accommodate other aspects of name management. We identify some possible extensions below.

Contextual Consistency and Compatibility As noted earlier, different components of a computing system may employ different contexts. When these components are coordinating some activity, they may require a consistent interpretation of their contexts in order to perform some computation. One potentially useful enhancement to PICCOLO would be a more rigorous definition or classification of *contextual consistency*. For example, when examining two different contexts, a programmer may want to know whether two (or more) contexts are equivalent (i.e., they each contain exactly the same set of bindings)[1], or whether some subset of the contexts is equivalent. Other useful classifications might include notions of ambiguity, i.e., two contexts containing the same set of names bound to different objects, or synonyms, two contexts containing bindings of different names to the same object.

[1] We actually demonstrated this kind of consistency in Chapter 5.

Context for Context Names are frequently used in specifying a context. For example, both in the UNIX environment variable specification

```
TEXINPUTS=/dir1:/dir2
```

and in the Ada with clause

```
with Package1, Package2
```

names are used to specify a desired context. In modeling such specifications, we have generally assumed that these names, i.e., the ones in the specification, have already been successfully resolved to valid binding space objects. A more thorough treatment, however, should relax this constraint. We believe that PICCOLO should be able to accommodate this additional aspect of context specification and formation. One possibility includes allowing CFTs to have extended closures associated with them. In essence, this would permit a context specification to have a context associated with it.

Partial Bindings PICCOLO makes a tacit assumption that bindings are *total*. By total, we mean that a binding always contains a name and an object. Many approaches to name management, however, permit the existence of *partial* bindings. For example, in programming languages such as Pascal and C, forward or external declarations behave as placeholders, so that compilers can proceed during a single pass when processing source code. Similarly, executables making use of dynamic, shared library mechanisms rely on similar kinds of partial bindings. This not only reduces the size of executables, but also provides a convenient method of incorporating updates to libraries while minimizing additional compilations. One way PICCOLO could be modified to account for such bindings would be to allow for *surrogate bindings*. A surrogate binding is similar to a traditional binding in PICCOLO except that it consists of a name-surrogate

pair instead of a name-object pair, where a surrogate represents some general description of the required object.

8.2.1.3 Formal Modeling

One shortcoming of our formal modeling activities is that we have only shown a formalization of a specific example in a specific system. Although it is open to question what a general formal model of name management might look like, it would seem appropriate to use PICCOLO and evolving algebras to guide the development of a general model for a particular system. For example, in Chapter 5 we showed a formalization of a specific Make example. A more ambitious effort would involve developing a general formalization of name management in Make/UNIX (including a subset of utilities that might be invoked by Make, e.g., a compiler, a linker, an archiver, etc.). Such a formalization would then allow for the analysis of arbitrary Makefiles.

The EA formalism has proven to be quite useful for our purposes. Its operational model makes its very accessible, while the availability of an EA interpreter permits EA descriptions to be executed and observed. It is conceivable possible, however, that other techniques (e.g., axiomatic semantics) might turn out to be better suited for describing name management in convergent computing systems. Other formalisms seem to have some possible connections to name management and so they might be worth pursing. Examples include the TAOS security model [152] and Milner's pi-calculus [88].

8.2.2 Mechanisms

Although the name management mechanisms described in Chapter 6 represent a substantial improvement over existing approaches, there are still a variety of directions for additional research in this area.

8.2.2.1 Generality and Applicability of Mechanisms

Analogous to work in modeling, one obvious area of future research involves a continued assessment of the generality and feasibility of our mechanisms in various settings. For example, since the basic mechanism is developed as a modular component of Open OODB, it should be possible to experiment with it in other OODBs. Such experimentation would not only help confirm the generality of our approach, but would also help lead to more realistic performance comparisons. (We discuss performance issues below.) Since many OODBs (both commercial and research prototypes) support C++, we anticipate that performing the actual port itself should be relatively straightforward.

Another assessment of the mechanism's generality and applicability would involve experimenting with other object-oriented languages, either in conjunction with Open OODB or some other OODB. Ada 95, Modula 3 and Java are certainly suitable candidates for an experiment. Open OODB, in its current form, supports only C++ and CLOS. Thus, performing this experiment would first involve integrating the language into Open OODB. Once this is accomplished, enabling the new language to use our existing name management mechanism would require producing the appropriate interfaces to the name management mechanism, implementing an appropriate TID-to-CID mapping mechanism (for the new language), and installing call-outs to the language-specific MakePersist and GetPersist routines in the (C++) core name management mechanism.

We also believe that the basic mechanism developed in Chapter 6 is applicable in other domains, i.e., besides OODBs and other POSs. Thus, the modular mechanism could be installed in other kinds of systems by distributing it as a library for use as a general purpose name management package. For example, one of the glaring weaknesses of the ILU system is its comparatively primitive approach to name

management.[2] An interesting experiment, therefore, might involve integrating our mechanism into ILU.

8.2.2.2 Extending Capabilities of the Basic Mechanism

Although we eventually selected a design that did not support any notion of type checking, we still believe that type plays an important role in a name management mechanism. When an object is assigned a name in a binding space that is reachable from a persistent root binding space, the type information associated with the object should be retained. Similarly, when an object is resolved in a context built from a persistent binding space, the type information for the resolved object should first be checked before returning an object. The persistence service in Open OODB guarantees that the types of persistent objects will be maintained by the database. In addition, the data dictionary service permits programs to dynamically *query type* information for objects. Thus, it may be worthwhile to revisit the second design (described in Chapter 6), or more likely some variant of it, as a starting point for a mechanism that supports type checking in conjunction with a name resolution operation.

The mechanisms described in Chapter 6 provide a uniform name management interface for objects defined in multiple languages. Another dimension related to uniformity concerns the unification of the transient name space of a programming language with the persistent name space of the persistent repository. In the enhanced name management we developed for Open OODB (and similarly in other persistent[X] POSs), the distinction between the transient and persistent name spaces is quite evident. The *de novo* class of POSs (e.g. Napier) have sometimes avoided the uniformity problem, since, as noted earlier, they have had the advantage of not having to be backwardly compatible with an existing language. Thus, an interesting topic of research might address how to unify these disparate approaches to name man-

[2]This shortcoming is recognized by the authors of ILU [63].

agement by using the basic abstraction of the systems and leaving the capabilities of the language intact (e.g., without extending the syntax of the language with special keywords).

8.2.2.3 Optimization and Performance

There is a great deal of work that can be done in the area of performance and optimization of name management mechanisms. One obvious area of improvement concerns the overall efficiency of the system. As we noted in Chapter 6, the focus of this dissertation on was demonstrating improved capabilities, not improved efficiency (although our performance evaluation shows that our extended mechanism does not appear to impose an appreciable overhead). A thorough analysis of our design and a concomitant run-time profile examination of our implementation would certainly lead to performance improvements in the system. For example, our current implementation of binding spaces uses unbalanced binary trees (clearly a naive choice of data structures). A much better design could make use of more efficient data structures, such as a hash table structure.

A related and more serious problem concerns the relationship of the data structure supporting the binding space structure to the underlying persistence mechanism. One disadvantage of building a modular name management mechanism for the Open OODB is that the data structure supporting the binding space organization is treated as a monolithic object by the persistence service. This means that when the root binding space is fetched from the persistence store, the persistence service (rather naively) attempts to retrieve the entire naming structure. The presence of UORs in bindings (as opposed to swizzled objects) helps the situation somewhat; however, such a policy is clearly infeasible for very large databases. One research direction involves fine tuning or developing a data structure that performs more efficiently with respect to the persistence service.

Another potential source of improvement can be traced to the underlying storage manager. Specifically, the Exodus storage manager can be dynamically configured to cluster objects. The Open OODB uses a simple policy such that objects are located near one another according to the time they were created. Combined with the name management mechanism, there are several improvements that can be made that might improve performance. For example, it might be worthwhile to cluster objects based on their containing binding space. Another interesting idea would involve extending compilers to exploit specific name management properties of persistent objects. Based on an analysis of an object's extended closure, objects could be clustered or even prefetched accordingly.

8.2.2.4 Extending Conch's Capabilities

Experiments with CONCH have suggested a number of improvements and extensions that could be made. Some of these involve additional capabilities for the shell, such as richer mechanisms for binding specification (e.g., allowing CFTs to specify local renaming of objects), for context specification (e.g., incorporating CFTPs into closure definitions or allowing CFTRs to define sources as combinations of binding spaces) and for creating and querying CFPs. Others would make the approach more user-friendly; the existing, relatively low level and explicit, shell commands are suitable for the fine-grained control needed in experimentation, but much of the effort involved in creating and manipulating CFTs could, and probably should, be automated or hidden to benefit POS users. Another interesting prospect is making extended closure information in a CFT/CFP-style format a standardly available part of the interface to objects in future systems. This would have significant potential for facilitating reuse, maintenance and interoperability. It addition, the development of analyses, based on the results of name management model research, could be incorporated into Conch.

As noted in Chapter 6, CONCH supports other functionality in addition to its context-control capabilities. In particular, it permits users to browse the persistent store, create new bindings and remove existing bindings. It does not, however, allow users to make copies of, or modifications to, objects in bindings. While file system shells provide such functionality, supporting similar functionality in an object-oriented database system is problematic. For example, the semantics for a copy operation in file systems is trivial, but the semantics for a copy operation in an OODB can be different for different objects. An alternative might be to convert the NameableObject class into an abstract class (in C++, for example, this would amount to associating pure virtual operations with the class), where its operations basically define a common protocol for objects supported by CONCH. For example, in file systems, all files have *edit* and *copy* operations associated with them. Of course, since files are just typeless byte streams, implementing these operations is trivial. In an object-oriented database, each nameable object would have to implement the core functions defined by the NameableObject super-class. For example, an image class might define the edit function to invoke an image editing tool (instead of a text editor).

Finally, we note that the current user interface for CONCH is text-based. Providing a graphical user interface would substantially improve the user-friendliness of the system. It might also facilitate new approaches to visualizing and modifying contexts and extended closures for objects in a persistent store.

8.2.3 Applications (Interoperability)

Experimentation with our name management-based approach to polylingual interoperability has suggested a number of future research directions. We highlight some of these possibilities in this section.

8.2.3.1 Generality and Applicability of PolySPIN

One evident shortcoming of the current framework and toolset is that they only support interoperation of components implemented in C++ and CLOS. Augmenting Open OODB with other languages will not only help confirm the generality of the language uniformity facet of our name management mechanism, but will allow continued experimentation with PolySPIN, in an effort to further confirm, or find ways to extend, its generality and applicability. As noted earlier, Ada 95 and Java are suitable candidates worth investigating for support in future versions of PolySPIN and PolySPINner.

We also note that our current approach to supporting interoperability is tightly intertwined with support for data object persistence. Thus, an open area of research involves investigating ways to loosen or remove this coupling in future versions of PolySPIN.

8.2.3.2 Type Compatibility

In our view, the most significant shortcoming in our initial prototypes is the very limited notion of type compatibility that they embody. Specifically, the determination of type compatibility embedded in PolySPINner is essentially an instance of *exact signature matching*, the most restrictive category in the Zaremski and Wing classification of signature matching approaches [157]. While we intentionally adopted this restrictive definition of type compatibility for our initial prototypes so as to focus on other aspects of supporting interoperability for polylingual systems, and even though a number of interesting examples can be successfully handled despite this limitation, there is nothing inherent in our approach that restricts us to this form of type compatibility. Therefore, an important direction of future research involves expanding the type compatibility aspect of PolySPIN and PolySPINner. A first step might include an investigation of how more general categories of signature matching

techniques in the Zaremski and Wing classification can be brought to bear on multilingual and polylingual interoperability situations (including development of automated support for doing so.) This will dramatically strengthen the compatibility checking component of the PolySPINner, allowing it to produce much richer and more powerful class compatibility mappings, and thus greatly expanding the applicability of our approach. Specifically, it will mean that components can interoperate not only by sharing data objects whose types are identical, but also by sharing data objects whose types are related by much weaker, and therefore probably more prevalent and hence more useful, kinds of compatibilities. A second step could investigate richer forms of type compatibility, based on deeper, semantics-based analyses of the relationships between types. Zaremski and Wing's work on *specification matching* [158] and Blaine and Goldberg's work on axiomatic approaches [15] represent two valuable points of departure for our research in this area.

8.2.3.3 Evolution

An important problem closely related to interoperability is the problem of type evolution in persistent object systems [8, 9, 98, 104, 124]. Indeed, the type evolution problem can be seen as a special case of interoperability, one in which the need for interoperation arises due to changes in the definitions of one or more data types that have existing persistent instances. Thus, an extension of our interoperability approach to the type evolution problem, in particular how enhanced versions of the PolySPINner tool can be brought to bear on this problem, is an interesting research question.

8.2.3.4 Transparent Persistent Polylingual Data Structures

Like polymorphism in programming languages, polylingualism is a powerful concept since it allows programs to ignore certain language implementation details of persistent objects and treat them as if they were all implemented in the same lan-

guage as the program. One limitation of the approach is that it does not support transparent persistent polylingual data structures. To help understand this concept, consider a C++ class definition for a Car that might have a data member, driver, of type Person. Suppose that a C++ application creates an instance of the Car class, fetches a persistent CLOS object of type Person (the program is of course unaware that the object is implemented in CLOS), and sets the driver field to this Person object. Since the Car object is constructed from objects that are implemented in multiple languages, we call it a polylingual data structure.

Based on the semantics of the name-based persistence service in PolySPIN, assigning a name to the Car object in a persistent binding space should result in making all the data reachable from the Car object persist. Hence, a different program that accesses this object should also be able to access the Person data represented in the driver data member. In other words, when the Car object is fetched from the persistent store, a surrogate for the Person object should be constructed since the object is implemented in CLOS. Unfortunately, the current version of PolySPIN does not support this. In its current form, the persistence service in PolySPIN only sees the C++ surrogate object in the driver data member when the Car object is initially saved. There is no information currently stored in the surrogate dictating how the surrogate should be reconstructed when the Car object is retrieved from the database. Although the Person object is available in the persistent store, the driver data member in the Car object will contain an invalid surrogate. Thus, an open research question concerns how to provide transparent polylingual persistence in polylingual OODBs. We believe that PolySPIN already has many of the necessary capabilities for providing this generalization of persistence.

8.2.3.5 Performance

In our initial prototypes we have completely ignored questions of performance and optimization in polylingual OODBs. Clearly, crossing language boundaries can

be very expensive. In Chapter 7, we showed a polylingual query accessing two objects, one defined in C++ and one defined in CLOS. Scaling the problem, suppose a query is applied to a set containing 10000 objects, half of which are actually implemented in C++ and half of which are actually implemented in CLOS. A naive approach might process the query on an object-by-object basis, possibly resulting in a large number of language boundary crossings. A relatively simple optimization technique might cluster the objects according to their implementing language, perform the query on each subset in each language's address space, and then combine the results. Having demonstrated the potential value of our approach to polylingual access, future research could focus on these and related topics, with the aim of improving the approach's practicality.

8.3 Concluding Remarks

In this dissertation, we have argued that improved models for name management lead to improved name management mechanisms. We have also shown how such models and mechanisms can help enable other capabilities, specifically polylingual interoperability, in convergent computing systems. The ultimate success and widespread adoption of next-generation convergent computing systems will, in part, hinge on a greater understanding of name management and on more flexible, powerful, seamless approaches to name management. We believe the work reported in this dissertation represents an important step in this direction.

APPENDIX

EVOLVING ALGEBRA FOR MAKE/UNIX

This appendix gives the complete EA description of $A_{\text{Make/UNIX}}$ (from Chapter 5), which can be run by the Evolving Algebra Interpreter [60].

```
/* Epochs */
FUNCTION startup
  HAS ARITY 0
  IS STATIC
  EXPRESSION "STARTUP"
ENDFUNCTION

FUNCTION hc
  HAS ARITY 0
  IS STATIC
  EXPRESSION "HALT"
ENDFUNCTION

FUNCTION da_cf
  HAS ARITY 0
  IS STATIC
  EXPRESSION "DA_CF"
ENDFUNCTION

FUNCTION da_nr
  HAS ARITY 0
  IS STATIC
  EXPRESSION "DA_NR"
ENDFUNCTION

FUNCTION ccc
  HAS ARITY 0
  IS STATIC
  EXPRESSION "CCC"
ENDFUNCTION

FUNCTION ia_db
  HAS ARITY 0
  IS STATIC
  EXPRESSION "IA_DB"
ENDFUNCTION

.
```

```
FUNCTION ia_cf
  HAS ARITY 0
  IS STATIC
  EXPRESSION "IA_CF"
ENDFUNCTION


FUNCTION ia_nr
  HAS ARITY 0
  IS STATIC
  EXPRESSION "IA_NR"
ENDFUNCTION

/* NextEpoch and CurrentEpoch functions */
FUNCTION NextEpoch
  HAS ARITY 1
  IS STATIC
  TUPLES
    (da_cf, da_nr)     &
    (da_nr, ccc)       &
    (ccc,   ia_db)     &
    (ia_db, ia_cf)     &
    (ia_cf, ia_nr)     &
    (ia_nr, da_cf)
ENDFUNCTION

FUNCTION CurrentEpoch
  HAS ARITY 0
  TUPLES
     (startup)
ENDFUNCTION

/* Universes */
FUNCTION object
  HAS ARITY 1
  IS UNIVERSE
ENDFUNCTION

FUNCTION line
  HAS ARITY 1
  IS UNIVERSE
ENDFUNCTION

/* Target Dependency Line */
FUNCTION TD
  HAS ARITY 0
ENDFUNCTION

/* Invoke Action Line */
FUNCTION IA
  HAS ARITY 0
ENDFUNCTION
```

```
FUNCTION name
  HAS ARITY 1
  IS UNIVERSE
ENDFUNCTION

FUNCTION bindingspace
  HAS ARITY 1
  IS UNIVERSE
ENDFUNCTION

/* Sources Binding Space */
FUNCTION Src
  HAS ARITY 0
ENDFUNCTION

/* Active Binding Space */
FUNCTION Cwd
  HAS ARITY 0
ENDFUNCTION

/* Binary Binding Space */
FUNCTION Bin
  HAS ARITY 0
ENDFUNCTION

/* Tool and Tool.C files */
FUNCTION TOOL
  HAS ARITY 0
ENDFUNCTION

FUNCTION TOOLDOTC
  HAS ARITY 0
ENDFUNCTION

/* Bindings in Binding Spaces */
FUNCTION LookUp
  HAS ARITY 2
ENDFUNCTION

/* Closures */
FUNCTION ContextDirective
  HAS ARITY 1
ENDFUNCTION

FUNCTION GetEpoch
  HAS ARITY 1
ENDFUNCTION

FUNCTION GetSources
  HAS ARITY 1
ENDFUNCTION
```

```
FUNCTION TheObject
  HAS ARITY 2
ENDFUNCTION

FUNCTION ContextFor
  HAS ARITY 1
ENDFUNCTION

FUNCTION SetupResolve
  HAS ARITY 0
  IS STATIC
  EXPRESSION "SETUP_RESOLVE"
ENDFUNCTION

FUNCTION CompleteResolve
  HAS ARITY 0
  IS STATIC
  EXPRESSION "COMPLETE_RESOLVE"
ENDFUNCTION

FUNCTION AttemptResolve
  HAS ARITY 0
  IS STATIC
  EXPRESSION "ATTEMPT_RESOLVE"
ENDFUNCTION

FUNCTION ResolveStep
  HAS ARITY 1
ENDFUNCTION

FUNCTION Sources
  HAS ARITY 1
ENDFUNCTION

FUNCTION CurrentSource
  HAS ARITY 1
ENDFUNCTION
```

```
/* Set up binding space organization */
INIT
  EXTEND (bindingspace, 3, "b") WITH
    EXTEND (object, 2, "o") WITH
      EXTEND (name, 4, "n") WITH
        EXTEND (line, 2, "l") WITH
          TOOLDOTC := Select("n#0");
          TOOL := Select("n#1");
          TD := Select("l#0");
          IA := Select("l#1");
          /* src/tool.C */
          LookUp (Select("b#2"), Select("n#0")) := Select("o#0");
          /* ./bin */
          LookUp (Select("b#0"), Select("n#2")) := Select("b#1");
          /* ./src */
          LookUp (Select("b#0"), Select("n#3")) := Select("b#2");
          /* bin/tool */
          LookUp (Select("b#1"), Select("n#1")) := Select("o#1");
        ENDEXTEND
      ENDEXTEND
    ENDEXTEND
    Cwd := Select("b#0");
    Bin := Select("b#1");
    Src := Select("b#2");
    ContextDirective (Select("l#0")) :=
                        MakePair (da_cf, Cons (Select("b#0"),
                                          Cons (Select("b#1"),
                                          Cons (Select("b#2"),
                                          Nil))));
    ContextDirective (Select("l#1")) :=
                        MakePair (ia_cf, Cons (Select("b#2"),
                                          Nil));

  ENDEXTEND

ENDINIT
```

```
PROGRAM

/* Some Macros */
#define Append Cons
#define First Car
#define Rest Cdr

/* Macro: ContextDefinedBy */
#define ContextDefinedBy (theContext,  theEpoch, theLine)
   IF Equal (theEpoch, GetEpoch (ContextDirective (theLine))) THEN
     theContext := GetSources (ContextDirective (theLine));
   ENDIF


/* Macro: CompleteResolution */
#define CompleteResolution (name1, name2)
   IF And (Equal (ResolveStep (name1), CompleteResolve),
          Equal (ResolveStep (name2), CompleteResolve)) THEN
     ResolveStep (name1) := SetupResolve;
     ResolveStep (name2) := SetupResolve;
     CurrentEpoch := NextEpoch (CurrentEpoch);
   ENDIF

/* Macro: Resolve */
#define Resolve(theContext, theName, theObject)
   IF Equal (ResolveStep (theName), SetupResolve) THEN
     Sources(theName) := theContext;
     CurrentSource(theName) := First (theContext);
     ResolveStep(theName) := AttemptResolve;
   ELSEIF Equal (ResolveStep(theName), AttemptResolve) THEN
      IF Equal(Sources(theName), Nil) THEN
        theObject := Undef;
        ResolveStep(theName) := CompleteResolve;
      ELSEIF NotEqual(Sources(theName), Nil) THEN
        IF Equal (LookUp (CurrentSource(theName),theName), Undef) THEN
          CurrentSource(theName) := First (Rest (Sources(n)));
          Sources(theName) := Rest (Sources (theName));
        ELSEIF NotEqual (
             LookUp (CurrentSource(theName), theName),
             Undef) THEN
          theObject := LookUp (CurrentSource (theName), theName);
          ResolveStep (theName) := CompleteResolve;
        ENDIF
      ENDIF
    ENDIF
```

```
/* Set up the EA */
IF Equal (CurrentEpoch, startup) THEN
  CurrentEpoch := da_cf;
  ResolveStep(TOOL) := SetupResolve;
  ResolveStep(TOOLDOTC) := SetupResolve;
    GetEpoch(ContextDirective(TD)) := da_cf;
    GetSources(ContextDirective(TD)) :=
           Append (Cwd,
                   Append (Bin,
                           Append (Src,
                           Nil)));
    GetEpoch(ContextDirective(IA)) := ia_cf;
    GetSources(ContextDirective(IA)) := Append (Src,Nil);
ENDIF

/* Dependency Analysis, Context Formation */
IF Equal (CurrentEpoch, da_cf) THEN
  ContextDefinedBy(ContextFor(TD), da_cf, TD)
  CurrentEpoch := NextEpoch (CurrentEpoch);

/* Dependency Analysis, Name Resolution */
ELSEIF Equal (CurrentEpoch, da_nr) THEN
  Resolve (ContextFor(TD), TOOL, TheObject(da_cf, TOOL))
  Resolve (ContextFor(TD), TOOLDOTC, TheObject (da_cf, TOOLDOTC))
  CompleteResolution (TOOL, TOOLDOTC)

/* Invoke Action, Define Binding */
ELSEIF Equal (CurrentEpoch, ia_db) THEN
  EXTEND (object, 1, "newfile") WITH
      LookUp (Src,  TOOL) := Select("newfile#0");
  ENDEXTEND
  CurrentEpoch := NextEpoch (CurrentEpoch);

/* Invoke Action, Context Formation */
ELSEIF Equal (CurrentEpoch, ia_cf) THEN
  ContextDefinedBy(ContextFor(IA), ia_cf, IA)
  CurrentEpoch := NextEpoch (CurrentEpoch);

/* Invoke Action, Name Resolution */
ELSEIF Equal (CurrentEpoch, ia_nr) THEN
  Resolve (ContextFor(IA), TOOL, TheObject (ia_cf, TOOL))
  Resolve (ContextFor(IA), TOOLDOTC, TheObject (ia_cf, TOOLDOTC))
  CompleteResolution (TOOL, TOOLDOTC)
```

```
/* Check Contexts, Name Resolution */
ELSEIF Equal (CurrentEpoch, ccc) THEN
  IF Equal (ContextFor (IA), Undef) THEN
     CurrentEpoch := NextEpoch (CurrentEpoch);
  ELSEIF And (Equal (TheObject (da_cf, TOOL),
                     TheObject(ia_cf, TOOL)),
              Equal (TheObject (da_cf, TOOLDOTC),
                     TheObject(ia_cf, TOOLDOTC))) THEN
     CurrentEpoch := NextEpoch (CurrentEpoch);
  ELSE
     CurrentEpoch := hc;
  ENDIF
ENDIF

ENDPROGRAM
```

BIBLIOGRAPHY

[1] Andrews, T. Designing linguistic interfaces to an object database or what do C++, SQL and Hell have in common? In *Fourth International Workshop on Database Programming Languages*, New York, NY, Aug–Sep 1993. (Invited Talk).

[2] Apollo Computer, Inc., Chelmsford, MA. *Network Computing System: A Technical Overview*, 1989.

[3] Atkinson, M. Persistent programming practices. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 352–353, San Miniato, Italy, 1992.

[4] Atkinson, M., Buneman, P., and Morrison, R. Binding and type checking in database programming languages. *The Computer Journal*, 31(2):99–109, Feb. 1988.

[5] Atkinson, M. and Morrison, R. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, Oct. 1985.

[6] Atkinson, M. and Morrison, R. Polymorphic names, types, constancy and magic in a type secure persistent object store. In *Proceedings of the Second International Workshop on Persistent Object Systems*, pages 1–12, Appin, Scotland, Aug. 1987.

[7] Atkinson, M. and Morrison, R. Types, bindings and parameters in a persistent environment. In *Data Types and Persistence*, pages 3–20. Springer-Verlag, 1988. (*Proceedings of the First International Workshop on Persistent Object Systems*, Appin, Scotland, August, 1985).

[8] Banerjee, J., Kim, W., Kim, H.-J., and Forth, H. K. Semantics and implementation of schema evolution in object-oriented languages. In *Proceedings of the ACM SIGMOD Annual Conference*, pages 311–322, San Francisco, CA, July 1987.

[9] Barbedette, G. Schema modifications in the LISPO_2 persistent object-oriented language. In *Proceedings of the Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July 1991.

[10] Berners-Lee, T., Cailliau, R., and Groff, J.-F. World-wide web: The information universe. *Electronic networking*, 2(1), Spring 1992.

[11] Berners-Lee, T., Fielding, R., and Frystyk, H. Hypertext transfer protocol http/1.0. INTERNET-DRAFT.HTTP Working Group, Feb. 1996.

[12] Bershad, B., Ching, D., Lazowska, E., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, Aug. 1987.

[13] Birrel, A. and Nelson, B. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2(1):39–59, Feb. 1984.

[14] Birrell, A., Levin, R., Needham, R., and Schroeder, M. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, Apr. 1982.

[15] Blaine, L. and Goldberg, A. Interoperability of abstract data values. Technical Note 6, DARPA Module Interconnection Formalism Working Group, Jan. 1991.

[16] Boehm, B. W. and Scherlis, W. L. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference*, pages 63–82, Los Angeles, CA, Apr. 1992.

[17] Börger, E., Gurevich, Y., and Rosenzweig, D. The bakery algorithm: Yet another specification and verification. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[18] Börger, E. and Rosenzweig, D. A mathematical definition of full Prolog. *Science of Computer Programming*, 1994.

[19] Bright, M., Hurson, A., and Pakzad, S. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50–60, Mar. 1992.

[20] Buhr, P. and Zarnke, C. Persistence in an environment for a statically-typed programming language. In *Proceedings of the Second International Workshop on Persistent Object Systems*, pages 317–336, Appin, Scotland, Aug. 1987.

[21] Burge, T. Reference and proper names. *Journal of Philosophy*, 70:435–439, Aug. 1973.

[22] Butterworth, P., Otis, A., and Stein, J. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, Oct. 1991.

[23] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report (revised). Research Report 52, Digital, Palo Alto, CA, Nov. 1989.

[24] Cardelli, L. and Wegner, P. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

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

[26] Carroll, J. M. Creative names for personal files in an interactive computing environment. *International Journal of Man-Machine Studies*, 16:405–438, 1982.

[27] Cattell, R., editor. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, 1994.

[28] Cattell, R., editor. *The Object Database Standard: ODMG 93, Release 1.1*. Morgan Kaufmann, 1994.

[29] Cheriton, D. R. and Mann, T. P. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7(2):147–183, May 1989.

[30] Church, A. *The Calculi of the Lambda-Conversion*. Princeton University Press, Princeton, NJ, 1941.

[31] Comar, C. *GNU-NYU Ada95 Translator/C++ Low-Level Interface*. Department of Computer Science, New York University, New York, NY, Nov. 1994.

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

[33] Connor, R., Brown, A., Cutts, Q., Dearle, A., Morrison, R., and Rosenberg, J. Type equivalence checking in persistent object systems. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 154–167, Martha's Vineyard, MA, Aug. 1990.

[34] Daley, R. and Neumann, P. A general-purpose file system for secondary storage. In *AFIPS Conference Proceedings, Fall Joint Computer Conference*, volume 27, pages 213–230, Washington, D.C., 1965.

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

[36] Dearle, A. and Brown, A. Safe browsing in a strongly typed persistent environment. *The Computer Journal*, 31(6):540–544, Apr. 1988.

[37] Dearle, A., Cutts, Q., and Kirby, G. Browsing, grazing and nibbling persistent data structures. In Rosenberg, J. and Koch, D., editors, *Proceedings of the Third International Workshop on Persisent Object Systems*, pages 56–69, Newcastle, Australia, Jan. 1989.

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

[39] DOD. *Reference Manual for the Ada Programming Language*. United States Department of Defense, Washington, D.C., Jan. 1983. (ANSI/MIL-STD 1815A).

[40] Donnellan, K. S. Proper names and identifying descriptions. In Davidson, D. and Harman, G., editors, *Semantics of Natural Language*, pages 356–379. D. Rediel Company, Dordrecht-Holland, 1972.

[41] Ellis, M. A. and Stroustrup, B. *The annotated C++ reference manual*. Addison-Wesley, Reading, MA, 1990.

[42] Farkas, A., Dearle, A., Kirby, G., Cutts, Q., Morrison, R., and Connor, R. Persistent program construction through browsing and user gesture with some typing. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 375–394, San Miniato, Italy, 1992.

[43] Feldman, S. Make – a program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–265, Apr. 1979.

[44] Forman, I. R., Danforth, S., , and Madduri, H. Composition of before/after metaclasses in som. *"AIXpert"*, May 1995.

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

[46] Frege, G. On sense and nominatum. In Freigl, H. and Sellars, W., editors, *Readings in Philosophical Analysis*. Appleton-Century-Crofts, New York, 1949.

[47] Ghezzi, C., Mandrioli, D., Morasca, S., and M.Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, Feb. 1991.

[48] Gibbons, P. A stub generator for for multilanguage rpc in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, Jan. 1987.

[49] Gottlob, G., Kappel, G., and Schrefl, M. Semantics of object-oriented data models - The evolving algebra approach. In Schmidt, J. and Stogny, A., editors, *Next Generation Information System Technology, First International East/West Database Workshop*, number 504 in Lecture Notes In Computer Science, pages 144–160, Kiev, USSR, Oct. 1990.

[50] Gurevich, Y. Logic and the challenge of computer science. In Börger, E., editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

[51] Gurevich, Y. Evolving algebras 1993: Lipari guide. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[52] Gurevich, Y. and Morris, J. Algebraic operational semantics and Modula-2. In Börger, E., Büning, H. K., and Richter, M., editors, *First Workshop on Computer Science Logic*, number 329 in Lecture Notes In Computer Science, pages 81–101, 1988.

[53] Hanson, S. J., Kraut, R. E., and Farber, J. M. Interface design and multivariate analysis of UNIX command use. *ACM Transactions on Office Information Systems*, 2(1):42–57, Mar. 1984.

[54] Harrison, W. and Ossher, H. Subject-oriented programming (a critique of pure objects). In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, Washington, D.C., Oct. 1993. (SIGPLAN Notices, 28 (10), 1993).

[55] Hauzeur, B. M. A model for naming, addressing, and routing. *ACM Transactions on Office Information Systems*, 4(4):293–311, Oct. 1986.

[56] Hayes, R., Manwieler, S., and Schlichting, R. A simple system for constructing distributed, mixed language programs. *Software-Practice and Experience*, 18(7):641–660, July 1988.

[57] Heimbigner, D. and McLeod, D. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.

[58] Howard, J., Kazar, M., Manees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[59] Huggins, J. Kermit: Specification and verification. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[60] Huggins, J. and Mani, R. The evolving algebra interpreter, version 2.0. University of Michigan, 1995.

[61] International Standard for Information Processing Systems. *Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, 1995. 8824-1.

[62] International Telegraph and Telephone Consultative Committee (CCITT). *The Directory, Recommendations X.500, X.501, X.509, X.511, X.518-X.521*. Dec. 1988.

[63] Janssen, B., Severson, D., and Spreitzer, M. *ILU Reference Manual*. Xerox Corporation, Palo Alto, CA, 1.8 edition, May 1995.

[64] Janssen, B. and Spreitzer, M. ILU: Inter-language unification via object modules. In *Workshop on Multi-Language Object Models*, Portland, OR, Aug. 1994. (in conjunction with OOPSLA'94).

[65] Johnston, J. The contour model of block structured processes. *SIGPLAN Notices*, 6(2):55–82, June 1971.

[66] Jones, J., Rashid, R., and Thompson, M. Matchmaker: an interface specification language for distributed processing. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 225–235, Jan. 1985.

[67] Jones, N. D. and Muchnick, S. S. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*. Number 66 in Lecture Notes in Computer Science. Springer-Verlag, 1978.

[68] Katzenberg, B. and Peila, P. Work language analysis and the naming problem. *Communications of the ACM*, 36(4):86–92, June 1993.

[69] Keene, S. E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[70] Keller, D. A guide to natural naming. *SIGPLAN Notices*, 25(5):95–102, May 1990.

[71] Khoshafian, S. N. and Copeland, G. P. Object identity. In Zdonik, S. B. and Maier, D., editors, *Readings in Object-Oriented Database Systems*, chapter 1, pages 37–46. Morgan Kaufmann, San Mateo, CA, 1990.

[72] Klug, M. Visicola, a model and a language for visibility control in programming languages. *SIGPLAN Notices*, 26(2):51–63, Feb. 1991.

[73] Knudsen, J. L. Name collision in multiple classification hierarchies. In *Proceedings of the European Conference on Object Oriented Computing*, number 322 in Lecture Notes in Computer Science, pages 93–109, Oslo, Norway, Aug. 1988.

[74] Korn, D. G. and Krell, E. A new dimension for the Unix file system. *Software-Practice and Experience*, 20(S1):S1/19–S1/34, June 1990.

[75] Kripke, S. A. *Naming and Necessity*. Harvard University Press, Cambridge, MA, 1980.

[76] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, Oct. 1991.

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

[78] Lampson, B. W. Designing a global name service. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 1–10, Calgary, Alberta, Canada, Aug. 1986. ACM.

[79] Landin, P. J. A correspondence between algol 60 and church's lambda-notation. *Communications of the ACM*, 8(2):89–101, Feb. 1967. Part I.

[80] Landin, P. J. A correspondence between algol 60 and church's lambda-notation. *Communications of the ACM*. 9(3):158–165, Mar. 1967. Part II.

[81] Litwin, W. and Abdellatif, A. Multidatabase interoperability. *IEEE Computer*. 19(12):10–18, Dec. 1986.

[82] Loomis, M. E. S. *Object databases : the essentials*. Addison-Wesley, Reading, MA, 1995.

[83] Lucid, Inc. *Lucid Common Lisp/Sun, Version 4.0. Advanced User's Guide*. first edition, 1991.

[84] Maybee, M. J., Heimbigner, D. H., and Osterweil, L. J. Multilanguage interoperability in distributed systems. In *Proceedings of the International Conference on Software Engineering*, Mar. 1996.

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

[86] Meyer, B. *Eiffel: The Language*. Prentice Hall Inc., 1992.

[87] Microsoft Corporation. *OLE 2 Programmer's Reference Library: Vol. 1 Working with Windows(R) Objects*.

[88] Milner, R. Elements of interaction. *Communications of the ACM*. 36(1):78–90, Jan. 1993.

[89] Minsky, N. Law-governed systems. *Software Engineering Journal*. 6(5):285–302, Sept. 1991.

[90] Mockapetris, P. V. Domain Names - Implementation and specification. RFC 1035, Network Working Group, Nov. 1987.

[91] Mockapetris, P. V. and Dunlap, K. Development of the domain name system. In *ACM SIGCOMM*, pages 123–133, Stanford, CA, Aug. 1988.

[92] Morgan, C. and Sufrin, B. Specification of the UNIX filing system. *IEEE Transactions on Software Engineering*. SE-10(2):128–142, 1984. Also in Specification Case Studies.

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

[94] Morrison, R., Brown, F., Connor, R., Cutts, Q., Dearle, A., Kirby, G., and Munro, D. *The Napier88 Reference Manual (Release 2.0)*. University of St. Andrews, Nov. 1993. (CS/93/15).

[95] Moss, J. E. B. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*. 18(8):657–673, Aug. 1992.

[96] Moss, J. E. B. and Hosking, A. L. Expressing object residency optimizations using pointer type annotations. In *Sixth International Workshop on Persistent Object Systems*, Tarascon, Provence, France, Sept. 1994.

[97] Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. R. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[98] Odberg, E. MultiPerspectives: The classification dimension of schema modification management for object-oriented databases. In *Proceedings of TOOLS USA*, Santa Barbara, CA, Aug. 1994.

[99] Oman, P. W. and Cook, C. R. A programming style taxonomy. *The Journal of Systems and Software*, 15(3):287–301, July 1991.

[100] OMG. Object management architecture guide, revision 2.0. OMG TC Document 92.11.1, Object Management Group, Framingham, MA, Sept. 1992.

[101] Open Systems Foundation, Cambridge, MA. *OSF DCE 1.0 Application Development Reference*, Jan. 1992.

[102] Oppen, D. and Dalal, Y. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, July 1983.

[103] Partee, B. H. Binding implicit variables in quantified contexts. In *Parasession on Language in Context*, Papers from the 25th annual regional meeting of the Chicago Linguistic Society, 1989.

[104] Penney, D. J. and Stein, J. Class modification in the GemStone object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 111–117, Orlando, FL, Oct. 1987.

[105] Pike, R., Presotto, D., Thompson, K., Trickey, H., and Winterbottom, P. The use of names in Plan 9. *Operating Systems Review*, 27(2):72–76, Apr. 1993.

[106] Popek, G., Walker, B., Chow, J., Edwards, D., Rudisin, G., and Theil, G. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 169–177, Pacific Grove, CA, Dec. 1981.

[107] Purtilo, J. M. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, Jan. 1992.

[108] Radia, S. *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, Jan. 1990. UW/ICR 90–01.

[109] Reiss, S. P. Generation of compiler symbol processing mechanisms from specifications. *Transactions on Programming Languages and Systems*, 5(2):127–163, Apr. 1983. General approach for describing symbol table processing for programming languages.

[110] Reiss, S. P. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–67, July 1990.

[111] Rice, M. and Seidman, S. A formal model for module interconnection languages. *IEE Transactions on Software Engineering*, 20(1):88–101, Jan. 1994.

[112] Richardson, J. E., Carey, M. J., and Schuh, D. T. The design of the E language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[113] Rowe, L. A. Programming language issues for the 1980's. *SIGPLAN Notices*, 19(8):51–61, Aug. 1984.

[114] Russell, B. On denoting. *Mind*, 14:479–493, 1905.

[115] Saltzer, J. Naming and binding of objects. In *Operating Systems: An Advanced Course*, number 60 in Lecture Notes in Computer Science, chapter 3A, pages 99–208. Spring-Verlag, 1978.

[116] Schmidt, J. and Matthes, F. Naming schemes and name space management in the DBPL persistent storage system. In *Proceedings of the Fourth International Workshop on Persisent Object Systems*, pages 39–58, Sept. 1990.

[117] Schroeder, M., Birrell, A., and Needham, R. Experience with Grapevine: Growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, Feb. 1984.

[118] Scott, D. and Strachey, C. Towards a mathematics semantics of computer languages. In Fox, J., editor, *Proceedings of the Symposium on Computres and Automata*, pages 19–46, New York, 1971. Polytechnic Institute of Brooklyn Press. also Technical Monograph PRG-6, Oxford University Computing Laboratory, Programming Research Group, 1971.

[119] Searle, J. Proper names. *Mind*, 67:166–173, 1958.

[120] Sheil, B. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, Mar. 1981.

[121] Sheltzer, A., Lindell, R., and Popek, G. Name service locality and cache design in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 515–522. IEEE, Computer Society Press, May 1986.

[122] Shoch, J. F. Inter-network naming, addressing, and routing. In *COMPCON*, pages 72–79, Washington, D.C., Sept. 1978.

[123] Simonyi, C. and Heller, M. The Hungarian revolution. *Byte*, 16(8):131–138, Aug. 1991.

[124] Skarra, A. H. and Zdonik, S. B. The management of changing types in an object-oriented database. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 483–495, Portland, OR, Sept. 1986.

[125] Spivey, J. *The Z Notation: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1980.

[126] Stallman, R. M. and McGrath, R. *GNU Make: A Program For Directing Recompilation*. Free Software Foundation, Cambridge, MA, 0.47, for make version 3.72q beta edition, Nov. 1994.

[127] Stemple, D. and Morrison, R. Specifying flexible concurrency control schemes: An abstract operational approach. In *Proceedings of the 15th Australian Computer Science Conference*, pages 873–891, Hobart, Tasmania, 1992.

[128] Stoy, J. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[129] Strachey, C. Towards a formal semantics. In Steele, T. B., editor, *Formal Description Languages for Computer Programming*, pages 198–220. North-Holland Publishing Company, Amsterdam, 1966.

[130] Strachey, C. *Fundamental Concepts in Programming Languages*. International Summer School in Computer Programming, Copenhagen, 1967.

[131] Stroustrup, B. *The design and evolution of C++*. Addison-Wesley, 1994.

[132] Sun Microsystems. *Network Programming*, May 1988. Part Number: 800-1779-10, Revision A.

[133] Sun Microsystems, Mountain View, CA. *Network Programming Guide*, revision a edition, Mar. 1990.

[134] Sun Microsystems, Inc., Mountain View, CA. *SPARCompiler C++ 3.0.1 Programmer's Guide*, revision a edition, Oct. 1992.

[135] Tarr, P., Wileden, J., and Clarke, L. Extending and limiting PGRAPHITE-style persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 74–86, Martha's Vineyard, MA, Aug. 1990.

[136] Tarr, P., Wileden, J., and Wolf, A. A different tack to providing persistence in a language. In Hull, R., Morrison, R., and Stemple, D., editors, *Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.

[137] Terry, D. An analysis of naming conventions for distributed computer systems. In *SIGCOMM Tutorials and Symposium, Communications Architecture and Protocols*, pages 218–224, Montreal, Quebec, June 1984. ACM.

[138] Texas Instruments, Inc., Dallas, TX. *Open OODB Query Language User Manual*, release 0.2 (alpha) edition, 1993.

[139] Thomas, E. A bibliography of programming style. *SIGPLAN Notices*, 25(2):7–16, Feb. 1990.

[140] Touretzky, D. S. and Lee, P. Visualizing evaluation in applicative langauges. *Communications of the ACM*, 35(10):49–59, Oct. 1992.

[141] Triantafilllou, P. and Bauer, M. Distributed name management in internet systems: A study of design and performance issues. *Journal of Parallel and Distributed Computing*, 9:357–368, Aug. 1990.

[142] Vorthmann, S. and LeBlanc, R. J. A naming specification language for syntax-directed editors. In *International Conference on Computer Languages*, pages 250–257, Miami Beach, FL, Oct. 1988. IEEE, IEEE Computer Society Press.

[143] Wallace, C. The semantics of the C++ programming language. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[144] Watson, R. W. Identifiers (naming) in distributed systems. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, number 105 in Lecture Notes in Computer Science, chapter 9, pages 191–210. Spring-Verlag, 1981.

[145] Wells, D. L., Blakely, J. A., and Thompson, C. W. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, Oct. 1992.

[146] Wiederhold, G. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, Mar. 1992.

[147] Wiederhold, G., Wegner, P., and Ceri, S. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, Nov. 1992.

[148] Wileden, J., Wolf, A., Fisher, C., and Tarr, P. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of the Third Symposium of Software Development Environments*, pages 130–142, Sept. 1988.

[149] Wileden, J. C. and Wolf, A. L. Object management technology for environments: Experiences, opportunities and risks. In *Proceedings of the International Workshop on Environments*, number 742 in Lecture Notes in Computer Science, Sept. 1989.

[150] Wileden, J. C., Wolf, A. L., Rosenblatt, W. R., and Tarr, P. L. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

[151] Wirth, N. *Programming in MODULA-2*. Springer–Verlag, second edition, 1983.

[152] Wobber, E. P., Abadi, M., Burrows, M., and Lampson, B. Acm transactions on computer systems. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.

[153] Wolf, A., Clarke, L., and Wileden, J. The AdaPIC Tool Set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, Mar. 1989.

[154] Wolf, A. L., Clarke, L. A., and Wileden, J. C. A model of visibility control. *IEEE Transactions on Software Engineering*, 14(4):512–520, Apr. 1988.

[155] Yeo, A., Ananda, A., and Koh, E. A taxonomy of issues in name systems design and implementation. *Operating Systems Review*, 27(3):4–18, July 1993.

[156] Younger, E. and Bennett, K. Functional approach to mathematical model of naming. *Information and software technology*, 34(3):185–191, Mar. 1992.

[157] Zaremski, A. M. and Wing, J. M. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2), Apr. 1995.

[158] Zaremski, A. M. and Wing, J. M. Specification matching of software components. In *The Third Symposium on the Foundations of Software Engineering*, pages 6–17, Washington, D.C., Oct. 1995.