Univers: The Construction Of An Internet-Wide Descriptive Naming System

C. Mic Bowman

TR 90-27

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA

Univers: The Construction Of An Internet-Wide Descriptive Naming System

C. Mic Bowman

TR 90-27

Abstract

Descriptive naming systems allow clients to identify a set objects by description. In a world where information is perfect, this amounts to a simple database query. However, descriptive naming systems operate in a very imperfect world: clients may provide inaccurate descriptions, the database may contain out-of-date or incomplete information, the database may be highly distributed, and so on.

This paper describes the construction of a descriptive naming system, called Univers, based on a model in which clients provide both an object description and some meta-information. The meta-information describes beliefs about the query and the naming system. Specifically, it is an ordering on a set of perfect-world approximations and it describes the preferred methods for accommodating imperfect information. The description is then resolved in a way that respects the preferred approximations.

August 10, 1990

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Contents

List of Figures

List of Tables

1 INTRODUCTION

Naming systems exist in nearly all computer systems. A compiler uses the naming services provided by a symbol table to map program identifiers into memory locations [Pric71]. An operating system relies on virtual memory hardware to translate virtual addresses into physical addresses [Fabr74], and on a directory service to map file names to disk addresses [Ritc74]. Computer networks support hostname translators that return the network address of a host in the network [Mock87] and white-pages directory services that determine the mailbox associated with a user name [Pete88]. In each of these systems, the naming system provides the same basic functionality: given a *name*, a naming system is a mechanism that answers queries of the form "what resource is denoted by this name?". A naming system resolves names relative to a collection of *contexts*, each of which contains a related group of name-to-resource bindings. The scope of the bindings in a context extends to a single administrative domain such as a compiler, a processor, or an administrative organization.

Although abstractly similar, there are significant differences between various naming systems. These differences are the result of the constraints placed on the naming system by the environment in which it must operate. For example, the environment associated with a compiler tightly constrains the symbol table to store information about the variable associated with an identifier and to resolve every name unambiguously. An operating system places similar constraints on a virtual memory mapper. In contrast, to satisfy system requirements, a directory service requires more functionality than is provided by the naming systems in a compiler or a virtual memory manager. For example, a directory service supports operations that resolve several different names into the same resource . As another example, a white-pages service provides an interface that allows a client to express all of the information that it possesses, but the service can respond to a client's query with ambiguous answers.

This thesis describes a naming system that manages the unique set of constraints associated with one particular environment: a large *internet*. Functionally, an internet provides a global communication facility for an aggregation of independent computer networks. Administratively, an internet allows a collection of autonomous and mutually suspicious organizations to share information and resources. A naming system for an internet environment must operate within the requirements of the organizations that provide information and of the clients that attempt to access it. In this environment, a naming system is a confederation of contexts, each of which is responsible for maintaining a collection of information for one of the organizations that participate in the internet. Each organization can have different policies for the type of information that is made available and for the methods used to access it. The information in a context describes the various resource classes—such as human users, processors, and printers—that are of special interest to the administrative organization. Clients—both humans and programs—use the naming system to discover and identify resources that exist in the internet. Each client has different requirements concerning the information used to identify a resource, the precision of the answer, and the time that the naming system may use to compute the answer.

1.1 The Problem: Satisfying System Constraints

An internet-wide naming system must accommodate three sets of constraints: the requirements placed on the naming system by the organizations that administer it, the requirements placed on the naming system

by its clients, and the requirements placed on the naming system by the characteristics of the information it contains. In satisfying these constraints, a fundamental conflict occurs between performance—the ability of a naming system to respond to queries within a reasonable amount of time—and functionality—the amount of expressiveness provided by the naming system. For each set of constraints, the design of the naming system reflects tradeoffs between performance and functionality. The remainder of this section describes how these constraints affect naming in an internet environment and contrasts these constraints with those that affect other naming systems.

Administrative Constraints

Every context in a naming system operates according to a set of requisite principles determined by the responsible administrative organization. The context must provide the services that the organization deems necessary, yet it must limit access to information that the organization deems inappropriate for other internet participants. For example, an organization may wish to use the internet environment to provide access to technical support for a software product. The context administered by this organization must contain information that allows clients to access the technical support service, such as a list of phone numbers or electronic mail addresses for the technical support staff. At the same time, the company may wish to limit access to some information; e.g. the phone number of the company president. A different organization may place a completely different set of requirements on the naming system.

Autonomous administration of each context accommodates the requirements placed on the naming systems by each participating organization. It allows the organization to determine what information will be available in the context, how it will be represented, and how it will be accessed. A context may contain information about a wide variety of resources—e.g. files, hosts or users—and for each resource, the context may maintain information about several different properties—e.g. mnemonic name, mailbox address, or architecture. Operationally, a context may support a set of name resolution functions that operates on different resource classes or provides different degrees of precision of answers.

Although administrative autonomy limits a client's ability to access information about resources in the internet, a set of contexts must support a common base upon which they can cooperate to provide an internet-wide naming service. Traditionally, naming systems have solved this problem by limiting the number of services provided by the naming system or by mandating a single administrative authority that ensures conformance to a set of system-wide conventions; e.g. an operating system acts as the only administrative authority responsible for a directory service for files. In a naming system where conventions and services may vary widely from one context to the next, clients must be able to query a context to learn the conventions it uses and the services it supports. There must be a common mechanism for discovering the conventions used by a context—a set of meta-conventions that specify the conventions used by a particular context and a set of primitive operations that access these meta-conventions. For example, a client might want to discover the functions in a context that resolve names associated with a certain class of resources, or to learn the meaningful properties associated with a given resource class. Thus, the context must support *a* well-defined interpretation of queries that allows clients to learn about any local conventions.

In summary, an internet-wide naming system may satisfy administrative constraints by providing

facilities for:

- Administrative autonomy among the organizations that participate in the naming system.
- A common foundation for the specification of the conventions used by each administrative organization.

Client Constraints

The clients of a naming system place a different set of requirements on the system: the naming system should accept as a name whatever information its clients possess for the resources they want to identify; it should respond with the precision expected by those clients; and it should respond within an acceptable amount of time. For example, the client of a symbol table—in this case another program—specifies an object with an identifier and expects at most one answer; a program that uses the symbol table cannot accommodate ambiguous answers. Furthermore, the client program, a compiler, generally expects rapid responses. The design of the symbol table reflects the need to respond quickly and unambiguously. In contrast, the client of a white-pages service—in this case a human user—often possesses several pieces of information about the object it wants to name and is sometimes willing to tolerate ambiguous results that take several seconds to compute, as long as the results contain the desired resource.

Generally, a client possesses information about the intrinsic properties of the resource it intends to identify such as architecture, speed, or available software. However, conventional naming systems sacrifice the ability to identify resources by the information that clients most often possess. Instead they focus on the need to satisfy the client's performance constraints. In order to respond to queries most efficiently, conventional naming systems presuppose some organization among the contexts participating in the service. According to Comer and Peterson [Come89], this organization, called a *name space*, is best thought of as a graph that coordinates communication between the contexts that constitute the naming system. A name represents a path through the name space. The name space may represent an actual physical network—called a *source-route name space*—or a virtual organization imposed on top of the physical network—such as a *hierarchical name space*.

If the name space represents a physical network, then a name corresponds to a path through the physical network. For example, in Figure 1 the name coatimundi!paloverde!cholla describes a path from gilamonster that passes through nodes coatimundi and paloverde before finally reaching the object cholla. A name space that corresponds to a virtual organization of the physical network is often hierarchical. A name corresponds to a path from one context—usually the root node—to another node—usually a leaf node. For example, Figure 2 shows a portion of the Domain Name System [Mock87]. The name cholla.cs.arizona.edu is a path from the root node to a representation for the resource cholla. Many conventional naming systems—such as the Domain Name System, the OSI/CCITT X.500 directory service [Int88], and the Unix file system [Ritc74]—impose some form of virtual hierarchy on top of a collection of contexts.

When a name identifies a path from one node to another in a name space graph, it is possible to locate and identify objects efficiently and precisely. However, this solution does not provide facilities for handling many of the functionality constraints placed on the system by its clients. If an object's name

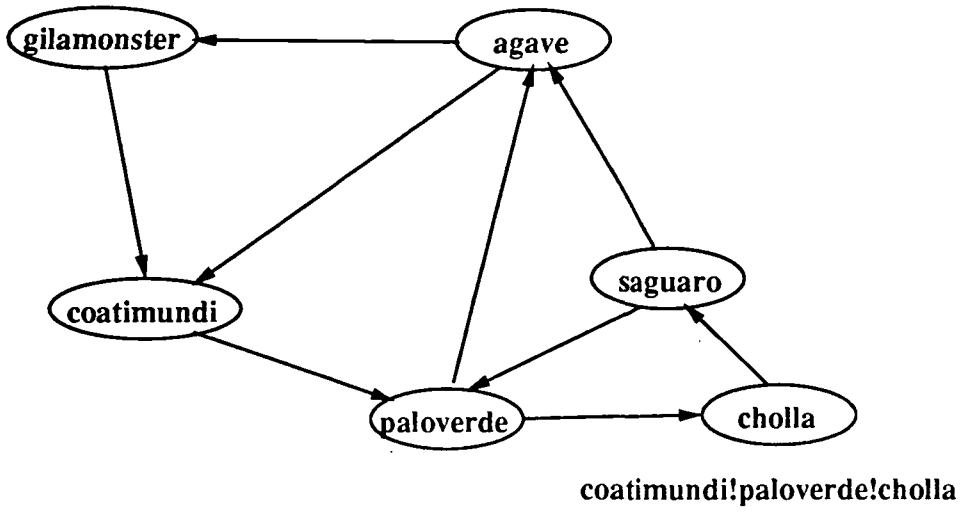

Figure 1: Source-route name space.

describes a path to the object through a graph then the name does not necessarily denote an inherent characteristic of the object. Instead, naming systems that identify a resource with a path through the name space place on the client the burden of understanding an artificial characteristic of the object—its location within the name space—that has no meaning and is generally not known outside the naming system. When the name represents a location-dependent path through a physical network, then the client must understand even more of the structure of the name space. For example, in Figure 1 gilamonster will not name cholla in the same way that saguaro might. In order to identify an object the client must understand the structure of the name space and the object's location within that structure. Whenever a name represents a path through some name space, the burden of understanding the structure of the name space falls on the client.

In contrast, the client of a software company that wants to locate the company's technical support staff often knows properties such as the name of the software product, the name of the company, and the company address; properties that identify the support staff resource independent of the naming systems in which it participates. The client is less likely to know that the name staff@techsup.wsp.com—a path through a hierarchical name space—identifies the technical support staff. If the naming system supports only this method for identifying objects then the client will be unable to locate the company's technical support staff. Consider a program that needs to locate a set of fast, lightly loaded processors; it will not be able to express the query using a path through some name space. The program should be able to identify the processors with the properties that it knows: the speed and the load. Names that contain information about the naming system may be resolved efficiently; however, the client sacrifices the freedom to present the information it naturally possesses for the objects it intends to identify. In order to accommodate the functionality requirements of its clients, the naming system should provide a method for describing objects by the properties that the clients are most likely to know.

In summary, an internet-wide naming system may satisfy user constraints by providing facilities for:

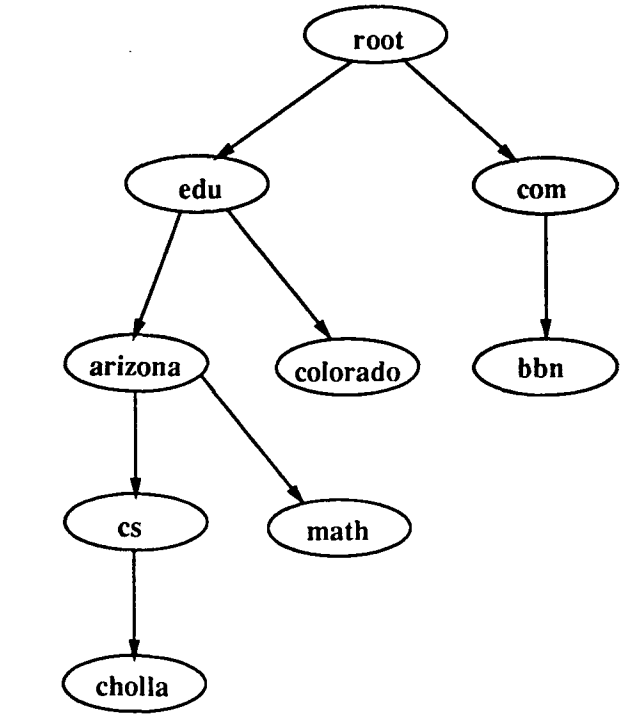

Figure 2: Hierarchical name space.

- An organization that allows the naming system to compute results within a reasonable amount of time.
- An interface that lets clients describe objects by their inherent characteristics.

Information Constraints

A naming system is constrained by the conditions under which it is implemented. In many cases, there are no limitations; the naming system exists in a "perfect world". For example, the parser for a compiler never fails to insert identifiers in the symbol table and the symbol table manager cannot fail independently of the entire compilation. In contrast, an internet-wide naming system must function in an imperfect world. The following characteristics of the internet naming environment negate the perfect data assumption.

First, a client may possess inaccurate information about the object it intends to identify. For example, a user might wish to find information about a person whose last name is "Clanton", whose first name is probably "Ike" and who is almost certainly in the Computer Science Department. The user would prefer information about any "Clanton" in Computer Science rather than information about some "Ike Clanton" not in Computer Science. Generally, a client knows when the information it possesses is potentially inaccurate.

Second, the information that a client specifies may vary in usefulness—portions of the client's information may provide more reasonable results. Consider a user that wants to identify a laser printer with a specific architecture that is located within a certain building. Intuitively, the client insists that the printer have the correct architecture—a file formatted for one architecture cannot be printed on a different architecture—but the client may not insist that the printer be located within the building—the file may be printed in another building within walking distance. The property that describes the printer's location is optional. Similarly, some information in a context may provide more efficient and precise responses to a query; i.e. it is more useful for resolving names. For example, serial numbers apply to at most one processor, but an architecture can describe several different processors.

Third, information within an internet-wide naming system may become out-of-date. Many properties, especially those for computational resources, change frequently. Communication delays between components in the system make it impossible to guarantee that the context contains up-to-date, accurate information about these properties. In general, if the value of a property changes more frequently than it is updated, then the information that the naming system uses may be out-of-date. Information that is guaranteed to be up-to-date can be used to answer queries more reliably than information that may change. For example a processor's serial number does not change as often as its load. Thus, a name that identifies an object by its serial number is more likely to provide accurate results than a name that identifies an object by its current load. When given a name that specifies both stable and quickly changing information, the naming system is more likely to produce the desired answer if it uses stable information in preference to dynamic information.

Finally, the completeness of the information known to the context may vary for different properties of objects. The database may not contain any information about some properties. For example, it is reasonable to believe that a naming system for an internet environment will not contain information about a person's food or clothing preferences. No system will contain every piece of information that its clients could use to identify the resources it administers. At the other extreme, some properties may be known for every applicable resource; e.g. a naming system may guarantee that the serial number of every processor is contained in the database. Other properties may be incomplete; i.e. a property may be contained in the database for some resources that it describes but not all. For example, a white-pages service can only resolve names that contain information put into the system by human users. If a user does not enter a particular fact in the naming system, then the system cannot respond to queries using that information. Also, naming systems that are implemented in distributed environments may lose information about some resources because independent components of the system fail.

In summary, an internet-wide naming system must be able to provide reasonable responses when the information in the system is imperfect. In particular, the information may be imperfect in any of the following ways:

- The information in a name can be inaccurate.
- The information in a name can vary in usefulness and importance.
- The value of a property contained in a context can be out-of-date.
- The value of a property can be unknown to the context.

1.2 The Solution: The Univers Naming System

Conventional naming systems are functionally simple: a context contains a table of name/value pairs, clients submit a single name, and the context returns the corresponding value. The simplicity of conventional naming systems exemplifies a decision to emphasize performance over functionality. Such conventional systems do not provide the power and flexibility to support general purpose naming in an internet environment.

This thesis describes a new naming system, called the *Univers* naming system, that focuses on the need to provide a powerful interface without neglecting performance issues. Univers satisfies administrative, client, and information constraints placed by a large internet environment. In particular, Univers supports a well-defined extension language that provides a foundation for communication between autonomously administered contexts. Univers provides a mechanism, called *descriptive naming*, that allows clients to identify an object using the object's intrinsic properties. When imperfect information exists in the system, the descriptive naming model is powerful enough to describe methods for computing preferred answers.

The Univers programmable name server is the fundamental building block of the Univers naming system. Each Univers name server implements one or more contexts by providing an infrastructure for binding and resolving naming information relevant to a collection of administrative organizations. In particular, a Univers name server consists of a database for storing information about resources, an interpreter that serves as a programmable interface to the database, and a server framework that supports remote access. Together, the database and interpreter implement the descriptive naming model.

To query the Univers naming system, a client submits a set of *attributes* and a *resolution function* that is to be applied to those attributes. For example, the query

```
(finduser ((= name "Ike Clanton") (= org "Computer Science")))
```

might be used to learn the attributes of a particular user. In this example, finduser is the name of a resolution function and (= name "Ike Clanton") is an example of an attribute. As another example, the query

```
(findprocessor ((= architecture 68020) (> mips 3) (< load 1.5)))
```

might be used to identify a processor with a 68020 architecture, a mips rating greater than 3, and a load less than 1.5. For both queries, the system replies with all of the attributes it possesses for each object identified by the query. For example, a Univers name server may return the following two sets of attributes, one for each processor matched by the attribute-based name:

```
(
  ( (name bodoni)
    (address 192.12.69.22)
    (architecture 68020)
    (load 1)
    (mips 4)
    (owner "Virgil Earp"))
  ( (name jenson)
    (address 192.12.69.33)
    (architecture 68020)
    (load 0)
    (mips 4)
    (owner "Doc Holiday"))
)
```

From the client's perspective, a descriptive naming system maintains a database of *objects*, each of which corresponds to either some "external" resource that exists outside the system—e.g. printers, processors, users, other name servers—or some "internal" abstraction that is meaningful only within the system itself. Each context in the system contains descriptions for a set of objects, usually those of special interest to the entity that administers that particular context. The description of an object consists of several attributes that denote specific properties of the object. For example, if a user is described by the property "phone-number is 621-1234" then the database may contain the attribute (phone-number 621-1234).

A descriptive naming system allows clients to identify objects by description; i.e. by a set of attributes and a resolution function. Each context in the naming system exports a set of resolution functions that allow clients to search for complete object descriptions given incomplete descriptions. For example, a client may learn a user's mailbox address by selecting a white-pages resolution function—one that provides access to human user resources—and specifying a description such as: (= name "Ike Clanton") and (= phone 621-1234). The naming system determines the set of objects described by these properties and returns a complete description—all of the information that the system possesses—for each object.

When information within the naming system is perfect—i.e. complete, accurate, and able to distinguish among a set of objects—then each resolution function corresponds to a simple database query. However, this is not the case in general. Instead, a preferred answer is computed using knowledge about the quality of information in the database and in the query. Specifically, clients provide the naming system with a description of an object and a resolution function that encapsulates some meta-information concerning the client's beliefs about the query and the naming system. This meta-information, called a *preference hierarchy*, orders a set of perfect-world approximations and it describes the preferred methods for accommodating imperfect information. The description is then resolved in a way that respects the preferred approximations. Each resolution function within the Univers naming system corresponds to a particular preference hierarchy.

The name space associated with the Univers naming system can be represented by an arbitrary directed graph. Each node in the graph corresponds to a context that participates in the naming system. An edge from one node to another node represents a communication path between the contexts. In particular, an edge in the name space graph from node x to node y implies that context x contains an object that

describes context **y** and that part of the description is a method for accessing information in **y**.

1.3 Research Contributions

The primary contribution of this thesis is a working demonstration of an internet-wide descriptive naming system. The construction of the system involves results in three different areas: a model that may be used to design and reason about naming systems, an attribute-based name server that implements the model, and a set of conventions that glues a collection of name servers together to form an internet-wide descriptive naming system. The conventions describe the contents of a name server and the methods used to connect that name server into the global name space.

We present a model, called the *preference hierarchy model*, that allows clients to define the meaning of preferred answers when the naming system responds to queries involving information that varies in usefulness. The preference hierarchy model provides insight into methods for solving problems such as imperfect information and decentralized, autonomous administration. It also provides a foundation for describing and comparing naming systems. Most importantly, the preference hierarchy model supports a naming paradigm that allows the construction of an internet-wide descriptive naming system. In addition to introducing the model and giving several examples of how it is used in the Univers descriptive naming system, we give a semantic foundation for the model. Chapter 3 describes the preference hierarchy model in detail.

The preference hierarchy model leads to a natural and easily optimized implementation, called the Univers name server. At the heart of each Univers name server is an inference mechanism that implements the preference hierarchy model. Unlike more conventional name servers that support a single name service, Univers is designed to be used as a building block from which naming systems may be constructed. Toward this end, Univers provides tools for communicating with other servers, for constructing resolution functions, and for extending and managing databases. Chapter 4 describes the facilities that Univers provides for constructing naming systems and the implementation of a Univers name server.

Using the tools provided by Univers, we describe the construction of an internet-wide descriptive naming system. This naming system supports an entire suite of resolution functions: white-pages functions used to locate system users, yellow-pages functions used to identify various types of computational resources, and functions that support conventional naming services. Specifically, we present a set of conventions for organizing the name space, methods for searching the name space, and resolution functions that handle various kinds of queries. Chapter 5 describes this system.

2 RELATED WORK

The ideas presented in this thesis bear a close relationship to both naming systems and database systems. This chapter discusses current research in descriptive and conventional naming systems, database management systems that handle imperfect information, extendible database management systems, and distributed database theory. In addition, this chapter surveys software systems that can be extended by defining new functions in an extension language.

2.1 Naming Systems

Conventional Naming Systems

Previous studies of naming focus on the operational aspects of conventional naming systems. For example, Fowler discusses the use of forwarding addresses [Fowl85]; Mann discusses methods for caching information in a decentralized naming system [Mann87]; The Grapevine [Birr82, Schr84] and Clearinghouse [Oppe83] naming systems, both developed at Xerox PARC, identify objects using a limited depth hierarchy; Lampson describes techniques for extending hierarchical naming to include unlimited hierarchical names that provide directory services within much larger environments [Lamp86]; Terry describes methods for using out-of-date information as a hint [Terr87]; and Mockapetris describes methods for naming objects within an internet using names that describe a path through an hierarchical graph of domains [Mock84, Mock87]. Each of these studies describes techniques for managing a conventional, decentralized naming service. They do not allow clients to identify objects by descriptive names.

X.500

The OSI/CCITT X.500 user directory service [Deut88, Int88] exhibits properties of both conventional and descriptive naming systems. An object is identified by a *distinguished name* that consists of a set of attributes as in other descriptive naming systems. However, the name does not necessarily contain attributes that describe the object being identified. Rather, the attributes in the name describe particular branches to be traversed in a global hierarchy. Each node in the hierarchy corresponds to an object and the object is identified by a path from the root node. This design places emphasis on the client's understanding of the hierarchy. In order to describe an object, a client must specify a complete, accurate description that properly traverses the nodes in the hierarchy—X.500 does not accommodate imperfect information. In order to alleviate some of the client's burden, X.500 provides tools for interactively searching the hierarchy. Neufeld extends X.500 to include true descriptive names—names that describe the object rather than a path through a hierarchy [Neuf89]. However, Neufeld's solution depends on locating objects relative to a hierarchy of contexts. A name contains attributes that describe a context object—such as the organization a person works for—as well as attributes that describe the object itself.

Networked Resource Discovery Project

The Networked Resource Discovery Project [Schw88] models the connections between autonomous information repositories in a way that resembles human social networks. The system is unique in that the resulting name space is organized as an arbitrary graph. Resources are discovered by asking an agent that either knows about the resource or knows an agent that knows about the resource. This model supports highly decentralized, autonomous system administration. Each participating organization may determine the level of access to provide without affecting access to any other organization. The name space used by the Univers naming system develops in a manner similar to NRDP; however, Univers makes explicit use of several embedded structures within the name space.

Heterogeneous Computer Systems Name Service

The Heterogeneous Computer Systems Name Service (HNS) developed at the University of Washington [Schw87] supports an extendible interface that allows a system administrator to augment the functionality of the naming system. The primary goal of HNS is to provide an umbrella server that hides differences between a heterogeneous collection of name servers. HNS supports an indirect naming mechanism in which a client consults a "name server clearinghouse" to learn the location of an agent (called a Naming Semantics Manager) that serves as a translator between the client and an existing, backend name server. In this way, HNS supports several different naming functions—one for each backend service. Similarly, a Univers name server can serve as a translator between representations that make use of a backend function. In addition, a Univers name server provides facilities for defining functions local to the server. This means that the functionality of a Univers server can change at run-time, making it more adaptable to the changing needs of a heterogenous computer system.

Centralized Directory Services

Two name servers, CSNET [Solo82] and the NIC [Pick79], provide centralized descriptive user directory services for internet participants. The CSNET name server defines a name resolution function in which a given set of attributes are partitioned into *mandatory* and *optional* subsets. Names are resolved giving preference to precise matches of the mandatory attributes. The NIC name server (also called WHOIS) limits queries to a single attribute and defines a resolution function that returns all objects that partially match the attribute. The NIC name server also enforces a restriction that an unambiguous attribute, called a *handle*, be registered for each object, such that if a client gives a handle, the naming system is guaranteed to return at most one object. Handles are implemented by attaching a unique prefix to a registered attribute so as to to ensure its uniqueness. Both the NIC and CSNET name server maintain a single context. That context is responsible for responding to queries and for maintaining the information in the database. Thus, NIC and CSNET have a single administrative authority and do not allow for distribution.

Profile

The work presented in this thesis is an outgrowth of the Profile naming system developed at the University of Arizona [Pete88]. Profile provides a suite of white-pages functions that access user descriptions in an autonomous confederation of name servers. A companion naming system provides a yellow-pages resolution function—one that allows clients to identify computational resources such as printers and processors [Pete87]. The Univers naming system is a generalization of the concepts that were used to design Profile. The preference hierarchy model helped us discover a flaw in the original specification of one of the functions and eventually rewrite the function to perform more like it was originally intended.

2.2 Database Systems

Descriptive naming is properly viewed as a highly specialized, extendible object database, where the process of resolving a name is conceptually similar to that of solving a database query [Gall84]. Special-

izations lie in two major areas. First, the goal of a descriptive naming system is to identify objects. This affects the system in several ways. A descriptive naming system always returns complete descriptions of objects. That is, it does not need the relational **join** operation that complicates the implementation of query optimization in database systems. A descriptive naming system works with small sets of objects. Most clients wish to identify only a single object or at most a small set of objects. Second, a descriptive naming system must attempt to answer queries that contain potentially inaccurate and incomplete information. In order to do this, attributes specified in a query may be removed or new ones may be added. Although some systems attempt to accommodate incomplete information in the database, very few database systems attempt to provide reasonable answers to client queries that can contain imperfect information.

Imperfect Information

Several recent research projects attempt to provide a theoretical basis for manipulating imperfect information within a database system. Two distinct approaches are taken. The first approach focuses on methods for representing missing information—information where the value of a property for a certain object is either known precisely or completely unknown. Codd extends the relational model [Codd79, Codd86, Codd87] to include null as an acceptable value. A three-value logic is used to respond to queries involving tuples with null values. Kocharekar [Koch89] further refines these concepts. Spyratos uses partition semantics of relational databases [Spyr87] to define the meaning of incomplete information [Laur88]. Demolombe discusses an algorithm for finding models of deductive databases that contain null values [Demo88].

The second approach examines methods for computing tight solution bounds given partial information; e.g. property A does not have value v_1 but may have value v_2 or v_3. Lipski focuses on methods for approximating solutions when partial information exists [Lips79, Lips81]. The model presented in this paper solves the problem of incomplete information by defining the meaning of a preferred answer, not by defining the meaning of missing information. Within a descriptive naming system, the client controls the methods used to handle imperfect information; it is not built into the system by its designers.

The solutions proposed by Codd and Lipski share a commonality: they tend to ignore the reasons why imperfect information may exist and focus instead on methods for using information that does exist. The approach presented in this paper more closely resembles work done by Wong in statistical inferencing [Wong82]. Wong describes a model for determining values given a history of past values. Using information about the kinds of errors that might occur, he attempts to provide solutions that are meaningful to the client. The technique presented in this thesis allows the system to respond to queries using meta-information about the database and the query. For example, a client may specify uncertainty about the accuracy of certain attributes in a query.

Extendible Database Systems

There is a trend in database systems toward extendible systems that address the needs of non-business application areas, e.g. RDL1 [Kier89], NAIL! [Morr86] at Stanford, POSTGRES [Ston86] from Berkeley, EXODUS [Care88a] at the University of Wisconsin, PROBE [Daya85] at CCA, and GemStone [Cope84]

at the Oregon Graduate Center. These systems attempt to extend the capabilities of database systems by adding logical induction, abstract data types, or programming language support (programming language support is discussed in a separate section).

The RDL1 and NAIL! systems use a form of logic (see [Ullm88, Ullm89] or [Gene87]) to extend the information in the database. POSTGRES uses a rule processing facility [Ston88b] to perform the same function. These systems allow new information to be inferred about the properties of objects or the relationships between objects from information that exists in the system.

The data models used by extendible database systems generally fall into one of three categories: object-oriented, extended relational, or functional. GemStone uses the object-oriented framework provided by Smalltalk-80, modified to include the notion of persistent objects. POSTGRES extends the relational model to include procedural and relational attribute values [Ston87]. DAPLEX [Ship81] models all data as functions on the relationships between objects. The EXODUS data model, EXTRA [Care88b], combines pieces of all three models to provide the type inheritance of GemStone, procedural and relational values of POSTGRES, and functional relationships between objects of DAPLEX. All of these models allow additional basic attribute value types to be constructed; i.e. they allow a client to define the structure of and the operations that may be applied to the value of a class of attributes. The object type system used within a Univers name server differs from more conventional type systems in that a type is defined by structural similarities in the object representation [Card88]. This is in contrast to viewing a type as a set of objects upon which a set of operations may be performed [Card85]. Structural typing allows types to be treated as first class objects without the inherent difficulties noted in [Card86] and [Meye85].

Distributed Database

A descriptive naming system avoids many of the problems that must be addressed by current distributed database systems. Stonebraker identifies seven issues that affect distributed database development [Ston88a, pages 190-191]:

- **Location transparency:** All objects in the system are equally accessible.
- **Performance transparency:** Query response time is independent of the node from which the query is submitted.
- **Copy transparency:** Several authoritative copies of an object may exist within the system.
- **Transaction transparency:** Updates are independent of the node from which they are submitted.
- **Fragment transparency:** The description of an object may be split into pieces across the system.
- **Schema change transparency:** A single transaction can update the structure of the entire database.
- **Local DBMS transparency:** The methods for storing local information are independent of the services provided by the distributed system.

IBM's R* distributed database management system demonstrates how these issues are resolved in a real database system [Lind85, Will81]. Several of these issues are simplified within an internet-wide

descriptive naming system. First, objects are described within a single context so that copy transparency is trivial. A single authoritative description exists for each resource. Any other context can cache the description but the cached information is not guaranteed to be up-to-date. Second, each system administrator is only responsible for maintaining the object descriptions within its own name server. Therefore, there are no non-local updates. Third, object descriptions remain within a single name server so there is no object fragmentation.

The transparency of local *DBMS* profoundly affects the development of descriptive naming systems. A descriptive naming system is a collection of autonomous name servers, i.e. there is no global system administrator. Each server within the system may choose different conventions for object representations, may export different resolution functions, and may have different security requirements. This property distinguishes descriptive naming from other distributed database systems.

Query Optimization

The preference hierarchy model presented in this thesis lends itself to optimizations that are not meaningful in other formalisms. Algorithms used in relational database systems focus on the relationship between **selects**, **projects**, and **joins** [Cham81, Hall76, Pech80]. These operations have meaning only in database systems that manipulate partial descriptions. In contrast, descriptive naming always works with complete object descriptions; the goal of the system is to provide a service for identifying objects not a general purpose database system. Thus, descriptive naming depends only on **select**, **union**, and **intersection** that may be optimized using different techniques.

Although the Univers naming system must infer the most preferred solutions, the methods used to optimize logical database queries do not apply. Query optimizations in logical databases usually involve solutions of recursive formulas [Banc87]. Descriptive naming, on the other hand, uses no recursive rules.

2.3 Programmable Systems

Many computer systems provide services through a language that provides capabilities for extending the interface or functionality of the system. For example, the Gnu Emacs editor [Stal87], the TEX document processing system, the GWM window manager [Naha89], and the Postscript language for display and print servers [Adob85] provide an extension language for modifying existing operations and for developing new ones. Both EXODUS [Rich87] and GemStone [Cope84] allow a database to be extended by adding features written in some programming language. New access methods, data types, and object operations may be added to an EXODUS system by developing code in E, a version of C++ that supports persistent objects. The GemStone language, OPAL, encompasses the functionality of both a database and an object-oriented programming language. OPAL is essentially an extension of Smalltalk-80. Within distributed systems both Falcone and Emer [Falc86], and Stamos and Gifford [Stam90] describe language support for remote execution.

The Univers name server described in Chapter 4 supports a programmable interface called Ulisp. Ulisp is a Lisp-based language that closely resembles Scheme [Stee78, Suss75]. We chose Scheme because of its function handling capabilities, most notably the ability to treat functions as first class objects. All name resolution functions are written in Ulisp.

3 A MODEL FOR DESCRIPTIVE NAMING

This chapter defines the notion of a *preference hierarchy* and describes the role it plays in resolving descriptive names. It introduces the intuition behind preference hierarchies and gives several example preferences, establishes a framework for designing resolution functions based on combinations of preferences, and shows how that framework can be used to reason about naming systems. To better understand preferences and to appreciate how preferences are used to resolve names within in a context in the Univers naming system, the chapter gives several examples of client and database preferences.

3.1 Preliminary Definitions

A descriptive naming system, in the abstract, answers queries about a universe of *resources*, each of which has certain *properties*. If a resource possesses a particular property, then the property is said to *describe* the resource. For example, if we are considering the universe of printers, then elements of this set might possess properties for "fonts", "location", and "resolution". The specific property "location is Room 723" describes a particular printer.

In practice, a descriptive naming system maintains and manipulates a database of syntactic information about a set of resources: It denotes each resource with a database *object* and each property with an *attribute*. For example, a naming system that knows about printers might store facts of the sort "printer ip2 supports italic font and is located in room 723", "printer lw6 is of make laserwriter and has 300 dots per inch resolution", and so on. The corresponding objects in the naming system database would be

((font italic) (location 723) (uid ip2))
((make laserwriter) (resolution 300) (uid lw6))

where (font italic) is an example of an attribute.

Each attribute consists of a *trait* and a *value*; denoted (t v). The attribute (font italic), for example, consists of the trait font and the value italic. For convenience, we assume each object contains a uid attribute that uniquely identifies the object. We often refer to the object by this attribute's value; e.g., object ip2. Also, if an attribute **a** is entered in the naming system database for object **x**, then **a** is said to be *registered* for **x**. When a property describes a resource and the corresponding attribute is registered for the corresponding object, we sometimes say that the attribute describes the object and that the object matches the attribute.

The meaning of objects and attributes is given by a *meaning function* μ; the resource represented by object **x** is given by $\mu(\mathbf{x})$ and the meaning $\mu(\mathbf{a})$ of an attribute **a** is the property specified by **a**. In the universe of printers, for example, μ(font italic) is the property of supporting the italic font. Furthermore, we say that a database is *honest* if an attribute **a** is registered for an object **x** implies that $\mu(\mathbf{a})$ describes $\mu(\mathbf{x})$. For example, if the attribute (phone 621-1234) is registered for the object jones in an honest database, then we expect the referent of jones—presumably a person—to have the phone number 621-1234.

Clients query a descriptive naming system with a set of attributes, called a *descriptive name*. The naming system responds with the set of objects that correspond, in some sense, to those attributes. Formally, a set of attributes **N** *names* object **x** if every attribute in **N** is registered for **x**. That is, *names* maps sets of attributes into sets of objects, where *names*(N) denotes the set of objects named by a set

of attributes **N**. If $\mathbf{x} \in$ *names*(N), then **N** is said to be a *name* for **x**. The corresponding semantic notion is *represents*: a set of properties, μ(N), represents a resource if every property in the set describes the resource.

The relationship between the semantic and syntactic domains, and the parallel between working with a single attribute and a set of attributes, is schematically depicted in Figure 3. The dotted edges link single items with sets of items.

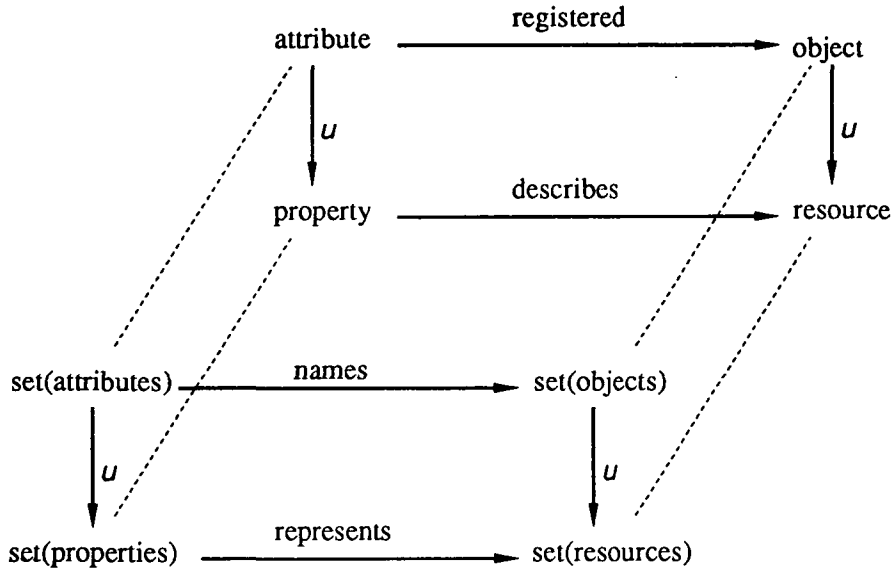

Figure 3: Relationship between Semantic and Syntactic Entities

In a "perfect" world—such as exists in a compiler or an operating system where the client accurately and completely identifies a resource and there is no possibility of missing or out-of-date information in the database—a naming system may implement a single operation that supports the *names* function. Within these naming systems, it always suffices to select only those objects that precisely match every attribute in a name, i.e. those objects for which every attribute in the name is registered. Indeed, this is essentially what is done when responding to queries in conventional database systems.

A naming system for an internet environment, however, does not operate in a perfect world; the information constraints placed on the system force a naming system to accommodate imperfect information. In an imperfect world, the naming system must cope with two potential problems: inaccuracies in the attributes specified by the client, and imperfect information in the database. Rather than provide a single method for resolving names, a descriptive naming system supports a set of resolution functions, each tailored for a class of attribute sets and databases that the designer considers interesting.

Intuitively, each resolution function considers certain assumptions about the quality of information in the system and resolves names according to the most preferred ones. Thus, a resolution function is constructed from an ordering on a set of *approximation functions*. Formally, a *preference*, denoted by $\prec$, is a total order on a set of functions that approximate portions of a perfect-world naming system. Each

of these approximation functions compensates for some imperfection believed to exist in the naming system, either in the information specified by the client (called *client approximation* functions) or in the information contained in the database (called *database approximation* functions). A client approximation function maps the set of attributes specified by a client into a set that, hopefully, accurately describes the object the client intends to identify. A database approximation function uses a set of attributes to select from a set of objects those that would "match" the attributes in a perfect world. The preference encapsulates some meta-information about the system by describing the assumptions that the client believes are most reasonable. The most preferred approximations provide answers based on what is believed to be the most reasonable and accurate information in the system. If the information from one set of approximations fails to distinguish among a set of objects, then the naming system attempts to resolve the name using another set.

3.2 Client Preferences

Using the attributes specified by the client, a client approximation function constructs a set of attributes that accurately describes the objects sought by the client. Generally, these functions use information about the property denoted by an attribute in order to determine its accuracy. For example, an approximation might consider social security numbers to be accurate because a client that specifies a social security number usually knows it accurately. Given a set of attributes, one kind of approximation function based on a preference for accurate attributes would remove those attributes that it believes to be inaccurate, returning the accurate ones. This kind of function always returns a subset of the attributes it is given. Another kind of client approximation function may augment the attributes specified by the client with additional attributes that specify the type of object being described or the object's location. This function uses information from the client's specification to add attributes to the description, i.e. it always returns a superset of the attributes it is given as an argument. In general, a client approximation function is formally defined by:

Definition 3.1 (Client Approximation Function) *A function* $f : 2^{\mathbf{A}} \longrightarrow 2^{\mathbf{A}}$ *over a domain of attributes* **A** *is a* **client approximation function** *if function* f *is monotonic increasing, i.e. for all sets of attributes* **M** *and* **N**, $\mathbf{M} \subseteq \mathbf{N}$ *implies* $f(\mathbf{M}) \subseteq f(\mathbf{N})$.

A client preference is a total order on a set of client approximation functions. Approximations high in the order are those the client considers most likely to be accurate. For example, a client may specify that an approximation that assumes that social security numbers are accurate is preferred to an approximation that assumes that social security numbers are inaccurate. The selection of the approximation functions and the order on them encapsulates information about the attributes supplied by the client. In practice, the selection of a specific preference depends on the client's assumptions about the information within a system and on the kind of objects that the client intends to identify. For example, a client that intends to identify processors will specify a different preference than one that intends to identify human users.

The remainder of this section describes several preferences that we have found useful in the naming systems we have designed. This list illustrates some possible preferences; it is not intended to be complete. Formal definitions for several of the preferences are given in Chapter 5 where we present several resolution functions that are constructed from these preferences.

Universal Preference

The universal preference consists of a single function *universal* that maps every set of attributes to itself. This function is used when the client believes that all of the attributes it specified are accurate and are sufficient to distinguish the set of objects to be identified. In other words, this function assumes that the client operates in a perfect world. The *universal* approximation function may be added to any other preference—generally as the most preferred approximation—to provide a perfect world approximation. Intuitively, this happens because a client would not specify all of the attributes in the name if it didn't believe that there is a chance that all of the attributes accurately describe the objects the client intends to identify.

Approximation	Assumes
universal	every attribute is accurate

Table 1: Universal preference approximation function.

Registered Preference

The registered preference, denoted $\prec_R$, prefers attributes that are guaranteed to be registered in the database over those that are not. It consists of the approximation functions *open* and *closed* where

$$open \prec_R closed$$

An attribute with trait t is *closed* if every attribute constructed from t is guaranteed to be registered for every object it describes. Conversely, a closed attribute—i.e., any of the attributes returned by the client approximation function *closed*—is guaranteed to be registered for every object it describes.[1] For example, a naming system may guarantee that every object with a serial number will have it registered in the database. If the client specifies an object's serial number such as (serial-number 932F2320) as part of the description, then the naming system can return all objects for which the attribute is registered and can guarantee that the objects returned are precisely those with the serial number 932F2320—there are no objects missing from the naming system's response.

An *open* attribute may not be registered for an object even though the corresponding property describes the resource the object denotes. If the naming system resolves a name that consists of open attributes then the result may not contain all of the objects in the database that represent resources described by the name. This preference is particularly useful when users are responsible for maintaining some of the information in the naming system. One user, for example, may decide to include a home phone number in the database but another user may choose to leave it out. In this case, home phone number attributes are not closed and may provide incomplete answers. Note that the registered preference order is similar

[1] Many approximation functions use only a portion of an attribute in order to determine its accuracy. For example, the *closed* approximation function determines an attribute's accuracy solely by its trait. Other approximations, such as the functional approximation, use the entire attribute.

to the idea of open/closed world databases [Reit78]. Instead of making the closed-world assumption over an entire database, however, we make the distinction on an attribute by attribute basis.

Approximation	Assumes
closed	attributes with traits that are completely registered are accurate
open	attributes with traits that are not completely registered are accurate

Table 2: Registered preference approximation functions.

Mutability Preference

The mutability preference, denoted $\prec_U$, considers the time interval over which an attribute describes an object, thereby accommodating changes in object properties over time. The mutability preference may be defined as:

$$dynamic \prec_U static$$

The *static* approximation function considers accurate those attributes that will always describe the objects that they currently describe. For example, the serial number and architecture of a processor remain constant throughout its lifetime. Thus, *static* believes that any attribute describing a processor's serial number or architecture is accurate. In contrast, *dynamic* returns only those attributes that may change over time. Properties that involve a processor's location or a professor's class load may change over time. The information that a client possesses about these properties may become out-of-date. In this way, the mutability preference encapsulates a client's belief that some of the specified information once described the object it seeks but may not describe it any longer.

Approximation	Assumes
static	attributes that denote properties that do not change over time are accurate
dynamic	attributes that denote properties that may change over time are accurate

Table 3: Mutability preference approximation functions.

Precision Preference

The precision preference, denoted $\prec_P$, prefers attributes that match a very small set of objects.

$$ambiguous \prec_P unambiguous$$

Given a set of attributes, the *unambiguous* client approximation function returns the attributes that are unambiguous, that is, those that are registered for at most one object in the database. Names based on unambiguous attributes—for example, (address 192.12.69.22)—tend to be very precise. If the client accurately specifies at least one unambiguous attribute then any object that matches the name is precisely the object that the client intended to identify. If no object matches the unambiguous attributes in a name then at least one of them does not describe the object that the client names. Most conventional naming systems only support unambiguous attributes.

An attribute is ambiguous if it can be registered for more than one object. Ambiguous attributes generally match several objects so that names based on them are less precise, e.g. (hostname pollux) currently matches at least four different machines in the Internet. The *ambiguous* approximation function returns those attributes in a name that are ambiguous. Thus, the precision preference can be used to optimize queries—i.e. optimize the search for objects that match a name. Unambiguous attributes match at most a single object so that the set of matching objects can be computed more efficiently. The actual preference that the Univers server uses to optimize queries provides more levels of discrimination; i.e. there are several levels of ambiguity.

Approximation	Assumes
unambiguous	attributes that match at most one object in the database are most useful
ambiguous	attributes that may match more than one object in the database are most useful

Table 4: Precision preference approximation functions.

Yellow-Pages Preference

When a person wishes to locate an inexpensive plumber to fix his sink, he searches the yellow pages of the phone book looking for plumbers who advertise inexpensive rates. Similarly, clients within computer systems often wish to locate a set of resources that provide a particular service. For example, a client might wish to locate a printer that supports a particular font and is located in a nearby building. When a person looks for an inexpensive plumber he may be willing to accept a plumber that is expensive, but he probably will not accept an inexpensive carpenter. In the same way, certain characteristics of the object that a client seeks are absolutely necessary, though others may not be. For example, the client may specify that the printer "must" have a particular font and it would be "nice" if it were also in a certain building. In this case, the naming system should first determine which printers have the specified font, and from this set, select one in the specified building if possible. In this example, one could define a yellow pages preference $\prec_Y$ as follows:

$$mandatory \prec_Y optional$$

where *optional* always returns all of the attributes—both mandatory and optional—that the client specifies and *mandatory* returns just those attributes that the client specifies as mandatory.

Explicit Preference

Finally, it is possible to have clients themselves partition the attributes in a name into several classes. Each partition corresponds to the result of applying an approximation function to the entire set of attributes in the name. Consider, for example, a user querying a bibliographic name server. For one citation, the user may be certain of the author's name but somewhat unsure about the exact title of the paper. For another, the user may be certain about the exact title, but not sure about the author or the date of publication. It would be desirable for the user to be able to specify the preferences to be used in each case: for the first citation, a paper that matches exactly on the author's name but partially on the title is to be preferred to one that matches exactly on the title but only partially on the author; for the second, the preferences are reversed.

3.3 Database Preferences

Preferences that consist of one or more client approximations provide information about the client's specification. Similarly, preferences that consist of database approximations encapsulate information about the descriptions of a set of objects. In particular, a database approximation function accommodates imperfect information in the database by approximating the selection of objects from the perfect database. Rather than reconstruct a perfect database, these functions select objects from the existing database in a way that mirrors the selection of objects from what the function believes to be the perfect database. This usually takes the form of some strategy for handling partial matches although it may use more sophisticated techniques such as the statistical inference methods described in [Wong82].

Definition 3.2 (Database Approximation Function) *A function* $m : 2^{\mathbf{A}} \times 2^{\mathbf{O}} \longrightarrow 2^{\mathbf{O}}$ *over a domain of attributes* **A** *and a domain of objects* **O** *is a* **database approximation function** *if*

- m *is a contraction, i.e. for any set of objects* **D**, $m(\mathbf{N}, \mathbf{D}) \subseteq \mathbf{D}$.
- m *is monotonic increasing relative to the set of objects, i.e. for sets of objects* **C** *and* **D**, $\mathbf{C} \subseteq \mathbf{D}$ *implies* $m(\mathbf{N}, \mathbf{C}) \subseteq m(\mathbf{N}, \mathbf{D})$.

Intuitively, the definition says that a database approximation function never adds new objects to the database and it returns more complete answers when it is given more information about the world. Note that most database approximation functions cannot approximate the set of objects that the client intended to identify if they have no information from the client; thus $m(\emptyset, \mathbf{D}) = \emptyset$ for any set of objects **D** is true for most database approximation functions. One specific function that does not conform to this principle is a database approximation function called *identity*. This function does not process its arguments. Rather, it always returns that set of objects that it is given no matter what the client specifies.

A database preference is a total order on a set of database approximation functions. As with client preferences, the selection of a particular preference supplies the naming system with information about

```
(
  ( (uid cmb)
    (office 737)
    (workstation gilamonster)
    (department cs))
  ( (uid rao)
    (office 737)
    (workstation jaguarundi))
  ( (uid llp)
    (office 725)
    (workstation cicada))
  ( (uid gmt))
)
```

Figure 4: Example database.

the assumptions that the client possesses regarding the database. A particular preference may supply information about how to handle different degrees of partial match or matches in different locations. The following examples describe potential solutions to these problems.

Match-Based Preference

The match-based preference, denoted $\prec_M$ provides approximations that match attributes to objects at four different precisions. The functions in this preference are ordered by

$$possible \prec_M partial \prec_M exact \prec_M unique$$

The least preferred approximation, *possible*, maps a set of attributes to the set of all objects that could possibly match. In other words, an object is selected as long as there is no conflict between an attribute in the name and in the object description contained in the database. This function computes $\| Q \|^*$ as defined by Lipski in the case where there is no partial information [Lips81, Lips79]. The second approximation, *partial*, returns all objects that possibly match and have at least one attribute in common with the name. The third approximation, *exact*, returns all objects that are described by all of the attributes in a name. *Exact* computes Lipski's $\| Q \|_*$. The final approximation, *unique*, returns either the single object that exactly matches the name or the empty set.

Consider the database presented in Figure 4 that contains descriptions for four users and the name that consists of the attributes (office 737) and (department cs). In this case, *possible* returns cmb, rao, and gmt. *Partial* would return cmb and rao, but not gmt—gmt does not match any of the attributes. *Exact* returns just cmb because it is the only object that is known to match both attributes in the name. For these two attributes, *unique* would return cmb. However, no objects would be returned for a name that consists of the single attribute (office 737) because there is no unique match.

The match-based preference may be used when the database contains accurate but potentially incomplete information. The approximation will not return an object that conflicts with any of the attributes that the client supplies. Note that this preference generalizes the idea of partial match retrieval [Ullm88].

Approximation	Returns
unique	the single object that matches all attributes in the name
exact	objects that match all attributes in the name
partial	objects that do not conflict with attributes in the name and match at least one
possible	objects that do not conflict with attributes in the name

Table 5: Match-based preference approximation functions.

Voting Preference

The voting preference, denoted $\prec_V$, views the attributes that are specified as votes for objects. In contrast with the match-based preference, the voting preference does not assume that all of the information in the database is correct. Rather, it attempts to return reasonable matches even though conflicts may exist with attributes specified by the client. The order is defined as:

$$\textit{also-ran} \prec_V \textit{majority} \prec_V \textit{unanimous}$$

The most preferred approximation, *unanimous*, prefers objects that receive all of the votes; i.e. all the attributes are registered for the objects. *Unanimous* is just another name for *exact*. The *majority* approximation returns the set of objects that receive a majority of the votes. Finally, *also-ran* matches the set of attributes to any object that receives at least one vote; i.e. any object for which one or more of the client specified attributes are registered.

Using the database in Figure 4, *majority* returns both cmb and rao for a name that consists of the three attributes: (office 737), (workstation jaguarundi), and (department cs). Note that the object cmb conflicts with the workstation attribute specified in the name. For the attributes (workstation gilamonster) and (office 725) *also-ran* returns cmb and llp. A client that selects the voting preference indicates concern for inaccurate information within the name and the database. An attribute that is registered for an object in the database may not actually describe the object. For example, the object cmb may not be described by the attribute (workstation jaguarundi). Generally, the client that selects this preference is willing to sift through additional object descriptions as long as the answer contains the objects that the client intended to identify.

Temporal Preference

The temporal preference, denoted $\prec_T$, differs from the voting and match-based preferences in that it does not attempt to match a set of attributes to an object. Rather, the approximation functions that constitute

Approximation	Returns
unanimous	objects that receive all of the votes
majority	objects that receive a majority of the votes
also-ran	objects that receive at least one vote

Table 6: Voting preference approximation functions.

the temporal preference distinguish between objects based on the length of time until the information contained in the database becomes stale. The temporal preference prefers objects that are described by authoritative information—information that is guaranteed to be accurate—over information that has been cached. This provides one method for handling out-of-date information within the database. The temporal preference may be defined as:

$$\textit{out-of-date} \prec_T \textit{cached} \prec_T \textit{authoritative}$$

The *authoritative* approximation assumes that the only accurate information in the database is authoritative information. As such, the only objects that match a name are those with authoritative attributes registered for each trait that appears in the name. Both *cached* and *authoritative* allow an object to be described by information that is cached. However, *cached* demands that the attributes registered for objects not be stale—the database expects that the attribute value still describes the object. *Out-of-date* allows information to be in any condition: the value in the database described the object at one time and the assumption is that even out-of-date information may provide hints about the correct value.

Note that the temporal database preference bears close resemblance to the mutability client preference. However, the basic assumptions are different. On one hand, the mutability preference assumes that the database contains information that actually describes the object in question. In this case, the client possesses information that has become out-of-date. For example, a client may remember that a printer lw11 was located in GS725. Since then, however, the printer has been moved to GS737 and the database has been updated to reflect the change. The information that the client possesses is out-of-date and will produce erroneous results if used to identify the printer. On the other hand, the temporal preference assumes that the client's information is accurate but that the value of an attribute in the database has become out-of-date. This corresponds to the case where the client knows the actual location of lw11 but the location of lw11 was never updated within the database. Here, the database contains information that will keep the client from identifying lw11 by location.

Location Preference

Finally, consider a distributed system in which several objects match a given set of attributes. It is intuitively desirable for the naming system to return an object that denotes a local resource to one that denotes a remote resource. That is, one could define a location preference as:

$$\textit{remote} \prec_L \textit{local}$$

Approximation	Returns
authoritative	objects described in the database by attributes that are guaranteed to be accurate
cached	objects described in the database by attributes that are cached but currently up-to-date
out-of-date	objects described in the database by attributes that are cached but out-of-date

Table 7: Temporal preference approximation functions.

where *local* returns objects that correspond to resources located in the local computing environment and *remote* returns objects that correspond to resources from any environment within the naming system.

Approximation	Returns
local	objects that describe resources in the local computing environment
remote	objects that describe resources in a remote computing environment

Table 8: Location preference approximation functions.

3.4 Induced Preferences

The composition of a series of database and client approximation functions encapsulates all of the approximations that are made about the naming system. In theory, if the assumptions made by each of the approximation functions are correct, then the set of objects selected by their composition should resemble the set that the client intended to identify. Definition 3.3 formally defines the composition of database and client approximation functions.

Definition 3.3 (Composite Approximation Function) *A nonempty, alternating series of approximation functions* $\langle f_1, m_1, f_2, m_2 \ldots, f_n, m_n \rangle$ *where each* f_i *is a client approximation and each* m_j *is a database approximation, defines a* **composite approximation function** $c : 2^{\mathbf{A}} \times 2^{\mathbf{O}} \longrightarrow 2^{\mathbf{O}}$ *over a domain of attributes* **A** *and a domain of objects* **O**, *and*

$$c(\mathbf{N}, \mathbf{D}) = m_n(f_n(\mathbf{N}), \ldots m_2(f_2(\mathbf{N}), m_1(f_1(\mathbf{N}), \mathbf{D})) \ldots)$$

A preference on a set of composite approximations is induced from the set of preferences on the component functions. Consider a situation where we wish to construct a preference on a set of approximations over the sets $\{open, closed\}$ ordered by $\prec_R$, and $\{also\text{-}ran, unanimous\}$ ordered by $\prec_V$, where

$\prec_R$ and $\prec_V$ are as previously defined. Here, we are faced with the problem of deriving an induced preference over the set

$$\{\langle closed, unanimous\rangle, \langle closed, also\text{-}ran\rangle, \langle open, unanimous\rangle, \langle open, also\text{-}ran\rangle\}$$

In this case, it is necessary to take into account the relative importance of the preferences being considered. For example, if we assume that the completeness of an attribute contributes more to its probable accuracy than its precision, then we have the induced preference

$$\langle open, also\text{-}ran\rangle \prec \langle open, unanimous\rangle \prec \langle closed, also\text{-}ran\rangle \prec \langle closed, unanimous\rangle$$

In the general case, we are given a *preference hierarchy* defined by a set of preferences $\Pi = \{\prec_1, ..., \prec_N\}$, totally ordered by $\preccurlyeq$; i.e., $\preccurlyeq$ acts as a metaorder. For any two members $\prec_j$ and $\prec_k$ of Π, we say that $\prec_k$ is *more important than* $\prec_j$ if $\prec_j \preccurlyeq \prec_k$. Without loss of generality, assume that $k < j$ implies that $\prec_k$ is more important than $\prec_j$. Let the set of approximations of $\prec_j$ be given by π_j. The preference, $\prec$, induced by $\langle \Pi, \preccurlyeq \rangle$ is defined to be the lexicographic order on the following set of composite approximation functions:

$$\pi_1 \times \pi_2 \times ... \times \pi_n$$

As an example, consider the case where $\Pi = \{\prec_R, \prec_P\}$, and $\preccurlyeq$ is defined by $\prec_P \preccurlyeq \prec_R$. The preference classes are $\pi_1 = \{open, closed\}$ and $\pi_2 = \{also\text{-}ran, unanimous\}$. Therefore, the induced preference order is

$$\langle open, also\text{-}ran\rangle \prec \langle open, unanimous\rangle \prec \langle closed, also\text{-}ran\rangle \prec \langle closed, unanimous\rangle$$

The elements of $\pi_1 \times \pi_2 \times ... \times \pi_n$ must form an alternating series of database and client approximation functions so that they may be composed as defined above. However, a client often orders preferences in a non-alternating series. For example, if a client is concerned about the precision of matches, the temporal nature of attributes, and whether an attribute is closed or open, the client might specify the following importance order:

$$\prec_V \preccurlyeq \prec_T \preccurlyeq \prec_R$$

A non-alternating series, such as this example, may be transformed into an alternating series by collapsing sequences of database or client approximation functions into a single function. That is, we combine $\prec_T$ and $\prec_R$ into a single preference on compositions of functions from $\prec_T$ and $\prec_R$.

To construct a single function from a series of client approximation functions, observe that any client approximation function may be defined as the composition of the two special kinds of client approximations: an *accuracy approximation function* that always returns a subset of the attributes that the client provided and a *completion approximation function* that always returns a superset of the client's specification. Specifically, a client approximation function f may be written as a combination of an accuracy approximation function f_a and a completion approximation function f_c by

$$f(\mathrm{N}) = (f_c(\mathrm{N}) - \mathrm{N}) \cup f_a(\mathrm{N})$$

where $f_a(\mathrm{N}) = f(\mathrm{N}) \cap \mathrm{N}$ and $f_c(\mathrm{N}) = f(\mathrm{N}) \cup \mathrm{N}$. This is because for every $\mathrm{N}, f(\mathrm{N})$ can be partitioned into two sets of attributes: $f(\mathrm{N}) \cap \mathrm{N}$ and $(f(\mathrm{N}) \cup \mathrm{N}) - \mathrm{N}$. Using this relationship, a sequence of client approximation functions $f_1, f_2, \ldots, f_n$ defines a single client approximation function g such that

$$g(\mathrm{N}) = [(f_{1c}(\mathrm{N}) \cup f_{2c}(\mathrm{N}) \cup \ldots \cup f_{nc}(\mathrm{N})) - \mathrm{N}] \cup [f_{1a}(\mathrm{N}) \cap f_{2a}(\mathrm{N}) \cap \ldots \cap f_{na}(\mathrm{N})]$$

where f_{ia} and f_{ic} are the accuracy and completion approximation functions that comprise f_i. Intuitively, g considers an attribute to be accurate only if all of the accuracy functions consider it accurate. For example, if a function from one preference believes that closed attributes are accurate while a function from a different preference believes that unambiguous attributes are accurate, then the composition considers only attributes that are both closed and unambiguous to be accurate. Similarly, g considers an attribute to be missing if any of the completion functions considers it missing.

A sequence of database approximation functions, $m_1, m_2, \ldots, m_j$ can be collapsed into a single database approximation function n such that

$$n(\mathrm{N}, \mathrm{D}) = m_1(\mathrm{N}, \mathrm{D}) \cap m_2(\mathrm{N}, \mathrm{D}) \cap \ldots \cap m_j(\mathrm{N}, \mathrm{D})$$

In other words, n returns the set of objects that are matched by all of the strategies encapsulated in the functions of the sequence. We have found this to be useful for precomputing portions of the database to use for an approximation. For example, the temporal preference may be used with one of the match-based preferences to support a set of approximations that handle both temporal issues and match precision. Although we have not yet attempted to construct any fault tolerant resolution functions, it may be that this technique will make it possible when each database approximation function accesses information within a distinct copy of the database.

3.5 Resolution Functions

Given an induced preference on a set of composite approximations, a name resolution function computes the set of objects described by a name in a way that respects the induced preference. This means that a resolution function may not return a set of objects using a composite approximation function, c_i, unless every more preferred approximation—that is any approximation c_j where $c_i \prec c_j$—failed to compute a non-empty set of objects. Intuitively, a composite approximation function supports several approximations of the perfect world system. The assumptions encapsulated in the set of approximations are considered "accurate" if the composition returns a non-empty set of objects. A resolution function computes the set of objects described by the name relative to the most preferred, "accurate" set of approximations. Formally,

Definition 3.4 (Name Resolution Function) *An induced preference order $\prec$ on a set of composite approximation functions $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ defines a* **name resolution function** $\rho : 2^{\mathbf{A}} \times 2^{\mathbf{O}} \longrightarrow 2^{\mathbf{O}}$ *over a domain of attributes* A *and a domain of objects* O *by $\rho(\mathrm{N}, \mathrm{D}) = \pi_i(\mathrm{N}, \mathrm{D})$ where*

1. $\pi_i(\mathrm{N}, \mathrm{D}) \neq \emptyset$
2. $\forall \pi_j \in \Pi$ [$\pi_i \prec \pi_j$ *implies that* $\pi_j(\mathrm{N}, \mathrm{D}) = \emptyset$].

Consider the following example where we wish to design a resolution function with two assumptions: the database contains accurate but potentially incomplete information and the set of attributes specified by the client are accurate but may match more objects than the client intended. For these assumptions it does not seem reasonable to use database approximation functions that return any object that conflicts with the client's description. For example, if the client specifies two accurate attributes: (name pollux) and (domain as.arizona.edu), an approximation such as *also-ran* may match an object with the attributes (name toto) and (domain as.arizona.edu), even though (name toto) conflicts with part of the client's specification. Since a processor may have only one name by definition, the object that is returned cannot be one that the client intends to identify.

We define a resolution function, *equal3*, specifically for this situation. The function *equal3* uses *partial* and *possible* in conjunction with closed attributes to guarantee that no object that is returned conflicts with attributes specified by the client. The resolution function is defined by the following preferences:

$\prec_P$: *ambiguous* $\prec_P$ *unambiguous*

$\prec_R$: *open* $\prec_R$ *closed*

$\prec_M$: *partial* $\prec_M$ *exact*

$\prec_p$: *ambiguous*

$\prec_r$: *open*

$\prec_m$: *possible* $\prec_m$ *partial* $\prec_m$ *exact*

where $\prec_m \Leftarrow \prec_r \Leftarrow \prec_p \Leftarrow \prec_M \Leftarrow \prec_R \Leftarrow \prec_P$. If the assumptions about the information in the system are valid, then *equal3* never returns objects that are known to conflict with the client's specification. In general, it prefers objects that are known to match the client's description over those that are not known to conflict with the description.

3.6 Semantics Of Resolution Functions

The discussion in the previous section focused on the structure of the overall preference induced by a set of preferences and an importance order over them. This makes it possible to understand and reason about some aspects of the overall behavior of resolution functions. This section describes the semantics of the preference hierarchy model and develops some tools that may be used to describe and reason about sets of related resolution functions. In the discussion that follows, we use Δ to denote the set of name/database pairs that serve as the domain of a resolution function. Specifically, Δ consists of pairs $(\mathbf{N}, \mathbf{D})$ where $\mathbf{N}$ is a set of attributes and $\mathbf{D}$ is a set of objects. Δ is determined by the characteristics of the naming system that will support the resolution function.

Soundness and Completeness

Clearly, it is possible to construct many resolution functions that differ in how many (or how few) objects they return for a given set of attributes. In this context, two properties are of interest: Given a set of attributes, it may be desirable that the resolution function (*i*) return only those objects named by the attributes; and (*ii*) return all those objects that are named by the attributes. In order to formalize these properties we postulate an *oracle* function that returns precisely the set of objects that the client intended to identify. The objects that *oracle* returns are those that would be returned if the client had presented its request to a naming system that contained perfect information. That is, *oracle* knows about all of the imperfections that exist in the system and compensates for them. We assume that every client attempts to identify at least one object, therefore *oracle* always returns at least one object. Using the *oracle* function, soundness and completeness can be formalized as follows:

Definition 3.5 (Soundness) *A name resolution function ρ is said to be* **sound** *for a name/database pair* $(\mathbf{N}, \mathbf{D})$ *in* Δ *if and only if* $\rho(\mathbf{N}, \mathbf{D}) \subseteq oracle(\mathbf{N}, \mathbf{D})$.

Definition 3.6 (Completeness) *A name resolution function ρ is said to be* **complete** *for a name/database pair* $(\mathbf{N}, \mathbf{D})$ *in* Δ *if and only if* $oracle(\mathbf{N}, \mathbf{D}) \subseteq \rho(\mathbf{N}, \mathbf{D})$.

We denote the set of name/database pairs in Δ for which ρ is sound by Σ_ρ and the set of name/database pairs for which ρ is complete by Γ_ρ.

Often, the designer of a naming system must ensure that the resolution functions the system provides satisfy constraints about soundness or completeness. For example, our experience suggests that clients generally want a white-pages naming system that allows clients to search for and name users to be complete. In contrast, clients want a yellow-pages naming system that allows clients to locate system services to be sound. In the first case, clients are willing to discard extra objects but they want the desired object to be contained in the result, while in the second case, the client often depends on all of the answers being equally valid; e.g. if a client asks for a processor with a 68020 architecture then an answer that contains a processor that has a different architecture cannot be tolerated.

Suppose we are given a resolution function defined by a set of basic preferences Π with an associated importance order $\leftarrow$. Then, given information about the soundness or completeness of each approximation function in the set of preferences, it is necessary to be able to reason about the soundness or completeness of the induced resolution function. For instance, Profile supports several white-pages resolution functions that attempt to handle databases that may contain incomplete information and clients that may specify inaccurate descriptions [Pete88]. It is important that the Profile functions be complete on many name/database pairs where the database is incomplete and the name contains some inaccuracies. The following results help a system designer to determine the extent to which such a resolution function meets its goals for soundness and completeness.

A resolution function that consists of a single composite approximation function is sound only if the approximation returns a subset of the objects the client intends to identify. Similarly, the resolution function is complete only if the objects that client seeks are contained in the set that the approximation returns. This observation leads to the definition of soundness and completeness for a composite approx-

imation: a composite approximation function c is sound on a set Σ_c if a resolution function constructed from just c is sound on Σ_c; c is complete on Γ_c if the resolution function is complete on Γ_c.

Proposition 3.1 *The composition $c \circ d$ of two composite approximation functions c and d, where c is sound on Σ_c and d is sound on Σ_d, is sound on $\Sigma_c \cup \Sigma_d$.*

Proof: Recall that database approximation functions always return a subset of the objects in the database. If d is sound on (N, D) then $c \circ d$ must be sound because c cannot add any objects—a database approximation always returns a subset of the objects in the database. Database approximation functions are monotonic increasing relative to the database so that $(\mathrm{N}, \mathrm{D}) \in \Sigma_c$ implies that every subset of D is also in Σ_c. Therefore, if c is sound on (N, D) then $c \circ d$ is sound on (N, D). □

Proposition 3.2 *The composition $c \circ d$ of two composite approximation functions c and d, where c is complete on Γ_c, d is complete on Γ_d and c is a point-wise function—c has the property that $c(\mathrm{N}, \mathrm{D}) = \{\mathrm{x} \in \mathrm{D} \mid \mathrm{x} = c(\mathrm{N}, \{\mathrm{x}\})\}$ for all sets of objects D—is complete on $\Gamma_c \cap \Gamma_d$.*

Proof: If d is complete on (N, D) then it returns at least the objects that the client seeks. If c is point-wise determined and complete on (N, D) then it is complete on every subset of D that contains $oracle(\mathrm{N}, \mathrm{D})$. Thus, $c \circ d$ is complete on all name/database pairs in $\Gamma_c \cap \Gamma_d$. □

These two propositions allow a system designer to determine when a composite approximation that consists of several client and database approximations is sound and complete. For example, it is easy to show that a composite approximation defined by ⟨ *closed, unanimous, open, also-ran* ⟩ is complete on $\Gamma_{\langle open, also-ran \rangle} \cap \Gamma_{\langle closed, unanimous \rangle}$. In particular, this composite approximation is complete when the database is accurate but potentially incomplete and the client's description contains at least one accurate open attribute and a non-empty set of closed attributes, all of which are accurate. Since every name resolution function consists of one or more composite approximation functions ordered by an induced preference, we may use the sets on which the composite functions are sound and complete to construct a set of name/database pairs on which the resolution function is sound or complete.

In general, we are concerned with the entire resolution function. A resolution function determines the set of objects described by a name relative to an induced preference on a set of composite approximations based on the failure of inaccurate approximations. Recall that an approximation fails when it computes an empty set of objects. The set of name/database pairs on which any composite approximation c fails is contained in Σ_c—the set of name/database pairs where c is sound. In fact, Σ_c may be partitioned into two disjoint sets P_c and E_c where c computes a non-empty set of objects for every (N, D) in P_c and an empty set of objects for every (N, D) in E_c. Given a resolution function ρ consisting of an induced order on several composite preferences, Σ_ρ and Γ_ρ may be computed using information about the name/database pairs in P_c and in E_c for each composite approximation c according to the following propositions.

Proposition 3.3 *A name resolution function ρ defined by the induced preference $c_n \prec c_{n-1} \prec \ldots \prec c_1$ is sound on*

$$\Sigma_\rho = P_1 \cup (E_1 \cap (P_2 \cup (E_2 \cap \ldots \Sigma_n)))$$

where c_i is sound on $\Sigma_i = P_i \cup E_i$ and c_n is sound on Σ_n.

Proposition 3.4 *A name resolution function ρ defined by the induced preference $c_n \prec \ldots c_2 \prec c_1$ is complete on*

$$\Gamma_\rho = \Gamma_1 \cup (E_1 \cap (\Gamma_2 \cup (E_2 \cap \ldots \Gamma_n)))$$

where c_i is complete on Γ_i and empty on E_i.

If it is the case that $E_1 \cap E_2 \cap \ldots \cap E_i \subseteq P_{i+1}$ for all i in the induced preference, then Σ_ρ is just the union of all P_i. In fact, the union is a good approximation for Σ_ρ whenever P_i is very small relative to E_i. A similar approximation may be made for Γ_ρ. That is, when Γ_i is very small relative to E_i, then Γ_ρ is simply the union of all Γ_i. These simplifications provide an adequate estimate of Σ_ρ and Γ_ρ for most resolution functions that we have written.

Discrimination

An important criterion when considering related naming systems is that of how many (or how few) objects they return for a given set of attributes. A more discriminating resolution function always returns a smaller set of objects for a given set of attributes relative to a particular database. This may be formalized as follows:

Definition 3.7 *Given two resolution functions ρ_1 and ρ_2 and a set of name/database pairs Δ, if $\rho_1(\mathbf{N}, \mathbf{D}) \subseteq \rho_2(\mathbf{N}, \mathbf{D})$, for all $(\mathbf{N}, \mathbf{D}) \in \Delta$ then ρ_2 is said to be* **less discriminating** *than ρ_1 on Δ (written $\rho_2 \sqsubseteq_\Delta \rho_1$).*

Let $\mathbf{Resolve}_\Delta$ denote the set of all resolution functions defined on Δ, partially ordered by $\sqsubseteq_\Delta$. $\mathbf{Resolve}_\Delta$ is a complete lattice, whose bottom element is the function that always returns the set of all objects, and whose top element is the function that always returns the empty set. Different resolution functions can therefore be compared and reasoned about based on their power of discrimination. For example, consider two resolution functions ρ_1 and ρ_2 where ρ_1 is defined by a single composite approximation $\langle universal, possible \rangle$ and ρ_2 is defined by a single composite approximation $\langle universal, exact \rangle$. It is easy to show that

$$\rho_1 \sqsubseteq_\Delta \rho_2$$

for Δ containing all name/database pairs.

In general, the choice of a particular resolution function from the family $\mathbf{Resolve}_\Delta$ depends on a consideration of tradeoffs between the computational cost and the precision of resolution offered by alternative functions. Elements of $\mathbf{Resolve}_\Delta$ that are low in the lattice defined by $\sqsubseteq_\Delta$ are relatively efficient, but typically not very discriminating. In other words, they may return multiple objects that the user then has to sift through. On the other hand, elements of $\mathbf{Resolve}_\Delta$ that are high in the lattice may be computationally more expensive, but are typically more discriminating. These functions, however, run the risk of being overly discriminating in that they may not return the object the user wants. Our experience is that in practice, useful resolution functions are tuned experimentally and strongly influenced by the client requirements and the underlying system constraints.

4 UNIVERS NAME SERVER

This chapter describes the Univers name server, the fundamental building block of the Univers naming system. The Univers name server provides facilities for implementing the descriptive naming model presented in Chapter 3. This chapter gives an overview of the structure of a Univers name server and describes in detail an extension language, called Ulisp, that defines a protocol for accessing naming information. Chapter 5 shows how to construct an internet-wide naming system using a set of Univers name servers. By analogy, this chapter defines a programming language and the next chapter shows how to implement a particular application using this language.

4.1 Structural Overview

A Univers name server consists of three major pieces as shown in Figure 5: an access manager, a database manager, and a program interpreter. The access manager provides a communication framework for accessing the server. A client accesses the server through one of the supported transport channels. The access manager acts as a filter, translating the client's requests into a Univers function call and translating the server's responses back into a form that the client can understand.

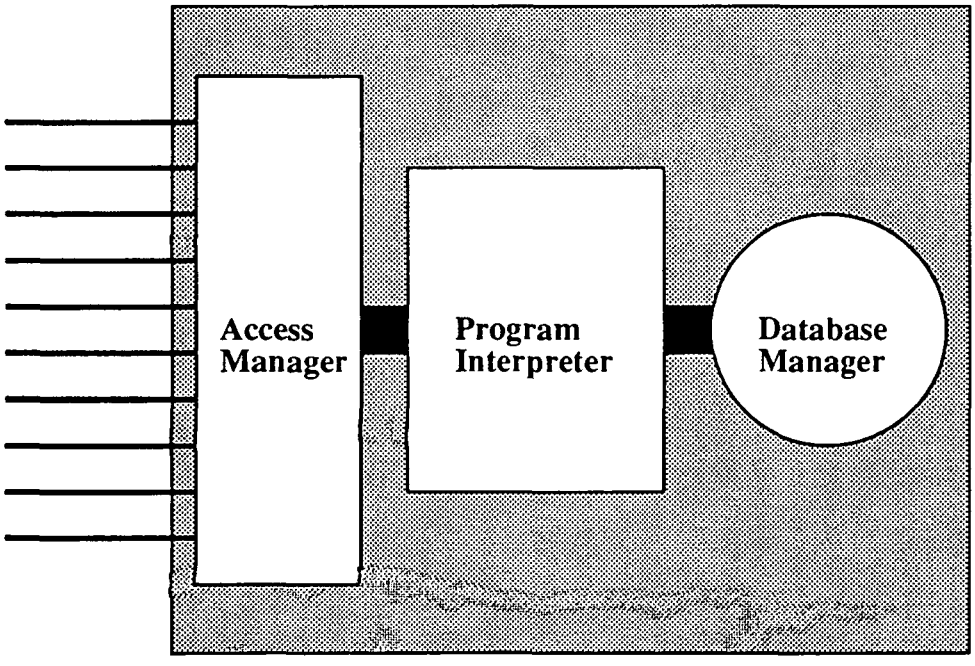

Figure 5: Internal structure of a Univers name server.

The database manager exports operations that access and maintain the information within the server's database. The database itself is organized into administrative domains called *contexts*. A context consists of a set of object descriptions; i.e. a set of attributes for each of a collection of objects. The database manager exports operations for selecting objects from a context, adding or removing objects from a

context, and updating the attributes that describe an object. The objects within a context are organized into resource classes, called *types*, according to structural similarities.

The program interpreter translates a client's query into operations that access the database. At the heart of the program interpreter is a function, called **resolve**, that implements the preference hierarchy model. Given a set of attributes and a preference hierarchy, **resolve** returns a set objects that are described by the attributes in a way that respects the preference hierarchy. In addition, the program interpreter supports a programmable interface for defining new constructs that extend the functionality of the server. This interface, realized by a Scheme-based language [Suss75, Stee78] called Ulisp, allows clients to define new operations such as approximation functions, comparison operations and update operators.

The access manager and database manager are entirely implemented in C. The program interpreter is implemented as a combination of Ulisp and C code. The foundation for the interpreter is a public domain, C-based implementation of Scheme, called Xscheme [Betz89], that supports byte-compilation of Scheme programs. In order to provide the necessary functionality and to improve efficiency, Univers adds new primitives and modifies existing ones. Each Ulisp primitive is a C procedure that implements either a function that affects the operation of the interpreter (e.g., **resolve**), a built-in database operation, or a well-known naming function.

4.2 Attributes

Recall that an attribute is a syntactic representation of some semantic property and is given by a trait/value pair that is denoted (**t v**). For example, the property "speed is ten pages per minute" is expressed as the attribute (speed 10).

Attribute traits are simple character strings that label an attribute. A trait serves two purposes. First, it acts as a hint about the interpretation of an attribute value. It does not, however, constrain the possible value of the attribute. For example, the speed trait indicates that the attribute value may be interpreted as an integer that denotes the speed of an object. However, the speed trait of a processor means something different than the speed trait of a printer. For a processor, speed denotes the number of instructions that can be processed per second, whereas the speed of a printer is the number of pages that can be printed per minute. Moreover, the value of two attributes with the same trait may be represented differently for two different kinds of objects; e.g. (name "Wyatt Earp") and (name saguaro.arizona.edu).

Second, an attribute trait relates objects to values. In particular, if the attribute (**t v**) is registered in the database for object **x**, then the trait **t** will map the object **x** into the value **v**. For example, if a printer lw1 supports the Times Roman font then the font relation can map lw1 to "Times Roman".

An attribute value denotes the actual property of an object; e.g. its address, name, location or load. The value of an attribute may be one of two possible forms. First, the value of an attribute may be a literal such as "Wyatt Earp", saguaro, or 3[2] Second, an attribute value may reference another object in the database; e.g. the value of the attribute (workstation **xyz**) refers to a processor object that is contained in the database. To facilitate unambiguous identification, a unique identifier is assigned to every object in the database and is registered for the object as the value of the uid trait.

[2]For the sake of clarity, we often use quotes to denote literal values that include a blank space. However, all literal values are treated as character strings.

The remainder of this section describes the differences between attributes that are registered for objects in the database—which we call database attributes—and attributes that the client specifies as part of the name—called naming attributes[3].

Database Attributes

Values are represented in the database as compound entities defined by the following structure:

(*print-value compare-value reference-value*)

An attribute's *print-value* is a character string that gives a printable or user-readable version of the value; i.e., it can be returned in response to a client's query. An attribute's *compare-value* gives the version of the value against which client queries are compared. Finally, an attribute's *reference-value* is a set of references to other objects in the database, i.e. a set of unique identifiers.

In general, it is possible that several attributes with the same trait identify an object. For example, if a printer supports several different fonts such as Helvetica, Times Roman, and Palatino, then the attributes (font helvetica), (font times-roman), and (font palatino) may be registered for the printer. The separation of print-value and compare-value accommodates potentially multi-valued attributes. A compare-value is represented by a character string that denotes a regular expression where each string generated by the regular expression is one of the literal values of the attribute. These regular expressions are similar to the ones used in Profile [Pete88]. For example, the font trait for a printer may have a compare-value "[Hh]elvetica | [Tt]imes [Rr]oman | [Pp]alatino" that corresponds to the three attributes listed above. Similarly, the name attribute for a person might have a compare-value Wyatt|[W[yatt]]Earp thereby allowing client's to select the object using the name trait with any of the values Wyatt, Wyatt Earp, W Earp, and Earp. A print-value summarizes the information contained in the compare-value in a form suitable for response to a client's query. For example, the print-value for the user's name may be Wyatt Earp whereas the printer's font attribute may specify the print-value (helvetica times-roman palatino).

Naming Attributes

A descriptive name consists of a set of attributes. In contrast with the database attributes, each attribute in the name has an associated operation that relates the attribute value in the name to the values of attributes that are registered in the database. Each operation/trait/value triple in the name specifies a property that can be used to select objects from the database. For example, the property "load less than three" is expressed as the attribute (< load 3). The operation associated with an attribute is a binary predicate and is used to compare the attribute value with the value of the corresponding database attribute. A naming attribute (f t v) describes an object x if the object is described by a database attribute (t u) such that f (u,v) is true. Thus, (< load 3) describes an object saguaro if the attribute (load 2) is registered for saguaro. Note that a database attribute may be considered a naming attribute with operation "=" since both describe the same set of objects.

Reference values can be specified in three ways: as an object's unique identifier, as an object's "common name", and as a query that identifies the object in the database. Client's can unambiguously

[3]Except when necessary to clarify the discussion, we will not distinguish between naming attributes and database attributes.

identify an object by specifying its unique identifier as the reference value; however, most clients will not possess any information about an object's uid attribute. Reference values can be specified by a literal value that denotes an object's "common name". The common name represents a property—usually an object's mnemonic name—that intuitively identifies the object—the literal saguaro is the common name of a particular workstation. A common name has the limitation that it may not uniquely identify the object it references—e.g. saguaro describes both a file system and a processor—or an object may not have any property that serves as a common name—e.g. many printer's are given addresses but not names. Finally, an object's reference value may be given as a query that selects the object from the database. For example, a client that wants to identify the person that uses a certain workstation may specify a reference attribute that describes an object that uses the workstation named gilamonster at address 192.12.69.20 as follows:

```
(ref= workstation
  (equal3 ((= name gilamonster) (= address 192.12.69.20)) ) )
```

The value of this attribute is a query that applies the equal3 resolution function to attributes that specify an object's name and address. The operation ref= compares reference values and is true whenever a database attribute refers to one of the objects specified by the client. Thus, this attribute describes any object that uses a workstation named gilamonster at address 192.12.69.20.

4.3 Naming System Objects

The primary function of the database manager is to provide facilities for maintaining information about resources that exist in the "external" world; e.g. printers, processors, users, or organizations. In addition, however, the database manager must also provide facilities for maintaining information about resources that have meaning only within the naming system. From the naming system's perspective, the database provides an ideal location for storing persistent information about objects that represent abstractions used by naming system. From the client's perspective, storing naming system objects in the database means that clients can use the naming system to discover information about the naming system. This section describes three special objects: type objects, context objects, and function objects.

Types

The database imposes a type structure on the objects it contains. A *type* is a collection of objects that exhibit structural similarities. For example, all printers have certain characteristics in common: they all have a pages per minute speed, a well-defined architecture, a mnemonic name, an address, and so on. Formally, type **T** is defined by a nonempty set of traits ($trait_1$ $trait_2$ ... $trait_n$). We refer to the set of traits that define a type as the type's *necessary traits*. An object is of type **T** if and only if the traits of its attributes are a superset of the traits in **T**. It is important to understand that an object's type is not explicitly stored as part of the object. Instead, type membership is determined by comparing the traits of an object's attributes with the traits that define the type. For example, one could define the type laser-printer with the set of attribute traits (name architecture speed resolution fonts). The following object would be of type laser-printer because the traits used to describe it contain all of the traits that constitute the laser-printer type.

```
( (uid lw11)
  (location GS737)
  (architecture postscript)
  (resolution 300)
  (speed 11)
  (fonts (helvetica palatino))
  (owner "Wyatt Earp"))
```

Note that if the type printer is defined by the set of attribute traits (name architecture speed) then the object is also of type printer. Intuitively, a type $\mathbf{T_1}$ is a subtype of another type $\mathbf{T_2}$ if $\mathbf{T_1}$ specifies more structure for an object; i.e. if the traits of $\mathbf{T_1}$ contain the traits of $\mathbf{T_2}$. In the example, laser-printer is a subtype of printer.

Because types are an important abstract resource, they are themselves represented as objects in the database. That is, just as there are user objects and printer objects in the database, there is a collection of type objects, each of which is defined by a set of attributes. Thus, every time we introduce a new type **T** that is defined by $(trait_1 trait_2 \ldots trait_n)$, we are able to give the database object that represents type **T** as follows:

((name T) (necessary-traits ($trait_1$ $trait_2$ ... $trait_n$)))

Moreover, because the collection of type objects in the database share the same set of attribute traits, they are of the same type. The type of all type objects is defined by the following object:

((name type) (necessary-traits (name necessary-traits)))

That is, all type objects have name and necessary-traits attributes.

Contexts

A second high-level structure imposed on objects, called *contexts*, partitions the set of objects in the database based on the authority that is responsible for administering the objects. For example, a database might support one "system" context, a collection of "application" contexts, and a set of "user" contexts. In this example, the system context would be owned by a trusted system authority (e.g., the super-user in Unix) and contain an object representing each resource in the computing system; each application context would be administered by the user or group responsible for the application and contain objects representing the resources and abstractions used by the application; and each user context would be administered by a user and would contain a collection of private aliases, private applications, and so on. Intuitively, the owner of a given context decides what objects to put into the context and sets the policy governing how information in the context may be accessed.

Note that while it is possible that there are objects contained in different contexts that represent (describe) the same resource, Univers treats such objects as *distinct*; i.e., Univers does not maintain the semantic connection among multiple database objects that describe the same physical resource. However, a context can cache objects and use them to provide hints about where to locate the authoritative object. Generally, a context registers for any cached object information about the context and server traits that describe the context and server that contain authoritative information about the object.

Like types, contexts are also stored as objects in the database. Context objects have the following type definition:

((name context) (necessary-traits (name server administrator objects)))

That is, a context has a mnemonic name (name), it is implemented in some Univers server (server), and it is administered by some organization (administrator). If the context is implemented within the same name server, then the reference-value of the objects attribute refers to the set of objects contained in the context. For example, the database object representing the system context might be given by the following set of attributes:

```
( (name system)
  (server cholla.arizona.edu)
  (administrator root)
  (objects "CS Resources") )
```

Note that the print-value of the objects attribute summarizes the content of the context but is only useful as a comment; the real worth of the objects attribute is its indirect-value which points to the set of objects in the database that belong to the system context.

An important consequence of representing contexts as objects in the database is that a given context can contain one or more context objects, thereby implicitly linking that context to other contexts. That is, it is by defining contexts that contain context objects that one implements links among the collection of contexts that make up a larger, perhaps hierarchical, name space. If a context named root contains a context object with the attribute (name users), for example, then it is possible to interpret a hierarchical name of the form root/users.

Each Univers name server contains a distinguished context, called the *meta context.* Intuitively, the names of the objects contained in the meta context play the role of reserved words. A name server's meta context contains a set of objects that correspond to each context (including itself) in the server's database; the type object for types, contexts, and functions; and objects corresponding to several distinguished name resolution functions.

In addition to the meta context, a *session* context is associated with each client that accesses the name server. A client's session context is created when the client establishes a connection to the server and it is destroyed when the client terminates the connection. The session context may be used as a cache to hold object descriptions retrieved from other contexts. The session context may also be used as a temporary storage for objects that the client creates. For example, a client that intends to access a special class of printer may define a new type object that describes the structure of the new printers. Such objects could be placed permanently in the database; however, if the type will never be used again, then it is be more reasonable to keep it in the temporary session context.

Functions

As with type and context resources, the database contains objects that represent functions. The type of a function object is given by:

((name function) (name ulisp-code))

The value of an attribute with trait ulisp-code defines a Ulisp function that implements the function object. Two subtypes of the function type are of especial interest to the program interpreter: resolution functions and approximation functions. Two other types support the functional interface: preference hierarchies and preferences. This section describes each of these types.

Resolution Function:

A resolution function can be defined in two ways. First, it can be defined by a preference hierarchy that can be submitted along with a set of attributes to **resolve**. This provides a general method for submitting queries to any server in the name system. Second, it can be defined by a Ulisp function that directly accesses the information in the database—the program interpreter does not call **resolve** to answer the query in this case. Although less portable, the Ulisp implementation of a resolution function is generally more efficient. This leads to the following type definition for objects that represent resolution functions:

```
( (name resolution-function)
  (necessary-traits (name preference-hierarchy ulisp-code)) )
```

A resolution function has a name that may be used to identify it to the interpreter and a preference hierarchy that specifies the particular assumptions that the function makes. The value of a preference-hierarchy attribute is a structure that may be sent to **resolve** as part of a query.

The ulisp-code attribute describes a Ulisp implementation of the function that does not rely on **resolve**. For example, the resolution function equal3, defined in Chapter 3, is specified by the following Ulisp function:

```
(define (equal3 N D)
  (cond
    [(possible (open (ambiguous N)) (exact (closed (unambiguous N)) D))]
    [(possible (open (ambiguous N)) (partial (open (unambiguous N)) D))]
    [(exact (ambiguous N) D)]
    [(partial (ambiguous N) D)]
    [(possible (ambiguous N) D)]
    [(exact (open (ambiguous N)) D)]
    [(partial (open (ambiguous N)) D)]
  )
)
```

The expressions within the **cond** statement are evaluated from top to bottom. The first non-empty set of objects computed by an expression is returned. Note that this implementation has been optimized—several redundant induced approximations have been removed and other induced approximations have been simplified.

Preference Hierarchy:

A preference hierarchy orders a set of preferences. The ordering can be stated explicitly or it can be computed. Therefore, objects that represent a preference hierarchy have the following type:

```
( (name preference-hierarchy)
  (necessary-traits (name order preferences)) )
```

The value of an order attribute can be either a list that explicitly orders a set of preferences or it can be

a Ulisp program that implements a comparison predicate for the preferences in the domain described by the preference attribute.

Preference:

A preference—a total ordering on a set of approximation functions—is described by objects of the following type:

```
( (name preference)
  (necessary-traits (name order approximation-functions)) )
```

An approximation-functions attribute lists the approximation functions in the preference's domain. The reference-value of the attribute contains pointers to the objects that implement the approximation functions. An attribute with trait order describes the preferred order for attempting the approximations contained in the domain. The value of this attribute can be either a function that orders the approximations or it may be a string that orders the common names of the approximations.

Approximation Functions:

An approximation function has the type:

```
( (name approximation-function) (necessary-traits (name ulisp-code assumptions)) )
```

The ulisp-code attribute describes the Ulisp program that implements an approximation function. Assumptions is a string that informally describes the assumptions that the approximation acts upon. A client approximation function can be distinguished by a sub-type of approximation-function:

```
( (name client-approximation)
  (necessary-traits (name ulisp-code assumptions accuracy-code completion-code)) )
```

Recall that any client approximation function may be decomposed into two special kinds of client approximation functions: an accuracy approximation and a completion approximation. An attribute with an accuracy-code trait describes the code that implements the accuracy approximation of a client approximation. Similarly, the completion-code attribute describes the completion approximation function. For example, the *closed* client approximation function is represented by the following object:

```
( (name closed)
  (code
     (lambda (attribute-set) (ul-closed attribute-set)))
  (assumptions "Attributes that are guaranteed to be registered are accurate")
  (accuracy-code
     (lambda (attribute-set) (ul-closed attribute-set)))
  (completion-code (lambda (attribute-set) attribute-set)) )
```

where **ul-closed** is a builtin Ulisp function that returns the closed attributes from a list of attributes.

4.4 Database Manager

Clients can access the database manager through a two level interface that is both flexible and efficient. The low-level interface to the database consists of a small set of primitives. Although somewhat terse, they provide a powerful and flexible interface to the database. In addition, every Univers name server provides a set of high level functions that are constructed from the low level primitives. For the sake of

efficiency, the high-level functions are actually implemented as Ulisp primitives within the name server and make use of the sophisticated indexing that is available within the database manager. In this way, clients can access the database using either an optimized high-level function or a function implemented with the low-level primitives. This section describes the interface supported by the database manager and its implementation.

Interface

The database manager provides four low-level functions that facilitate database updates. The context administrator can use these functions to create new objects, to destroy old objects, and to modify existing objects. For example, if the memory size of a processor with unique identifier xyz changes from 4 megabytes to 8 megabytes, then the system administrator may update the object's description using the commands

(delete-attribute xyz (memory 4))
(add-attribute xyz (memory 8))

The following is a list of the fundamental database operations that can be used to manage the information within the database:

- **(create-object)**: Create a new database object and add it to the current session context.
- **(destroy-object** object): Destroy the specified object.
- **(add-attribute** object attribute): Register the given attribute for the specified object.
- **(delete-attribute** object attribute): Unregister the specified attribute from the given object.

In general, these functions are used to create more general purpose, high-level functions that provide update services. Univers provides four builtin update functions that are constructed from the fundamental functions:

- **(change-attribute** object trait old-value new-value): Change the value of the attribute with the specified trait registered for the given object from the old value to a new value.
- **(make-object** attribute-set context-name): Create an object, add the specified attributes to its description, and include it in the named context.
- **(add-object** object context-name): Add the specified object to the named context.
- **(remove-object** object context-name): Remove the specified object from the name context.

Appendix A contains the Ulisp program that implements **make-object** in a Univers name server.

Similarly, Univers provides a single, low-level function for accessing information within the database. The functions, called **get-value**, takes an object and a trait as arguments and returns the value of the attribute with the specified trait for the given object. This function may be used to construct high-level operations such as:

- **(select** object-set naming-attribute): For the given set of objects, return the subset that match the specified naming attribute.
- **(dereference** object-set trait): For the given set of objects, return the set of objects referenced by the attributes with the specified trait.
- **(select-type** object-set type-name): For the given set of objects, return the subset that are of the named type.
- **(select-context** context-name): Return the set of objects in the named context.

Each of these functions can be constructed using **get-value**. The code that implements **select** is provided in Appendix A as an example. For the sake of efficiency, these particular functions are actually implemented as builtin Ulisp primitives.

Implementation

The Univers database is conceptually a table of attributes as illustrated in Figure 6(a). Columns of the table correspond to attribute traits and rows correspond to objects. The intersection of row i and column j corresponds to the unique attribute of object i with trait j. The table is implemented in memory as a sparse array. Figure 6(b) illustrates the key data structure: a hash table that maps object_index/trait_index pairs into a pointer to a record that holds the corresponding attribute. A second hash table maps trait strings into trait indices. Both hash tables are dynamic [Lars88].

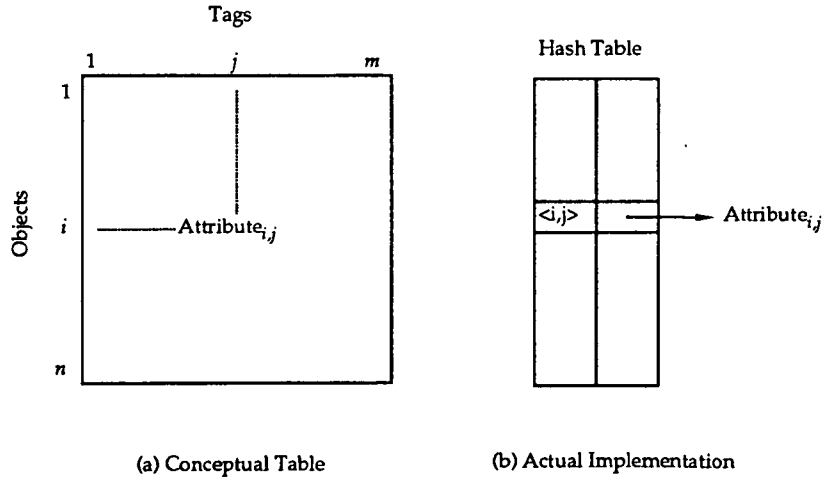

Figure 6: Univers Database

The most common operation on the database is to manipulate sets of objects. To accommodate a wide variation of set sizes efficiently, Univers supports two different representations. Whenever an object set has fewer elements than a certain threshold, Univers represents the set as a list of object indices. Whenever an object set is larger than the threshold, Univers represents the set as a bit vector. The current

list threshold is 16 elements and the bit vector is configured to store up to 4K elements. Both limits can be reconfigured at compile time to reflect the needs of a particular database.

Univers makes extensive use of object sets. First, database approximation functions directly operate on sets of objects; e.g., they search an object set for those objects with a particular attribute, they take the union or intersection of two object sets, and so on. Univers caches computed object sets so as to optimize common subexpression evaluation. Second, the reference-value field of an attribute is implemented as an object set. Third, Univers pre-computes the set of objects that have a particular trait. The select-type function supplied by Univers is then implemented by searching the session and meta contexts (in order) for the named type, extracting the set of necessary traits for the type, and taking the intersection of the object sets corresponding to each of those traits. Object sets that represent commonly used types are also cached. Fourth, each context in the name server is implemented as a set of objects. This set is stored as the reference-value of the context object's objects attribute. The C procedure that implements the Ulisp primitive select-context searches the meta context for the named context and then applies the dereference operation to that object's objects attribute.

The compare-value of an attribute may be given by a regular expression. When entered into the database, the expression is parsed by a recursive descent parser that builds a non-deterministic finite state machine. The machine is implemented with left and right child and state value arrays, resulting in approximately a six-fold increase in space over the textual representation of the regular expression. When making a comparison with a value supplied in a query, Univers does a depth-first examination of this machine graph.

Recall that resolution functions are database objects whose ulisp-code attribute stores the Ulisp program that implements the function. Internally, the value of a function's ulisp-code attribute is represented by both the program's source text and a compiled version of the function. The compiled code is generated when the function is entered into the database and it is interpreted when the function is invoked. The textual source is used when the attribute is displayed and when the database is written to disk. When the database manager is started, the function is reparsed from the stored program text.

Univers employs checkpointing and transaction logging to ensure database integrity and to facilitate failure recovery. For checkpoint writing, an ad-hoc translation is made from in-memory structures to the representation on disk; the reverse translation is made for restarting from a checkpoint. The ratio of log size to checkpoint frequency is configurable. Univers does not currently support automatic replication of the database across machines.

4.5 Function Interpreter

The Univers function interpreter accommodates conflicting performance and functionality goals in the same way as the database manager—it operates efficiently at several levels of abstraction. At the lowest level, a query is resolved independently of the function objects contained in the database. The client specifies the entire preference hierarchy as ordered lists of function expressions. Thus, a client can query the name server without any knowledge of the conventions that affect the structure and contents of the context's database. At the highest level, a client's query contains nothing more than a set of attributes and the name of a resolution function to be applied to those attributes. The resolution function

completely defines the preference hierarchy. For the sake of efficiency, many resolution functions are Ulisp primitives; i.e. they are implemented as hand optimized C procedures that directly access information within the system. In this way, the function interpreter accommodates both functionality and performance constraints.

Low-Level Interface

At the most fundamental level of abstraction, the client interacts directly with the **resolve** function. This function takes a set of naming attributes and a preference hierarchy and returns the most preferred set of objects. The client specifies a preference hierarchy as an ordered list of preferences where each preference is an ordered list of approximation functions. For example, the following preference hierarchy may be submitted to **resolve**:

$$((f_{1,1}\ f_{1,2} \ldots f_{1,n})\ (f_{2,1}\ f_{2,2} \ldots f_{2,m}) \ldots)$$

where preference f_1 is more important than preference f_2 and for all i and j, $f_{1,i} \prec f_{1,j}$ whenever $j < i$; i.e. the preference is specified left to right. Note that at this level, each approximation function $f_{i,j}$ is a **lambda** expression that defines a Ulisp function. Consider, for instance, an expression that defines an approximation that assumes social security attributes are accurate:

```
(lambda (attribute-set)
  (get-attributes-by-trait 'social-security attribute-set))
```

The meaning of the client's query does not depend on the conventions used within the name server. Rather, the operations that are available at this level—a subset of the functions available with standard Scheme, the functions provided by the database manager, and a small set of utility macros such as **get-attributes-by-trait** that simplify the code—and the format mandated by **resolve** define the fundamental protocol used to access naming information in the Univers naming system.

High-Level Interface

Although the fundamental level of interface provides a well-defined interface to a name server, the complexity of specifying queries becomes cumbersome. In order to alleviate this burden, the program interpreter provides several levels of abstraction that correspond to the function types described in Section 4.3. At the lowest level of abstraction—i.e. just above the level presented in Section 4.5—a client can denote an approximation function using a single identifier that uniquely describes the representation of the function in the database. Thus, in place of a list of lists of **lambda** expressions in the previous example, the client may specify the following preference hierarchy:

$$((f_{1,1}\ f_{1,2} \ldots f_{1,n})\ (\text{universal closed open})\ (\text{exact}\ f_{3,2}\ \text{partial}))$$

The program interpreter preprocesses the preference hierarchy specified by the client, searches the database for objects named in the preference hierarchy, and replaces every identifier in the preference hierarchy with the proper **lambda** expression from the database. The new expression created by the interpreter can be evaluated using **resolve**. References are resolved by searching in order the client's session context, the meta context, and then the contexts that the client is searching. Note that a client can override

the standard definition of a naming system object by placing in its session context another object that describes a slightly different resource.

Clients can substitute, at the next higher level, the name of a preference object contained in the database for an ordered list of approximation functions. For example, if the database contains an object that represents the registered preference consisting of the approximation functions universal, closed, and open then the client could specify the preference hierarchy as:

$$((f_{1,1}\ f_{1,2} \ldots f_{1,n}) \text{ registered } (\text{exact } f_{3,2} \text{ partial}))$$

Finally, at the highest level, clients can substitute the name of an object that represents a preference hierarchy for a list of preferences. Although **resolve** takes a preference hierarchy as an argument, most often a client specifies the preference hierarchy as the name of a resolution function. This has the advantage of being able to use an optimized implementation of the function, if one exists. If the client specifies a resolution function that is described by a ulisp-code attribute, then the interpreter simply evaluates the expression provided by the client. For example, if bib-lookup identifies a resolution function with a non-empty ulisp-code attribute, then the query

(bib-lookup ((= type citation) (∈ title Cactis) (= author Hudson)))

can be evaluated without ever entering **resolve**.

4.6 Access Manager

The access manager provides a framework for submitting queries to the program interpreter and for retrieving the results. Clients query a Univers name server using one of several access protocols, each of which connects to Univers via one or more transport channels. For example, a white-pages access protocol might be used to connect to a Univers server on either a UDP channel [Post80] or a TCP channel [USC81]. The Domain Name protocol [Mock87] might be used to query Univers on two other UDP and TCP channels, and various Unix library routines can query Univers using Sun RPC [Sun86]. The access manager also allows clients to query Univers directly via UDP and TCP channels. For example, a client may update the information in a context by directly submitting programs that invoke the **create-object**, **destroy-object**, **add-attribute**, and **delete-attribute** operations.

The access manager implements a filter for each access protocol by which clients can query Univers. Figure 7 depicts this process. The filter receives raw bytes from a channel, translates them into a query according to the semantics of the access protocol, invokes the interpreter to resolve the query, translates the results back into the correct protocol format, and sends the reply message to the client. In particular, the interpreter's *read-eval-print* loop interacts directly with the access manager through a single communication channel that serializes queries. In the case of direct queries, the filter is null, but the access manager is responsible for authenticating the client. Such authentication is used to restrict what contexts the client is allowed to access and modify. The current implementation uses a simple Unix-based authentication; i.e., it trusts that the client program runs on a Unix machine.

The access manager creates a session context for each established connection. This context is removed when the connection is closed. A client may submit multiple queries on a given connection, meaning

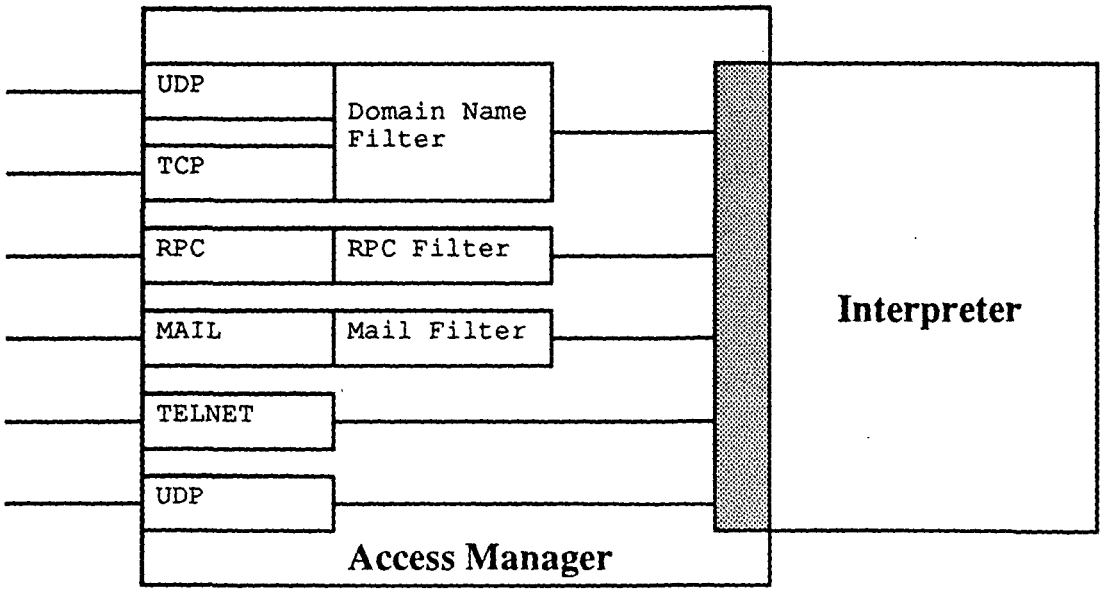

Figure 7: Univers Access Manager

that a single session context potentially services a sequence of queries. Thus, the result of one query can be saved in the session context to be used by subsequent queries.

At any given time, the access manager is responsible for multiplexing several connections onto Univers. The access manager is implemented by a single Unix process. This process services the connections in a round-robin fashion: it buffers incoming messages on each connection and invokes the interpreter when a complete query has been received. A limit is placed on the number of function calls allowed per query. Infinitely recurring user functions and queries requiring time (as measured in number of calls) more than linear in the size of the database are preempted.

4.7 Attribute Generators

Attributes denote resource properties. Some properties rarely change; e.g., a processor's architecture. Attributes that represent slowly changing properties are said to be *static*. The **add-attribute** and **delete-attribute** operations are sufficient for maintaining static attributes. Other properties change rapidly; e.g., a processor's load. Attributes representing these properties are said to be *dynamic*. The **add-attribute** and **delete-attribute** operations are not sufficient for maintaining dynamic attributes. Instead, Univers supports a mechanism, called *attribute generators*, that allows dynamic attributes to be maintained accurately in the object database.

An attribute generator is a C procedure that is called to compute the current value of an attribute. Attribute generators are stored in the database in place of static values. The procedure that implements a generator directly queries the resource represented by the object to retrieve its current value. The value returned by an attribute generator is cached in the database and reused to resolve subsequent queries. A

timer associated with this cache controls whether the currently cached value can be used or the generator is called again. Notice that this implies a lazy evaluation scheme: the generator is called only when the value is needed to resolve a query and the currently cached value is stale.

In practice, client queries operate on a particular attribute for an entire class of objects; e.g., the load attribute for all the processor objects. Because it is often just as easy to determine the value associated with each of a group of objects as it is for a single object, Univers is optimized to allow a single generator to be associated with a set of attributes. Whenever the generator is invoked for one attribute in the set, it returns fresh values for each of the attributes in the set. For example, Univers implements a single generator that computes the load attribute for all the processor objects in the database. The procedure that implements this generator does an RPC broadcast to the rstat program running on all the hosts in the local network. Any host that fails to respond is assigned an infinite load.

4.8 Query Optimizations

The naive implementation of a preference hierarchy does not lend itself to optimization by conventional database techniques that primarily focus on methods for efficiently joining large relations. Fortunately, certain characteristics of preferences allow the number of database operations to be significantly reduced for most queries. Intuitively, a query may be resolved by elaborating every approximation in the induced preference order and evaluating them from most preferred to least preferred. This method guarantees that the number of operations that must be performed will be less than the size of the induced preference. Often, the induced preference contains more than twenty composite approximations. It is possible to reduce significantly the number of database accesses for a large class of useful queries. Intuitively, this is because clients tend to select preferences that form increasing or decreasing set approximations. For example, the client preference used to define the yellow-pages resolution function first assumes that all of the client's attributes are accurate. If that approximation fails then it assumes that all of the attributes except the "most optional" are accurate. Thus, the set of attributes assumed accurate by this client preference always decreases in size as less preferred approximations are attempted.

Consider the following propositions that describe methods for optimizing resolution functions that use the exact database approximation function; we have derived similar optimizations for other approximation functions such as partial, possible, and also-ran. To characterize optimizations that might be used with exact, we assume that the preference hierarchy includes an arbitrary client preference and a database preference on the single approximation, exact.

Proposition 4.1 *If* **N** *and* **M** *are sets of attributes then* exact(**N**,**D**) ∩ exact(**M**,**D**) = exact(**N** ∪ **M**,**D**).

Proof: All objects in exact(**N**,**D**) are described by all of the attributes in **N**. All objects in exact(**M**,**D**) are described by all of the attributes in **M**. Therefore the objects in exact(**N**,**D**) ∩ exact(**M**,**D**) are described by all of the attributes in **N** ∪ **M**. □

Theorem 4.1 provides a method for composing and decomposing attribute sets. The sets of object that exactly match a set of attributes are those objects that are matched by all decompositions of the attribute set. This theorem may be used to prove the following theorems:

Proposition 4.2 *If* **N** *and* **M** *are sets of attributes such that* $\mathbf{N} \cap \mathbf{M} \neq \emptyset$ *then* $\text{exact}((\mathbf{N} \cap \mathbf{M}),\mathbf{D}) \supseteq \text{exact}(\mathbf{M},\mathbf{D})$.

Proof: From Theorem 4.1 $\text{exact}(\mathbf{N} \cap \mathbf{M},\mathbf{D}) \cap \text{exact}(\mathbf{M},\mathbf{D}) = \text{exact}(\mathbf{M},\mathbf{D})$. Therefore, $\text{exact}(\mathbf{N} \cap \mathbf{M},\mathbf{D}) \supseteq \text{exact}(\mathbf{M},\mathbf{D})$. □

Proposition 4.3 *If* **N** *and* **M** *are sets of attributes such that* $\mathbf{N} \subseteq \mathbf{M}$ *and* $\text{exact}(\mathbf{N},\mathbf{D}) = \emptyset$ *then* $\text{exact}(\mathbf{M},\mathbf{D}) = \emptyset$.

Proof: $\mathbf{N} \subseteq \mathbf{M}$, therefore, $\mathbf{N} \cap \mathbf{M} = \mathbf{N}$. From Theorem 4.2, $\text{exact}(\mathbf{N} \cap \mathbf{M},\mathbf{D}) = \text{exact}(\mathbf{N},\mathbf{D}) \supseteq \text{exact}(\mathbf{M},\mathbf{D})$. But, since $\text{exact}(\mathbf{A},\mathbf{D}) = \emptyset$, $\text{exact}(\mathbf{M},\mathbf{D}) = \emptyset$. □

Proposition 4.4 *If* **N** *and* **M** *are sets of attributes such that* $\mathbf{N} \subseteq \mathbf{M}$ *and* $\text{exact}(\mathbf{M} - \mathbf{N},\mathbf{D}) \cap \text{exact}(\mathbf{N},\mathbf{D}) = \emptyset$ *then* $\text{exact}(\mathbf{M},\mathbf{D}) = \emptyset$.

Proof: $\mathbf{N} \subseteq \mathbf{M}$ so that $(\mathbf{M} - \mathbf{N}) \cup \mathbf{N} = \mathbf{M}$; and $\text{exact}(\mathbf{M} - \mathbf{N},\mathbf{D}) \cap \text{exact}(\mathbf{N},\mathbf{D}) = \emptyset$. Therefore, from Theorem 4.1, $\text{exact}(\mathbf{M},\mathbf{D}) = \emptyset$. □

Theorem 4.2 says that adding more attributes to a name will decrease the number of objects that exactly match it. Theorem 4.3 shows that if a set of attributes matches no objects, then a superset of the attributes also matches no objects. Finally, according to Theorem 4.4, if a set of attributes matches a set of objects but a superset does not, then the attributes in the difference of the sets match none of the objects. Given these facts, there are two principles that allow a series of attribute sets to be collapsed.

Proposition 4.5 *If* $\mathbf{N}_1 \subseteq \mathbf{N}_2 \subseteq \ldots \subseteq \mathbf{N}_n$ *is a series of preferred attribute sets, then* $\text{exact}(\mathbf{N}_i,\mathbf{D})$ *is the set of objects preferred by the series if* $\mathbf{N}_i$ *is the first non-empty set of attributes in the series.*

Proof: Let $\mathbf{N}_i$ be the first non-empty set of attributes. If $\text{exact}(\mathbf{N}_i,\mathbf{D}) \neq \emptyset$ then it is the most preferred response. If $\text{exact}(\mathbf{N}_i,\mathbf{D}) = \emptyset$ then by repeated application of Theorem 4.3, $\text{exact}(\mathbf{N}_j,\mathbf{D}) = \emptyset$ for all $j > i$. In this case, the preferred answer is empty as determined by $\text{exact}(\mathbf{N}_i,\mathbf{D})$. □

Proposition 4.6 *If* $\mathbf{N}_1 \supseteq \mathbf{N}_2 \supseteq \ldots \supseteq \mathbf{N}_n$ *is a series of preferred attribute sets, then* $\text{exact}(\mathbf{N}_i,\mathbf{D})$ *is the set of objects preferred by the series if* $\text{exact}(\mathbf{N}_i,\mathbf{D})$ *is non-empty and* $\text{exact}(\mathbf{N}_{i-1} - \mathbf{N}_i,\mathbf{D}) \cap \text{exact}(\mathbf{N}_i,\mathbf{D})$ *is empty.*

Proof: Pick i such that $\text{exact}(\mathbf{N}_i,\mathbf{D}) \neq \emptyset$ and $\text{exact}(\mathbf{N}_{i-1} - \mathbf{N}_i,\mathbf{D}) \cap \text{exact}(\mathbf{N}_i,\mathbf{D}) = \emptyset$. By Theorem 4.4, $\text{exact}(\mathbf{N}_{i-1},\mathbf{D}) = \emptyset$. Furthermore, for all $j < i$, $\text{exact}(\mathbf{N}_j,\mathbf{D}) = \emptyset$ by Theorem 4.3. Therefore, $\text{exact}(\mathbf{N}_i,\mathbf{D})$ is the first non-empty set of objects and is the preferred answer. □

Thus, a query may be optimized according to Theorem 4.5 by looking for one or more series of preferred attribute sets that can be ordered by subset inclusion. When such a series is found, it can be replaced by the first non-empty set of attributes. For example, if the approximation functions in a client preference returned, in preferred order, the attribute sets (**a**), (**a b**), and (**a b c**) then the interpreter need only compute $\text{exact}((\mathbf{a}),\mathbf{D})$. Theorem 4.6 concerns series that are ordered by superset inclusion. This kind of series can be evaluated in reverse. That is, starting with the smallest non-empty set, one can work backwards through the series, intersecting the objects matched by the difference set with the objects

previously matched. If the intersection is empty, then the system returns the set of previously matched objects. If the client approximation functions compute the attribute sets (a b) and (a), for example, then the interpreter need only compute exact((a),D) and exact((a),D) ∩ exact((b),D) . In other words, the first optimization collapses a series of attribute sets to the smallest non-empty set, while the second optimization collapses the series to an attribute set no larger than the first set in the series.

The optimizations are implemented as a three stage process. During the first stage, each client approximation function is evaluated to generate a list of attribute sets ordered by the client preference. In the next stage, the sets are used to build an evaluation tree. The attribute sets are processed in reverse order, i.e. from least preferred to most preferred. An attribute set is added to the tree depending on its relationship to the leftmost path in the tree. Figure 8 shows the construction of a tree for the four cases that may exist. Given a tree that contains B and C as branches from the top node, an attribute set A may be added depending on its relationship to B. If $A \subseteq B$, then according to Theorem 4.5, A may replace B as in Figure 8(a). If $B \subseteq A$, then $A - B$ becomes the leftmost child of B as in Figure 8(b). If B already has children, then recursively process $A - B$ relative to the leftmost child of B. This has the effect of ordering sets of attributes according to Theorem 4.6. If $D = A \cap B \neq \emptyset$, then using Theorem 4.2 D may precede both $A - D$ and $B - D$, i.e. an object that is not selected by D will not be selected by either A or B. Figure 8(c) depicts this situation. Finally, if there are no common attributes in the two sets then A is added as a new leftmost branch of the tree as in Figure 8(d). The nodes of the tree are marked if they have computed enough information to return a matched set of objects.

In the final stage, each node in the evaluation tree is visited using a pre-order traversal. The expression associated with a node is evaluated relative to the set of objects associated with the parent of the node. If no objects are matched by the set of attributes then the tree is trimmed below that node and evaluation returns to the parent node. If all of the children of a node fail to compute a non-empty set of objects and the node is marked then the set of objects that matched the node's attributes are returned. If the node is unmarked then the traversal returns to the node's parent.

5 UNIVERS NAMING SYSTEM

This chapter describes the construction of the Univers naming system using the facilities provided by a set of Univers name servers described in Chapter 4. In the sense that a Univers name server supports a programming language, this chapter describes a "program" that provides three kinds of naming services: a conventional lookup service, a yellow-pages service, and a white-pages service. It specifies several resolution functions and types that can be loaded into a single context, and the methods that can be used to organize and search a set of contexts to find a particular object.

5.1 Intra-Context Organization

Each context in the Univers naming system is autonomously administered by some entity. Each entity determines the set of resources to be represented in the context it administers. The exception to this rule is a set of well-known functions and types that are contained in the meta-context of every name server to provide some common naming services. This section describes various type and resolution function

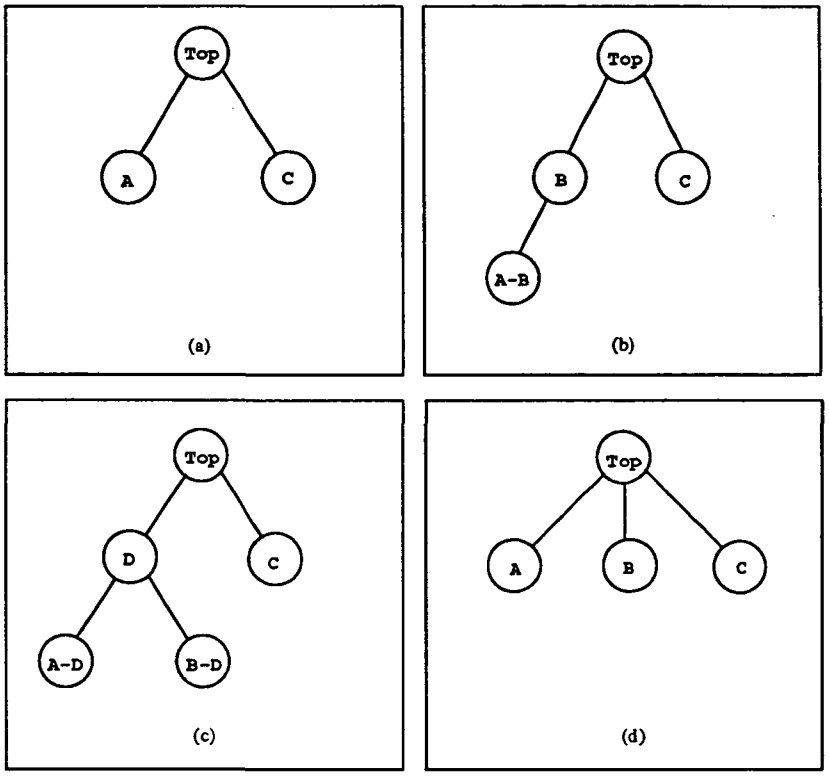

Figure 8: Cases for tree construction.

objects that have been especially useful for providing a conventional lookup service, a yellow-pages resource locator, and a white-pages directory service. In addition, this section contains an analysis of the suitability of the resolution functions that are presented and a discussion of their performance within a single Univers name server.

Object Types

Several type objects play an especially important role in resolving names. As discussed in Chapter 4, types help clients precisely distinguish the kind of object to be identified. For example, the attribute (= name gothic) describes both a processor and a font. The processor type helps a client distinguish between the processor named gothic and the font named gothic.

The traits that constitute a type are chosen with two goals in mind. First, the traits must be sufficient to distinguish a certain class of object, yet be able to describe every object in the class. If the set of common traits for a class of objects is too small to be useful—it does not serve to distinguish objects in the class—then it is often the case that the type can be decomposed into two or more subtypes. The primary type object can point clients to objects that represent its subtypes. Second, the traits in a type should represent properties that will be maintained in the database. For example, if the naming system cannot guarantee that a user's home phone will be included in the database for every user object then the

type user should not contain the trait home-phone. Generally, the traits that constitute a type should be closed; i.e. the system should guarantee that any object in the class represented by the type will have at least one attribute for each trait in the type registered in the database.

The type objects that are included in the meta-context of a server are those that provide special support for the service provided by the naming system or those that are commonly used by clients. In general, a client can apply a resolution function to any type of object.Resolution functions are often tailored to be most useful for naming objects of one particular type as with the white-pages family of resolution functions; however, the family of conventional resolution functions are designed to resolve names that describe any type of object. For example, the Univers naming system includes the following two types for objects that are commonly used by the system's clients:

```
( (name citation)
  (necessary-traits (author title)) )
( (name alias)
  (necessary-traits (alias name)) )
```

The citation type describes the structure of objects that represent entries in a bibliography. Often the database will contain several subtypes of citation that represent the different kinds of entries in a bibliography such as article, manual, or tech-report. The alias type describes the structure of objects that represent a user's personal list of aliases.

Generally, a client that uses a yellow-pages service wants to locate one or more computational resources of a single type such as a set of processors or printers. The Univers naming system defines the following types for objects that represent computational resources:

```
( (name processor)
  (necessary-traits (name address os architecture speed protocol service)) )
( (name printer)
  (necessary-traits (name address office architecture speed resolution)) )
( (name service)
  (necessary-traits (name processor protocol)) )
```

The service type describes the structure of objects that represent an available service. For example, the object

```
( (name webster) (processor yucca) (protocol tcp) (port 2049) )
```

describes a Webster dictionary service provided by the processor yucca using tcp as the access protocol.

A white-pages naming service is used exclusively to locate user and organization objects. Usually, clients use the service to determine properties such as a user's mailbox or phone number. The following white-pages type objects are defined by the Univers naming system:

```
( (name user)
  (necessary-traits (name mailbox login phone office domain)) )
( (name organization)
  (necessary-traits (name postmaster liaison context)) )
```

Resolution Functions

The Univers naming system provides three families of resolution functions: conventional lookup functions, yellow-pages functions, and white-pages functions. The family of conventional lookup functions provides a generic service for resolving names; i.e. the functions know nothing of the semantics of the objects that the client identifies. The yellow-pages functions are tailored for discovering a set of computational resources. The family of white-pages functions resolves names with the assumption that the client is a user and that the object to be identified is either a user or an organization. These resolution functions are defined using the various approximation functions informally defined in Chapter 3. Several of these approximations will be formally defined later in this section.

The resolution functions defined in this section make use of a special preference hierarchy designed to compute the set of objects from a given context with a specified type. The hierarchy is defined as:

$\prec_{CONTEXT}$: context-attribute

$\prec_{context}$: resolve-context

$\prec_{TYPE}$: empty-attribute $\prec_{type}$ type-attribute

$\prec_{type}$: select-type

where $\prec_{type} \leftarrow \prec_{TYPE} \leftarrow \prec_{context} \leftarrow \prec_{CONTEXT}$. The most preferred method is to select those objects from the given context with the conjunction of types specified in the query. If that fails, then the type attribute is assumed to be inaccurate. In that case, all objects are returned regardless of their type.

Conventional Lookup Functions:

The conventional lookup service provided by the Univers naming system consists of three resolution functions: equal1, equal2, and equal3. The function equal3 is presented in Chapter 3. Each attempts to respond precisely to a client's query using general properties of the database. For example, equal3 assumes that the client intends to identify any object that is uniquely matched by the unambiguous attributes in a name.

The equal1 function returns the object that uniquely matches the attributes in the name. It encompasses the functionality provided by conventional naming systems that map unambiguous names to a value; e.g. the object's address. The preference hierarchy encapsulates the belief that the database contains accurate and unambiguous attributes. Resolution function equal1 is formally defined by the preference hierarchy:

$\prec_U$ universal

$\prec_M$ unique

where $\prec_M \leftarrow \prec_U$. The Univers function that implements equal1 takes a set of attributes and an object type. Clients often use equal1 to resolve aliases or to map mnemonic names to addresses. For example, a client can query Univers with

```
(equal1 ((= name saguaro)) processor system-context)
```

and the naming system will respond with the description of the processor object named saguaro from the system context. Note that a mnemonic name can be unambiguous relative to certain sets of objects; e.g. those objects in a single local context with type processor.

The equal2 resolution function returns the set of objects that are matched by all of the specified attributes. Like equal1, equal2 encapsulates the assumption that the information specified by the client is accurate and that the database is honest, i.e. it contains accurate information. Function equal2 differs from equal1 in that the information may be ambiguous; i.e. equal2 may return more than one object. Formally, equal2 is defined as $\prec_M \Leftarrow \prec_U$ over the preferences:

$\prec_U$: universal

$\prec_M$: exact

Yellow-Pages Function:

A yellow-pages resolution function allows clients to discriminate among a set of similar computational resources—such as printers, processors, databases, network services, and so on—according to their particular characteristics. The descriptive yellow-pages function [Pete87], denoted yp is defined as follows.

The set of attributes presented to yp are partitioned into mandatory and optional subsets, denoted by N_m and N_o, respectively, where the optional attribute set is ordered; i.e., $N_o = \{a_1, a_2, ...a_n\}$. The most preferred answer matches the mandatory attributes and all of the optional attributes; however, if no such match exists then a match based on optional attribute a_i is preferred over a match based on attribute a_j if $i < j$. Thus, n optional attributes define $n + 1$ different client approximation functions $o_0, o_1, \ldots o_n$ where o_i returns the mandatory attributes and the first i optional attributes. Thus, for a given set of attributes, yp is defined by $\prec_M \Leftarrow \prec_A$, where $\prec_A$ and $\prec_M$ are given by:

$\prec_A$: $o_0 \prec_A o_1 \prec_A \ldots \prec_A o_n$

$\prec_M$: *exact*

The Univers resolution function that implements yp accepts two sets of attributes that correspond to the mandatory and optional attributes, an object type, and an optional context that defaults to the system context. A sample call of yp would look like:

(yp ((= arch 68020) (= os unix)) ((< load 2) (> speed 3)) processor)

This query selects a processor with a 68020 architecture and the Unix operating system. If more than one processor exists that matches that description, then yp first eliminates those with a load greater than two and then those with a speed rating less than three.

White-Pages Functions:

The family of white-pages resolution functions are used to identify users and organizations [Pete88]. The five functions in this family identify objects relative to assumptions about the usefulness of closed and open attributes. In particular, the white-pages functions assume that precise matches—conjunctive

matches based on the unanimous approximation—on closed attributes are likely to include the object that the client seeks. Imprecise matches—disjunctive matches based on the also-ran approximation—on open attributes compensate for incomplete information in the database.

Function all, returns those objects matched by any of the attributes that the client specifies. This resolution function assumes that the client possesses inaccurate information about the object it wants to identify. Rather than chance not returning the objects that the client wants to identify, all returns any object that the client might be identifying. Formally, all is defined by $\prec_V \Leftarrow \prec_U$ on the following preferences:

$\prec_U$: *universal*

$\prec_V$: *also-ran*

Function lookup is similar to equal1 and equal2 but it returns the single object matched by the closed attributes in the name. This function encapsulates an assumption that the database contains incomplete information about some of the objects that the client intends to identify. Formally, lookup is defined by $\prec_V \Leftarrow \prec_R$ on the following preferences:

$\prec_R$: *closed*

$\prec_V$: *unique*

Function wp1 computes the conjunction of the specified attributes and gives preference to closed attributes over open attributes. It encapsulates the assumption that the set of objects precisely matched by the closed attributes are most likely to include the objects that the client attempts to identify. It is defined by $\prec_V \Leftarrow \prec_R$ on the following preferences:

$\prec_R$: *open* $\prec_R$ *closed*

$\prec_V$: *also-ran* $\prec_V$ *unanimous*

The resolution function wp2 is like wp1 except that it uses open attributes to reduce the set of objects returned. Formally, wp2 is defined by $\prec_v \Leftarrow \prec_r \Leftarrow \prec_V \Leftarrow \prec_R$ on the following preferences:

$\prec_R$: *open* $\prec_R$ *closed*

$\prec_V$: *also-ran* $\prec_V$ *unanimous* $\prec_V$ *unique*

$\prec_r$: *open*

$\prec_v$: *identity* $\prec_v$ *also-ran* $\prec_v$ *unanimous*

Finally, wp3 completely avoids precise matches of open attributes. It encapsulates the assumption that the database does not contain enough information about the objects the client seeks to identify them with an exact match of the open attributes in the name. Rather, the open attributes are used to reduce the number of objects precisely matched by closed attributes. Formally, wp3 is defined by $\prec_v \Leftarrow \prec_r \Leftarrow \prec_V \Leftarrow \prec_R$ on the following preferences:

$\prec_R$: *closed*

$\prec_V$: *identity* $\prec_V$ *also-ran* $\prec_V$ *unanimous*

$\prec_r$: *open*

$\prec_v$: *identity* $\prec_v$ *also-ran*

The resolution functions that constitute the white-pages family limit objects to those of type user or organization. Thus, the Ulisp functions that implement the white-pages resolution functions take a single set of attributes as an argument. For example, a client may present this query to the naming system:

```
(wp2 ((= name "Wyatt Earp")
      (∈ research-associates "Ike Clanton")
      (= research-interests fast-guns)) )
```

and the naming system would respond with the objects of type user or organization that are computed by the most preferred composite approximation in the induced preference defined by wp2.

Analysis

Using the tools provided in Chapter 3, it is possible to argue the suitability of the resolution functions in the Univers naming system. This section discusses the assumptions used by the white-pages family of resolution functions and the suitability of the resolution functions within that family. It also comments on conventional resolution functions and their relationship to white-pages functions.

White-Pages Functions:

The design of the white-pages resolution functions is constrained by three assumptions. The first assumption is that the database is honest but potentially incomplete. Univers automatically manages some of the information in the database using attribute generators. These attributes are considered closed because they are maintained by a reliable system administrator. The remainder of the information is added to the database by the system's clients. These attributes are considered open because one client might register different information than another. Second, the attributes contained in a name may be inaccurate: uses commonly forget or incorrectly remember information about other people. Third, as mentioned earlier, clients of a white-pages naming service are generally interested in complete answers—that is, ones that definitely contain the objects the client intended to identify. This section shows how the first two assumptions affect the completeness of the white-pages family of resolution functions.

The preferences used to construct the white-pages resolution functions consist of three client approximation functions and four database approximation functions. Although based on the example approximation functions informally presented in Chapter 3, we define these functions formally so that we may reason about the resolution functions that use them.

Definition 5.1 (White-Pages Client Approximation Functions) *The client approximation functions used by the white-pages family of resolution functions are:*

- $universal(\mathbf{N}) = \mathbf{N}$

- *closed*(**N**) = {**t** : **v** ∈ **N** | *for all* **u**, μ(**t** : **u**) *describes* μ(**x**) *implies that* **t** : **u** *is registered for* **x**}
- *open*(**N**) = **N** − *closed*(**N**)

Definition 5.2 (White-Pages Database Approximation Functions) *The database approximation functions used by the white-pages family of resolution functions are:*

- *identity*(**N**, **D**) = **D**
- *also-ran*(**N**, **D**) = {**x** ∈ **D** | *there exists an* **a** ∈ **N** *such that* **a** *is registered for* **x**}
- *unanimous*(**N**, **D**) = {**x** ∈ **D** | *for every* **a** ∈ **N**, **a** *is registered for* **x**}
- *unique*(**N**, **D**) = *if* |*unanimous*(**N**, **D**)| = 1 *then unanimous*(**N**, **D**) *otherwise* ∅.

Given these definitions, it is easy to show that a discrimination order $\sqsubseteq_\Delta$ exists for the white-pages resolution functions. The order is as follows:

Proposition 5.1 *For* Δ *containing all name/database pairs:*

$$\mathsf{all} \sqsubseteq_\Delta \mathsf{wp1} \sqsubseteq_\Delta \mathsf{wp2} \sqsubseteq_\Delta \mathsf{lookup}$$

Before considering the completeness of these resolution functions, we make the following assumption to simplify the analysis: an inaccurate attribute in the client specification—one that does not describe any of the objects that the client intends to identify—does not match any object within the database. While in practice the assumption cannot be guaranteed, it is unlikely that a client will specify an inaccurate attribute that actually describes another object.[4] That is, the information in the naming system is sparse in the same sense that error detection codes such as parity and Hamming codes use the distance between reasonable values to detect the presence of errors in binary information.

The following claims about completeness are valid under this assumption:

Proposition 5.2 *The function* all *is complete for all sets of attributes that contain at least one attribute that is registered for each object that the client intended to identify.*

Proposition 5.3 *Function* wp1 *is complete on names that contain a non-empty set, C, of closed attributes that are accurate—that is, each attribute in C describes every object that the client intends to identify.*

Proof: Consider Γ_{cu} = {(**N**, **D**)|**x** ∈ *oracle*(**N**, **D**) implies that μ(*closed*(**N**)) *represents* μ(**x**)}. If **x** ∈ *oracle*(**N**, **D**) then μ(*closed*(**N**)) *represents* μ(**x**). By the definition of *closed*, each of these attributes is registered in **D**. Since ⟨*closed*, *unanimous*⟩ returns the objects that match all of the closed attributes, it returns **x**. Therefore, **x** ∈ *oracle*(**N**, **D**) implies that **x** ∈ *unanimous*(*closed*(**N**), **D**) and ⟨*closed*, *unanimous*⟩ is complete on Γ_{cu}.

[4]It is possible to formally argue the completeness of resolution functions without this assumption, however, it is far more cumbersome to describe the set of name/database pairs where an approximation fails.

Let $\Gamma_{ca} = \{(\mathrm{N},\mathrm{D})|$ there exists a non-empty set $C \subseteq$ *closed*(N) such that $\mathrm{x} \in$ *oracle*(N,D) implies that $\mu(C)$ *represents* $\mu(\mathrm{x})\}$. If $\mathrm{x} \in$ *oracle*(N,D) then there is a non-empty set of closed attributes, C, that represents x. By definition, each attribute in C is registered for x in D. *also-ran(closed(*N*),*D*)* is the set of objects in D for which at least one of the closed attributes is registered. Therefore, x is contained in *also-ran(closed(*N*),*D*)* if x is in *oracle*(N,D) and $\langle$*closed, also-ran*$\rangle$ is complete on Γ_{ca}.

E_{cu} consists of the name/database pairs that have either no closed attributes or at least one erroneous closed attribute and $\Gamma_{cu} \subseteq \Gamma_{ca}$. At this point it is trivial to show that Proposition 3.4 implies that $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca}) = \Gamma_{ca} \subseteq \Gamma_{wp_1}$. Thus, Proposition 5.3 holds. □

We previously defined a closed attribute as one that is guaranteed to be registered for every object it describes. A resolution function that uses open attributes may be incomplete if some open attribute in the name is not registered for some objects that the client intends to identify. For example, $\langle$*open, unanimous*$\rangle$ returns a set of objects that match every attribute in the name. However, one of the attributes in the name may not be registered for an object that the client seeks, and as a consequence that object does not match every attribute in the name according to the database. This leads to the following definition: An attribute a is *relevant* to a set of objects D if for every object x in D, $\mu(\mathrm{a})$ describes $\mu(\mathrm{x})$ implies a is registered for x. (Thus, an attribute is closed if it is relevant to the entire database). Using this definition, it is possible to show the following two propositions:

Proposition 5.4 *Function* wp1 *is complete on names where all open attributes in the name are accurate and relevant to the set of objects that the client intends to identify.*

Proof: Let $\Gamma_{ou} = \{(\mathrm{N},\mathrm{D})|\mathrm{x} \in$ *oracle*(N,D) implies that $\mu($*open*$(\mathrm{N}))$ *represents* $\mu(\mathrm{x})$ and each attribute in *open*(N) is registered for x$\}$. If $\mathrm{x} \in$ *oracle*(N,D) then $\mu($*open*$(\mathrm{N}))$ *represents* $\mu(\mathrm{x})$ and each attribute in *open*(N) is registered for x in D. Recall that, $\langle$*open, unanimous*$\rangle$ returns the objects that match all of the open attributes. Therefore, $\mathrm{x} \in$ *unanimous*$($*open*$(\mathrm{N}),\mathrm{D})$ if $\mathrm{x} \in$ *oracle*(N,D) so that according to the definition of completeness $\langle$*closed, unanimous*$\rangle$ is complete on Γ_{ou}.

When wp1 resolves names using $\langle$*open, unanimous*$\rangle$ both of the earlier composite approximation returned empty sets. This occurs whenever all of the closed attributes are inaccurate; i.e. $E_{cu} \cap E_{ca}$ is the set of all (N,D) where *closed*(N) contains no closed attributes that describe the objects the client intended to identify. $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca})$ includes any name that contains a non-empty set of closed attributes that describe the client's objects. Γ_{ou} may be partitioned into two disjoint sets—one where the names contain a accurate closed attributes and one where they do not contain any accurate closed attributes. It is trivial to show that the former is contained in $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca})$ and the latter in $(E_{cu} \cap E_{ca} \cap \Gamma_{ou})$. Thus, $\Gamma_{ou} \subseteq \Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca}) \cup (E_{cu} \cap E_{ca} \cap \Gamma_{ou})$. Proposition 3.4 implies that $\Gamma_{ou} \subseteq \Gamma_{wp_1}$. Therefore, wp1 is complete on Γ_{ou} and Proposition 5.4 holds. □

Proposition 5.5 *The resolution function* wp1 *is complete on names that contain at least one inaccurate, open attribute and a non-empty set of accurate, open attributes such that at least one attribute in this set is registered for every object that the client intended to identify.*

Proof: Let $\Gamma_{oa} = \{(\mathrm{N},\mathrm{D})|\mathrm{x} \in$ *oracle*(N,D) implies that there exists a non-empty set $O \subseteq$ *open*(N) such that $\mu(O)$ *represents* $\mu(\mathrm{x})$ and each attribute in O is registered for x$\}$. If $\mathrm{x} \in$ *oracle*(N,D) then

there is a non-empty set of open attributes, O, that represents **x** and each attribute in O is registered for **x** in **D**. *Also-ran(open(***N***),***D***)* is the set of objects in **D** for which at least one open attributes is registered. Therefore, **x** is contained in *also-ran(open(***N***),***D***)* if **x** is in $oracle(\mathbf{N}, \mathbf{D})$ and $\langle open, unanimous \rangle$ is complete on Γ_{oa}.

If every name **N** contains at least one inaccurate attribute, then Γ_{ou} is empty. In this case, $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca}) \cup (E_{cu} \cap E_{ca} \cap \Gamma_{ou}) \cup (E_{cu} \cap E_{ca} \cap E_{ou} \cap \Gamma_{oa})$ is equivalent to $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca}) \cup (E_{cu} \cap E_{ca} \cap \Gamma_{oa})$. As before, this expression contains Γ_{oa}. Therefore, wp1 is complete on Γ_{oa} and Proposition 5.5 holds. □

Since wp2 uses the open attributes to "trim" the set of objects returned, it may not always be complete. However, we can give sufficient conditions for its completeness. First, notice that if there are no open attributes present in a set of attributes submitted by the client, then no filtering is possible, and wp2 is identical to wp1. Thus, we have

Proposition 5.6 *Function* wp2 *is complete on names that do not contain any open attributes but do contain a non-empty set of closed attributes that describe the objects the client intends to identify.*

We can, however, do better than this. Proposition 5.6 is based on the fact that if there are no open attributes in the set, then they cannot contribute to the discarding of an intended object. Thus, the reason for any possible incompleteness is that some open attribute in the given set of attributes was not registered for the intended object. If an open attribute is relevant to the objects that the client intends to identify, then it does not contribute to the discarding of an intended object.

Proposition 5.7 *Function* wp2 *is complete on names that contain a non-empty set of accurate attributes that are relevant to the set of objects that the client intends to identify.*

Note that Proposition 5.6 is as a special case of Proposition 5.7 because a closed attribute is by definition relevant to all objects.

The resolution functions provided by the white-pages family attempt to respond with complete answers. The analysis presented in this section points out one design decision that adversely affects this goal within the definition of wp1 and wp2. The composite function $\langle open, unanimous \rangle$ selects objects that match all open attributes in a name. This function is complete only if every open attribute in the name is relevant to all of the objects that the client identifies; i.e. only if the attribute is closed relative to the set of objects that the client identifies. Because the very definition of an open attribute is one that is not relevant to a large portion of the database, it is unlikely that more than one or two open attributes will be relevant to the set of objects that a client attempts to identify. Based on this understanding, the design of wp3 resembles wp1 except that it never attempts a unanimous match on open attributes. The following two propositions hold for wp3.

Proposition 5.8 *Function* wp3 *is more discriminating than* wp1 *except when:*

1. *Every open attribute in the name is accurate and relevant to every object that the client identifies.*
2. *Every attribute in the name is either inaccurate or irrelevant to every object that the client identifies.*

Proposition 5.9 *Function* wp3 *is complete if for every object that the client intends to identify there is an accurate, open attribute in the name that is relevant to it.*

This example shows that the preference hierarchy can help a system designer understand how well a resolution function meets its goals. wp1 sacrifices completeness for a substantial set of name/database pairs because it uses an exact match on open attributes, whereas wp3 overcomes this weakness at the expense of being less discriminating for certain unlikely sets of attributes.

Resolution Functions for Conventional Naming Systems:

The family of conventional resolution functions operates under the assumption that the database contains accurate information. It is important to note that conventional resolution functions, because of the assumptions that constrain them, generally sacrifice completeness for the sake of soundness. White-pages functions, on the other hand, often sacrifice soundness for the sake of completeness. As a result, conventional resolution functions are likely to deal exclusively with induced approximations that are high in the discrimination hierarchy—e.g. $\langle unambiguous, exact \rangle$ and $\langle closed, unique \rangle$ —whereas white-pages functions are likely to use induced approximations that are low in the hierarchy—e.g. $\langle ambiguous, possible \rangle$ and $\langle open, also\text{-}ran \rangle$.

The conventional resolution function equal1 is an example of a conventional resolution function which assumes that the database is accurate and complete. Given this assumption, it is easy to to see the following:

Proposition 5.10 *Function* equal1 *is sound for all names that accurately describe the objects that the client intends to identify.*

Notice, however, that equal1 is not complete unless the attributes in the name uniquely describe the object that the client intends to identify. Thus,

Proposition 5.11 *Function* equal1 *is complete for all names that contain an unambiguous attribute.*

The family of conventional resolution functions provides several levels of discrimination. The order is given by the following two propositions:

Proposition 5.12 *If* Δ *contains all name/database pairs then*

$$\text{equal2} \sqsubseteq_{\Delta} \text{equal1}$$

Proposition 5.13 *If* Δ *containing name/database pairs where the database is honest and the name contains attributes that accurately describe the objects that the client intends to identify then*

$$\text{equal3} \sqsubseteq_{\Delta} \text{equal2} \sqsubseteq_{\Delta} \text{equal1}$$

Context	Object Types	Count
Meta	Type	7
	Function	7
	Context	25
System	User	306
	Processor	471
	Printer	22
Bibliography	Citation	1511
Biology	DNA Pattern	53
Private		720

Table 9: Breakdown of Univers Configuration.

Performance

This section reports on the space requirements and performance of Univers. For the purpose of this section, we evaluate the configuration of Univers in the Department of Computer Science at the University of Arizona. That configuration contains 3096 total objects distributed across 25 contexts. There are a total of 21789 attributes for an average of seven attributes per object. Attribute traits and values average seven and 16 characters in length, respectively. There are a total of 75 distinct traits. Table 9 contains a breakdown of the various resources that are represented in the database.

The database occupies 2.08 megabytes of dynamically allocated memory. This implies 95 bytes per attribute, or approximately 675 bytes per object. Given that the raw attributes are on average 23 bytes in length, this means a 4-fold increase from the size of raw data to the size of the database representation of the data. Of the approximately one and a half megabytes of overhead, a quarter of a megabyte corresponds to the overhead imposed by malloc which requires four bookkeeping bytes for every chunk of allocated memory, and half a megabyte corresponds to attribute records, each of which is 24 bytes long. The other three-quarters of a megabyte correspond to various indices. For example, the database contains 75 large object sets (implemented as bit vectors) and 30 small object sets (implemented as lists). Given the small size of each data item (23 bytes) and the significant amount of indexing supported by Univers, we find the storage requirements of Univers to be acceptable.

In order to evaluate the performance of Univers, we measured the time necessary to apply several resolution functions to the database. Note that these functions were implemented as Ulisp primitives. The tests were run on a Vax 6850 with the functions submitted locally; i.e., without sending and receiving messages over a network. The reported times were derived by invoking each function 1000 times and dividing the total elapsed time by 1000. Function equal2 was given a single attribute, function yp was given three mandatory and two optional attributes, and function wp1 was given two attributes. The results are summarized in Table 10. Note that the performance of these functions depends on the number of objects of the specified context and type; they are independent of the total number of objects in the database. This is because each function uses the select-context and select-type operations to quickly

Function	Time	Search Space
equal1	30 msec	493 objects of type alias
equal2	109 msec	1511 objects in bibliography context
yp	106 msec	471 objects of type processor
wp1	75 msec	306 objects of type user

Table 10: Performance of Various Resolution Functions

eliminate from consideration those objects that are not of interest. Also note that the caching done by the object database is used only within a single call; it does not carry over between calls.

The preceding evaluation ignores two important possibilities. First, all of the attribute values are given by simple strings rather than regular expressions. While this is appropriate for most attributes, Univers' utility is greatly improved if certain attributes—e.g., user names—are given by regular expressions. Configuring the database to have the name attributes for the 306 user objects given by a Profile-based regular expression adds 50k-bytes to the overall size of the database. It also increases the running time of wp1 to 184 msec. Second, the performance figures do not include the cost of invoking an attribute generator; i.e., the values are already contained in the database. Clearly, generators can have a significant impact on performance. For example, the generator that computes the load attribute for a collection of processors has a 5 second interval during which it waits for processors to respond. While this is significant in comparison to 106 msec, it is quite acceptable for a client that is trying to find a processor with a light load. In general, the time it takes to compute a dynamic value depends on the generator and cannot be improved by Univers.

Compared with more specialized name servers, Univers performs noticeably slower. For example, the DNS lookup function takes approximately 3 msec to search 500 objects. Similarly, a stand alone implementation of the Profile naming system [Pete88] performs the same search as wp2 in about one tenth the time. The reason for such performance differences is that Univers does not optimize for the comparison operations that will normally be performed on an attribute; for example, string equality. That is, in order to allow general comparison operations on all attributes, the column of values associated with a trait is always searched linearly. In contrast, the DNS lookup function uses a hash table to search for values. In the future, Univers could be optimized to account for the expected operations on a property as described in Chapter 6.

5.2 Inter-Context Organization

The previous section focused on the function and type objects within a single context. This section outlines several methods for organizing a collection of contexts in a way that allows clients of the system to search for objects in more than one context. Other methods are certainly possible.

Recall that the client of a conventional naming system identifies an object using its location in the name space provided by the naming system. Although the client of the Univers naming system can identify an object independently of its location in the name space, the naming system relies on an organization of the

contexts that comprise the naming system to resolve the client's description. The client of the naming system specifies a set of methods for searching the name space by selecting a resolution function that includes combinations of client and database approximation functions supporting non-local searching. This section describes the structure of the Univers name space and gives several examples of methods that have proven useful for locating objects within the name space.

Name Space Organization

The Univers name space organizes a set of contexts into an arbitrary graph as shown in Figure 9. In this diagram, a rectangle represents a name server that implements one or more contexts; contexts are represented by large circles. Within each context are a possibly empty set of context objects—represented in Figure 9 by the small boxes—that reference another context resource in the naming system; either in the local name server or in a remote name server. In the latter case, there will also exist a server object that enumerates the access protocols that can be used to query the remote server. For example, context D3 contains the following three objects that represent the connections to context E1 and D2.

```
( (name E1)
  (server "Server E")
  (administrator "Dept E")
  (objects "Employees of Department E") )

( (name D2)
  (server "Server D")
  (administrator "Dept D")
  (objects "Meta Context of Server D") )

( (name "Server D")
  (processor cholla)
  (protocol (TCP UDP RPC) )
```

Thus, it is the context attribute that effectively links together name servers that make up the name space. Note that the meta context in each name server contains a context object for each context implemented by the server. For example, in Figure 9 contexts A1, B1, C1, D2, and E1 are meta contexts.

The Univers name space is an arbitrary "knows about" graph—a graph where an edge represents another object known to a context. This design has several advantages. First, there is no need for a central authority to maintain the connections between contexts. Instead, a new context is added by adding a context object to the meta context of a Univers name server. Any context that knows about the server's meta context can discover the new context. Similarly, the new context, which must contain an object that represents the meta context, can discover how to contact any other context that the meta context knows about. In order to add a name server to the system, the new meta context must be advertised to other servers in the system. This can be done by adding an object to the meta context of one or more "well known" name servers. Thus, the system can grow and change without the need for any central authority.

Second, administrative organizations cooperate with one another to provide more structured organization. The organizations that participate in the naming system described in Figure 9, for example, can

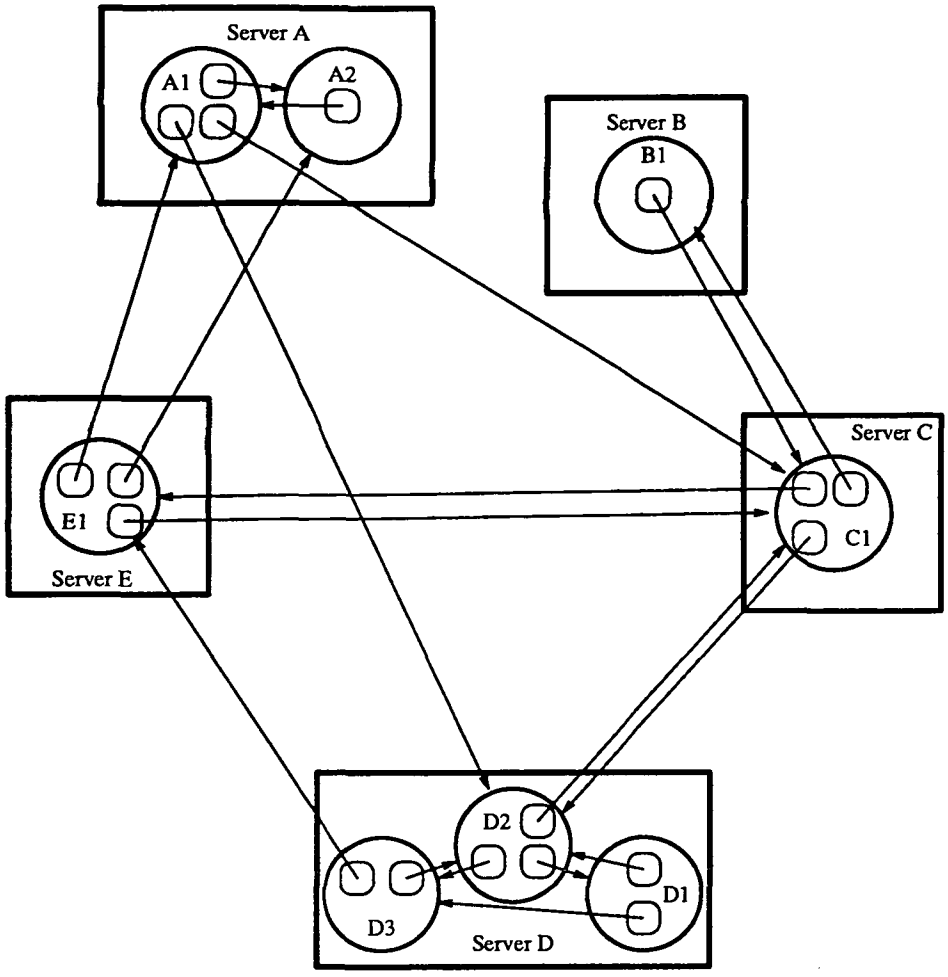

Figure 9: Univers Name Space.

access one another through an hierarchical name space rooted in context C1. The meta-context of each server—contexts A1, B1, C1, D2, and E1—contains an object that represents the C1 context so that every context in the system can contact C1 through the meta context of the server that implements it. The resolution functions that can be used to access information about remote clients take advantage of structure provided by these embedded name spaces.

Finally, contexts can be moved freely within the system with little or no loss in availability. For example, to move context A2 from server A to server B, the object that denotes A2 within server A's meta context changes from authoritative to cached and references the object's new location in server B. An authoritative object that represents A2 is then added to server B's meta context. A client attempting to locate context A2 through the reference in E1 finds that the context is no longer in server A. In order to find the new location, the client queries context A1 for the forward reference to A2 in server B. At this point, server E updates the cached object that represents A2 to reflect the change. In this way, the

name space evolves over time with minimal loss of functionality.

Searching The Name Space

Within conventional naming systems, the client provides as part of the name an object's location within the name space supported by the naming system. In contrast, clients of the Univers naming system are not required to specify information about an object's location within the name space. Rather, the naming system must infer that information using the information that the client provides in its description of the object. Once the naming system identifies a set of candidate contexts, each is searched for the client's object. Thus, locating an object is a two step process: determine a set of candidate contexts and then search each candidate for the objects that are identified by the client's description. This section describes several methods for identifying a set of candidate contexts.

The method used to locate objects within the Univers name space is determined by one or more client preferences within the preference hierarchy specified by the client as part of the query. In general, completion approximation function are used to construct search preferences. Each approximation adds to the name an attribute that refers to a set of contexts that might contain the object. The value of the context attribute—an attribute that describes the context containing an object—is a query consisting of a resolution function that is specifically designed to locate context objects, and a set of attributes that describe candidate contexts.

The context that is responsible for administering an object is not an intrinsic property of the resource the object represents; it is unlikely that the client will specify information about an object's context. However, information in the client's query can often be used to describe the entity responsible for administering the object and, as a result, the context it manages. A client approximation function maps properties of the object that the client identifies into properties of the context that administers the object. Approximation functions map attributes differently according to the type of object to be identified and the degree to which information in the name is used to identify a context.

Consider the problem of locating an object described by the following attributes:

```
( (name "Doc Holiday")
  (department "Department of Computer Science")
  (university "University of Arizona") )
```

To locate a set of candidate contexts, a client approximation function might construct a reference attribute as follows:

```
(ref= context
   (context-lookup ( (= organization "University of Arizona")
                     (= administrator "Department of Computer Science"))
   )
)
```

The value of this attribute is a Univers query that selects a set of contexts. In particular, the result of the query is a set of objects that represent contexts administered by the Department of Computer Science at the University of Arizona. Note that the approximation function simply maps the value of the attribute with trait department into an attribute with trait organization, and the value of the attribute with trait university

into an attribute with trait organization. Each selected context object contains information concerning access methods for the context represented by the object. Eventually, a database approximation function will call the builtin Ulisp primitive **get-context** to resolve these references.

The responsibility for actually locating the contexts falls on the resolution function in the query that the client approximation constructs. The preference hierarchy of one of these resolution functions generally consists of the universal client approximation and several database approximations. For example, database approximations within the preference hierarchy that defines context-lookup—a resolution function specifically designed to locate context objects—use information within the local context to identify remote contexts. This information can identify which contexts are local, which contain a large cache of context objects, or which participate in certain embedded name spaces. In other words, these database approximation functions encapsulate information about potentially effective methods for manipulating the current configuration of the name space.

The remainder of this section describes three methods for locating objects in remote contexts. The first method, breadth-first search, does not rely on help from the client but can be very inefficient. The second method attempts to extract information about the context from the client's description of the object to be identified. The last method directly queries the client for information that might help locate a set of candidate contexts. Note that these methods are intended to provide examples for locating objects in the name space. There are also many other ways to use the preference hierarchy model to locate objects in the Univers naming system.

Breadth-First Search:

Consider a client preference, denoted $\prec_{BFS}$, that implements a bounded breadth-first search of the name space.

$$forward \prec_{BFS} remote \prec_{BFS} local$$

The most preferred approximation function, *local*, adds a context attribute that contains just contexts contained within the local name server. The next most preferred approximation, *remote*, assumes that any context described in the local name server might contain the object that the client intends to identify. That is, *remote* constructs an attribute that describes contexts with cached descriptions in the local name server. Finally, the least preferred approximation, $forward_n$, adds an attribute that uses the resolution function bfs-lookup to locate the set of contexts that are at most n hops away from the local context in the name space graph.

The resolution function bfs-lookup is defined by the following preference hierarchy where $\prec_{search} \leq \prec_U$:

$\prec_U$ universal

$\prec_{search}$ bfs-search

The database approximation function bfs-search takes an attribute with trait distance—for example, (= distance n)—and returns all contexts that are in name servers exactly n hops away from the name server that is queried. For example, when $n = 0$, bfs-search queries the meta-context of the local name server to discover information about any contexts that the server contains. When $n = 1$ the function

Approximation	Assumes
local	the object is contained in the local server
remote	the object is contained in a server that is described locally
$forward_n$	the object is contained in a server within n hops of the local server

Table 11: Breadth-first search preference approximation functions.

queries the contexts in the local name server for the set of objects with type context; it returns objects for all of the contexts that are represented by objects cached in the local server. In order to locate objects many hops away, the approximation function may need to query the naming system many times, using the results from each query to formulate the next.

Consider the name space represented in Figure 9. If a client presents a query to server B then *local* would add an attribute (ref= context (bfs-lookup ((= distance 0)))) that selects the context B1 as a candidate. Function *remote* would add an attribute that identifies context C1 that is described by objects contained in server B. Finally, $forward_2$ would add an attribute to the name that specifies contexts E1 and D2. In this case, contexts A1, A2, D1, and D3 would never be considered candidates to be searched.

Using Hints:

Often, the type of resource being identified provides hints about methods that might successfully resolve a client's description. To take advantage of these hints we organize an embedded name space and we define a set of preferences that manipulate it. Each preference encapsulates information about the semantics of the objects being identified and the name space provides an organization that can be searched efficiently using those semantics. Throughout the remainder of this section we discuss methods for using hints in a white-pages user directory service; similar methods can be used to identify other resource types.

A white-pages service can make effective use of several attribute traits that can describe both the user being identified and the context that administers the object that represents the user. A user's telephone number gives geographical information—in the area code of the phone number—about the location of contexts that might contain the object. Similarly, a surface mailing address contains information about the city, state, and zip code of contexts to be searched. For example, suppose that the name space has three hierarchical embedded name spaces organized according to zip code, area code, and organizational domain (i.e. the top level of the hierarchy is split into several domains such as education, industry, and government). Of these three, the zip code most precisely specifies the location of a context, followed by area code and then the organizational domain. Therefore, it is reasonable to define a client preference $\prec_{hint}$ for using these three embedded name spaces as follows:

$$org\text{-}domain \prec_{hint} area\text{-}code \prec_{hint} zip\text{-}code$$

The *zip-code* function returns an attribute that specifies a query that consist of the zip code attribute in the client's specification and a resolution function that is designed to use the zip code name space. If

the client does not specify a zip code attribute then the approximation fails. In the same way, *area-code* and *org-domain* construct reference attributes with values that consist of a resolution function that uses an embedded name space and the attribute or attributes that describe the location of a set of contexts within that name space. In general, client approximation functions extract information from the client's specification that might help describe the some characteristic of the context and place it in a reference attribute with trait context.

The resolution function specified in the context attribute must search the name space for contexts that match the description. To improve efficiency, the resolution functions assume the name space includes a hierarchy of contexts that maintain information about other contexts. For example, Figure 10 shows an hierarchical embedded name space rooted at A1. Contexts A2 and A3 contain information about properties such as geographical location, area code, address, and administrative organization for other contexts in the naming system. Note that at least one context in each name server contains an object that represents A1. Using this object, a resolution function queries A1 for information about contexts. A1 then passes the query on to one of the contexts below it in the hierarchy. Eventually, a set of candidate contexts is returned to the original resolution function. The resolution function then queries each context with the remainder of the client's query; i.e. the resolution function prepares a query that contains the attributes that the client specified and the unevaluated portion of the preference hierarchy.

Interactive Browsing:

It is reasonable to believe that the client of a naming system cannot accurately specify information about an object's location but can distinguish among a set of possibilities. This assumption leads to another method for locating objects that involves direct interaction between the client and the approximation functions that comprise a preference in the resolution function. An interactive browsing search preference consists of a set of client approximation functions that interact with the client to determine a set of candidate contexts. For example, an approximation function uses information from the client's original object description and from earlier interaction with the client to provide the client a set of candidate contexts. The client then selects one or more contexts from the list or informs the approximation function that none of the contexts contain the object, in which case the approximation fails and another is tried.

The browsing technique closely resembles interactive search facilities provided by hypertext environments [Shne87]. In a hypertext system, a client views a piece of information and then dereferences pointers to related information. Similarly, the Univers naming system attempts to resolve a name within a certain context. If the resolution fails, then the system queries the client with a set of objects that describe additional contexts that can be searched. In a sense, the naming system acts to locate pointers to contexts and to dereference those pointers. The client specifies a set of pointers that should be dereferenced.

6 CONCLUSIONS

This thesis describes a new naming system, called Univers, that accommodates the practical constraints associated with a large internet environment. Unlike other naming systems that operate within an internet environment, Univers stresses the need to provide a powerful interface over the need to provide optimal performance. This chapter discusses the major contributions of this thesis and outlines directions for future research.

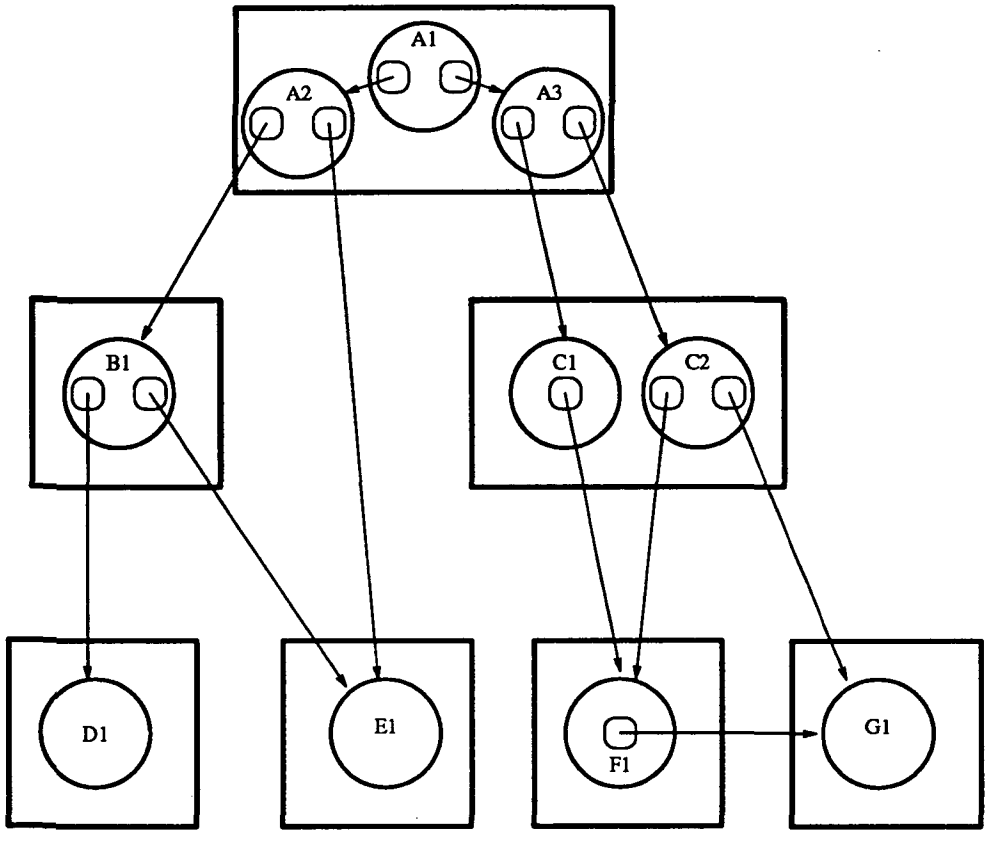

Figure 10: Embedded Hierarchical Name Space.

6.1 Contributions

The Univers naming system is a novel solution to the problem of satisfying the constraints placed on naming by an internet environment. The contributions of this thesis are the preference hierarchy model, the Univers programmable name server, and the set of conventions that describe the actual construction of the Univers naming system. This section details the contributions in each of these three areas.

Preference Hierarchy Model

The preference hierarchy model presented in Chapter 3 describes a new technique for handling imperfect information within naming systems. Rather than addressing the question of imperfect information from the perspective of a system designer, the preference hierarchy model allows clients to specify meta-information about the status of information within the naming system. That is, the preference hierarchy model is a client controlled method of accommodating imperfect information. The preference hierarchy also provides tools for formalizing the resolution functions associated with a naming system. This section describes the application of the preference hierarchy to comparing naming systems and to deriving new systems from a set of constraints.

Comparing Systems:

The preference hierarchy model provides a mechanism for reasoning about naming systems as specialized inference mechanisms. It defines a preference hierarchy that is used to specify the resolution function(s) associated with a given naming system, including both conventional and descriptive systems. Although it is not clear that the preference hierarchy is the only framework that could be used to describe the structure of naming systems, it has proven powerful enough to describe the naming systems we have encountered. The preference hierarchy provides a handle on comparing naming systems. For example, one can directly compare the most discriminating function in the family of white-pages resolution functions with equal1 as follows:

$$lookup \sqsubseteq_{\Delta} equal1$$

In other words, it is accurate to view equal1 as a restrictive member of the white-pages family of resolution functions.

While it is the case that we cannot compare naming systems unless the preference hierarchy of one can be embedded in the other, being able to conclude that the naming systems are inherently different is itself useful information. For example, one might be interested in knowing if white-pages and yellow-pages services are fundamentally different, or if they are simply synonyms for the same thing. The answer, at least relative to the systems with which we have experience, is that they are fundamentally different: white-pages services are based on *open* and *closed* approximations, while yellow-pages services are based on *mandatory* and *optional* approximations.

Furthermore, our experience strongly suggests that the approximations correctly represent the environment in which the two systems operate and the requirements placed on the two systems by the entities that use them. In the case of white-pages services, the fact that not all useful information about users is known to the naming system is a significant constraint on the system; the resolution functions must be designed to accommodate missing information. Also, because users of a white-pages system generally have a particular object in mind when they submit a name, it is implied that the object identified by the attributes should possess all of the attributes in the name; i.e., the distinction between mandatory and optional attributes is not relevant. In the case of yellow-pages services, it is possible to mandate that all necessary information be stored in the database; i.e., the distinction between open and closed attributes is not an issue. Moreover, the entity that names a computational resource is most interested in distinguishing between similar resources—users doesn't care which processor they get as long as it has the appropriate blend of attributes. Thus, the fine-grain control afforded by partitioning the name into mandatory and optional attributes is useful.

Deriving New Systems:

The preference hierarchy model also provides a prescriptive tool for use in designing naming systems. Consider the following three examples.

First, the white-pages resolution functions introduced in Chapter 5 were designed as part of the Profile naming system [Pete88] before the preference hierarchy model had been formally defined, but clearly, an intuitive understanding of preferences guided the design. It is interesting to note that the formal specification of the preference hierarchy helped understand and correct a subtle flaw in the original

definition of wp_2 and to define a new function, wp_3, that avoided one of the shortcomings that existed in the original Profile resolution functions.

Second, after having defined the preference hierarchy, we were able to apply it to the problem of designing a yellow-pages service. In particular, we wanted a descriptive yellow-pages service that would allow users to discriminate among a set of similar resources; i.e., resources that provided approximately the same service. The result was the yp resolution function based on mandatory attributes that identify the basic service, and optional attributes which identify preferred characteristics of the resources that provide the service.

Third, we have observed that the domain naming system (DNS) [Mock87] evolved in a way that suggests an implicit understanding of the preference hierarchy. Specifically, DNS provides the same functionality as conventional systems: it maps host names into network addresses. Unlike simpler systems, however, the domain system is implemented in a network environment in which the database is distributed over multiple hosts. Several years of experience with the system led its designers to understand that the DNS mechanism must be able to deal correctly with out-of-date data. In particular, an updated specification of the system reads: "Cached data should never be used in preference to authoritative data...". This informal definition directly corresponds to the temporal preference *cached* $\prec_t$ *authoritative*. The important point is that even in functionally simple systems that map names into addresses, it is possible for the environment in which the system is implemented to impose constraints on the system, and the technique used to deal with these constraints can, in turn, be expressed in terms of the preference hierarchy.

Univers Name Server

Univers is a programmable name server that serves as the foundation for the Univers naming system. A key aspect of Univers' design is to shift responsibility for determining the conventions used for context administration from a central authority to the organizations that participate in the naming system. Except for a limited number of well-known objects, a context administrator loads Univers server with the type, function, and context objects that implement a particular service.

Beyond this basic design, Univers exploits four novel ideas. First, it associates a set of attributes with each object and supports lookups based on any combination of attributes. Thus, rather than mapping names into addresses, Univers accepts queries containing whatever information clients know about an object (not just its name) and responds with whatever information it possesses for each of identified objects (not just its address). Second, Univers accepts arbitrary naming programs as queries. In contrast, most conventional name servers support only a small collection of lookup operations. Third, Univers allows users to define new object types. Thus, users are able to isolate interesting classes of objects upon which they want to operate. Fourth, Univers provides explicit support for keeping the attributes contained in the database accurate. Keeping name servers up-to-date is one of the most difficult problems faced by naming systems.

High-Level and Low-Level Names:

Although it is often the case that name spaces can be classified into levels, Univers does not enforce, or depend upon, a level structure. For example, an organization's Univers name server might participate in a high-level directory service that translates attribute-based names for users into domain names. The

same name server might also be called upon to translate domain names into Internet addresses as part of the domain name hierarchy. As another example, a Univers name server could be asked to translate the attribute (= name "John Doe") into the tty most recently accessed by John Doe. This could be done with the query (project (wp2 "name=John Doe") tty). At the same time, Univers could be asked to report information about the person currently logged onto tty12 with the query (dereference (get_tty name=12) user). In general, Univers can be used to map one attribute for an object into another attribute for the same object. In effect, different levels of names for an object simply correspond to different attributes for the object; each client chooses the attribute(s) it prefers to use.

Programmable Interface:

Being able to submit naming programs rather than being restricted to a pre-defined set of name server operations is a significant feature of Univers. It allows a single server to encapsulate the functionality of widely different naming services. Moreover, representing functions as objects in the database makes the server easy to extend. In fact, one could store useful functions in a local instance of Univers, and later retrieve them and submit them to a remote Univers server that does not support them. It is interesting to note that there seems to be a trend toward building servers that are programmable in the sense that they support an interpreter rather than a fixed set of operations. Chapter 2 identifies several examples.

Types:

Representing types as objects in the database has two desirable consequences. First, the system is easily extendible. Any client can define its own set of type objects and then use them to query the database. Moreover, it is possible for clients to define independently a collection of object types without interfering with other types that might also be defined. For example, it is possible to define two different user types: one that is used relative to a global white-pages service and another that is used relative to a local white-pages service. This is because the type system is imposed on top of the set of objects; an object's type is not represented as part of the object. Second, users can query Univers to discover the set of meaningful types and the traits that those types depend on. This helps users to construct queries that identify other objects.

Note that a type can either be explicitly given by name—such a type is stored as an object in the database—or it can be implicitly specified in a client's description. That is, while in the abstract there are 2^N types associated with a database if there are N distinct traits, only a small subset of them are of any practical use. It is this small collection of types that are actually *named* and stored as objects in the database. On the other hand, when a particular type is not stored in the database, it may still be used in a query. This is because every query involves the specification of a set of attributes; the attribute traits used by the query specify a type, which in turn limits the query to an interesting set of objects.

Registering Attributes:

Conventional name servers provide two primary functions: *bind* and *resolve*. The bind function associates a name with a value. The resolve function takes a name as an argument and returns the corresponding value. The implication is that a name must be bound before it can be resolved. For attributes that don't change very often, this is not a problem; once the name is bound, any number of resolutions can take place. However, for objects with attributes that change frequently, such a scheme has two problems. First, in order to maintain accurate bindings, there may be many more bindings than resolutions. That is, many values might be bound to a name for each time the value is read. Second,

there does not necessarily exist an active entity that faithfully establishes (and reestablishes) the binding. Without a mechanism to rebind values as they change, stale information may be used to resolve names.

Univers avoids these two problems by registering dynamic values (generators) in the database rather than static values. Generators are evaluated in a lazy fashion so as to avoid unnecessary updates of the value. Also, generators query the object itself for the value. In other words, rather than maintaining a database of static information, it is more accurate to think of Univers as providing a clearinghouse for a collection of procedures that monitor the state of various resources.

Univers Naming System

In a conventional naming system, an object is identified by its location within the name space that is defined by the naming system. While this method provides an efficient way to locate objects, it limits the system's functionality in two areas: a client must supply information about an object's location and the name space must be static and strictly hierarchical. The design of the name space helps Univers overcome these limitations.

Using Semantics:

Conventional naming systems rely on the client to locate an object within a system's name space. In contrast, Univers does not require clients to specify information about an object's location in the name space. Rather, Univers maps intrinsic properties of an object into properties of the context that contains the object using an understanding of the semantics of the object type to be identified. In the case where the client specifies no information that can be used to identify the context, the system exhaustively searches the name space for the object. Although very inefficient, this method can resolve names that conventional naming systems cannot. More often, however, the client specifies information about an object that provides hints about the context that contains the object. For example, the employer property of a user object may describe the administrator of the context that contains the user object. In general, Univers translates queries about an object into two sub-queries: one that locates a context and one that locates the object within the context. When a client specifies information about an object's location in a name space—such as its domain name—then the query can be resolved in the same manner as conventional systems taking advantage of an embedded hierarchical name space. The difference between conventional systems and the Univers system is that the Univers client is not required to possess information about the name space.

Embedded Name Space:

Conventional naming systems that force a client to identify an object by its location in the name space must address issues such as location independence and time independence; i.e. an object is identified independent of when and where the client presents its query. At one end of the spectrum of solutions, source-route naming systems provide neither location independence nor time independence; an object may be named differently at different times or from different places. At the other end of the spectrum, hierarchical name spaces provide both location and time independence by naming objects relative to a strictly hierarchical name space that rarely changes. Both solutions limit the functionality of the system. The Univers naming system, however, avoids these limitations while still providing many of the benefits.

The Univers name space achieves location and time independence by identifying objects by their

intrinsic characteristics. An object is identified differently only when the object itself changes. By solving these problems separately from the construction of the name space, Univers does not need to maintain a static, strict hierarchy within the name space. Rather than forcing the system to use connections in a hierarchical fashion, it can take advantage of all the connections between contexts. Thus Univers has the advantage that it can use "short-cuts" through the name space when they exist. A hierarchical name space can be embedded within the Univers name space to provide the same potential for efficient searches as conventional systems. The difference is that Univers does not mandate strict adherence to the hierarchy. Furthermore, the structure of the name space can change without affecting the way that a client identifies an object. In this way, contexts can change locations and the name space can reorganize itself without affecting the way that a client identifies objects in the system.

6.2 Future Work

The research presented in this thesis can be continued in several areas, including modifying the preference hierarchy model to include partial information and reimplementing the Univers server to improve efficiency and usability. This section describes each area of future work in greater detail.

Partial Information

The preference hierarchy model described in Chapter 3 assumes that the value of a property is either completely known—though possibly inaccurate—or completely unknown. It does not take into account information that may be partially known. For example, there is no method for representing the information that an object is contained in one of several possible contexts or that a processor's speed is in a range of values. To facilitate partial information, several changes must be made. First, the notion of an attribute needs to be generalized to account for a disjunction of values. Second, preferences and preference hierarchies need to be redefined as partial orders instead of total orders. A partial order provides for several results to be computed simultaneously as is necessary when an attribute can take several different values. Finally, the tools for proving soundness and completeness need to be changed to accommodate the new definitions.

Implementation Improvements

Based on our experiences building and using Univers, we could modify the implementation in the following three ways. First, the values of attributes need to be typed. Typing attribute values allows us to optimize the most common comparison operations. For example, the domain name attribute has unique string values for each object and the only comparison used is equality. These values can be implemented as a hash table optimized for rapid lookup. Other types include integers, reals, and general comparison strings. We expect this modification to allow Univers to perform competitively with existing special purpose name servers in most cases.

Second, facilities need to be added to allow the replication of Univers servers. This allows an organization to continue to participate in the naming system even if its Univers server fails. Additionally, a name server that contains the root context in an embedded space can be replicated. This alleviates

some of the bottleneck of accessing a single server for information within the embedded name space. Replicating information means that several issues associated with distributed databases such as copy and transaction transparency must be addressed.

Finally, the binary predicates used by naming attributes need to be replaced by more general purpose aggregate operations. For example, operations such as < can be defined on a value and a relation of object/value pairs. Aggregate operations have two benefits: they are more powerful (operations such as max and min can be defined) and they can be implemented more efficiently. To facilitate aggregate operations, the preference hierarchy model needs to be generalized to account for the increasing distinction between database attributes and naming attributes.

Increased Support For Object Organizations

The reference value of an attribute provides a powerful facility for organizing objects within the naming system. Currently, Univers takes advantage of this ability in two ways: types and contexts. In the future, the Univers naming system could support several different name spaces simultaneously by adding objects that organize collections of name space objects. For example, a *domain* object might reference a collection of domain and context objects. Thus, each client could organize a name space that meets its particular needs.

A Univers Code Examples

This appendix contains several Ulisp function definitions used to implement the name resolution functions described in Chapter 5. This is how we actually define resoulution functions in the Univers name server. These functions may be included as part of the query or they may be stored in the Univers server and referenced by the query. Note that several optimizations have been made in order to avoid redundant computations of composite approximation functions.

A.1 Database Functions

```
(define (make-object attribute-set context-name)
  (let ((uid (create-object)))
    (for-each (lambda (attribute) (add-attribute uid attribute)) attribute-set)
    (add-attribute context-name '(objects ,uid))))


(define (select object-set naming-attribute)
  (let
    ( [result-set '()]
      [op (car naming-attribute)]
      [trait (cadr naming-attribute)]
      [value (caddr naming-attribute)])
   (do ([objects object-set (cdr object-set)])
```

```
      ((null? objects) result-set)
       (let ([object (car objects)])
        (if (op (get-value object trait) value)
           (set! result-set (cons object result-set)))))))
```

A.2 Resolution Functions

White-Pages Functions

```
(define (all N D)
   (also-ran (universal N) D)
)


(define (lookup N D)
   (unique (closed N) D)
)


(define (wp1 N D)
   (cond
      [(unanimous (closed N) D)]
      [(also-ran (closed N) D)]
      [(unanimous (open N) D)]
      [(also-ran (open N) D)]
   )
)


(define (wp2 N D)
   (cond
      [(unique (closed N) D)]
      [(unanimous (open N) (unamimous (closed N) D))]
      [(also-ran (open N) (unamimous (closed N) D))]
      [(unanimous (closed N) D)]
      [(unanimous (open N) (also-ran (closed N) D))]
      [(also-ran (open N) (also-ran (closed N) D))]
      [(also-ran (closed N) D)]
      [(unanimous (open N) D)]
      [(also-ran (open N) D)]
   )
)
```

```
(define (wp3 N D)
  (cond
    [(also-ran (open N) (unanimous (closed N) D))]
    [(unanimous (closed N) D)]
    [(also-ran (open N) (also-ran (closed N) D))]
    [(also-ran (closed N) D)]
    [(also-ran (open N) D)]
    [D]
  )
)
```

Yellow-Pages Functions

```
(define (yp M O D)
  (if (null? O)
    (exact M D)
    (cond
      [(exact O (exact M D))]
      [(yp M (cdr O) D)]
    )
  )
)
```

Conventional Functions

```
(define (equal1 N D)
  (unique (universal N) D)
)
```

```
(define (equal2 N D)
  (exact (universal N) D)
)
```

References

[Adob85] Adobe Systems Incorporated, . *Postscript Language Reference Manual.* Addison-Wesley Publishing, Reading, Mass., December 1985.

[Banc87] Bancilhon, F. and Ramakrishnan, R. An amateur's introduction to recursive query-processing strategies. In *1987 Proceedings of SIGMOD International Conference on Management of Data*, pages 16–52, 1987.

[Betz89] Betz, D. M. *Xscheme: An Object-oriented Scheme*. Peterborough, NH, 1989.

[Birr82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25 Nr 4:260–274, April 1982.

[Card85] Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Card86] Cardelli, L. A polymorphic λ-calculus with type:type. Technical Report 10, DEC Systems Research Center, May 1986.

[Card88] Cardelli, L. Structural subtyping and the notion of power type. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.

[Care88a] Carey, M. J. and DeWitt, D. J. The architecture of the EXODUS extensible DBMS. In Stonebraker, M., editor, *Readings in Database Systems*, chapter 7.3, pages 488–501. Morgan Kaufmann Publishers, Inc., 1988.

[Care88b] Carey, M. J., DeWitt, D. J., and Vandenberg, S. L. A data model and query language for EXODUS. In *1988 Proceedings of SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, June 1988.

[Cham81] Chamberlin, D. D. and et. al., . Support for repetitive transactions and ad-hoc queries in system r. *ACM Transactions on Database Systems*, 6(1):70–94, March 1981.

[Codd79] Codd, E. F. Extending the database relational model to capture more meaning. In *1979 International Conference on Management of Data*, Boston, Mass, May 1979.

[Codd86] Codd, E. F. Missing information (applicable and inapplicable) in relational databases. *SIGMOD Record*, 16(4), 1986.

[Codd87] Codd, E. F. More commentary on missing information (applicable and inapplicable) in relational databases. *SIGMOD Record*, 16(1), March 1987.

[Come89] Comer, D. E. and Peterson, L. L. Understanding naming in distributed systems. *Distributed Computing*, 3(2):51–60, May 1989.

[Cope84] Copeland, G. and Maier, D. Making Smalltalk a database system. In *1984 Proceedings of SIGMOD International Conference on Management of Data*, Boston, MA, May 1984.

[Daya85] Dayal, U. and Smith, J. Probe: A knowledge-oriented database management system. In *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Demo88] Demolombe, R. and Farinas Del Cerro, L. An algebraic evaluation method for deduction in incomplete data bases. *The Journal of Logic Programming*, 5:183–205, 1988.

[Deut88] Deutsch, D. An introduction to the X.500 series network directory service. Technical report, BBN Laboratories Incorporated, June 1988.

[Fabr74] Fabry, R. Capability-based addressing. *Communications of the ACM*, 17(7):403–411, 1974.

[Falc86] Falcone, J. and Emer, J. A programmable interface language for heterogeneous distributed systems. Tech Report DEC-TR-371, Digital Equipment Corporation, Hudson, Mass, August 1986.

[Fowl85] Fowler, R. J. *Decentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington, December 1985.

[Gall84] Gallaire, H., Minker, J., and Nicolas, J. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2):153–186, June 1984.

[Gene87] Genesereth, M. R. and Nilsson, N. J. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., 1987.

[Hall76] Hall, P. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, may 1976.

[Int88] International Organization for Standardization. *Information processing systems: open systems interconnection — the directory — overview of concepts, models, and service*, 1988. Draft International Standard ISO 9594-1:1988(E).

[Kier89] Kiernan, J., de Maindreville, C., and Simon, E. The design and implementation of an extendible deductive database system. *SIGMOD Record*, 18(3):68–77, September 1989.

[Koch89] Kocharekar, R. Nulls in relational databases: Revisited. *SIGMOD Record*, 18(1):68–73, March 1989.

[Lamp86] Lampson, B. Designing a global name service. In *Proceedings of Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, August 1986.

[Lars88] Larson, P. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[Laur88] Laurent, D. and Spyratos, N. Partition semantics for incomplete information in relational databases. In *1988 Proceedings of SIGMOD International Conference on Management of Data*, pages 66–73, Chicago, IL, June 1988.

[Lind85] Lindsay, B. A retrospection on R*: A distributed database management system. Technical Report RJ4859, IBM Research Center, San Jose, CA, September 1985.

[Lips79] Lipski, W., Jr. On semantic issues connected with incomplete information databases. *ACM Transactions on Database Systems*, 4(3), September 1979.

[Lips81] Lipski, W., Jr. On databases with incomplete information. *Journal of the ACM*, 28:41–70, 1981.

[Mann87] Mann, T. *Decentralized naming in distributed computer systems.* PhD thesis, Stanford University, Palo Alto, California, May 1987.

[Meye85] Meyer, A. R. and Reinhold, M. B. 'Type' is not a type: Preliminary report. In *Proceedings POPL 1985*, 1985.

[Mock84] Mockapetris, P. The domain name system. In *Proceedings of the IFIP 6.5 Working Conference*, Nottingham, England, May 1984.

[Mock87] Mockapetris, P. Domain names—implementation and specification. Request For Comments 1035, USC Information Sciences Institute, Marina del Ray, Calif., November 1987.

[Morr86] Morris, K., Ullman, J. D., and Gelder, A. V. Design overview of the NAIL! system. Technical Report STAN-CS-86-1108, Stanford University, Stanford, CA, May 1986.

[Naha89] Nahaboo, C. and Joloboff, V. *The Generic Window Manager.* Massachusetts Institute of Technology, 1989.

[Neuf89] Neufeld, G. A descriptive name service. In *Proceedings of the SIGCOMM '89 Symposium*, September 1989.

[Oppe83] 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.

[Pech80] Pecherer, R. M. Efficient evaluation of expressions in a relational algebra. In *Proceedings ACM Pacific Conference*, pages 44–49, 1980.

[Pete87] Peterson, L. L. A yellow-pages service for a local-area network. In *Proceedings of the SIGCOMM '87 Workshop: Frontiers in Computer Communications Technology*, pages 235–242, Stowe, VT, August 1987.

[Pete88] Peterson, L. L. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, November 1988.

[Pick79] Pickens, J., Feinler, E., and Mathis, J. The NIC name server—a datagram based information utility. In *Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979.

[Post80] Postel, J. User datagram protocol. Request For Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., August 1980.

[Pric71] Price, C. Table lookup techniques. *Computing Surveys*, 3(2):49–65, 1971.

[Reit78] Reiter, R. On closed world databases. In Gallaire, H. and Minker, J., editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, NY, 1978.

[Rich87] Richardson, J. E. and Carey, M. J. Programming constructs for database systems implementation in EXODUS. In dayal, U. and Traiger, I., editors, *Proceedings of ACM SIGMOD*, pages 208–219, San Francisco, CA, May 1987.

[Ritc74] Ritchie, D. and Thompson, K. The Unix time-sharing system. *Communications of the ACM*, 17(7):365–375, july 1974.

[Schr84] Schroeder, M. D., Birrel, A. D., and Needham, R. M. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, 2(1):3–23, February 1984.

[Schw87] Schwartz, M. F., Zahorjan, J., and Notkin, D. A name service for evolving, heterogeneous systems. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, November 1987.

[Schw88] Schwartz, M. F. The networked resource discovery project: Goals, design, and research efforts. Technical Report CU-CS-387-88, University of Colorado, May 1988.

[Ship81] Shipman, D. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1), March 1981.

[Shne87] Shneiderman, B. User interface design and evaluation for an electronic encyclopedia. Technical report, Department of Computer Science, University of Maryland, College Park, MD, March 1987.

[Solo82] Solomon, M., Landweber, L., and Neuhengen, D. The CSNET name server. *Computer Networks*, 6:161–172, 1982.

[Spyr87] Spyratos, N. The partition model: a deductive database model. *ACM Transactions on Database Systems*, March 1987.

[Stal87] Stallman, R. *GNU Emacs Manual*, sixth edition, March 1987.

[Stam90] Stamos, J. W. and Gifford, D. K. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[Stee78] Steele Jr., G. L. and Sussman, G. J. The revised report on Scheme, a dialect of Lisp. Technical Report Memo 452, MIT Artificial Intelligence, January 1978.

[Ston86] Stonebraker, M. and Rowe, L. A. The design of POSTGRES. In *1986 Proceedings of SIGMOD International Conference on Management of Data*, pages 340–355, Washington, DC, May 1986.

[Ston87] Stonebraker, M., Anton, J., and Hanson, E. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, September 1987.

[Ston88a] Stonebraker, M. *Readings in Database Systems*. Morgan Kaufmann Publishers, Inc., 1988.

[Ston88b] Stonebraker, M., Hanson, E., and Hong, C.-H. The design of the POSTGRES rules system. In Stonebraker, M., editor, *Readings in Database Systems*, chapter 8.3, pages 556–565. Morgan Kaufmann Publishers, Inc, 1988.

[Sun86] Sun Microsystems, Inc., Mountain view, Calif. *Remote Procedure Call Programming Guide*, February 1986.

[Suss75] Sussman, G. J. and Steele Jr., G. L. Scheme: an interpreter for extended Lambda calculus. Technical Report Memo 349, MIT Artificial Intelligence, December 1975.

[Terr87] Terry, D. Caching hints in distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):48–54, January 1987.

[Ullm88] Ullman, J. D. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ullm89] Ullman, J. D. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[USC81] USC, . Transmission control protocol. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., September 1981.

[Will81] Williams, R. and et al, . R*: An overview of the architecture. Technical Report RJ3325, IBM Research Center, San Jose, CA, December 1981.

[Wong82] Wong, E. A statistical approach to incomplete information in database systems. *ACM Transactions on Database Systems*, 7(3):470–488, September 1982.