Reasoning About Naming Systems

Mic Bowman
Penn State University

Saumya K. Debray and Larry L. Peterson[1]
University of Arizona

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*query languages* H.2.4 [**Database Management**]: Systems—*query processing* H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*selection processes* H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*question-answering (fact retrieval) systems.*

General Terms: Design, Theory

Additional Key Words and Phrases: Descriptive Naming Systems, Inference Mechanisms

---

1 Introduction

A name is a syntactic entity that denotes an object and a naming system is a mechanism that answers queries of the form "what object is denoted by this name?". A naming system resembles a restricted database system that infers the object(s) referenced by a name. The class of queries that can be handled by such a system and the precision of the answers it returns depends directly on the power of its inference mechanism.

Conventional naming systems, such as those found in compilers [Pric71], virtual memory managers [Fabr74], and file systems [Ritc74], are functionally simple: the database contains a table of name/value pairs, clients submit a single name, and the system returns the corresponding value. More elaborate naming systems, such as *white-pages* services used to identify users [ISO88, Pete88, Pick79, Solo82], *yellow-pages* services used to identify network resources [Pete87, Sun86], and document preparation utilities for managing bibliographic data [Budd86], allow clients to identify an object by giving a collection of attributes that describe the object; we refer to such systems as *descriptive* or *attribute-based* naming systems. For example, a client may learn a user's mailbox address by querying a white-pages service with a name of the form

"name=John Doe, phone=621-1234, org=State University"

Similarly, a client may learn the address of a cycle server by querying a yellow-page service with a name of the form

"service=cycle, arch=vax, load<2, mips=max"

In both cases, the answer returned by the naming system depends on the information available in its database and the power of its inference mechanism.

This paper defines a simple model, called a *preference hierarchy*, that provides a framework for using the information available to a naming system to compute the object(s) identified by a given name. The preference hierarchy therefore serves as a formal tool for designing and reasoning about naming systems.

[1]Authors' address: Department of Computer Science, University of Arizona, Tucson, AZ 85721. This work supported in part by National Science Foundation Grant DCR-8609396 and National Aeronautics and Space Administration Grant NCC-2-561.

In particular, it allows us to evaluate naming systems based on how *discriminating* they are and the set of names for which they are *sound* and *complete*.

The rest of this section motivates the preference hierarchy and discusses related work. Section 2 then presents some preliminary definitions and Section 3 describes how preference hierarchies can be used to specify the structure of a naming system's inference mechanism. Finally, Section 4 demonstrates the utility of preference hierarchies and Section 5 makes some concluding remarks.

1.1 Motivation

To appreciate the impact and utility of the preference hierarchy, it is important to understand that the naming system designer must accommodate two sets of constraints: the requirements placed on the naming system by the entities that use the system, and the conditions under which the naming system must operate.

First, the clients of a naming system place a set of requirements on the system: The naming system must accept as a name whatever information its clients possess for the objects they want to identify and it must respond with the precision expected by those clients. For example, the client of a symbol table manager—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. In contrast, the clients of a white-pages service—in this case human users—often possess several pieces of information about the object they want to name and they are sometimes willing to tolerate ambiguous results as long as it contains the desired object.

Second, the naming system is constrained by the conditions under which it is implemented; i.e., the properties of the environment in which it must function. 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, many naming systems must function in an imperfect world. For example, a white-pages service can only resolve names that contain information put into the system by users. If a user does not enter a particular fact in the naming system, then the system cannot respond to queries using that information. As another example, naming systems that are implemented in distributed environments may lose information because independent components fail, or information may become out-of-date because of communication delays between the components that make up the system.

The preference hierarchy defined in this paper can be used to guide the design of naming systems that must satisfy various requirements and constraints. While it is the case that the preference hierarchy is most useful when applied to more complex descriptive naming systems, we have found the that preference hierarchy provides insights into the design of more conventional naming systems as well. In fact, Section 5 points out a naming system originally conceived as a conventional system, but which has evolved over time in a way that suggests the unconscious use of the preference hierarchy.

1.2 Related Work

Previous studies of naming focus on the operational aspects of naming systems, describing their architecture and the implementation of the architecture's base elements. For example, Fowler [Fowl85], Lampson [Lamp86], Mann [Mann87], Oppen and Dalal [Oppe81], and Terry [Terr87] each describe techniques for managing a decentralized naming service. In addition, studies by Comer and Peterson [Come89] and Watson [Wats81] give general characterizations of the resolution process. In contrast, this paper is concerned with the functional aspects of naming. In other words, we consider the question of <u>what</u> objects are identified by a given name rather than the question of <u>how</u> the system is structured.

In addition to work that explicitly addresses the problem of naming, many of the underlying ideas found in this paper can be traced to two other related areas. First, naming systems can be thought of as specialized database systems, where the process of resolving a name is conceptually similar to that of solving a database query [Gall84]. In particular, the preference hierarchy unifies ideas from work being done in open-world and closed-world databases [Reit78], partial match retrieval [Ullm88], and incomplete information [Lips79, Lips81]. Second, one can view naming as a specific constraint-satisfaction problem [Born87]. More specific comparisons to these related topics are made throughout the rest of the paper.

2 Basic Concepts

In the abstract, naming systems answer 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 the properties any element of this set might possess include "fonts", "location", and "resolution". The specific property "location is Room 723" describes a particular printer.

In practice, a naming system maintains and manipulates a database of syntactic information about a set of resources: It denotes each resource with a database *object*—an object is essentially a record—and each property with an *attribute*—a tagged field in the record. 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 type laserwriter and has 300 dots per square inch resolution", and so on. The corresponding objects in the naming system database would be

$$\langle\ \texttt{font:italic, location:723, uid:ip2}\ \rangle$$
$$\langle\ \texttt{type:laserwriter, resolution:300, uid:lw6}\ \rangle$$

where `font:italic` is an example of an attribute. Each attribute consists of a *tag* and a *value*; denoted **t:v**. For example, the attribute `font:italic` consists of the tag `font` and the value `italic`. For convenience, we assume each object contains a `uid` attribute and we refer to the object by this attribute's value; e.g., object `ip2`. Also, if an attribute $\mathbf{a}$ is entered in the naming system database for object $\mathbf{x}$, then $\mathbf{a}$ is said to be *registered* for $\mathbf{x}$.

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

Clients query a naming system with a set of attributes and the naming system responds with the set of objects that correspond, in some sense, to those attributes. Formally, a set of attributes $\mathbf{N}$ *names* object $\mathbf{x}$ if every attribute in $\mathbf{N}$ is registered for $\mathbf{x}$. That is, *names* maps sets of attributes into sets of objects, where $names(\mathbf{N})$ denotes the set of objects named by a set of attributes $\mathbf{N}$. If $\mathbf{x} \in names(\mathbf{N})$, then $\mathbf{N}$ is said to be a *name* for $\mathbf{x}$. The corresponding semantic notion is *represents*: a set of properties, $\mu(\mathbf{N})$, 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 1. The top layer corresponds to database objects—i.e., operations on fields and records—while the bottom layer corresponds to resources—i.e., operations real world entities such as a printer or a person. The dotted edges link single items with sets of items. Note that 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.

Conventional naming systems that operate in a "perfect" world—e.g., symbol tables, virtual memory systems—have only to implement a procedure that supports the *names* function. That is, they simply return the set of objects for which all the given attributes are registered. In general, however, naming systems do not operate in a perfect world. The information that a client uses to query a naming system may contain imperfections. For example, the client may not possess enough information to uniquely identify the desired objects or the information that the client does possess may be inaccurate. Likewise, the information contained in the naming system database may not be perfect—it may contain incomplete or out-of-date information. Thus, a naming system must use information specified by the client and registered in the database to approximate the results that would be obtained by *names* if it operated in a perfect world. The preference hierarchy described in the next section suggests a method for deriving various procedures, called *resolution* functions, that approximate the *names* mapping in an imperfect world.

3 Preference Hierarchy

Many naming systems distinguish between the "quality" of different kinds of information. For example, a descriptive naming system must be able to resolve names that contain a multiplicity of attributes. In doing so, it is reasonable to give some attributes more importance than others. Suppose a user is trying identify John Smith at State University, where the user is quite sure the individual is at State University, but uncertain as to whether his first name is John. Such a user would prefer that the naming system respond with information about any Smith at State University rather than information about some John Smith not at State University. As another example, because naming systems implemented in a distributed environment such as the Domain Name System [Mock87] must accommodate out-of-date information, they generally prefer attributes that are guaranteed to be current over cached attributes that may have become stale.

In a perfect world, where the client accurately and completely identifies a set of objects and there is no possibility of missing or out-of-date information in the database, it suffices to always use *names* to resolve a set of attributes. Indeed, this is essentially what is done when responding to queries in conventional database systems. In an imperfect world, however, the naming system must cope with two potential problems: inaccuracies in the attributes specified by the client, and imperfect information in the database. Furthermore, rather than provide a single resolution function that compensates for these problems, a naming system might support a number of different resolution functions, each tailored for a certain class of attribute sets the designer considers interesting and the databases the designer expects to manipulate.

Intuitively, each resolution function in a naming system uses certain assumptions about the quality of information in the system and resolves names according to the most preferred set of assumptions. 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 (these are called *client approximation* functions) or in the information contained in the database (these are called *database approximation* functions). 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 that information fails to distinguish among a set of objects, then another approximation is tried.

This section defines the notion of a preference hierarchy and describes the role it plays in resolving names. In particular, it introduces the intuition behind preference hierarchies and gives several example preferences, it establishes a framework for designing naming systems based on combinations of preferences, and it shows how that framework can be used to reason about naming systems. To better understand preferences and to appreciate how preferences are used in real attribute-based naming systems we provide several examples of client and database preferences defined in the Univers and Profile [Pete88] attribute-based name servers.

3.1 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 number attributes 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. Formally, a client approximation function is defined by:

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

A client preference is a finite, totally ordered 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 a client that intends to identify human users.

The following informally 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 Section 4 when we consider specific resolution functions in more detail.

Universal Preference: The universal preference consists of a single function *universal* that maps every set of attributes to itself. This function is used most often when the client believes that all of the attributes it specified are accurate and are sufficient to distinguish the set of objects that are identified. In other words, this function assumes that the client operates in a perfect world.

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 tag $\mathbf{t}$ is *closed* if every attribute constructed from $\mathbf{t}$ is guaranteed to be registered for every object it describes. For example, a naming system may guarantee that every object with a name will have that name registered in the database. The client approximation function *closed*—which returns the closed attributes in a name—limits the scope of resolution to portions of the database with complete information.[2] An open attribute may not be registered for an object even though the corresponding property describes the

[2]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 tag. Other approximations, such as the *explicit* approximation use the entire attribute.

resource the object denotes. The resolution of a name based on open attributes may not contain all of the objects in the database that represent resources described by the attributes. This preference is particularly useful when users are responsible for maintaining some of the information in the naming system. For example, one user 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 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.

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.

Precision Preference: The precision preference, denoted $\prec_P$, prefers attributes that match a very small set of objects. 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. An attribute is ambiguous if it is 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 on the Internet. The *ambiguous* approximation function returns those attributes in a name that are ambiguous. Thus,

$$ambiguous \prec_P unambiguous$$

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.

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. Within computer systems, clients 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:

$$optional \prec_Y mandatory$$

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 citation, 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 and for the second citation the preferences are reversed.

3.2 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* $\mathbf{A}$ *and a domain of objects* $\mathbf{O}$ *is a* **database approximation function** *if* m *is a contraction, i.e. for any set of objects* $\mathbf{D}$, $m(\mathbf{N}, \mathbf{D}) \subseteq \mathbf{D}$.

Intuitively, the definition says that a database approximation function never adds new objects to the database. Most useful database approximation functions have two other characteristics. First, most database approximation functions return more complete answers with more information about the world. In other words, as the size of the database increases, the number of objects matching a name increases. More formally, a database approximation function m is monotonic increasing relative to the set of objects, i.e. for sets of objects $\mathbf{C}$ and $\mathbf{D}$, $\mathbf{C} \subseteq \mathbf{D}$ implies $m(\mathbf{N}, \mathbf{C}) \subseteq m(\mathbf{N}, \mathbf{D})$. The *unique* approximation described below is an exception to this rule; *unique* returns the single object that uniquely matches the set of attributes. Second, most database approximation functions cannot approximate the set of objects that the client intended to identify without some information from the client; thus $m(\emptyset, \mathbf{D}) = \emptyset$ for any set of objects $\mathbf{D}$ is true for most database approximation functions. One specific function that does not conform to this principle is a database approximation function called *identity*; *identity* always returns the set of objects that it is given no matter what attributes 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 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. For example, consider a database that contains descriptions for four users:

⟨ `uid:cmb, office:737, department:cs` ⟩
⟨ `uid:rao, office:737` ⟩
⟨ `uid:llp, office:725` ⟩
⟨ `uid:gmt` ⟩

and a name that consists of the attributes `office:737` and `department:cs`. In this case, *possible* returns `cmb`, `rao`, and `gmt`. 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. *Partial* would return `cmb` and `rao`, but not `gmt` in the example. The third approximation, *exact*, returns all objects that are described by all of the attributes in a name. For this example, *exact* returns just `cmb`—it is the only object that is known to match both attributes in the name. *Exact* computes Lipski's $\| Q \|_*$. The final approximation, *unique*, returns either the single object that exactly matches the name or the empty set—in this case `cmb` would be returned by *unique* but no objects would be returned for a name that consists of the single attribute `office:737` because there is no unique match. This preference may be used when the accuracy of the database is not in question though it may not be complete. That is, no object will be returned if it conflicts with one of the attributes that the client supplies. Note that this preference generalizes the idea of partial match retrieval [Ullm88].

<u>*Voting Preference*</u>: The voting preference, denoted $\prec_V$, views the attributes that are specified as votes for objects. The order is defined as:

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

The most preferred approximation, *unanimous*, prefers objects that receive all of the votes; i.e. all the attributes are registered for the objects. 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. 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.

<u>*Temporal Preference*</u>: 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 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:

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

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.

3.3 Resolution Functions

A resolution function is defined by an ordering on a set of client and database preferences; this ordering is called a *preference hierarchy*. The preference hierarchy specifies the relative importance of each preference when resolving names. Functionally, this means that a resolution function selects an approximation function from each preference, constructs a composition of the functions, and evaluates the composition relative to the given name and database. It continues to select functions until one combination computes a non-empty set of objects. The preferences specify the order in which approximation functions are selected and the preference hierarchy specifies the order of composition for the sets of approximations. 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.

For example, consider the construction of a resolution function for a naming system with the following three assumptions. First, the client always presents the naming system with accurate information. Second, the database contains information that may be incomplete. Finally, some of the information in the database is authoritative—i.e., it is always up-to-date—and some is cached. For a naming system based on these assumptions, three preferences seem most applicable:

$$\prec_R\text{: } \textit{open} \prec_R \textit{closed}$$

$$\prec_T\text{: } \textit{cached} \prec_T \textit{authoritative}$$

$$\prec_M\text{: } \textit{partial} \prec_M \textit{exact}$$

This collection of preferences means that the resolution function can select from eight different sets of approximations:

$\langle$ *open* , *authoritative, exact* $\rangle$	$\langle$ *open* , *authoritative, partial* $\rangle$
$\langle$ *open* , *cached, exact* $\rangle$	$\langle$ *open* , *cached, partial* $\rangle$
$\langle$ *closed* , *authoritative, exact* $\rangle$	$\langle$ *closed* , *authoritative, partial* $\rangle$
$\langle$ *closed* , *cached, exact* $\rangle$	$\langle$ *closed* , *cached, partial* $\rangle$

It is reasonable to assume that the registered preference $\prec_R$ is more important than the match based preference $\prec_M$ and at least as important as the temporal preference $\prec_T$. This assumption implies that the relative importance of the three preferences defines a preference hierarchy $\lhd$ as $\prec_M \lhd \prec_T \lhd \prec_R$.

The preference hierarchy has two effects on the evaluation of sets of approximations. First, the approximations in any set must be applied in order from most important to least important. For example, the first set of approximations above must be evaluated in order: *closed*, *authoritative*, and *exact* because *closed* is the approximation from the most important preference and *exact* is from the least important. Three rules guide the evaluation of an ordered set of approximations $f_1, f_2, \ldots f_n$:

1. If f_i and f_{i+1} are database approximation functions then the composition of f_i and f_{i+1} is a database approximation function f defined as $f(N, D) = f_i(N, D) \cap f_{i+1}(N, D)$.

2. If f_i and f_{i+1} are client approximation functions then the composition of f_i and f_{i+1} is a client approximation function f defined as $f(N) = ((f_i(N) \cup f_{i+1}(N)) - N) \cup (f_i(N) \cap f_{i+1}(N) \cap N)$.

3. If f_i is a client approximation and f_{i+1} is a database approximation then the composition of f_i and f_{i+1} is a database approximation function f defined as $f(N, D) = f_{i+1}(f_i(N), D)$.

The first two rules are applied until the ordered set alternates between database and client approximation functions. At this point the third rule is applied to construct a single approximation function. Thus the composition of $\langle$ *closed* , *authoritative, exact* $\rangle$ —called a composite approximation function—is a function f defined as $f(N, D) = authoritative(closed(N), D) \cap exact(closed(N), D)$.

The second effect of the preference hierarchy is the order in which a set of composite approximation functions are applied to the name. A preference hierarchy specifies an induced preference on a set of composition functions. In this example, the resolution function attempts both exact and partial matches on the closed attributes before attempting any match on open attributes. The preference hierarchy induces the following preference on the sets of approximations:

$$\begin{array}{llll}
\langle \textit{ open , cached, partial } \rangle & \prec & \langle \textit{ open , cached, exact } \rangle & \prec \\
\langle \textit{ open , authoritative, partial } \rangle & \prec & \langle \textit{ open , authoritative, exact } \rangle & \prec \\
\langle \textit{ closed , cached, partial } \rangle & \prec & \langle \textit{ closed , cached, exact } \rangle & \prec \\
\langle \textit{ closed , authoritative, partial } \rangle & \prec & \langle \textit{ closed , authoritative, exact } \rangle &
\end{array}$$

In other words, $\langle$ *closed* , *authoritative, exact* $\rangle$ is the most preferred approximation. If this composite approximation function computes a non-empty set of objects, then that set is returned by the resolution function. The resolution function evaluates compositions in order from most preferred to least preferred. It returns the first non-empty set computed by a composition.

Formally, if we are given a set of preferences $\Pi = \{\prec_1, ..., \prec_N\}$, totally ordered by the preference hierarchy $\lhd$, then for any two members $\prec_j$ and $\prec_k$ of Π, we say that $\prec_k$ is *more important than* $\prec_j$ if $\prec_j \lhd \prec_k$. 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 $\langle \Pi, \lhd \rangle$ is defined to be the lexicographic order on the following set of compositions:

$$\pi_1 \times \pi_2 \times \ldots \times \pi_n$$

Given an induced preference on a set of compositions, the 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 composition function, c_i, unless every more preferred composition—that is any composition c_j where $c_i \prec c_j$—failed to compute a non-empty set of objects. Formally,

Definition 3.3 (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* $\mathbf{A}$ *and a domain of objects* $\mathbf{O}$ *by* $\rho(\mathbf{N}, \mathbf{D}) = \pi_i(\mathbf{N}, \mathbf{D})$ *where*

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

Intuitively, a composition function supports several approximations of the perfect world system—one from each preference in the metaorder. 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.

To complete the example, consider a resolution function defined by a Univers name server. This function implements the preference order $\prec_M \lhd \prec_C \lhd \prec_R$. Several redundant compositions have been removed from this definition. For example, the approximations *partial* and *exact* compute the same set of objects for names that consist of closed attributes. Thus $partial(closed(N), D)$ is equivalent to $exact(closed(N), D)$.

```
(define (lookup N D)
  (cond
    [(intersect (authoritative (closed N) D) (exact (closed N) D))]
    [(intersect (cached (closed N) D) (exact (closed N) D))]
    [(intersect (authoritative (open N) D) (exact (open N) D))]
    [(intersect (authoritative (open N) D) (partial (closed N) D))]
    [(intersect (cached (open N) D) (exact (open N) D))]
    [(intersect (cached (open N) D) (partial (open N) D))]
  )
)
```

3.4 Tools For Reasoning About Resolution Functions

The discussion in the previous section focused on the structure of the preference induced by a preference hierarchy. This makes it possible to understand and reason about some aspects of the 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.

3.4.1 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 3.4 (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 3.5 (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 preference hierarchy $< \Pi, \lhd >$. Then, given information about the soundness or completeness of each approximation function in the preferences of Π, 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 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 composition 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 composition returns. This observation leads to the definition of soundness and completeness for a composite approximation function: 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 monotonic database 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 $(\mathbf{N}, \mathbf{D})$ 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. Since c and d are monotonic increasing relative to the database, $(\mathbf{N}, \mathbf{D}) \in \Sigma_c$ implies that every subset of $\mathbf{D}$ is also in Σ_c. Therefore, if c is sound on $(\mathbf{N}, \mathbf{D})$ then $c \circ d$ is sound on $(\mathbf{N}, \mathbf{D})$. □

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 $c(\mathbf{N}, \mathbf{D}) = \{\mathbf{x} \in \mathbf{D} \mid \mathbf{x} = c(\mathbf{N}, \{\mathbf{x}\}\}$ for all sets of objects $\mathbf{D}$—is complete on $\Gamma_c \cap \Gamma_d$.*

Proof: If d is complete on $(\mathbf{N}, \mathbf{D})$ then it returns at least the objects that the client seeks. If c is point-wise determined and complete on $(\mathbf{N}, \mathbf{D})$ then it is complete on every subset of $\mathbf{D}$ that contains $oracle(\mathbf{N}, \mathbf{D})$. 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 $(\mathbf{N}, \mathbf{D})$ in P_c and an empty set of objects for every $(\mathbf{N}, \mathbf{D})$ 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.

3.4.2 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 3.6 *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 Applications

The preference hierarchy provides a framework for precisely specifying and reasoning about naming systems. This section shows how the preference hierarchy can be applied to several existing naming systems. It also demonstrates how the preference hierarchy can be used as a prescriptive model for designing new naming systems. This section concludes with of discussion of how different naming systems can be compared to each other. The appendix contains procedures that implement each resolution function.

4.1 Profile Resolution Functions

The Profile naming system [Pete88] provides a *white-pages* service that is used to identify users and organizations. Profile supports a suite of resolution functions that were designed with three assumptions in mind. First, Profile assumes that the database is honest, but that the data it contains is potentially incomplete. Profile automatically generates some of the information in the database. These attributes are considered closed because they are entered 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 enter different information than another. Second, the attributes contained in a name may be inaccurate, because people may forget or incorrectly remember information about other people. Third, as mentioned earlier, clients of the Profile system 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 resolution functions in Profile.

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

Definition 4.1 (Profile Client Approximation Functions) *The client approximation functions used by the Profile naming system are:*

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

- $closed(\mathbf{N}) = \{\mathbf{t} : \mathbf{v} \in \mathbf{N} \mid$ *for all* $\mathbf{u}$*,* $\mu(\mathbf{t} : \mathbf{u})$ *describes* $\mu(\mathbf{x})$ *implies that* $\mathbf{t} : \mathbf{u}$ *is registered for* $\mathbf{x}\}$
- $open(\mathbf{N}) = \mathbf{N} - closed(\mathbf{N})$

Definition 4.2 (Profile Database Approximation Functions) *The database approximation functions used by the Profile naming system are:*

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

We now formally define each of the four resolution functions that comprise the Profile naming system. For each function, we give an intuitive overview of the function, the preferences used by the function, and the importance order on the preferences.

<u>*Profile*$_1$</u>: Returns all objects that match any of the given attributes. *Profile*$_1$ is defined by $\prec_V \lhd \prec_U$ on the following preferences:

$\prec_U$: *universal*

$\prec_V$: *also-ran*

<u>*Profile*$_2$</u>: Computes the conjunction of the given attributes, given preference to closed attributes over open attributes. *Profile*$_2$ is defined by $\prec_V \lhd \prec_R$ on the following preferences:

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

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

<u>*Profile*$_3$</u>: Like *profile*$_2$ except it uses open attributes to reduce the set of objects returned. *Profile*$_3$ is defined by $\prec_v \lhd \prec_r \lhd \prec_V \lhd \prec_R$ 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*

<u>*Profile*$_4$</u>: Matches at most one object. *Profile*$_4$ is defined by $\prec_V \lhd \prec_R$ on the following preferences:

$\prec_R$: *closed*

$\prec_V$: *unique*

Given these definitions, it is easy to show that a discrimination order$\sqsubseteq_\Delta$ exists for Profile's resolution functions. The order is as follows:

Proposition 4.1 *For* Δ *containing all name/database pairs:*

$$profile_1 \sqsubseteq_\Delta profile_2 \sqsubseteq_\Delta profile_3 \sqsubseteq_\Delta profile_4$$

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.[3] 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 errors in binary information.

The following claims about completeness are valid under this assumption:

Proposition 4.2 *The function profile*$_1$ *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 4.3 *Profile*$_2$ *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 $\Gamma_{cu} = \{(\mathbf{N}, \mathbf{D}) | \mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ implies that $\mu(closed(\mathbf{N}))$ *represents* $\mu(\mathbf{x})\}$. If $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ then $\mu(closed(\mathbf{N}))$ *represents* $\mu(\mathbf{x})$. By the definition of *closed*, each of these attributes is registered in $\mathbf{D}$. Since $\langle$ *closed*, *unanimous* $\rangle$ returns the objects that match all of the closed attributes it returns $\mathbf{x}$. Therefore, $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ implies that $\mathbf{x} \in unanimous(closed(\mathbf{N}), \mathbf{D})$ and $\langle$ *closed*, *unanimous* $\rangle$ is complete on Γ_{cu}.

Let $\Gamma_{ca} = \{(\mathbf{N}, \mathbf{D})|$ there exists a non-empty set $C \subseteq closed(\mathbf{N})$ such that $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ implies that $\mu(C)$ *represents* $\mu(\mathbf{x})\}$. If $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ then there is a non-empty set of closed attributes, C, that represent $\mathbf{x}$. By definition, each attribute in C is registered for $\mathbf{x}$ in $\mathbf{D}$. *also-ran(closed(*$\mathbf{N}$*),*$\mathbf{D}$*)* is the set of objects in $\mathbf{D}$ for which at least one of the closed attributes is registered. Therefore, $\mathbf{x}$ is contained in *also-ran(closed(*$\mathbf{N}$*),*$\mathbf{D}$*)* if $\mathbf{x}$ is in $oracle(\mathbf{N}, \mathbf{D})$ 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 $\Gamma_{cu} \cup (E_{cu} \cap \Gamma_{ca}) = \Gamma_{ca} \subseteq \Gamma_{profile_2}$. Thus, Proposition 4.3 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 so that object does not match every attribute in the name according to the database. This leads to the following definition: An attribute $\mathbf{a}$ is *relevant* to a set of objects $\mathbf{D}$ if for every object $\mathbf{x}$ in $\mathbf{D}$, $\mu(\mathbf{a})$ describes $\mu(\mathbf{x})$ implies $\mathbf{a}$ is registered for $\mathbf{x}$. (Thus, an attribute is closed if it is relevant to the entire database). Using this definition, it is possible to show that:

Proposition 4.4 *Profile*$_2$ *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 $\Gamma_{ou} = \{(\mathbf{N}, \mathbf{D}) | \mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ implies that $\mu(open(\mathbf{N}))$ *represents* $\mu(\mathbf{x})$ and each attribute in $open(\mathbf{N})$ is registered for $\mathbf{x}\}$. If $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ then $\mu(open(\mathbf{N}))$ *represents* $\mu(\mathbf{x})$ and each attribute in $open(\mathbf{N})$ is registered for $\mathbf{x}$ in $\mathbf{D}$. Recall that, $\langle$ *open*, *unanimous* $\rangle$ returns the objects that match all of the open attributes. Therefore, $\mathbf{x} \in unanimous(open(\mathbf{N}), \mathbf{D})$ if $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ so that according to the definition of completeness $\langle$ *closed*, *unanimous* $\rangle$ is complete on Γ_{ou}.

[3] 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.

When *profile*$_2$ 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 $(\mathbf{N}, \mathbf{D})$ where $closed(\mathbf{N})$ 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 $\Gamma_{ou} \subseteq \Gamma_{profile_2}$. Therefore, *profile*$_2$ is complete on Γ_{ou} and Proposition 4.4 holds. □

Proposition 4.5 *Profile*$_2$ *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 $\Gamma_{oa} = \{(\mathbf{N}, \mathbf{D}) | \mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ implies that there exists a non-empty set $O \subseteq open(\mathbf{N})$ such that $\mu(O)$ *represents* $\mu(\mathbf{x})$ and each attribute in O is registered for $\mathbf{x}\}$. If $\mathbf{x} \in oracle(\mathbf{N}, \mathbf{D})$ then there is a non-empty set of open attributes, O, that represent $\mathbf{x}$ and each attribute in O is registered for $\mathbf{x}$ in $\mathbf{D}$. *Also-ran(open(*$\mathbf{N}$*),*$\mathbf{D}$*)* is the set of objects in $\mathbf{D}$ for which at least one open attributes is registered. Therefore, $\mathbf{x}$ is contained in *also-ran(open(*$\mathbf{N}$*),*$\mathbf{D}$*)* if $\mathbf{x}$ is in $oracle(\mathbf{N}, \mathbf{D})$ and $\langle$ *open*, *unanimous* $\rangle$ is complete on Γ_{oa}.

If every name $\mathbf{N}$ 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, *profile*$_2$ is complete on Γ_{oa} and Proposition 4.5 holds. □

Since *profile*$_3$ 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 *profile*$_3$ is identical to *profile*$_2$. Thus, we have

Proposition 4.6 *Profile*$_3$ *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 4.6 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. Thus, 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 4.7 *Profile*$_3$ *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 4.6 is as a special case of Proposition 4.7 because a closed attribute is by definition relevant to all objects.

The resolution functions provided by the Profile naming system attempt to respond with complete answers. The analysis presented in this section points out one design decision that adversely affects this goal. 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. 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, it is possible to design a new function similar to *profile*$_2$ that does not use a precise match on open attributes, and thus, avoids this problem. The function, called *profile*$_{new}$, is defined 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*

where $\prec_v \lhd \prec_r \lhd \prec_V \lhd \prec_R$. The following two propositions hold for $profile_{new}$:

Proposition 4.8 *Function $profile_{new}$ is more discriminating than $profile_2$ 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 4.9 *Function $profile_{new}$ 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. $Profile_2$ sacrifices completeness for a substantial set of name/database pairs because it uses an exact match on open attributes, whereas $profile_{new}$ overcomes this weakness at the expense of being less discriminating for certain unlikely sets of attributes.

4.2 Lookup Resolution Function

Consider a naming system where 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 seems unreasonable 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, domain:as.arizona.edu` ⟩, *also-ran* may match an object with the attributes ⟨ `name:toto, domain:as.arizona.edu` ⟩, even though `name:toto` conflicts with part of the client's specification. In this case, since by definition a processor may have only one name, the object that is returned cannot be one that the client intends to identify.

We define a new resolution function, *lookup*, specifically for this situation. *Lookup* uses *partial* and *possible* in conjunction with closed attributes to guarantee that no object that is returned conflicts with attributes specified by the client. Formally, *partial* and *possible* are defined as follows:

Definition 4.3 (Possible Database Approximation Function) $possible(\mathbf{N}, \mathbf{D}) = \{\mathbf{x} \in \mathbf{D} |$ *for every* $\mathbf{t} : \mathbf{v} \in \mathbf{N}$ *either* $\mathbf{t} : \mathbf{v}$ *is registered for* $\mathbf{x}$ *or no attribute with tag* $\mathbf{t}$ *is registered for* $\mathbf{x}$ $\}$

Definition 4.4 (Partial Database Approximation Function) $partial(\mathbf{N}, \mathbf{D}) = \{\mathbf{x} \in possible(\mathbf{N}, \mathbf{D}) |$ *there is a* $\mathbf{a} \in \mathbf{N}$ *such that* $\mathbf{a}$ *is registered for* $\mathbf{x}\}$

The *lookup* 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 $\prec_m \lhd \prec_r \lhd \prec_p \lhd \prec_M \lhd \prec_R \lhd \prec_P$. If the assumptions about the information in the system are valid, then *lookup* 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.

4.3 Other Non-Trivial Resolution Functions

This section discusses several other non-trivial resolution functions.

4.3.1 Yellow-Pages

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 $\mathbf{N}_m$ and $\mathbf{N}_o$ respectively, where the optional attribute set is ordered; i.e., $\mathbf{N}_o = \{\mathbf{a}_1, \mathbf{a}_2, ...\mathbf{a}_n\}$. 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 $\prec_M \lhd \prec_A$, 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*

For example, *yp* allows a client to ask for a processor that has a 68020 architecture and supports a Pascal compiler, and of all processors that possess those properties, select those with a load less than 1.5, should any exist.

4.3.2 CSNET

Consider the CSNET name server that is used to identify users and organizations [Solo82]. Like *yp*, the CSNET name server defines a client approximation in which a given set of attributes are partitioned into *mandatory* and *optional* subsets. The resolution function used by the CSNET name server, denoted *csnet*, is defined by $\prec_m \lhd \prec_f \lhd \prec_M \lhd \prec_F$ on following preferences:

$\prec_F$: *mandatory*

$\prec_M$: *exact* $\prec_M$ *unique*

$\prec_f$: *optional*

$\prec_m$: *identity* $\prec_m$ *maximum*

where *maximum* implies the set of objects that match the maximum number of attributes. The function-based preference is given priority over the match-based preference.

4.3.3 NIC

The NIC name server (also called WHOIS) limits queries to a single attribute and defines a resolution function that returns all objects partially matched by that attribute [Pick79]. The resolution function is therefore functionally equivalent to $profile_1$. 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.

4.4 Resolution Functions for Conventional Naming Systems

While descriptive naming systems such as Profile, the CSNET name server, and the NIC name server take advantage of the full expressive power of the preference hierarchy, understanding preferences provides insight into more conventional systems.

Conventional naming systems support a simple inference system: They restrict the database to accurate, unambiguous attributes and they restrict queries to names containing a single, accurate attribute. This implies that all matches are unique. As a consequence, conventional naming systems are trivially given by resolution function *conv*, which returns the the set of objects uniquely matched by the set of unambiguous attributes in the name. Given this definition, it is easy to see the following:

Proposition 4.10 *The function conv is sound for all singleton sets of attributes.*

Notice, however, that that the restriction that only unambiguous attributes may be registered implies that such systems cannot be complete for attribute sets other than those that contain a single, unambiguous attribute. Thus,

Proposition 4.11 *The function conv is complete for all sets of attributes that contain no ambiguous attributes.*

Consider a common scenario in which the naming system is used to map a symbolic name into a machine-readable address. In this case, the symbolic name registered for a particular object is an *always* attribute—one that always describes the object—and the object's machine-readable address is either a *now* attribute—one that currently describes the object—in which case the system must be designed to flush the binding if there is any chance that it has become stale, or it is a *sometime* attribute—one that described the object at one time but may not describe it now—in which case users of the naming system must be able to accommodate out-of-date information. The later case is commonly referred to as a *hint* [Terr87].

It is important to note that conventional naming systems, because of the requirements placed on them, generally sacrifice completeness for the sake of soundness. Descriptive naming systems, on the other hand, often sacrifice soundness for the sake of completeness. To accomplish this, conventional (low-level) system are likely to deal exclusively with preference classes that are high in the preference hierarchy (e.g., *unambiguous*, *unique* and *always* attributes), whereas descriptive (high-level) systems are likely to accept preference classes that are low in the hierarchy (e.g., *ambiguous* and *sometime* attributes). Moreover, while high-level naming systems are more likely to accommodate multiple preference classes, even low-level naming systems that accommodate only one preference class have implicitly made a decision regarding that preference. That is, they mandate a certain preference class.

4.5 Comparing Systems

The preference hierarchy provides a handle on comparing naming systems. For example, because conventional naming systems assume *unambiguous* attributes and *unique* matches, and both these preference classes are present in the preference hierarchy upon which Profile is based, one can to directly compare Profile's most discriminating resolution function with *conv* as follows:

$$profile_4 \sqsubseteq_{\Delta} conv$$

for all sets of name/database pairs in Δ. In other words, it is accurate to view *conv* as a restrictive member of the Profile 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 that of 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* preference classes, while yellow-pages services are based on *mandatory* and *optional* preference classes.

Furthermore, our experience strongly suggests that these preference classes correctly represent the environment in which the two systems operate and the requirements placed on the two systems by the clients 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 in a way that accommodates missing information. Also, because clients 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 <u>all</u> 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 ensure all the necessary information be stored in the database—a program is responsible for registering attributes—the distinction between open and closed attributes is not an issue. Moreover, the client that names a computational resource is most interested in distinguishing between similar resources; i.e., clients generally do not 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.

5 Concluding Remarks

This paper reasons 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. The preference hierarchy has proven powerful enough to describe the naming systems we have encountered. We have implemented a general name server that explicitly enforces this model and we are using it to implement many of the naming systems described in this paper.

In addition to providing a formal model for understanding existing naming systems, we have also found the preference hierarchy to be a prescriptive tool that can be use when designing new naming systems. For example, the Profile naming system introduced in Section 3 was designed before we had formally defined the preference hierarchy, but clearly, an intuitive understanding of preferences guided the design. It is interesting to note that the formal specification of the preference hierarchy led us to understand and correct a subtle flaw in the original definition of $profile_3$. It also led to the definition of the new function $profile_{new}$ given in Section 4.1

As another example, after having defined the preference hierarchy, we were able to apply it to the problem of designing a yellow-pages service used to identify computational resources such as printers, processors, databases, network servers, and so on [Pete87]. 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 resolution function *yp* given in Section 4.3.

As a final example, we have observed that the domain naming system (DNS) [Mock87] evolved in a way that suggests an implicit understanding of the preference hierarchy. Specifically, the domain naming system 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 correctly deal 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 $sometime \prec_t now$. The important point is that even in functionally simple systems that maps names into addresses, it is possible for the environment in which the system is implemented to impose constraints on the system, and that the technique used to deal with these constraints can, in turn, be expressed in terms of the preference hierarchy.

Appendix

This appendix contains several Scheme function definitions used to implement the name resolution functions described in Section 4. 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 serveral optimizations have been made in order to avoid redundant computations of composite approximation functions.

```
(define (profile1 N D)
  (also-ran (universal N) D)
)


(define (profile2 N D)
  (cond
    [(unanimous (closed N) D)]
    [(also-ran (closed N) D)]
    [(unanimous (open N) D)]
    [(also-ran (open N) D)]
  )
)


(define (profile3 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 (profile4 N D)
  (unique (closed N) D)
)
```

```
(define (profile-new 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]
  )
)
```

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

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

```
(define (csnet N D)
  (cond
    [(unique (mandatory N) D)]
    [(maximum (optional N) (exact (mandatory N) D))]
    [(exact (mandatory N) D)]
  )
)
```

```
(define (conv N D)
  (unique (unambiguous N) D)
)
```

References

[Born87] Borning, A., Duisberg, R., and Freeman-Benson, B. Constraint hierarchies. In *Proceedings of OOPSLA '87*, pages 48–60, October 1987.

[Budd86] Budd, T. Bib — a program for formatting bibliographies. In *UNIX User's Supplementary Documents*. 4.3 Berkeley Software Distribution, April 1986.

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

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

[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.

[ISO88] ISO, . Information processing systems: open systems interconnection—the directory—overview of concepts, models, and service. Technical Report Draft International Standard ISO 9594-1:1988(E), International Standards Organization, 1988.

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

[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.

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

[Oppe81] Oppen, D. and Dalal, Y. The clearinghouse: a decentralized agent for locating named objects in a distributed environment. Technical Report OPD-T8103, Xerox Office Products Division, Palo Alto, California, October 1981.

[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.

[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.

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

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

[Sun86] Sun Microsystems Inc., Mountain View, California. *Yellow pages protocol specification*, February 1986.

[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.

[Wats81] Watson, R. Identifiers (naming) in distributed systems. In Lampson, B., Paul, M., and Siegert, H., editors, *Lecture Notes in Computer Science: Distributed Systems—Architecture and Implementation*, pages 191–210. Spinger-Verlay, New York, New York, 1981.

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