An Analytical Taxonomy of Naming Systems

Bryan Bayerdorffer
Department of Computer Sciences
University of Texas at Austin

TR-92-48 December, 1992

Abstract

A *naming system* consists of the mechanisms that govern the definition, binding, and access to the names upon which communication among active objects in a parallel or distributed system depends. A concurrent algorithm may require certain patterns of communication among objects. Naming systems differ significantly in the patterns of communication that they support, and this diversity suggests the existence of efficiency tradeoffs and specialization among the various approaches. To enable the impact of these differences upon the structure of concurrent computations to be examined systematically, we present a *taxonomy of naming systems* that isolates a small set of fundamental naming system properties, and ranks naming systems in (partial) order of the expressiveness they derive from their properties. We compare the properties of the naming systems that underlie several representative concurrent programming systems, and give examples of algorithms that require specific naming system properties.

Keywords: classification, communication properties, concurrent programming, expressiveness, naming systems.

1 Introduction

An important component of any parallel or distributed system is the *naming system* that underlies communication among active objects. A naming system consists of the mechanisms that define names and make them accessible to objects, bind names to objects, and resolve name references during execution. These mechanisms are incorporated into concurrent programming languages as well as runtime environments. Certain fundamental properties of naming systems affect the structure of computations that may be expressed in concurrent programs by enabling or prohibiting specific patterns of communication among objects. Because naming systems differ significantly in their properties, a thorough understanding of naming systems is useful to both the designers and users of parallel and distributed systems. However, unlike other important elements of such systems—e.g. synchronization, resource allocation, or scheduling—naming systems have not been widely studied as an independent problem domain. In particular, the lack of a precise understanding of the role of binding and name access (modification of the set of names known to an object) in communication contributes to the difficulty of concurrent programming.

1.1 Role of Naming Systems in Communication

A concurrent computation consists of a set of active, communicating *objects*. An object is a process, task, or similar entity. In order for object A to successfully initiate a communication with object B, there must exist a name that is *known* to A and *bound* to B. A name is a symbol that either directly points to an active object, or denotes a passive point of interaction among objects, such as a channel, port, mailbox, or shared variable. Irrespective of the mechanism, a communication occurs from a *source* object to a *destination* object, *over* a name. The relationship between the name and the destination object is called a *binding*. It is the function of the naming system to establish the necessary bindings and to control which names are known to a given object.

Beyond the fundamental role of names and bindings in distinguishing objects from one another there is great variety in the ways that names and bindings are established and manipulated. Naming systems differ, for example, in the number of bindings in which one name may simultaneously appear, or in whether the set of bindings may be modified during execution. This diversity suggests the existence of tradeoffs and specialization among the various approaches. It also leads one to search for a unifying framework within which the tradeoffs and design choices can be systematically evaluated.

A concurrent algorithm may require certain patterns of communication among objects. We have identified several fundamental properties of naming systems that determine their ability to express such patterns. For example, a naming system may allow a single name to be bound to a set of objects to enable broadcasting, or may allow each name to be used by at most one object (e.g. exclusive access to a channel). Consequently, for a given algorithm specification there may exist a straightforward implementation under one naming system, while the same algorithm may be awkward or inefficient to implement under another naming system. Constraints on the form of the implementation (e.g. "clients may not poll servers") may preclude the existence of a solution altogether. The *taxonomy* that we define in this paper ranks naming systems in (partial) order of the expressiveness they derive from their properties.

1.2 A Preliminary Example: Resource Tracking

Consider a system in which a set of *resources* is shared among several objects that communicate via message passing. Each resource is of a specific type, and there exists exactly one instance of each type. A resource is *owned* by one object at a time and can only be accessed by its owner. At any time, an object may relinquish its ownership of a resource in response to a request from another object, which then becomes the new owner of the resource. Periodically, an object must acquire several resources in order to perform some operation. It sends requests to the current owners of the needed resources and then waits for these to relinquish the resources before it can proceed. Aside from the possibility of deadlock, the fundamental problem encountered in implementing such a system is that of locating the current owner of a resource to whom a request is to be issued.

Suppose that we wish to solve this problem without using a central coordinator that keeps track of the ownership of the resources. If the communication primitive used is, e.g. **send**(*name, message*), then the

most natural solution is one in which an object simply executes **send**(*"owns resource R", request*) whenever it requires resource R. However, such a solution assumes that the name *"owns resource R"* is in fact bound to the correct object, and that it is *rebound* whenever the ownership of R changes. The ability to rebind a name during execution is a property of the naming system called *dynamic binding*, which we define formally in section 2.3.1. Without this property, the objects must execute some more complex and less efficient protocol, either sending every request to *all* objects, or forwarding a request along a chain of previous owners of the requested resource. In section 3.6 we discuss two naming systems that differ from most in that they allow dynamic binding.

A simple generalization of the resource tracking problem is to allow multiple instances of each type of resource. In this case, it suffices for an object to acquire any *one* instance of a resource type, but the request should still be delivered to *all* objects that hold an instance of the requested type. To support this, the naming system should thus allow the name *"owns resource R"* to be bound to the *set* of objects that hold an instance of R. This is the *1–N multiplicity* property, defined in section 2.3.3.

1.3 Impact of Naming Systems

It will become apparent that the definitions of names and bindings used here are quite broad. The concept of a name encompasses such familiar items as process identifiers and entries, but also includes such things as unbound variables in parallel logic languages and the data values satisfying the condition of a selective accept statement in Concurrent C[11]. Similarly, a binding may be reflected by the association of a constant value with an object (e.g. the static naming of processes in CSP), by explicitly storing a name in some naming data structure (e.g. opening a port), or it may be implicit in the state of an object (e.g. executing selective accept with a particular expression). An important step in identifying the properties of a naming system is to determine how names and bindings are represented under that naming system. We give several examples of this procedure in section 3.

Several significant characteristics of concurrent systems are influenced by properties of the underlying naming system. Among these are:

- *Blocking and queueing of communications:* In initiating a communication, reference to a name that is not bound to any object may cause the communication to block or be queued until the name is bound.
- *Modularity:* Independent specification of the shared components of a system (e.g. servers) requires that they be accessed (e.g. by clients) using shared names.
- *Dynamic structures:* To allow creation of new communicating objects during a computation, naming systems must enable objects to learn new names (i.e. names of objects created on the fly), or must permit existing names to be dynamically bound to new objects.
- *Abstraction:* Typical communication primitives specify one-to-one communication among directly-named objects. By allowing objects to be referenced through indirect names, and allowing names to be bound to sets of objects, communication may be specified at a higher level of abstraction.

1.4 Syntax vs. Semantics

A naming system consists of two complementary sets of mechanisms: those that establish bindings of names to objects, and those that subsequently make use of these bindings to resolve name references. Comer and Peterson have observed that name resolution is a strictly syntactic process[6], and that all semantic information (i.e. the meaning of a name with respect to a particular concurrent computation) resides in the bindings of names to objects. A name is given a meaning by binding it to an object (or set of objects), and that meaning is independent of the method by which references to that name are later resolved. An analogy with compilers serves to illustrate the relationship between binding and resolution: Compilation of a program requires a language definition step, which assigns operations to syntactic constructs, and a translation step, which takes a given syntactic construct and generates the corresponding operations. *Resolution is to binding as translation is to language definition.*

1.5 Overview

Typically, the influence of communication mechanisms (and hence naming systems) on the structure of concurrent computations has been examined in an ad hoc manner. It has been shown, for example, that limitations of the Ada[8] rendezvous mechanism (which relies on a comparitively restrictive naming system) lead to programs that poll[12], and that the requirement that communicating processes in CSP[17] name each other precludes certain desirable communication patterns. Improvements have been suggested for both languages. Such analyses, while useful, have two shortcomings: they lack generality because they are stated in terms of the syntax and semantics of the particular language or system in question, and they do not clearly distinguish the roles of names and bindings, even though these are often critical. Therefore the results do not extend in an obvious way to other concurrent systems.

The taxonomy enables a more systematic analysis of naming systems and their impact upon the structure of computations by defining a set of naming system properties that are independent of any particular concurrent programming language or system. Once a particular subset of these properties have been shown to hold for a naming system, it becomes possible to determine whether the naming system permits the communication patterns necessary for the solution of a given problem. Instead of attempting to implement the solution, it is only necessary to determine at a high level the naming system properties required by the problem.

1.5.1 Structure of the Taxonomy

The naming system properties are defined using a model of concurrency that differs from conventional models in two ways. First, it distinguishes exactly that part of the state of a computation that enables us to determine the bindings and known names that exist at any point in the computation. Second, it specifies for each communication in a computation the name(s) used in carrying out that communication.

We capture the relationship between names, bindings, and communication by augmenting the standard partial-order model of concurrency[21] with information about the state of the naming system, as follows: A computation is a partial order of *events* interconnected by explicitly represented *communications*. Each event is the execution of an *action* by an object. The events are annotated with the names known to and bound to the object at which the event occurs. Communications are triples containing the source and destination events, and the name over which the communication occurs. The only significant characteristics of events are their annotations, their positions in the partial order that constitutes a computation, and whether they participate in a communication.

Each naming system property is a predicate in first-order logic that describes the structure of these computations. A naming system is said to have a given property **iff** the computations it permits satisfy the predicate. An ordering of the properties into six orthogonal pairs defines a lattice-structured hierarchy of *categories*, which ranks naming systems according to their expressiveness. A category is a combination of properties, and a naming system is a member of a category **iff** it has all of the corresponding properties.

1.5.2 Characteristics of the Taxonomy

To summarize: The model of concurrency, properties, and hierarchy together form a *taxonomy* of naming systems with the following characteristics.

- *Specification of communication semantics of naming systems:* Work on naming in concurrent systems has focused on resolution, the syntactic aspect of naming[6, 9, 20, 22]. The taxonomy identifies those semantic properties that are significant in the specification of communication, which have received little previous systematic study.

- *Separation of concerns:* Naming systems are examined in isolation both from other characteristics of concurrent systems and from the implementation details of naming system properties.

- *Utility:* The hierarchy enables systematic evaluation of the relative expressiveness of naming systems. This includes the selection or design of a naming system for a particular problem domain, comparison of different implementations of equivalent naming systems, or checking the tractability of a problem under a given naming system prior to implementing a solution.

- *Coverage:* The taxonomy covers the naming system properties underlying variety of communication models, including message-passing systems, shared data structures, rendezvous, dataflow, and single-assignment variables (e.g. in parallel logic languages). Often models that differ substantially in other respects have many naming system properties in common.

The rest of this paper is organized as follows: In section 2, we give definitions for the properties and the hierarchy. Section 3 traces a path through the hierarchy that illustrates the correspondence between successively more expressive naming system properties and increasingly general communication mechanisms, which parallels the evolution of concurrent systems from CSP to Ada, Concurrent C, and Linda.

2 Definitions

2.1 Model of Concurrency

A computation is a a partially ordered multiset, or *pomset*[15], of events, together with a multiset of communications. A pomset is a set in which an element may occur more than once, together with an ordering relation. The relation is defined on the distinct *occurrences* of the elements of the pomset. That is, one can think of the individual occurrences of a particular element as being indexed so that they can be distinguished. Similarly, the equality relation, $=$, only holds reflexively for each occurrence of an element—two distinct occurrences of the same element are not considered to be equal.

2.1.1 Objects and their Behaviors

The *objects* in a computation are the "communicating entities" that comprise a concurrent system. Examples of objects are CSP processes, Ada tasks, and clause evaluations in parallel logic languages. A *behavior* of an object is a linear sequence of events (i.e. an execution thread), where each event is an occurrence of an *action*. For each object there exists a *program*, which specifies a finite set of possible actions, and a (possibly infinite) set of possible behaviors of the object. Actions are not distinguished by the values of any variables to which they refer. In each computation, an object exhibits exactly one of its behaviors.

2.1.2 Names

A *name* is a *tuple* containing the *identifiers* that specify the destination of a communication. Often the only such identifier is one that appears in the action that initiates a communication; e.g. **send**(P, msg) to send a message to receiver P. In this case there is a 1–1 correspondence between identifiers and messages. Some naming systems use multiple identifiers to specify a communication. In CSP, for example, the sender and receiver of a message identify each other, as well as type of the message to be transmitted (see section 3.2).

2.1.3 Characteristics of Events

An event is distinguished from others by exactly four attributes: the object at which it occurs, its *binding*, and *domain* (see section 2.1.5), and its position in the partial order of events in the computation.

Definition: The set of *events* is $\mathcal{E} = \mathcal{O} \times \mathcal{D} \times \mathcal{B}$, where $\mathcal{O}$ is the set of all objects, and $\mathcal{D}$ and $\mathcal{B}$ are the set of all domains and the set of all bindings, respectively. □

Notation: We denote the components of an event e as $e.o$ (object), $e.d$ (domain), and $e.b$ (binding) □

Events are the "endpoints" of communications among objects, i.e., each event may be the *source* or *destination* of one or more communications. We needn't represent every underlying execution of an action with a corresponding event. The only interesting events are those that participate in communications, or those at which the state of the naming system changes (i.e. the set of names known to or bound to an object is modified).

2.1.4 Communications

A *communication* is a (*source event, name, destination event*) triple. The name that appears in the communication is an element of both the domain of the source event and the binding of the destination event, indicating that the name is known at the object at which the source event occurs, and is bound to the object at which the destination event occurs. The source and destination events of a communication are temporally ordered such that the source event precedes the destination event. Examples of source/destination event pairs are sending/receiving a message, writing/reading a shared variable, or invoking/accepting a remote procedure call.

Definition: The set of *communications* is $\mathcal{C} = \mathcal{E} \times \mathcal{N} \times \mathcal{E}$. □

Notation: We denote the components of a communication, c, as $c.s$ (source), $c.n$ (name), and $c.r$ (destination). □

2.1.5 Domains and Bindings

Generalizing from the the definitions in [6], a resolution mechanism is a function, $R(n, C)$ whose arguments are n, a name, and C, a context. A context is a function from names to sets of objects, defined by a set of tuples. Each such tuple is a binding, i.e. (*name*, {*set of objects*}). The function R returns the set of objects to which n is bound within context C.

States and Mechanisms A context is a component of the state of the naming system. The resolution function *reads*, but does not modify, this state. It is possible to define analogous functions for those mechanisms that do modify the state of the naming system.

The mechanism responsible for modifying contexts is the binding mechanism: $B(n, C, O)$, where n is a name, C a context, and O a set of objects. This function binds n to the set of objects O in context C, and possibly unbinds n from other objects. That is, it replaces (n, X) in C with (n, O), where X is the previous set of objects to which n was bound.

The naming system state consists not only of one or more contexts, but also of the set of names known to each object. The corresponding function that modifies this component of the state is $A(o, K, N)$, where o is an object, N is a set of names, and K is a function from objects to sets of names, representing the names known by a given object, and defined as a set of tuples, *(object,* {*set of names*}*)*. The function A replaces (o, Y) with (o, N) in K, where Y is the previous set of names known to object o.

Since resolution is outside the scope of, and orthogonal to, our model of concurrency, we assume that resolution always succeeds, and therefore does not affect the structure of computations that may be expressed under a given naming system. Consequently, a simplification that we can make immediately is to assume a single universal context, C (i.e. a global namespace). Without loss of generality, we can also assume that there exists a single space, K, for known names.

Annotation of Computations The state of the naming system, (C, K), at any point is the result of applying the binding and resolution functions in some finite sequence, and starting from some initial state. However, our model of concurrency is based upon partial orders of events, rather than on a state-transition model, because this enables a much simpler definition of the concept of a communication, which is central to the subsequent definitions of naming system properties. To capture information about the state of the naming system, each event in a computation is annotated with the appropriate elements of C and K.

There are two annotations for each event, called the *domain* and *binding* of the event, whose elements correspond to elements of the known-names function, K and the context, C, respectively.

Definition: The *domain* (*binding*) of an event is the set of names known (bound) to the object at which the event occurs, at the occurrence of the event. □

Domain of an Event An object "knows" a name at a point in its behavior **iff** it could originate a communication over the name, *were it free to execute any of its actions at that point.* Let T be the finite set

of actions that an object can execute, and let $e_0 \ldots e_{i-1} e_i$ be an initial subsequence of a possible behavior of the object. Let event x be an occurrence of an action from T. For all such events x, if the naming system allows a computation in which the behavior $e_0 \ldots e_{i-1} x$ appears (i.e. x is substituted for e), and that computation contains a communication (x, n, y), then the name n is in the domain of e_i.

Note that $e_0 \ldots e_{i-1} x$ is not necessarily a possible behavior of the object, since an object is not always free to execute any of its actions. However, it indicates a set of names that is accessible to the object at the point at which event e_i occurs.

Binding of an Event Where the domain of an event describes the maximal set of communications that may originate at the point in the behavior at which the event occurs, the binding of an event describes the maximal set of communications that may terminate at the point in the behavior at which the event occurs. A name may be *known* at a point in a behavior at which it is not possible for the object to use the name. However, a name is *bound* to an object exactly when it is possible for a communication to terminate at that event. In defining the binding of an event, we can therefore confine our attention to the possible behaviors of an object:

Let $e_0 \ldots e_i x$ and $e_0 \ldots e_i y$ be initial prefixes of possible behaviors of an object that are identical except for their last event (x differs from y). If the naming system allows a computation containing $e_0 \ldots e_i y$, and that computation contains a communication (z, n, y), then n is in the binding of x. That is, if the object is capable of receiving a communication over the name n at the point in its behavior where x occurs, then n is bound to the object at x.

Correspondence There is a direct correspondence between the domains and bindings of certain sets of events and the global state of the naming system. Consider taking a "snapshot" or "consistent cut"[5], of a computation. This cut, s, constitutes a globally consistent state. The context that exists at s is simply the union of the binding annotations of the events that immediately precede the cut. Similarly, the set of known names that exists at s is the union of all the domain annotations of the events that immediately precede the cut.

The domain of an event is determined by the semantics of the actions by which identifiers are created, destroyed, and propagated among objects. For example, where identifiers are treated as data items, a name may be added to the domains of subsequent events by receiving a message containing the name. Conversely, a protection scheme may cause a name to be removed from the domains of subsequent events by revoking access rights to the name.

The binding of an event is determined by the semantics of the actions by which identifiers are bound (explicitly or implicitly) to objects. Examples of actions that cause bindings to be modified are attaching a message port to a process, executing a read operation on a channel, or instantiating a new unbound variable in a parallel logic language.

2.1.6 Computations

Definition: (*precedes*, $\mapsto$) Let e and f be two events. The relation $e \mapsto f$ indicates that e necessarily precedes f in time[19]. $\mapsto$ is transitive, antisymmetric, and irreflexive. □

Definition: Two events, e, f are *concurrent* **iff** $\neg((e \mapsto f) \vee (f \mapsto e))$ □

Definition: A *computation*, γ, is a pair (Π, Δ), where Π is a pomset $(\langle e, \ldots \in \mathcal{E} \rangle, \mapsto)$, and Δ is a multiset of communications, $\langle c | (c \in \mathcal{C}) \wedge (c.s \in \Pi) \wedge (c.r \in \Pi) \rangle$ such that:

$$\begin{aligned}
&\forall (s, n, r) \in \Delta :: s \mapsto r \\
&\wedge \forall e, f \in \Pi :: e.o = f.o \Longrightarrow (e \mapsto f) \vee (f \mapsto e) \\
&\wedge \forall e, f \in \Pi :: (e.o \neq f.o) \wedge (e \mapsto f) \Longrightarrow \exists (x, n, y) \in \Delta :: (x = e \vee e \mapsto x) \wedge (y = f \vee y \mapsto f)
\end{aligned}$$ □

There are only two universal constraints on the structure of a computation: first, the source event of a communication always precedes the corresponding destination event and second, there is a total ordering of events at every object. In addition, two non-collocated events are ordered only where there exists

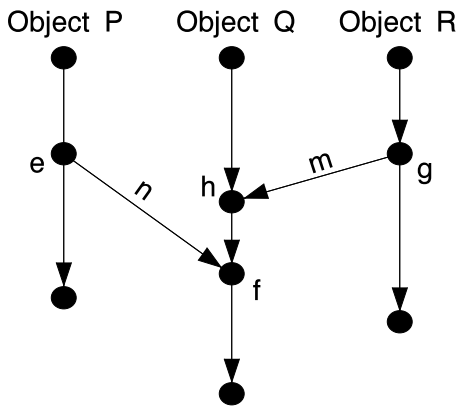

Figure 1: A computation

(transitively) a communication between the two events. Naming systems are differentiated according to the additional constraints they impose upon computations.

Figure 1 shows a possible computation with three objects. The computation contains two communications: (e, n, f) (from object P to object Q) and (g, m, h) (from object R to object Q). Name n is in the domain of event e and the binding of event f. Name m is in the domain of event g and the binding of event h.

2.2 Naming Systems

Definition: A *naming system* is a specification of the allowable domains, bindings, communications, and order of events of computations. For each naming system, a, there exists a *maximal computation set* Γ_a, the set of all computations specifiable under a. □

If a naming system admits the existence of particular domains, bindings, or communication patterns, then each of these will exist in at least one of the computations in its maximal computation set. Therefore, the naming system properties defined below characterize naming systems by stating the existence or nonexistence of such computations in a maximal computation set.

2.3 Properties of Naming Systems

There are six orthogonal characteristics of naming systems: Mutability of bindings at runtime, name access (ability of objects to learn new names during execution), binding of names to sets of objects, shared access to names, "descriptive" names, and binding of multiple names to an object. Each naming system property is a statement of the presence or absence of one of these characteristics in the set of all computations that may be specified under a given naming system. These properties meet the following criteria: None of the properties holds for every naming system; therefore, the properties reflect design decisions made in the development of the naming system. Each property may be necessary to, or may preclude the existence of solutions to certain classes of problems; therefore, the properties are relevant to the selection of a naming system that is appropriate for the solution of a given problem. Finally, the definition of a naming system property does not specify how the property is to be implemented.

In the definitions below, quantification is over the elements of a maximal computation set, Γ. For the sake of brevity, we use the following shorthand:

Notation:

"$\exists$ event e" means $\exists e \in \Pi \in \gamma \in \Gamma$
"$\exists$ behavior b" means $\exists b \in \gamma \in \Gamma$
"$\exists$ communication c" means $\exists c \in \Delta \in \gamma \in \Gamma$
"$\exists$ communication set Δ" means $\exists \Delta \in \gamma \in \Gamma$
"$\exists$ name n" means $\exists (e, n, f) \in \Delta \in \gamma \in \Gamma$ □

The definitions of the properties also refer to the following auxiliary function:

Definition: $\beta(n, e, \Delta) = \{o | ((e, n, f) \in \Delta) \wedge (f.o = o)\}$ □

The function $\beta(n, e, \Delta)$ denotes the set of objects at which the destination event(s) occur for those communications in Δ that have source event e and name n in common.

2.3.1 Mutability of Bindings

A naming system that provides a mechanism for altering bindings at runtime is said to have the *dynamic binding* property. The dynamic binding property provides the abstraction of indirect naming, which is comparable to the abstraction provided by pointers. This property is required where a name (e.g. a "service") must remain constant while the object to which it is bound (e.g. the server that implements the service) varies during the computation. While many naming systems allow dynamic binding, there are some significant exceptions. CSP and languages based upon it explicitly eschew dynamic binding in favor of statically-named processes (*static binding*), as does Ada (in which names used in communication are a concatenation of task ID and entry ID). Dynamic binding is typically achieved by associating a fixed port or mailbox with successive objects, or by making use of some shared data conduit, such as a channel, which is accessible to multiple objects.

Definition: Dynamic binding (**B**):

$\exists$ events e, f, communication set Δ, and name $n :: (e \neq f) \wedge (\beta(n, e, \Delta) \neq \beta(n, f, \Delta))$ □

Definition: Static binding (**b**): $\neg$**B** □

2.3.2 Name Access

The properties *dynamic domain* and *static domain* characterize the ability of objects to learn new names during a computation. Where the dynamic domain property holds, names are typically treated as ordinary data values that may be passed from object to object during execution, and appear as arguments in communication primitives. Thus the set of objects with which a given object is able to communicate may grow as the computation progresses. This property is required by problems having an inherently dynamic structure, where new objects (and hence new names) are created during a computation. Examples include simulations of physical processes and data-parallel algorithms where the number of "worker processes" created varies according to the size of the workload. Nearly all concurrent programming languages allow names to be manipulated as data, CSP being a notable exception.

Definition: Dynamic domains (**D**):

$\exists$ event f and name $n :: (n \in f.d) \wedge \forall$ events $e :: (e.o = f.o) \wedge (e \mapsto f) \Longrightarrow (n \notin e.d)$ □

Definition: Static domains (**d**): $\neg$**D** □

2.3.3 Multiply-Bound Names

Where a name may be simultaneously bound to multiple objects (i.e. *sets* of objects), the naming system is said to have the *1–N multiplicity* property. The complementary property is *1–1 multiplicity*. 1–N multiplicity provides the abstraction of allowing communication with arbitrary groups of objects (e.g. broadcasting).

That is, communication over a multiply-bound name is a generalization of communication between exactly two objects. This property is required where the size of the set of objects with which it is to communicate is transparent to the originator of the communication. This is common in systems where objects are replicated for fault tolerance (group broadcast in ISIS[4] is an example), and group communication is the primary mode of object interaction in several systems, such as BSP[13], broadcast channels[2], and Associative Broadcast[3]. In addition, concurrent systems in which objects communicate through shared data structures (e.g. Linda[14]) have the 1–N multiplicity property as well, since multiple objects may read any value written to such a data structure.

Definition: 1–N multiplicity (**M**):

$\exists$ event e, communication set Δ, and name $n :: |\beta(n, e, \Delta)| > 1$ □

Definition: 1–1 multiplicity (**m**): $\neg$**M** □

2.3.4 Multiply-Named Objects

Some naming systems allow a given object to refer to another object by more than one name. This dual of 1–N multiplicity is called the *aliased names* property. Where every object is known to every other object by at most one name throughout the computation, the *unaliased names* property holds. Typically, multiple aliases are required where the name signifies an action to be performed by the destination object, as in an entry (e.g. Ada), or remote procedure call. Multiple aliases enable a destination object to selectively accept communications from a given source object. Such selectivity is often required to prevent polling (see section 3.3).

Definition: Aliased names (**A**):

$\exists$ communications $(e, n, f), (g, m, h) :: e.o = g.o \wedge f.o = h.o \wedge n \neq m$ □

Definition: Unaliased names (**a**): $\neg$**A** □

2.3.5 Name Sharing

A shared name is one that is accessible by more than one object, i.e., two communications over a single name may be initiated by distinct objects. Naming systems that permit such behaviors are said to have the *shared-name* property; if all names are accessible by at most one object then the naming system has the *private-name* property. Shared names are required where an object to which exactly one name is bound is to be the destination of communications from more than one source object, or where names are to be passed as data between objects. While most naming systems permit name sharing, it is sometimes desirable to prevent the nondeterminism that can arise from multiple communications over the same name. The BSM system[16], designed for real-time process control, allows each process to receive messages from at most one other process. This restriction assumes that each process has at most one name, the names are not shared, and bindings are not mutable. In section 3.2 we show that the CSP naming system also does not permit names to be shared, because it requires senders and receivers to name each other, so that each (*sender, receiver*) identifier pair constitutes a unique name.

Definition: Shared names (**S**):

$\exists$ communications $(e, n, f), (g, n, h) :: e.o \neq g.o$ □

Definition: Private names (**s**): $\neg$**S** □

2.3.6 Descriptive names

Descriptive names carry information about the states of objects to which they are bound. That is, instead of associating arbitrary identifiers with objects (e.g. process "X"), it is often useful to be able to assign meanings to identifiers (e.g. "idle") that are bound to objects exactly when they are in the corresponding

state. The naming system must provide mechanisms for establishing and removing such bindings so that they always reflect the current states of the objects. Since a descriptive name reflects an object's state, and the range of states is potentially infinite, it must further be possible for each name from an infinite set of names to be bound to the object during a computation.

Such naming systems are said to have the *descriptive reference* property, while naming systems in which names are uninterpreted identifiers are characterized by *nondescriptive reference*. Descriptive reference is required where the identities of a set of objects that are to participate in a given communication depends upon the local states of the potential participants. For example, the initiator of a deadlock-detection computation may need to send a message to "all processes waiting for resource R." Most conventional naming systems do not support descriptive reference. The designers of Concurrent C addressed the limitations in Ada that are due to its lack of descriptive reference by introducing the selective accept action, which effectively binds a descriptive name to the object (process) that executes it (see section 3.5). Both Associative Broadcast and Linda integrate descriptive reference with dynamic binding and 1–N multiplicity, allowing dynamic, descriptively-named sets of objects to be specified as the destinations of communications, although the manner in which these implement their naming system properties differ fundamentally.

Definition: Descriptive reference (**R**):

$\exists$ behavior $e_0 \ldots :: \bigcup_i e_i.b$ is countably infinite □

Definition: Nondescriptive reference (**r**): $\neg$**R** □

2.4 Hierarchy of Naming Systems

The six mutually exclusive pairs of naming system properties form the axes of a coordinate space, within which exists a complete lattice of naming system *categories*. A category is a unique point in the coordinate space, having one property from each axis.

Definition: A *coordinate* is one of the properties {**B, b, D, d, M, m, A, a, S, s, R, r**} □

Definition: (*satisfies*, $\vdash$) Let Γ be a maximal computation set and P a coordinate. Γ *satisfies* P **iff**

$\forall \gamma : \gamma \in \Gamma :: P$ is true for γ. □

Definition: (*subsumes*, $\sqsupset$, w.r.t. coordinates) Let Γ_0 be the set of all maximal computation sets, and P, Q two distinct coordinates. $\sqsupset$ is a reflexive, asymmetric, transitive relation such that:

$$P \sqsupset Q \iff (\forall(\Gamma_a, \Gamma_b \in \Gamma_0) :: (\Gamma_a \vdash Q \wedge \Gamma_b \vdash P) \Longrightarrow$$
$$(\neg(\Gamma_a \vdash P)) \wedge (\neg(\Gamma_b \vdash Q)) \wedge (\Gamma_a \cup \Gamma_b \vdash P) \wedge \neg(\Gamma_a \cup \Gamma_b \vdash Q))$$ □

From their respective definitions, it follows that

B $\sqsupset$ **b**, **D** $\sqsupset$ **d**, **M** $\sqsupset$ **m**, **A** $\sqsupset$ **a**, **S** $\sqsupset$ **s**, and **R** $\sqsupset$ **r**.

Definition: An *axis* is one of the ordered sets of coordinates

(**B**, **b**), (**D**, **d**), (**M**, **m**), (**A**, **a**), (**S**, **s**), (**R**, **r**) □

Definition: A *category* is a tuple of coordinates $R_0 \ldots R_5$, one from each axis □

Definition: Let $J = R_0 \ldots R_5$ be a category. Naming system a is a *member* of J **iff**

$\forall i : 0 \leq i \leq 5 :: \Gamma_a \vdash R_i$ □

Definition: (*subsumes*, $\sqsupset$, w.r.t. categories) Let J and K be categories. $J \sqsupset K$ **iff** each coordinate of J subsumes the corresponding coordinate of K □

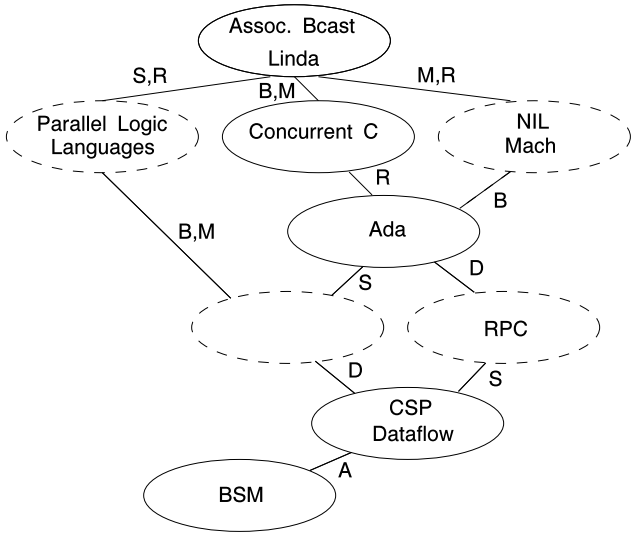

Figure 2: A portion of the hierarchy

The core of the hierarchy is the *subsumes relation*, $\sqsupset$. If two distinct coordinates P and Q are related by $P \sqsupset Q$, then a larger set of computations satisfies P than Q. This larger set contains computations that violate Q but not P; i.e. P is a weaker constraint than Q. Thus a naming system that has property P allows computations that cannot be obtained under a naming system with property Q. Where category J subsumes category K, each naming system that is a member of J allows computations that cannot be obtained under any naming system that is a member of K.

3 Examples

While every naming system belongs to one of the categories of the hierarchy, an exhaustive classification of existing naming systems is beyond the scope of this paper. This section illustrates the $\sqsupset$ relation among several representative categories with a traversal of one of the many possible paths through the hierarchy from the category **bdmasr**, also called $\perp$, to **BDMASR** ($\top$). The intervening categories contain several familiar naming systems. Figure 2 shows a portion of the hierarchy with representative naming systems for each category. Lines between categories are labeled with the properties by which the categories differ. Solid outlines indicate categories discussed in this section.

3.1 Classification Procedure

To classify a naming system as a member of a particular category, we must show for each property (coordinate) of the category that the naming system's maximal computation set satisfies the property. The first step in this procedure is to identify the elements of computations (objects, names, communications, domains, bindings), which, for example, a given concurrent system may define in terms of processes, process identifiers, communication operators, and messages.

The next section contains a detailed classification of CSP. CSP has influenced the design of many concurrent programming systems, and so its classification provides a relatively well-understood starting point for a traversal of the hierarchy of naming systems. Subsequent sections examine successively more expressive naming systems. In the interest of brevity, however, their full classification is omitted, and only the properties by which they differ are discussed in detail.

3.2 Classification of CSP

Communicating Sequential Processes[17] is a model of concurrent computation that emphasizes simplicity of communication in order to make reasoning about the interactions of processes manageable. By classifying it within the taxonomy, we show that its naming system is correspondingly simple—in fact it differs from the category containing the simplest possible naming systems only in that it has the aliased names property. By comparing the position of the CSP naming system in the hierarchy to the positions of other naming systems, we gain a more precise understanding of the relative simplicity of CSP, and of its applicability to problems that require specific naming system properties.

3.2.1 Elements

A CSP program specifies the behavior of a fixed set of *processes*. Each such process is labeled with a unique identifier. Communication in CSP consists of synchronous message-passing among exactly one sender and one receiver, and the participants for each communication are fixed by the program text. The value of a variable x local to process P is assigned to the variable y at process Q when P executes the statement $Q!c(x)$ and Q executes the statement $P?c(y)$, where c is an optional *constructor*, also called a message type. Each action of a CSP process is either a send action, a receive action, or an internal action.

- *Objects:* Each object is a distinct CSP process.
- *Names:* Let I be the set of possible CSP process identifiers, and C the set of possible constructors. A name is a (*sender-id, constructor, receiver-id*) triple from the set $I \times C \times I$.
- *Communications:* Each communication arises from the execution of a corresponding pair of send/receive actions. Let event e be the execution of the send action $j!c(x)$ by the process labeled i. Let event f be the execution of the corresponding receive action $i?c(y)$ by the process labeled j. Then the computation contains the communication (e, n, f), **iff** n is identifier triple (i, c, j).
- *Domains:* The identifiers appearing in send actions are fixed by the program text. Therefore, the domain of each event contains a name n **iff** n is the identifier triple (P, c, j), where P is the process identifier, and $j!c(x)$ is a send action of the process, for some variable x.
- *Bindings:* CSP processes can choose nondeterministically to execute one of a finite set of receive actions. Let $e_0 \ldots x$ be an initial prefix of a behavior of a process whose identifier is Q, such that event x is a receive action chosen nondeterministically from the set $\{t_0, \ldots, t_k\}$. Let $t_j = i?c(y)$ for $0 \leq j \leq k$, and some variable y. Then the name n is in the binding of x **iff** n is the identifier triple (i, c, Q).

As an example, consider the following simple CSP program:

$[\ P :: Q!c(x); R!d(y)$
$||Q :: P?c(u); R!e(v)$
$||R :: P?d(w)|Q?e(z)\ ]$

Process P has a single behavior consisting of two events, one for each of its sequential send actions. The behavior of Q is a receive action followed by a send action, while R has two possible single-event behaviors: either to receive from P or from Q. The domain of both events in the behavior of P is $\{(P, c, Q), (P, d, R)\}$. The domain of both events in the behavior of Q is $\{(Q, e, R)\}$. The binding of Q's first event is $\{(P, c, Q)\}$, while the binding of the second event is null. Finally, the domain of the single event in both possible behaviors of R is null, while in each case the binding is $\{(P, d, R), (Q, e, R)\}$.

3.2.2 Properties

The CSP naming system is a member of the category **bdmAsr**:

- *Static binding:* If a name $n = (i, c, j)$ appears in the binding of an event, e, then the identifier of $e.o$ is j (see *Bindings*, above). Since CSP process identifiers are unique, all events e where $n \in e.b$ are collocated, and the naming system has the static binding property.

- *Static domains:* For any two events e, f, if $e.o = f.o$, then $e.d = f.d$. Therefore the naming system has the static domains property.
- *1–1 multiplicity:* From *Static binding*, above, we have $e.b \cap f.b = \emptyset$ for two events e and f, if $e.o \neq f.o$. Therefore each name is bound to at most one object, and the naming system has the 1–1 multiplicity property.
- *Nondescriptive reference:* The binding of each event in a behavior is fixed by the program text that specifies the possible actions of the process. The set of possible actions is finite, and each receive action specifies a finite set of sender-ids. Therefore, in a behavior $e_0 \ldots$, $\bigcup_i e_i.b$ is finite, and nondescriptive reference holds.
- *Aliased names:* Let P be the identifier of a process with actions $Q!c(x)$ and $Q!d(y)$. Let Q be the identifier of a process with actions $P?c(v)$ and $P?d(w)$. Then there exists a computation containing two communications (e, n, f) and (g, m, h), such that e and g are events of P, f and h are events of Q, and $n = (P, c, Q)$, $m = (P, d, Q)$. Therefore the naming system allows aliased names.
- *Private names:* For any name $n = (i, c, j)$, $n \in e.d$ **iff** e is an event of the process whose identifier is i. Therefore, for any two communications (e, n, f) and (g, n, h), $e.o = g.o$, and the naming system allows only private names.

3.3 Category bdmAsr (CSP) vs. $\perp$

The CSP naming system resides in a category immediately above $\perp$, which is the greatest lower bound on the lattice, and the category containing the simplest naming systems in the hierarchy. We have found no general-purpose models of concurrency in the literature whose naming systems reside in $\perp$, although some specialized naming systems like the one described in [16] are members of that category. Nevertheless, it is instructive to study $\perp$ as a lower limit on the expressiveness of naming systems. Naming systems that reside higher in the hierarchy presumably incur some overhead in their implementation. This additional complexity is only justified if these naming systems are also more expressive. One may question, for instance, why even naming systems such as that of CSP, for which simplicity was a primary design goal, are not members of $\perp$, and whether the aliased names property in particular imparts any additional expressiveness to CSP.

3.3.1 A Resource Allocation Problem

The combination of properties that comprise $\perp$ are sufficiently restrictive to make even some simple problems impossible to solve efficiently. Consider a simple resource allocation algorithm, in which client objects request and release units of a single resource from a centralized manager object. To limit overhead within the manager, it does not queue requests that it can't immediately satisfy. Unsatisfiable requests are rejected and may subsequently be reissued by the client. Since no particular communication mechanism is assumed, it is not specified whether communication is synchronous or asynchronous. For purposes of this example, we ignore issues of fairness. The objects act as follows:

1. There are two types of communications from clients to the manager: one which signals a **request**, and one which signals a **release**. Clients issue **request** and **release** in arbitrary order, provided that # of **release**s issued $\leq$ # of **succeed** responses received. Clients must wait for either the **succeed** or the **fail** response from the manager before issuing subsequent **request**s.
2. There are two types of communications from the manager to a client, one which signals success of the preceding request, and one which signals failure of the request. Upon receiving **request**, the manager responds with **succeed** or **fail** before processing more **request**s or **release**s.
3. A **request** succeeds **iff** initial # of resource units $-$ # **succeed** issued $+$ # **release**s received > 0.
4. The manager may not refuse to accept a *release*.
5. There are no communications between clients.

An implementation of this algorithm under almost any naming system is straightforward. However, it is inefficient in that it relies upon *polling*. Polling occurs where an unbounded number of communications may pass between two objects before at least one object makes progress[12]. In a case where a client requires a request to succeed in order to make progress, the manager may issue an unbounded number of failures to a given client before issuing a success response.

The tendency to poll is often viewed as an artifact of a synchronization mechanism, or simply poor programming practice. However, we show here, as is alluded to in [12], that polling can be a consequence of a restrictive naming system.

3.3.2 Avoiding Polling

In order to prevent clients from polling the manager, we must eliminate the condition under which the manager is forced to issue a failure in response to a request. Since the problem specification states that the manager responds immediately to a **request** with **fail** when it has no free resources, eliminating the possibility of **fail** responses implies that the manager must not receive a **request** unless it is prepared to issue **succeed**. Thus it must be possible to specify the type of communication that the manager will accept at a given point in its behavior.

Theorem: A non-polling solution to the resource allocation problem requires the aliased names property. □

Proof: Consider a computation consisting of the manager and two clients, A and B, in which the manager never issues **fail**. Let $e_0 \ldots sxr \ldots$ be the behavior of the manager, such that s is an event at which the manager issues **succeed**, and the number of free resources has become 0.

1. Let a be the source event of a communication at A, and b the source event of a communication at B, such that a, b, and s are concurrent. By condition 1, each is either a **request** or **release**. By conditions 5 and 2, the clients have no information about each others state, or the state of the manager. Therefore, each communication may be either a **request** or a **release**.

2. If the communication originating at a is a **request**, and the communication originating at b is a **release**, then the communication (a, n, x) does not exist, since otherwise the manager would have to issue *fail* at r. The communication (b, m, x) exists, since the manager may not refuse a **release** (condition 4), and no other possible communications exist.

3. If instead the communication originating at a is a *release*, the communication originating at b is a **request**, then the communication (a, l, x) exists, and the communication (b, p, x) does not exist.

4. Since either a **request** or **release** may originate at a, $l \in a.d$ and $n \in a.d$. However, $n \notin x.b$ (step 2), and $l \in x.b$ (step 3). Therefore, $l \neq n$ (l and n are distinct names), and the aliased names property holds. □

Informally, polling can be avoided only if distinct names for requests and releases are known to each client, so that the manager can accept only releases when a request would result in a failure response. If a client may communicate with the manager over only a single name, then it is impossible to determine whether the manager will receive a request or a release at any given point. Since it may not refuse to accept releases, it cannot refuse to accept requests either, and may be forced to respond with **fail**. Thus a non-polling solution to the resource allocation problem does not exist under any naming system that is a member of $\perp$. However, the CSP naming system and all others that subsume it have the properties necessary for non-polling solution.

3.3.3 Expressiveness Principle

Although the manager does not queue requests, clearly an unsatisfiable request must be blocked or queued (depending on the synchronization properties of communication) at *some* level in order to avoid polling. Thus one might observe that the responsibility for queueing requests has merely been shifted to the communication

subsystem. This is true, and it is in fact desirable in most circumstances to implement the necessary queuing once in the communication subsystem, rather than repeatedly at the object level. Note, however, that it is not possible to force the communication subsystem to handle queueing of requests in the first place *unless* there exists some action by which the communication subsystem can be informed when the manager is prepared to accept a request. That action is precisely the binding of the name associated with requests to the manager object, which in turn requires the existence of distinct names for requests and releases.

This illustrates a general principle of the relationship between naming systems and communication: A more expressive naming system does not alter the communication requirements of an algorithm. However, it allows the required actions (e.g. queueing) to be specified at the interface between objects and the communication subsystem, and thereby allows the actions to be implemented at a lower level.

3.4 Category bDmASr (Ada) vs. bdmAsr (CSP)

Ada[8] is a language designed from the beginning with concurrent programming in mind, and addresses some of the perceived limitations of CSP. Ada allows objects (called tasks) to be created on the fly, and thus must allow existing tasks to learn the names of newly created ones. Ada also relaxes the constraint in CSP that requires objects to identify each other in order to carry out communication.

Communication in Ada is by means of the rendezvous[1], using the *entry call* and the *accept* action. A task initiates a communication by specifying a *task identifier* and an *entry identifier* of the called task. The task identifier denotes the task that must execute the *accept* action for the specified entry.

3.4.1 Properties of the Ada Naming System

A name as defined by the Ada naming system is a pair, (*destination-task-id, entry-id*). This differs from CSP in that the source task is not identified.

The domain of an event is determined by two types of identifiers: entry identifiers and identifiers of *declared tasks*, which are specified as constants in the program, and identifiers of *created* tasks, which are pointer values that can be assigned to local variables of the task. A name (t, e) is in the domain of an event **iff** e is an entry-id fixed by the program text of the task, and t is a declared-task-id fixed by the program text of the task, or t is a created-task-id assigned to a local variable of the task at the occurrence of the event. Thus the Ada naming system has both the dynamic domains property and the shared names property. However, Ada's support for dynamic domains is not as general as it could be, since tasks cannot learn new entry identifiers, nor can they learn the identifiers of declared tasks. If a task is to have access to these identifiers, they must be specified as constants in the program.

Ada tasks may choose nondeterministically to *accept* one of a set of entries. Thus the binding of an event that is an accept action is $\{(i, d), (i, e), \ldots\}$, where i is the identifier of the task at which the event occurs, and $d, e, \ldots$ are the entries specified in the accept action.

As does CSP, Ada has the aliased names property, because two names (t, d) and (t, e) may coexist in a domain. It has the static binding property and the 1–1 multiplicity property because a name (t, e) is bound only at events of the object whose identifier is t, and task identifiers are globally unique. Finally, nondescriptive reference holds because the binding of each event is fixed by the program text, which specifies a constant and finite set of entry identifiers.

3.4.2 Advantages of the Ada Naming System

One of the early criticisms of CSP was that its use of symmetric names (receivers identify senders) prevented its use in problems that require "anonymous" senders. For example, it is often the case that an object that implements a service (e.g. in an operating system or network routing protocol) must receive communications from an indeterminate set of clients. While it is accurate to say that this limitation of CSP is due to its symmetric names, it is important to note that this use of symmetric names is only one of the possible manifestations of the private names property. The shared names property of the Ada naming system distinguishes it not only from CSP, but from all naming systems that allow only private names, regardless of how the

[1] Ada also permits tasks to share variables, but the semantics of shared variables are not completely specified.

private names property is realized. Examples of systems in which communication among objects is specified using private names include dataflow languages[1] (streams), parallel logic languages[23] (single-assignment variables), and point-to-point channels.

The Ada naming system is further distinguished from that of CSP by the dynamic domain property. This property is required for the implementation of "reactive systems," where events in the environment (e.g. a user login) necessitate the creation of new objects at runtime. Dynamic domains are also useful in data-parallel computations, where the number of objects among which a data set is partitioned varies according to the size of the input.

3.5 Category bDmASR (Concurrent C) vs. bDmASr (Ada)

Concurrent C[11] is an extension of the C language[18] that supports distributed programming. It resembles Ada in that a computation consists of a dynamic set of processes that communicate using the rendezvous mechanism[2] through *transactions* (equivalent to entries in Ada). A process initiates a communication by executing a *transaction call*. The transaction call is completed when the destination process executes the *accept* action for the given transaction.

The designers of Concurrent C have observed that Ada would benefit from the ability to accept outstanding entry calls conditionally, based on the values of the parameters[12]. In Concurrent C, an *accept* action may be augmented with the qualifier *suchthat (expr)*, causing an outstanding transaction call to be accepted when *expr* is true for the parameters of the transaction call. This generalization of *accept* is the principle advantage of the Concurrent C naming system over that of Ada, as it implements the descriptive reference property.

3.5.1 Properties of the Concurrent C Naming System

A transaction call specifies two static identifiers that indicate the called process and a transaction within that process respectively. In this respect Concurrent C names resemble those defined by the Ada naming system. However, because of the *suchthat* qualifier, Concurrent C names have a third component, consisting of the values of the parameters of the transaction call, i.e., a name is a triple, (*process-id, transaction-id, parameter-list*). Let event e be the execution of the action *accept t(...) suchthat (expr)* by the process whose identifier is p. Then a name n is in the binding of event e **iff** $n = (p, t, r)$, where r is a parameter list for which *expr* evaluates to true. Because *expr* may refer to the local variables of the process, the set of names that is bound to an object at an event may depend upon the local state of the object. This in turn implies that the set of names that may be bound to an object over its entire behavior is potentially infinite, and thereby the descriptive reference property holds.

The remaining properties of the Concurrent C naming system are identical to those of the Ada naming system. The way in which Concurrent C implements dynamic domains is more general than Ada's implementation because both process identifiers and transaction identifiers are values that can be passed from one process to another. While an object may have any number of names bound to it, each name, when bound, is always bound to the same object, because process identifiers are unique and static. In section 3.6 we discuss naming systems that combine descriptive reference with dynamic binding and 1–N multiplicity.

3.5.2 A Preemptive Scheduling Problem

The descriptive reference property is required when the set of communications an object is prepared to receive varies according to the state of the object, and the state space of the object is potentially infinite. Consider the problem of managing prioritized access to a single resource (perhaps a specialized processor or device controller). As in the resource allocation problem described in section 3.3.1, client objects request and release the resource from a manager object, and the manager does not queue requests. Associated with each request is an integer priority that is used to determine when one client may preempt another in accessing the resource. The objects act as follows:

[2] More recent versions of Concurrent C allow asynchronous communication as well.

1. Clients may initiate either a (**request**, p) communication, or a **release** communication to the manager, where p is an integer priority. Clients must wait for either the **succeed** or the **fail** response from the manager before issuing subsequent requests. Upon receiving **succeed**, a client issues **release** before issuing another request, **iff** it has not received **preempt** subsequent to the most recent **succeed**.

2. There are three types of communications from the manager to a client, one that signals success of the preceding request, one that signals failure of the request, and one that preempts the most recent **succeed** . Upon receiving (**request**, p), the manager responds with **succeed** or **fail** before processing more **request**s or **release**s.

3. A (**request**, p) succeeds **iff** the resource is currently free, or if $p > q$ where q is the priority of the most recent successful request. In the latter case, the manager issues **preempt** to the client to which it issued the most recent **succeed**, before responding with **succeed** to the current request.

4. The manager may not refuse to accept a release nor a satisfiable request.

5. There are no communications between clients.

As in the resource allocation algorithm, a client may make an unbounded number of unsuccessful attempts to obtain the resource before succeeding. However, in this case it is not sufficient for the naming system to allow the manager to distinguish only between requests and releases, it must also distinguish satisfiable from unsatisfiable requests.

3.5.3 Avoiding Polling, Revisited

Again, eliminating polling requires that the manager not be forced to issue **fail** in response to a request. The manager can satisfy a request when the priority of the request exceeds the priority of the most recent successful request. Thus the name over which the request communication is initiated must encode the priority of the request, and that name must be bound to the manager object exactly when it is able to satisfy the request.

Theorem: A non-polling solution to the preemptive scheduling problem requires the descriptive reference property. □

Proof: Assume that the set of names that may be bound to the manager object contains at most m names. Consider a computation consisting of the manager and k ($k > m$) clients, in which the manager never issues **fail**.

1. By conditions 1, 2 and 5 the first event at each client may be the initiation of a request communication, all these events may be concurrent, and the priorities of all requests may be distinct.

2. Since k requests are initiated concurrently and at most $k-1$ names are ever bound to the manager, there exist at least two communications that are initiated over a common name. Let a be the source event of the first such communication, (**request**, p). Let b be the source event of the second, (**request**, q). Let n be the common name.

3. Let $e_0 \ldots sxr \ldots$ be the behavior of the manager, such that s is an event at which the manager issues **succeed**, in response to a request whose priority is w, such that $p < w < q$. Then (b, n, x) is a possible communication, since $q > w$ and the manager may not refuse satisfiable requests (condition 4). Therefore, n is bound to the manager at event x. Since a and b are concurrent, and n is bound at x, (a, n, x) is also a possible communication. By condition 3, the manager must issue **fail** at event r in this case. Contradiction.

4. By step 3, there are at least k names that are bound to the manager. As $k \to \infty$, the number of names that must be bound to the manager grows without bound, and the descriptive reference property holds. □

Note that it is not necessary for there to be k clients in order to generate a set of requests that require k names to be bound to the manager, nor need the requests be originated concurrently. These assumptions merely shorten the proof.

The above example further illustrates the principle mentioned in section 3.3.3: A descriptive name makes visible at the interface to the communication subsystem some portion of the state of the object to which it is bound. This allows decisions about blocking or queueing communications to be made at a lower level, even when these decisions are a function of the local states of objects.

A mechanism that implements descriptive reference is useful not only when the set of names that are bound to an object during a computation is potentially infinite, but also when it is finite, yet large. It is much more clear, concise, and efficient to write *accept t(x) suchthat (*$0 < x < 100$*)* in Concurrent C, for instance, than it is to enumerate a family[12] of 99 entries in Ada, one for each possible value of x.

3.6 Category $\top$ (Linda and Associative Broadcast)

There exist naming systems that are members of category $\top$, which subsumes all the categories of the hierarchy. Despite the existence of such naming systems, it is often preferable to use a less expressive naming system because it can be implemented more efficiently. We have already seen that the aliased names and descriptive reference properties may require the communication subsystem to block or queue communications, and this may not be desirable in all cases. Similar tradeoffs exist for other naming system properties. For example, one of the naming systems in $\top$ relies upon a logically shared data structure, which can be difficult to realize on distributed architectures where processors do not share physical memory.

Below, we contrast the naming systems of Linda[14] and Associative Broadcast[3], both of which are members of $\top$, in order to show that naming systems whose properties are identical may nevertheless have markedly different implementations. Unlike Linda processes, Associative Broadcast objects communicate using asynchronous message passing, subject to only a weak ordering constraint. This shows that the set of properties that constitutes the category $\top$ can be implemented in the absence of logically shared data structures.

3.6.1 Linda

A Linda program specifies a set of processes that communicate by inserting, reading, and removing *tuples* of data values in a logically shared data structure called the *tuple space* (TS). The tuples are stored unordered in TS. Processes manipulate TS using three operations: *in(t)*, which removes the tuple t from TS, *out(t)*, which places t in TS, and *read(t)*, which reads t but does not remove it. Each data value in a tuple has a *type*. t may be a *pattern* that contains a formal parameter in some of its fields. The tuple returned by the *read(t)* or *in(t)* operation is one in which data values match the types of formal parameters. For example, the operation *in("size",* **var** *x)*, where x is of type **integer** might match the tuple *("size", 20)*. If more than one tuple in TS matches, one is chosen arbitrarily. If none match, the operation blocks until some process inserts a matching tuple in TS.

3.6.2 Associative Broadcast

Associative Broadcast uses message broadcasting as the fundamental mode of communication among objects. Communication is 1–N: A *sender* specifies a *target set* of objects that are to receive a message. 1–1 communication is merely a special case broadcast in which the size of the target set is 1. Associated with each object is a *profile*, which consists of a set of *attributes*. An attribute is either a *symbol* (a simple attribute) or a *(symbol, value)* pair (a compound attribute). The symbols may be treated as data values and passed from one object to another in a message. An attribute may represent the *type* of an object (operations it supports) or an abstraction of the object's state. For example, in a distributed hashing algorithm, the profile {`INSERT, FIND, (MIN, 0), (MAX, 10)`} might indicate that the object supports the `INSERT` and `FIND` operations on key values in the range 0–10.

An object consists of a set of local data structures, and a set of *operations*. Receipt of a message causes the operation indicated in the message to be executed. The operation may modify local data, alter the profile of the object, and broadcast messages. A *message* is a tuple, *([selector], arg, ...)*, where *[selec-*

tor] is a propositional formula over attributes that specifies the target set of the message. For example, *broadcast(*[`INSERT` $\wedge$ `MIN` < 3]*, x)* might send the value x to all object whose `MIN` attribute is less than 3, and cause them to perform the `INSERT` operation using the value x. After a message is broadcast, the selector contained in the message is matched against the profile of each object. If the selector is *true* for an object's profile, the message is received by the object. The *broadcast* primitive is not atomic. The profile of an object may change while the message is in transit. In this case, the object is not guaranteed to receive the message. Two messages are guaranteed to arrive in the order they were sent only if they were sent by the same object.

3.6.3 Elements

- *Communications:* In Linda, the execution of an *out* action may give rise to any number of communications, including 0. The number of communications is defined by the number of subsequent *read* and *in* actions on the tuple placed in TS by *out*. Thus a communication (e, n, f) occurs where event e is the execution of an *out* action, and f is either a corresponding *read* or *in*. In an Associative Broadcast communication, (e, n, f), e is the execution of the *broadcast* action, and f is the receipt of the message.

- *Names:* Communication in Linda is through the TS, which is in effect a content-addressable buffer. Thus the name over which a communication (the "transmission" of a tuple) occurs is the tuple itself, and the set of names defined by the Linda naming system is exactly the set of tuples that may appear in TS. Names and data are one and the same in Linda. Associative Broadcast names are simply the selectors appearing in messages.

- *Bindings:* Let event e be the execution of the action *in(t)* or *read(t)* by a Linda process, where t is a pattern. Then $e.b$ is exactly the set of tuples that match t. In Associative Broadcast, the binding of an event is determined by the composition of the profile of the object at which the event occurs. Let event f be the receipt of a message, such that the profile of object $f.o$ at the occurrence of f is P. Then $n \in f.b$ **iff** P satisfies n. That is, a message may be received at f only if its selector is satisfied by P.

- *Domains:* Since names are data tuples in Linda, a name is known to a Linda object exactly when it knows all the elements of the name. Let $n = (d_0, \ldots, d_k)$. Then $n \in e.d$ **iff** for all i, d_i is either a constant fixed by the program text of the object at which e occurs, or a value of a local variable of the object. Similarly, an object in Associative Broadcast knows a name (selector) when it knows all the symbols appearing in the formula that constitutes the name.

3.6.4 Properties

$\top$ differs significantly from the categories discussed earlier in that it combines the descriptive reference property with 1–N multiplicity and dynamic binding. The dynamic domains, shared names, and aliased names properties of Linda and Associative Broadcast are similar to those of naming systems examined earlier, so we omit discussion of these properties here.

- *Dynamic Binding:* Any number of copies of a given Linda tuple may appear in TS, either sequentially or simultaneously. Thus it is possible for two communications (e, n, f) and (g, n, h) to exist, where e and g are occurrences of distinct *out* actions of n, and f and h are occurrences of distinct *in* or *read* actions. Since f and h may occur at distinct objects, the Linda naming system has the dynamic binding property. Dynamic binding also holds for Associative broadcast. For communications (e, n, f) and (g, n, h), n may be satisfied by profile P at f, and by profile Q at h, where P and Q are profiles of distinct objects.

- *1–N multiplicity:* Let e be the execution of *out(t)*, and f and g the executions of *read(t)* by two distinct objects. Then the communications (e, t, f) and (e, t, g) satisfy the definition of the 1–N multiplicity property. Similarly, a message in Associative Broadcast may be received by at least two distinct objects.

- *Descriptive Reference:* For an action of a Linda process *in(t)*, where t is a pattern containing formal parameters whose range is infinite (e.g. formal parameters of type **integer**), there exists an infinite

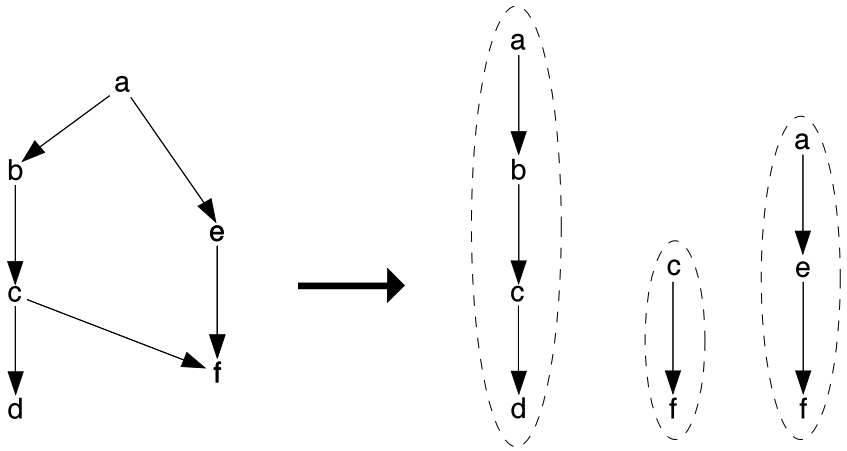

Figure 3: Partitioning of edges among objects

set of tuples that match t. Therefore, the set of names bound at the occurrence of such an action is infinite. In Associative Broadcast, the set of potential profiles of an object is infinite. For each such profile, there exists a distinct set of selectors that are satisfied by the profile. Therefore, in an infinite behavior, an infinite set of names may be bound to the object.

3.6.5 Communication Abstraction

Communication abstraction extends to communication the same principles of abstraction that are commonly applied to data types, synchronization, and control sequencing[10]. A naming system with the above three properties enables communications to be specified at a level that is significantly more abstract than do naming systems in which these properties are absent. *Description* is the form of abstraction provided by descriptive reference, which incorporates the state of an object into the names that are bound to it. When combined with the *grouping* abstraction of 1–N multiplicity, arbitrary *sets* of objects can be descriptively named. Dynamic binding adds *indirection*, allowing the membership of a descriptively named group to vary. These forms of abstraction are, respectively, generalizations the *structured naming*, *space uncoupling*, and *continuation passing* properties of Linda mentioned in [14].

3.6.6 A Distributed Cycle-Detection Problem

The manager object in section 3.5.2 represents a special case of descriptively-named groups in which the group is always of size 1 or 0, and there is only one possible member of the group. However, there exist problems that naturally engender groups of objects in which each object is a potential member of *every* group, and the membership of each group is dynamic and determined by input from the environment. In these cases group membership is determined by the *global* state of the execution. Because of this, such problems are difficult to solve using a distributed (no global data structures), and symmetric (objects execute identical programs, no object is distinguished as a coordinator) algorithm in the absence of descriptive reference, dynamic binding, and 1–N multiplicity.

Consider the problem of detecting cycles in a dynamic graph that is stored by partitioning its edge set among a set of objects. Each edge is an ordered pair of nodes, (u, v), and is stored at exactly one object (figure 3). Such graphs arise, for instance, from the precedence relation among transactions in a replicated, distributed database. It is necessary to check these graphs for cycles to determine whether transactions executed during a network partition failure are serializable[7]. The following solution uses "probe" communications among objects to traverse the graph in place. The graph is not frozen while cycle detection occurs—edges may be added to the graph concurrently with ongoing executions of the algorithm.

1. There are two types of communications: (**add**, (u, v)) is a communication from the environment to

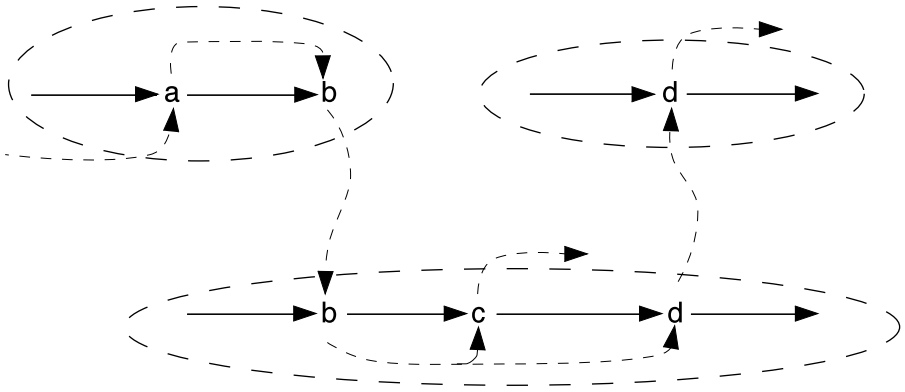

Figure 4: Probe propagation

an object, causing it to add the edge (u, v) to the set of edges it maintains. Receiving (**probe**, n, p) causes the object to propagate the probe p along all edges reachable from node n.

2. Upon receiving (**add**, (u, v)), the object propagates a new, unique probe through the graph (figure 4). Beginning at node v, the object labels the nodes of a traversed edge with p, and issues (**probe**, n, p) for each node, n, encountered in the traversal. If a node is labeled twice, a cycle exists.

3. Upon receiving (**probe**, n, p), p is propagated as above, starting at node n.

4. An object receives (**probe**, n, p) only if n is a node of one of the edges it maintains.

Because there is no provision for objects to communicate their edges to each other, nor for any one object to coordinate an instance of the algorithm, an implementation must be both symmetric and distributed. Furthermore, condition 4 requires *minimal communication*: an object may receive a probe only if it actually maintains an edge along which the probe is to be propagated. This disallows a brute-force implementation in which every object receives every probe.

The algorithm requires the descriptive reference property because of the requirement for minimal communication: Each object maintains a subset of an infinite set of possible nodes. 1–N multiplicity is required because a given node may belong to any number of edges, and a probe that traverses an edge incident on node n must reach all objects where n appears. Finally, dynamic binding is required because edges may be added while probes are traversing the graph: The set of objects at which node n appears may differ for each communication (**probe**, n, p).

We omit a formal proof in favor of giving an example of how the algorithm is expressed using Associative Broadcast. Each object has the following identical specification:

initially
    profile = {(NODES, ∅), ADD, PROBE}

/* Upon receiving (**add**, (u, v)) */
add(u, v) {
    NODES := NODES ∪ $\{u\}$;
    i := **unique_id**();
    probe(v, i);
}

```
/* Upon receiving (probe, w, p) */
probe(w,p) {
        foreach local node n reachable from w {
                if n labeled with p
                        declare cycle;
                else {
                        label n with p;
                        broadcast([PROBE ∧ (n ∈ NODES)], n, p);
                }
        }
}
```

Here again the principle of expressiveness applies. The descriptive reference, dynamic binding, and 1–N multiplicity properties of Associative Broadcast allow us to specify an algorithm that leaves it up to the naming system to ensure that a probe message is delivered to the correct set of recipients. In the absence of these properties, it would be necessary for the objects to explicitly implement a protocol that does the work done here by the naming system. This would require the abrogation of one or more of the stated assumptions, either by removing the minimal communication requirement and using a brute-force approach, or by breaking the symmetry of the algorithm.

4 Conclusions

In designing a naming system, one is faced with tradeoffs between expressiveness and complexity that result from the diversity of ways in which names and bindings may be established and manipulated to create patterns of communication among objects. Such decisions should be made with an awareness of both the range of possible choices, and the impact upon the structure of programs and computations of each choice. The taxonomy of naming systems described here is intended to allow the alternatives to be examined in a systematic way. We have sought to identify a set of fundamental properties that clarify the role of names and bindings in concurrent programming, and that capture the practical distinctions among naming systems that are often implicit in informal comparisons of concurrent programming systems. The arrangement of the properties into orthogonal sets of mutually exclusive elements imparts simplicity and regularity to the structure of the taxonomy, and ensures that every naming system belongs to one of the categories of the hierarchy.

The taxonomy is a tool for comparing, selecting, and evaluating naming systems: The position of a naming system within the hierarchy determines the set of naming systems whose properties it subsumes. For an algorithm whose required naming system properties are known, the taxonomy allows the selection of a naming system having the necessary properties. Finally, one may use some appropriate measure to determine if a given naming system is the best possible implementation of its properties for a given target architecture (cf. Linda vs. Associative Broadcast). We have seen that more expressive naming systems allow such operations as queueing and filtering of communications to be implemented below the object level. This eliminates error-prone duplication of effort, encourages more efficient implementation of the operations, and offers the possibility of exploiting or developing specialized features of the underlying architecture.

The composition of the taxonomy is driven to some extent by observation: We have sought to define properties that capture significant distinctions among existing naming systems, and that affect the ability of a naming system to express solutions to important classes of problems. It is possible that the suitability of the taxonomy to a particular purpose may be improved by the adding more axes to the coordinate space that contains the lattice of categories, or subdividing the coordinates of an axis to improve the "resolution" of the hierarchy. Any such changes would not invalidate the results obtained using the current structure. Nevertheless, any modifications should be approached with caution, as they might complicate the process of classifying naming systems. In fact, the taxonomy has been simplified significantly over the course of its development.

We have explicitly avoided discussing resolution, as the resolution mechanism of a naming system is

orthogonal to its binding and name-access mechanisms. Nevertheless, properties of resolution mechanisms may in some cases also influence patterns of communication, and thus the structure of computations. To account for these effects, the taxonomy could be augmented with properties of resolution that are orthogonal to the existing sets of properties.

References

[1] W. Ackerman, Dataflow languages, *IEEE Computer*, vol. 15, no. 2, Feb. 1982

[2] M. Ahamad and A. Bernstein, An application of name based addressing to low level distributed algorithms, *IEEE Trans. Software Eng.*, vol. SE-11, no. 1, Jan. 1985

[3] B. Bayerdorffer, Associative broadcast: a comprehensive approach to naming in concurrent programming, *PhD. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin*, (expected) May 1993

[4] K. Birman and T. Joseph, Reliable communication in the presence of failures, *ACM Trans. Comp. Systems*, vol. 5, no. 1, Feb. 1987

[5] K. Chandy and L. Lamport, Distributed snapshots: Determining global states of distributed systems, *ACM Trans. on Comp. Sys.*, vol. 3, no. 1, Feb. 1985

[6] D. Comer and L. Peterson, Names and name resolution, In *Concurrency Control and Reliability in Distributed Systems*, B. Bhargava, ed., Van Nostrand Reinhold, 1987

[7] S. Davidson, Optimism and consistency in partitioned database systems, *ACM Trans. on Database Systems*, vol. 9, no. 3, Sept. 1984

[8] Department of Defense, Reference manual for the Ada programming language, *ANSI/MIL-STD-1815A*, DoD, Washington, D.C., Jan. 1983

[9] J. Dutton, Naming and object reference: towards a rigorous model, *PhD Dissertation Proposal, UT Austin CS Dept.*, July 1987

[10] N. Francez and B. Hailpern, Script: a communication abstraction mechanism, *Proc. ACM Symp. Principles Distributed Computing*, Aug. 1983

[11] N. Gehani and W. Roome, Concurrent C, *Software Pract. Exper.*, vol. 16, no. 9, Sep. 1986

[12] N. Gehani and T. Cargill, Concurrent programming in the Ada language: the polling bias, *Software Pract. Exper.*, vol. 14, no. 5, May 1984

[13] N. Gehani, Broadcasting sequential processes (BSP), *IEEE Trans. Software Eng.*, vol. SE-10, no. 4, July 1984

[14] D. Gelernter, Generative communication in linda, *ACM Trans. Prog. Languages and Systems*, vol. 7, no. 1, Jan. 1985

[15] J. Gischer, Partial orders and the axiomatic theory of shuffle, *PhD. Thesis, Computer Science Dept., Stanford Univ.*, Dec. 1984

[16] R. Gueth, J. Kriz, and S. Zueger, Broadcasting source-addressed messages, *Proc. 5th Int. Conf. on Distributed Computing Syst.*, May 1985

[17] C. Hoare, Communicating sequential processes, *CACM*, vol. 21, pp. 666–677, Aug. 1978

[18] B. Kernighan and D. Ritchie, The C programming language, Prentice-Hall, Englewood Cliffs, N.J., 1978

[19] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *CACM*, vol. 21, no. 7, July 1978

[20] D. Oppen and Y. Dalal, The clearinghouse: a decentralized agent for locating named objects in a distributed environment, *XEROX Office Products Division Research Report*, Oct. 1981

[21] V. Pratt, Modeling concurrency with partial orders, *Int. Journal of Parallel Programming*, vol. 15, no. 1, Feb. 1986

[22] J. Saltzer, Naming and binding of objects, In *Operating Systems: An Advanced Course*, R. Bayer, et al., eds., Springer-Verlag, 1979

[23] A Takeuchi and K. Furukawa, Parallel logic programming languages, *Proc. 3rd Int. Conf. Logic Prog.*, Springer-Verlag, Berlin, pp. 242–254, 1986