ASSOCIATIVE BROADCAST AND THE COMMUNICATION SEMANTICS OF NAMING IN CONCURRENT SYSTEMS

APPROVED BY

DISSERTATION COMMITTEE:

To my family

ASSOCIATIVE BROADCAST AND THE COMMUNICATION SEMANTICS OF NAMING IN CONCURRENT SYSTEMS

by

BRYAN CARL BAYERDORFFER, B.S., M.S.C.S

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1993

Acknowledgments

I am greatly indebted to all those whose patience, encouragement, and wisdom sustained me in producing this work. All the members of my dissertation committee gave generously of their time and talents. My supervisors, Professor J.C. Browne and Dr. Richard Brice, provided invaluable technical and philosophical advice, leavened with equally appreciated enthusiasm. Professor Browne's ability to quickly get to the core of a problem both aided and inspired me. I am grateful to Dr. Brice for spending many an hour helping me to turn the germ of an idea into a viable research topic, and for many interesting discussions and valuable comments since then. I thank Professor Mohamed Gouda for teaching me early on the importance of distinguishing the good not only from the bad, but also from the ugly. The thoughtful comments of professors Jeffrey Brumfield and Simon Lam helped me immeasurably in learning to communicate my ideas to others.

Thanks are due as well to friends and fellow graduate students Jim Anderson, Anish Arora, Ken Calvert, Al Carruth, Sarah Chodrow, Ernie Cohen, Ulf Hermjakob, Markus Kaltenbach, Edgar Knapp, Jacob Kornerup, Dave Naumann, Peter Newton, J.R. Rao, and Ambuj Singh. They made life inside and outside the university both interesting and enjoyable.

Special thanks must go to Nancy Macmahon and Gloria Ramirez for their help in overcoming the innumerable microchallenges of academic life, which without the benefit of their experience and skills would surely have added up to an insurmountable barrier.

Finally, my heartfelt gratitude goes to my family, whose love, emotional support, and confident belief in my eventual success were irreplaceable in seeing me through.

Bryan Carl Bayerdorffer

The University of Texas at Austin

December, 1993

ASSOCIATIVE BROADCAST AND THE COMMUNICATION SEMANTICS OF NAMING IN CONCURRENT SYSTEMS

Publication No.______________

Bryan Carl Bayerdorffer, Ph.D.
The University of Texas at Austin, 1993

Supervisors: James C. Browne and Richard S. Brice

Much of the complexity of concurrent program design lies in the specification of *patterns of communication*: flows of information among objects (e.g. processes) expressed as functions of the global computation state. Underlying the communication mechanisms of every concurrent system is a *naming system*, which is used to specify the objects participating in each communication. We call those characteristics of naming systems that determine the patterns of communication that can be specified the *communication semantics* of naming systems. A more expressive naming system allows communication to be specified at a higher level of abstraction. The available communication abstractions are significant in the choice of an algorithm, in which specific patterns of communication are manifest, to solve a particular problem. Thus the choice or design of a naming system often significantly affects the design of concurrent

programs. Yet naming systems have not been widely studied as independent components of concurrent systems. This has led to empirical design and inappropriate choices of naming systems, which hinder the specification of complex patterns of communication and constrain algorithm design, yielding awkward programs and inefficient executions. To simplify the use of naming systems in specifying communication, and to make more precise our understanding of their communication semantics, we adopt a twofold approach: The development of a formal *taxonomy of naming systems*, and the design of a specific naming system and communication primitive, called Associative Broadcast, that enable straightforward specification of complex communication patterns. The taxonomy defines a set of orthogonal properties that characterize the ability of naming systems to express fundamental patterns of communication. Naming systems are classified and ranked in a partially ordered hierarchy according to their properties. The hierarchy allows systematic comparison of naming systems, and selection of naming systems for the solution of problems requiring specific properties. Associative Broadcast resides at the top of the hierarchy, and allows the specification of a dynamic, descriptively named *target set* of objects as the destination of a message by defining names to be propositional expressions over sets of object attributes, and by using delayed resolution. We apply Associative Broadcast to obtain algorithms with desirable symmetry, robustness, concurrency, and efficiency properties for problems in the areas of distributed constraint reduction, database consistency in the presence of network partition failures, and virtual time synchronization.

Table of Contents

List of Figures

Chapter 1

Introduction

The process of designing a concurrent program consists of specifying a set of active *objects* (e.g. processes) that cooperate to perform a given task. The greatest source of complexity in concurrent programming lies in the specification of communication. The programmer must introduce a structure upon the order of events in a computation that achieves the desired behavior, and this structure is obtained through communication among objects. Within a given computation, we may observe *patterns of communication*, in which information flows from a source object to a set of destination objects whose membership is some function of the the global state of the computation. Because the necessary patterns of communication vary significantly with the requirements of the problem being solved, the choice and the design of mechanisms for specifying communication is a task to be approached with care, and with the goal of subjecting the programmer to as few arbitrary restrictions as possible.

Underlying the communication mechanisms of every concurrent system is a *naming system*, which provides for the *binding* of names to objects and for the *resolution* of name references. The naming system is of central importance to communication, because specifying a particular communication is essentially a matter of indicating its participants. It follows therefore, that more expressive mechanisms for indicating (i.e. *naming*) sets of objects allow more complex and varied patterns of communication to be specified, and lead to simpler specifications.

Our work is aimed at simplifying the task of specifying complex patterns of communication, and thereby the programming of concurrent algorithms for which such patterns are required. This requires a more precise and fundamental understanding of the role of naming systems in the specification of communication than presently exists, since naming systems have not been widely studied as independent components of concurrent systems. We call those characteristics of naming systems that determine the patterns of communication that can be specified the *communication semantics of naming systems*. Specifically, the roles of binding and knowledge (modification of the set of names known to an object) in communication are not well understood. This has led to empirical design and inappropriate choices of naming systems that hinder the specification of complex patterns of communication and constrain the thinking of programmers, and thus yield awkward programs and inefficient executions.

To simplify the use of naming systems in specifying communication, and to make more precise our understanding of their communication semantics, we adopt a twofold approach: The development of a formal framework for the analysis of naming systems in general on the one hand, and the design of a specific naming system that enables straightforward specification of complex communication patterns on the other.

1.1 Syntax vs. Semantics

A naming system contains two complementary mechanisms: that which establishes bindings of names to objects, and that which subsequently make use of these bindings to resolve name references. The behavior of a resolution mechanism is completely specified by the requirement that it return a set of objects, given a name. Its distinguishing properties have entirely to do with its implementation, e.g. its execution efficiency. In contrast to this, there are a variety of types of bindings, which are used to create distinct patterns of communication.

Comer and Peterson have observed that name resolution is a strictly syntactic process[19], 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.*

All the naming systems we will consider are *sound*. A naming system is sound **iff** any binding that can be established by the binding mechanism can be resolved by the resolution mechanism. Naming systems are typically unsound only when they allow partitioned name spaces, in which names may be bound not only to objects but also to sets of names, called *contexts*. Partitioned name spaces can introduce ambiguous bindings and cycles.

Most naming systems enforce soundness by design, and for those that do not, we are effectively assuming that the programmer refrains from writing programs that generate unresolvable bindings, which by almost any definition are incorrect programs. This assumption guarantees that resolution always succeeds, and thereby ensures that resolution and binding are strictly orthogonal operations. Since resolution is comparatively well understood, we are concerned exclusively with the communication semantics of naming systems, which are determined by their binding mechanisms. The orthogonality of resolution and binding allows us to place resolution and partitioned name spaces outside the scope of our consideration. We discuss the extension of our results to include resolution in chapter 9.

1.2 Models and Systems

In subsequent sections we discuss both models and systems, in particular naming systems, naming models, and concurrent systems. We adopt the definition that a *system* is a particular instance of a *model*, and conversely that any given model may be instantiated as any one of several distinct systems. The purpose of maintaining this distinction is that models are generally denotational, abstract definitions of a set of properties in which we are interested, while systems are concrete, operationally specified implementations of those properties. We will often discuss the properties of a particular naming model, and how these properties affect its applicability to a particular problem independent of how those properties are implemented.

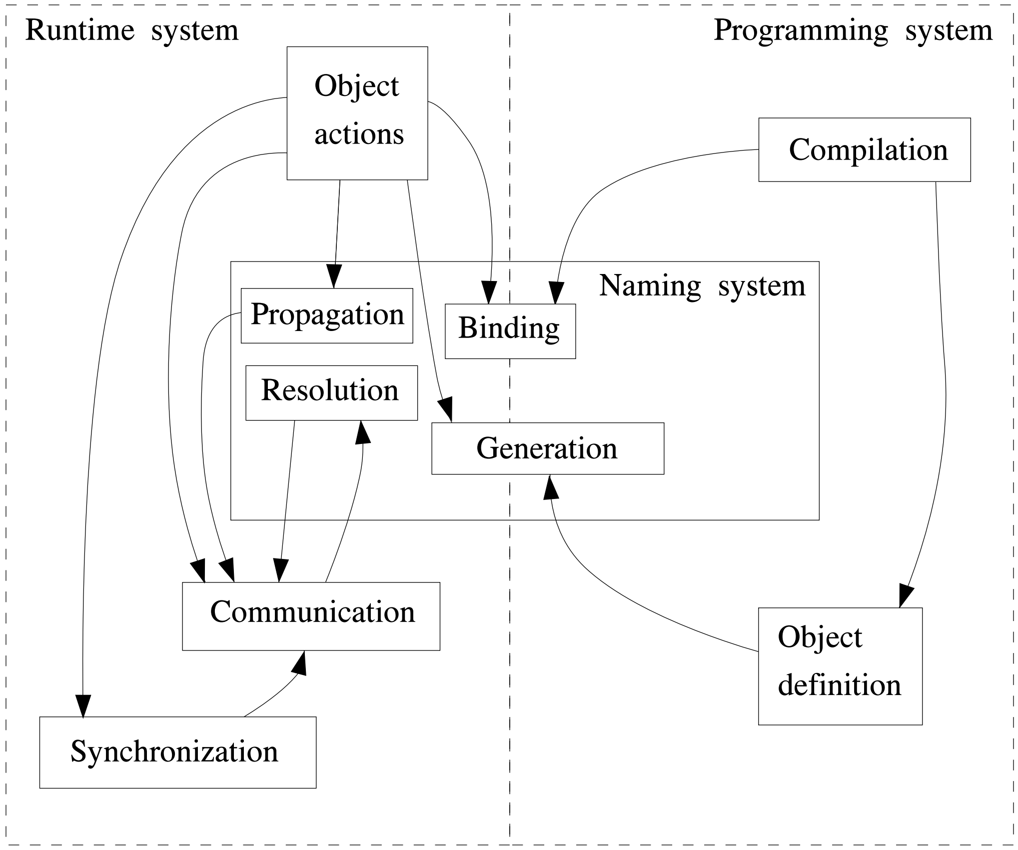

Figure 1.1: Components of a concurrent system

A naming system is one component of a *concurrent system* within which concurrent programs are defined, compiled, and executed. Figure 1.1 illustrates the possible dependencies among the operations in a concurrent system. The binding and name-creation mechanisms of the naming system may be incorporated into both the programming and runtime systems because it is possible for names to be created and bound during program definition, compilation, and execution. During execution, objects may cause the naming system to create names and bind them to objects. Conversely, the communication system calls upon the resolution mechanism to resolve a given name against the set of existing bindings, that is, to identify the object(s) to which the name is bound so that a communication may be carried out.

A further distinction between models and systems is that we will define the properties of a naming model in a manner that is independent of any concurrent system in which an instance of that model might be embedded. That is, a naming model abstracts away the details of concurrent systems that do not have to do with naming. On the other hand, when discussing specific naming systems it is necessary to refer to details of the overall concurrent system when the naming system is defined implicitly and operationally.

1.3 Role of Naming Systems in Communication

Having argued briefly that naming systems are important yet overlooked components of concurrent systems, let us examine their role and significance in more detail.

We will view the *events* of a computation as being instances either of local actions of objects, or as "endpoints" of communications among objects. 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*. The set of names known to an object at a particular point in its execution (an *event*) is the *domain* of the

Function	*Specification*	*Implementation*
Knowledge	Naming convention Propagation rules	Generation mechanism Propagation mechanism
Binding	Binding rules	Binding mechanism
Resolution	Context	Resolution mechanism

Table 1.1: Elements of a naming system

event, and the set of names bound to an object is the *binding* of the event. A naming system thus consists of three functional elements, for each of which there is a specification and an implementation:

- *Knowledge:* The set of names that can be known to an object is specified by a *naming convention*, which determines how names are generated, and a set of *propagation rules*, which determine how names may be transmitted from one object to another. These are implemented by a *name generation mechanism*, and *name propagation mechanism*, respectively.
- *Binding:* A set of *binding rules* determines how names may be bound. The binding rules are implemented by a *binding mechanism*.
- *Resolution:* The object(s) to which a given name will be resolved at a given point in an execution is determined by the set of bindings that exist at that point, called the *context*. A name is resolved against a context by a *resolution mechanism*.

These elements are summarized in table 1.1.

The concept of a name encompasses such familiar items as process identifiers and task 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[28]. 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[36]), 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.

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.

For a given problem there may exist a straightforward algorithm that solves the problem under one naming system, while a solution to the problem may be awkward to express or inefficient to implement under another naming system. Constraints on the form of the implementation (e.g. symmetry, decentralization, minimal communication, absence of polling) may preclude the existence of some families of solutions altogether. We recognized that the patterns of communication required by a range of algorithms reveal a small set of fundamental patterns that occur repeatedly. We define a set of orthogonal properties of naming systems, each of which determines the ability of a naming system to express one of these fundamental 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).

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

- *Blocking and queueing of communications:* In initiating a communication, reference to a name that is not currently 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 distinct 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 Analysis of Naming Systems

A naming system is typically implicitly defined within a specific concurrent system, and the definitions of its communication properties are therefore embedded within the context of that system. The influence of naming systems on the structure of concurrent computations is commonly examined in an ad hoc manner. It has been shown, for example, that limitations of the Ada[24] rendezvous mechanism (which relies on a comparatively restrictive naming system) lead to programs that poll[29], and that the requirement that communicating processes in CSP[36] 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 the naming system is the source of the problem. Therefore the results do not extend in an obvious way to other concurrent systems.

It would be far easier to assess the capabilities of a naming system if its communication properties could be stated explicitly, and modeled at an appropriate level of abstraction, independent of a particular concurrent system. One is lead to search for a unifying framework within which tradeoffs and design choices can be systematically evaluated. Such a theory of the communication semantics of naming systems should have the following goals:

- *To delimit the capabilities of naming systems, and to enable rigorous comparisons between them.* The theory should allow us to determine whether a solution to a given problem exists under a particular naming system, if both the properties of the naming system and the properties required by the problem are known. Given a solution to a particular problem under one naming system, the theory should also allow us to determine whether a similar solution exists under another, i.e. the theory should define an expressiveness relation among naming systems.

- *To facilitate the selection or design of the most suitable naming system for a particular problem domain.* The advantages of being able to specify precisely the desired properties of a naming system are significant, amounting at the very least to less duplication of effort when suitable existing candidates can be recognized. Yet, in practice, naming systems are often developed without a clear understanding of what is required of them.

1.5 Design of Naming Systems

It is reasonable to expect that an improved understanding of the communication semantics of naming systems will lead to improvements in the design of naming systems. There exists a large class of problems for which it is difficult to specify algorithms that are symmetric, fully distributed, and in which communication occurs only between those objects among which dependencies exist. Example problem domains include distributed simulation (e.g. n-body simulations), distributed constraint satisfaction (parallel execution of logic programs[61]), and consistency protocols for replicated databases (recovery from network partition failures[23]). Algorithms that are the most natural solutions to these problems produce *1–N* patterns of communication—that is, communication between a *source* object and several *destination* objects—where the membership of the set of destination objects is dynamic and determined by the global state of the computation. These patterns are impossible to specify before execution, because the global state may be determined in part by input from the environment. They are also difficult to specify during execution because the global state is distributed.

Closer inspection reveals that the difficulty in specifying such communications has its origin in the lack of expressiveness of the naming model used to specify the algorithm, and in the tendency of distributed naming systems to resolve names *before* communication occurs. The latter requires the global state to be known at the source of the communication.

Straightforward specification of dynamic, *1–N* communication patterns require three forms of *communication abstraction*, which extend to communication the same principles of abstraction that are commonly applied to data types, synchronization, and control sequencing[26]:

- *Grouping* is the binding of a single name to a set of objects. This allows specification of *1–N* communications that are independent of the size or composition of the set of destination objects (called the *target set*). In message-passing communication, for example, this means that a message may be delivered to multiple recipients without requiring the sender to enumerate, or even be aware of, the list of recipients.

- *Indirection* is the binding of a name to distinct objects or sets of objects during a computation. Indirection exists where the referent of a name may vary, much as does the referent of a pointer in a conventional programming language. This abstraction separates the interpretation of a name from the object to which the name is bound. For example, in a client/server system a name may indicate a service that is currently provided by the object to which the name is bound.

- *Description* incorporates into a name an abstraction of the state of the object(s) to which the name is bound. This allows communications to be specified in terms of the local states of objects.

1.6 Approach

The scope of our work extends to both the analysis and design of naming systems. As a contribution to the former, we define a formal hierarchy of naming models (the *taxonomy of naming systems*) that, while not forming a complete theory of the communication semantics of naming systems, consists of two elements that comprise the kernel of such a theory: a set of fundamental properties of naming systems, and a relation that ranks naming systems according to the expressiveness imparted by their properties. Our second contribution is the design of a concurrent system, Associative Broadcast, whose naming system enables straightforward, distributed, and symmetric solutions to problems that engender patterns of communication requiring grouping, indirection, and description. We use the taxonomy to rank the expressiveness of the naming system of Associative Broadcast among those of well-known concurrent systems from the literature.

1.6.1 Taxonomy of Naming Systems

The taxonomy defines a set of six orthogonal pairs of naming system properties. The taxonomy ranks naming systems in (partial) order of the expressiveness they derive from their properties. We capture the relationship between names, bindings, and communication by augmenting a partial-order model of a computation[52] 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 their participation 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. The 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.

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

1.6.2 Associative Broadcast

We propose a new concurrent system, called *Associative Broadcast*[1] in which broadcasting is the fundamental mode of communication. Associative Broadcast allows *1–N* communication to be specified at a high level of abstraction, and enables programs to exploit the broadcasting capabilities of distributed hardware, such as Ethernet[45], cellular radio, or satellite networks. Such systems support asynchronous message broadcast at a cost that is close to that of sending a message to a single recipient.

The central feature of Associative Broadcast is a naming system that allows the target set of a broadcast message to be defined in terms of the local states of the potential recipients. Because the specification of the target set travels with each message, it is not up to the sender to determine the identity of the destination objects. This enables straightforward specification of distributed, symmetric, and asynchronous algorithms that require communications in which the sender knows what the local states of the target objects should be, but does not know which potential recipient objects are in that state.

Associative Broadcast associates with each object a program-specific, mutable set of *attributes*, called a *profile*. A sender includes in an outgoing message a name that is a propositional expression over attributes. The name specifies the profiles of those objects that are to receive the message. An

[1] Because its operation is reminiscent of the content-addressability of an associative memory.

attribute is either a *symbol* (a *simple* attribute) or a *(symbol, value)* pair (a *compound* attribute). An attribute may encode the *type* of an object (denoting operations it supports), an abstraction of the object's state, or it may be a unique identifier. The symbols may be treated as data values and passed among objects in messages.

The conventional notion of a static object or group name is subsumed by Associative Broadcast names, which descriptively name dynamic sets of objects of arbitrary cardinality. Matching of the name against object profiles is performed locally and asynchronously as the message arrives at the sites where objects reside. Objects may be created on the fly. Each object controls its own profile and may add, remove, or alter the attributes in its profile at any time, although the matching of a name against an individual profile must be atomic with respect to the modification of that profile. The only global knowledge[2] upon which communication depends is the program-specific meaning of the attribute symbols.

The profile/name naming scheme results in programs that are clearer and more concise than those in which individual recipients filter messages from system-wide broadcasts. It is much easier to specify the intended membership of a target set as a single formula in first-order logic at the originator of a broadcast than it is to do the same at each potential recipient. The code that an object executes to alter its profile is typically significantly simpler than any message-filtering code would be. Associative Broadcast also offers advantages for implementation. The matching of names to profiles can be implemented once at a low level in the communication subsystem, rather than

[2]We use "global knowledge" informally here to mean information that is known to at least two objects.

repeatedly at the object level as is the case for arbitrary message-filtering code. This is particularly important where several unrelated computations may be in progress simultaneously.

1.7 Overview

The remainder of the dissertation comprises the following chapters. Chapter 2 describes the notation used in our formal definitions and proofs, and summarizes the partial-order model of concurrency upon which the definitions of both the taxonomy and Associative Broadcast are based. In chapter 3, we give the complete definition of the taxonomy of naming systems, followed by the definition of Associative Broadcast in chapter 4. We summarize related work on naming theory, broadcast-based programming models, and communication abstraction in chapter 5. Chapter 6 classifies several representative naming systems within the literature, and identifies the naming system requirements of a number of example problems. We give new solutions to several of these problems using Associative Broadcast in chapter 7. In chapter 8 we describe a generalization of the fault-tolerance properties of some of the algorithms in chapter 7. Chapter 9 contains concluding remarks.

Chapter 2

Preliminaries

We introduce here the concepts and notation used in the definitions appearing in subsequent chapters.

2.1 Partial Orders vs. Interleaving

In formally specifying the taxonomy of naming systems and Associative Broadcast, we are faced at the outset with a fundamental choice between using interleaving model of concurrency, or using a partial-order one.

An interleaving model defines a computation as a total order, or sequence, of *events*, where an event is the execution of an atomic action by an object, by the environment, or by several of these in synchrony. This sequence of events is an abstraction of what actually happens in a concurrent computation: For any two events, it may not be possible to determine which occurred first. Such events are said to be *concurrent*. In the case of a distributed system, for instance, the absence of a global clock makes it difficult to determine which of two instructions that are not causally related and are executed by distinct processors occurred first, or even to determine whether one completed before the other began. An interleaving model chooses an order for concurrent events arbitrarily. Thus, for any computation there may exist several sequences of events, each of which is a possible order in which the events occur, but none of which is necessarily the actual order. A computation, in turn, may be one of several possible executions of a given program. Thus a program typically specifies a large (possibly infinite) set of possible interleavings of events.

In contrast, a partial order of events models a computation in a manner that reflects more accurately the temporal relationship between events. For any two events that are causally related, e.g. because one represents the sending of a message and the other the receipt of that message, the relation $\mapsto$ (pronounced "precedes") holds. $\mapsto$ is irreflexive, antisymmetric and transitive. That is, if $a \mapsto b$, and $b \mapsto c$, then neither $a \mapsto a$, nor $b \mapsto a$ hold, but we may conclude that $a \mapsto c$. The partial-order model thus retains information about which events are concurrent and which are ordered by causality. A program specifies a possibly infinite set of partial orders of events, but a given computation is modeled by exactly one partial order of its events. In an interleaving model, we cannot distinguish in a single sequence those events that are ordered by causality from those that are interleaved arbitrarily; we must look at all the possible interleavings.

The advantage of interleaving models is that it is generally easier to reason formally about sets of sequences of events than about partial orders. Thus it is easier to prove general program properties such as deadlock-freedom, termination, etc. There exist several well-known logics and algebras for the analysis and specification of computations that are based on interleaving semantics, such as Unity[18], CSP[37], and temporal logic[51]. However, several researchers have pointed out that partial order models are a more natural, and thus often more convenient representation of computations[42, 52]. In [52], Pratt argues that partial-order models are better suited to complex systems, nonatomic events, and relativistic systems. We do not contribute to this discussion beyond stating that our work is an example of an additional situation in which a partial-order model of concurrency is preferable. Our definitions state the communication properties of naming systems primarily as precedence (or concurrency) relations among small sets of events. These definitions are, however, quantified over large sets of computations. Since a partial-order model captures precedence among events explicitly, we have found it a much more convenient vehicle for expressing the properties of concern to us.

2.2 Pomsets

We view computations as partially-ordered multisets[33], or *pomsets* of events. A multiset is simply a set in which an element may occur more than once. In a pomset, the ordering relation is defined on the distinct *occurrences* of the elements of the multiset. 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 an element are not considered to be equal.

We make the simplifying assumption that each event represents the execution of an action by exactly one object, and that the events of each object are totally ordered. Neither of these assumptions is restrictive: We can model synchronous communication, as well as objects with internal parallelism. However, the partitioning of events among objects and the certainty that $\mapsto$ holds for any two collocated events significantly simplifies some definitions and proofs.

We are interested in two general types of events: *communication* events and *naming* events. A communication event occurs when an object participates in a communication, either as the source object or as a destination object. A naming event occurs when an object executes an action that changes the state of the naming system. A computation may contain other, *internal*, events. Collocated sequences of internal events may be coalesced into a single event, or omitted entirely, as dictated by convenience.

The ordering relation we use is the aforementioned $\mapsto$, indicating causality among events. The only source of causal ordering among the events of distinct objects is communication. That is, $a \mapsto b$, where a and b occur at distinct objects, implies that a communication passes the respective objects "after" a and "before" b.

While there exists an algebra for the specification of pomsets[52], we have chosen to forego algebraic specification, and confine ourselves instead to first-order logic in describing the structure of computations (pomsets). This leads to more readable specifications that are, in this case at least, no less compact than the equivalent algebraic ones.

2.3 Notation

We use the following notational conventions: Names, objects, and events are represented by lower-case letters $[a \ldots z]$, sets thereof by capital letters $[A \ldots Z]$. Other entities are represented by the Greek alphabet, where capitals again indicate sets of their corresponding lower-case elements.

Our operators, appearing in increasing order of binding power, are:

$\Longrightarrow, \Longleftarrow, \Longleftrightarrow$

$\vee, \wedge$

$\neg$

$\forall, \exists$

$=, \neq, <, \leq, >, \geq, \subset, \subseteq, \supset, \supseteq, \sqsupset, \approx, \vdash, \in$

$\cap, \cup, +, - \mapsto$

"." (Infix dot denoting components: $a.b$ denotes component b of a)

All operators are left-associative. $\mapsto$ is the precedence operator among events, as stated above. We will define $\sqsupset, \vdash$, and $\approx$ in subsequent chapters. All other operators have their usual meanings.

Chapter 3

Taxonomy of Naming Systems

3.1 A Preliminary Example: Resource Tracking

The following example illustrates informally the nature of the naming system properties we define in this chapter.

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 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 3.4.1. 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 6.8 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 *multiplicity* property, defined in section 3.4.3.

3.2 Computations as Partial Orders of Events

3.2.1 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.

3.2.2 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. $\mathbf{send}(P, \text{msg})$ 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 6.2).

3.2.3 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 3.2.5), 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. $\Box$

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

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).

3.2.4 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 $\mathcal{C} = \mathcal{E} \times \mathcal{N} \times \mathcal{E}$, where $\mathcal{N}$ is the set of all names. □

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

3.2.5 Domains and Bindings

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. □

Generalizing from the the definitions in [19], 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. Since we assume that resolution always succeeds, it 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).

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 computation 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. Without loss of generality, we can assume (as with the universal context) 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.

The elements of the domain and binding annotations of each event correspond to elements of the known-names function, K and the context, C, respectively.

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"[17], 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.

3.2.6 Computations

Definition: (*precedes*, $\mapsto$) Let e and f be two events. The relation $e \mapsto f$ indicates that e necessarily precedes f in time[40]. $\mapsto$ is transitive, antisymmetric, and irreflexive. $\Box$

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

Definition: A *computation*, γ, is a pair (Π, Δ), where Π is a pomset $(\langle e, ... \in \mathcal{E}\rangle, \mapsto)$, 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:

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

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

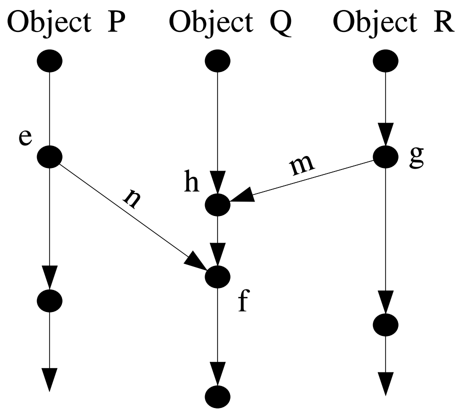

Figure 3.1: A computation

every object. In addition, two non-collocated events are ordered only where there exists (transitively) a communication between the two events. Naming models are differentiated according to the additional constraints they impose upon computations.

Figure 3.1 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.

3.3 Naming Models

Definition: A *naming model* is a specification of the allowable domains, bindings, communications, and order of events[1] of computations. For each naming model, a, there exists a *maximal computation set* Γ_a, the set of all computations specifiable under a. □

If a naming model 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 properties defined below characterize naming models by stating the existence or nonexistence of such computations in a maximal computation set.

3.4 Properties of Naming Models

There are six orthogonal characteristics of naming models: 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 model. These properties meet the following criteria: None of the properties holds for every naming model; therefore, the properties reflect design

[1] A model of concurrent computation that includes the naming model may further constrain the order of events.

decisions made in the development of the naming model. 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 model that is appropriate for the solution of a given problem. Finally, the definition of a naming model property does not specify how the property is to be implemented. Note that for each of the properties defined here, it is the case that a property holds for a naming model **iff** the property holds for every naming system that is an instance of the model. We therefore use the terms "naming system properties" and "naming model properties" interchangeably, depending upon context.

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 $\exists (e, n, f) \in \Delta \in \gamma \in \Gamma$ $\square$

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

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

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.

3.4.1 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.

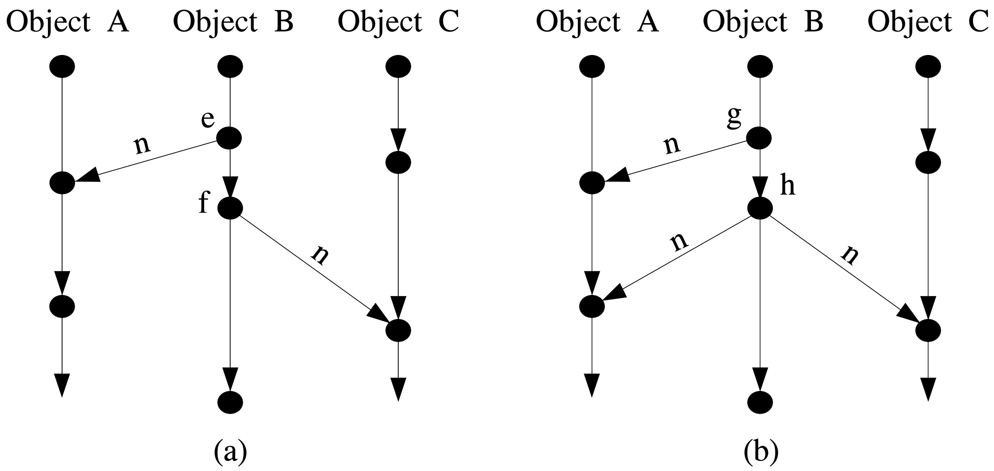

Figure 3.2: Dynamic binding

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

$\exists$ events e, f, communication set Δ, and name n ::

$$(e \neq f) \wedge (\beta(n, e, \Delta) \neq \beta(n, f, \Delta)) \qquad \Box$$

Definition: Static binding (**b**): $\neg\mathbf{B}$ $\Box$

Figure 3.2 illustrates two partial computations that contain instances of dynamic binding. In case (a), name n is bound to object A for the communication originating at event e. For the subsequent communication originating at event f, however, n is bound to object C. In the case (b), n is bound to a single object (A) for the communication originating at g, but to a set of two objects (including A) for the communications originating at h.

3.4.2 Knowledge

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 ::

$$(n \in f.d) \wedge \forall \text{ events } e :: (e.o = f.o) \wedge (e \mapsto f) \Longrightarrow (n \notin e.d) \qquad \Box$$

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

3.4.3 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 *multiplicity* property. The complementary property is *singularity*. 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[11] is an example), and group communication is the primary mode of object interaction in several systems, such as BSP[30], broadcast channels[3], and Associative Broadcast. In addition, concurrent systems in which objects communicate through shared data structures (e.g. Linda[31]) have the multiplicity property as well, since multiple objects may read any value written to such a data structure.

Definition: Multiplicity (**M**):

$$\exists \text{ event } e, \text{ communication set } \Delta, \text{ and name } n :: |\beta(n, e, \Delta)| > 1 \qquad \square$$

Definition: Singularity (**m**): $\neg\mathbf{M}$ $\square$

Figure 3.3 illustrates multiplicity. The two communications have a common source event, and both occur over the name n.

3.4.4 Multiply-Named Objects

Some naming systems allow a given object to refer to another object by more than one name. This dual of multiplicity is called the *aliased names* property. Where every object is known to every other object by at most one

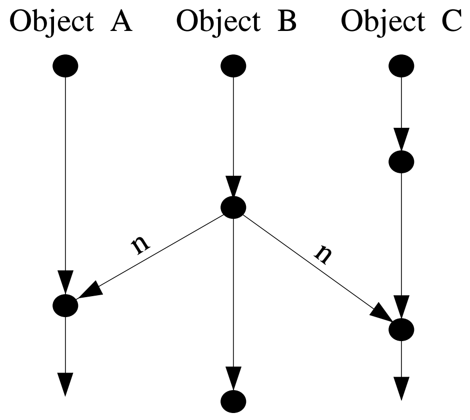

Figure 3.3: Multiplicity

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 6.3).

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

$\exists$ communications $(e, n, f), (g, m, h)$ ::

$$e.o = g.o \wedge f.o = h.o \wedge n \neq m \qquad \square$$

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

Figure 3.4 illustrates a partial computation in which names n and m are both bound to object B, exemplifying the aliased names property.

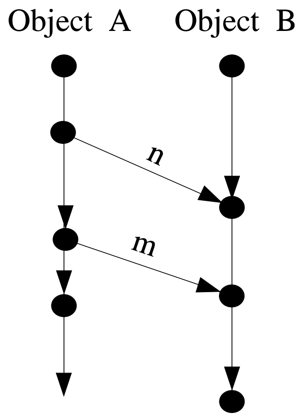

Figure 3.4: Aliased Names

3.4.5 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[34], 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 6.2 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

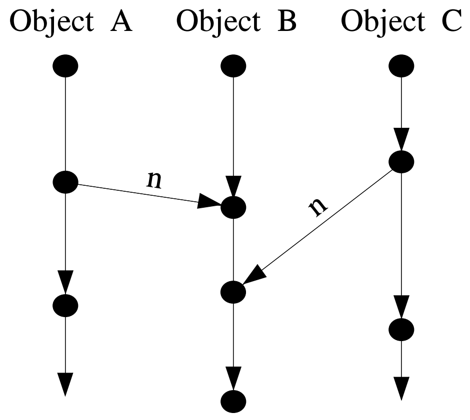

Figure 3.5: Shared names

each (*sender, receiver*) identifier pair constitutes a unique name.

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

$$\exists \text{ communications } (e, n, f), (g, n, h) :: e.o \neq g.o \qquad \square$$

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

In figure 3.5, a partial computation is shown in which name n is shared by objects A and C.

3.4.6 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 6.6). Both Associative Broadcast and Linda integrate descriptive reference with dynamic binding and 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**):

$$\exists \text{ behavior } e_0 \ldots f \ldots g_0 \ldots, \text{ name } n, \text{ and } j, k ::$$

$$\bigcup_i e_i.b \cup f.b \cup g_i.b \text{ is countably infinite } \wedge$$
$$n \in e_j.b \wedge n \in g_k.b \wedge n \notin f.b \wedge$$
$$\forall \text{ events } h :: h \in \text{ a consistent cut taken at } f \Longrightarrow n \notin h.b \qquad \square$$

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

That is, a naming system has the descriptive reference property **iff** it permits computations in which the set of all names bound to an object over its entire behavior is infinite, and such names may be *intermittently* bound only to one object. This allows a mapping from the states of an object to the names that are bound to it to be constructed.

3.5 Hierarchy of Naming Models

The six mutually exclusive pairs of naming model properties form the axes of a coordinate space, within which exists a complete lattice of naming model *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**

$$\forall \gamma : \gamma \in \Gamma :: P \text{ is true for } \gamma. \qquad \square$$

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:

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

$$(\neg(\Gamma_a \vdash P)) \wedge (\neg(\Gamma_b \vdash Q)) \wedge (\Gamma_a \cup \Gamma_b \vdash P) \wedge \neg(\Gamma_a \cup \Gamma_b \vdash Q)) \qquad \square$$

From their respective definitions, it follows that

$$\mathbf{B} \sqsupset \mathbf{b},\ \mathbf{D} \sqsupset \mathbf{d},\ \mathbf{M} \sqsupset \mathbf{m},\ \mathbf{A} \sqsupset \mathbf{a},\ \mathbf{S} \sqsupset \mathbf{s}, \text{ and } \mathbf{R} \sqsupset \mathbf{r}.$$

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

$$(\mathbf{B}, \mathbf{b}), (\mathbf{D}, \mathbf{d}), (\mathbf{M}, \mathbf{m}), (\mathbf{A}, \mathbf{a}), (\mathbf{S}, \mathbf{s}), (\mathbf{R}, \mathbf{r}) \qquad \square$$

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

$\square$

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

$$\forall i : 0 \leq i \leq 5 :: \Gamma_a \vdash R_i \qquad \square$$

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. $\square$

Notation: We denote the least upper bound of the hierarchy (category **BDMASR**) as $\top$, and the greatest lower bound (category **bdmasr**) as $\perp$. $\square$

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 model that has property P allows computations that cannot be obtained under a naming model with property Q. Where category J subsumes category K, each naming model that is a member of J allows computations that cannot be obtained under any naming model that is a member of K.

Note that if $\Gamma_a \vdash P$ and $\Gamma_b \vdash Q$, then in most cases, $\Gamma_b \subset \Gamma_a$. That is, under a naming model characterized by P, it will usually be possible to specify all the *computations* specifiable under a naming model characterized by Q. However, we cannot rule out the possibility that some naming system, a, will impose a peculiar constraint on the computations it allows, such that

$\Gamma_b \not\subset \Gamma_a$. We have thus far encountered only one such pathological case, namely the BSM system[34]. It is to allow for such cases that $P \sqsupset Q$ is defined in terms of $\Gamma_a \cup \Gamma_b$, rather than $\Gamma_b \subset \Gamma_a$.

Chapter 4

Associative Broadcast

As we discuss in chapter 6, few of the naming systems that are incorporated into existing concurrent systems reside near the top of the hierarchy defined in the previous chapter. Only the naming system of Linda[31] is a member of the topmost category, $\top$, of the hierarchy, and Linda is difficult to implement efficiently as a distributed system because its processes communicate through a logically-shared data structure, of which a globally consistent view must be maintained at all sites using a complex protocol. Associative Broadcast incorporates a naming system that is a member of $\top$, yet relies for communication only upon unordered message broadcast, such as that provided by broadcast-bus local-area networks and satellite networks. It is therefore a concurrent system suitable for the specification of algorithms that constitute distributed solutions to problems requiring category $\top$ naming system properties.

Recall that a category $\top$ naming system allows names to be rebound dynamically, propagation and generation of names, binding of names to sets of objects, multiply-named objects, shared names, and descriptive names. Of these, the naming system of Associative Broadcast distinguishes itself from other *distributed* naming systems by combining dynamic binding, multiplicity, and descriptive reference, and thereby providing the indirection, grouping, and description communication abstractions described earlier. These three properties allow us to specify the destination of a communication (a message in the case of Associative Broadcast) as a dynamic, descriptively-named *target set* of objects.

In section 1.5 we mentioned some of the problems whose most natural solutions are algorithms whose patterns of communication are functions of the global state. Associative Broadcast allows the design of algorithms to solve these problems that exhibit three desirable characteristics: They are symmetric because description eliminates the need for a central controller. They allow a high degree of parallelism because binding, communication, and resolution requires no global synchronization. Finally, they are efficiently executable: Associative Broadcast performs at a lower level the function of the object-level protocol that would be necessary to implement the same algorithm under a less expressive naming system.

Since the distinguishing feature of Associative Broadcast is its naming system, the definitions and discussion presented in this chapter focus upon the naming system and the communication system that depends upon it. While we also define the behavior of objects, we do so only to the extent necessary to allow us to write programs that use the Associative Broadcast communication primitive. We chose a simple task model for specifying object behaviors mainly because of its suitability for the algorithms that we develop in chapter 7. However, the communication and naming systems are specified independently, and

Function	*Specification*	*Implementation*
Knowledge	Naming convention Propagation rules	Generation mechanism Propagation mechanism
Binding	Binding rules	Binding mechanism
Resolution	Context	Resolution mechanism

Table 4.1: Elements of a naming system

may be incorporated without change into concurrent systems that use alternate object models.

4.1 Requirements

To specify the naming system used by Associative Broadcast, we must define each of the elements summarized in the table appearing in section 1.2, reproduced here for convenience (table 4.1). We first define the naming convention, binding rule, and propagation rule. Subsequently we define the operation of primitives that constitute the binding mechanism and propagation and (vacuous) generation mechanism, and the operation of the resolution mechanism.

In order to arrive at a suitable set of definitions, let us look at our requirements for the naming system in more detail. The naming convention, propagation rule, and binding rule must impart to the Associative Broadcast naming system all the properties of category $\top$. The generation, propagation, binding, and resolution mechanisms must not assume that the underlying distributed architecture provides any communication facilities beyond asynchronous, unordered message broadcast.

There are three characteristics of objects that may be may be used to specify the membership of a target set: the *type* of an object, the *state* of an object, and *identifiers*. The naming convention and binding rule should allow each of these characteristics to be represented within the naming system, and to be used in specifying target sets. Furthermore, the naming convention and binding rule should specify a uniform representation of all three characteristics, as well as a uniform manipulation of that representation by the propagation, binding, and resolution mechanisms. This simplifies the specification of target sets by abstracting information about types, states, and identifiers into uniform "naming information."

Though there are many sophisticated mechanisms for organizing and manipulating type information[62], we will denote an object's type simply as a set of symbols, each of which is associated with one of the object's actions.

An appropriate representation of the state of an object, i.e. the values assigned to the variables of an object at a particular point in a computation, is crucial to our approach. The naming system should allow a certain portion of an object's state to be "made visible" to other objects while hiding that part of the state that is irrelevant to communication. The part of an object's state that should be reflected in the name(s) that are bound to the object is problem-specific. The naming system must allow not only an object's changing state to be represented, but it must be defined in a way that provides control over which part of an object's state is visible at any time. Furthermore, an exact reflection of a given state variable may not be the most desirable one. For example, given an object that implements a stack, it might be that it should receive certain messages from other objects based only on whether the stack is empty or full, and not on the current value of its internal stack pointer. Thus it is important for the naming system to be capable of representing a suitable abstraction of an object's state.

Finally, an object may be characterized by one or more identifiers. An identifier is an arbitrary symbol that may be associated with an object. In more familiar naming systems, *unique* identifiers are the only means of referring to objects, and are thus the names in such a system. However, we avoid equating unique identifiers with names for two reasons: to avoid confusion with our definition of names, which is quite different, and because the role of identifiers is considerably weaker in our naming system. Identifiers are necessary only to distinguish between objects that are in identical states (at least as is discernible by other objects), or to refer to an object without regard to its current state.

4.2 Rules

In order to satisfy the above requirements, we first define a scheme in which an object's type, state, and identifiers may be abstracted into a concise symbolic representation. We then define the naming convention and binding rule.

An attribute is a symbol (a *simple attribute*) or a (symbol, value) pair (a *compound attribute*) used to represent object characteristics (types, states, or identifiers). While the actual symbols and values used to construct attributes are problem-specific, without loss of generality we can call the universes from which these are chosen S and V, respectively.

Definition: Let S be an infinite set of symbols, and V be an infinite set of values. An *attribute* is defined to be an ordered pair (s, v) or a singleton, (s), such that $s \in S$ and $v \in V$. $\square$

Attributes are *similar* **iff** their symbols are identical. The definition of $\approx$ is needed in order to properly define a profile as a set of attributes in which the symbol components of all the attributes are distinct. Since each symbol has a unique interpretation, it is nonsense for a profile to contain, for example, two attributes in which different values are associated with the same symbol, even though such a profile would fit the usual definition of a "set."

Definition: (*similarity*, $\approx$) For any two attributes a and b
($a = (s_a, v_a)$ or $a = (s_a)$, $b = (s_b, v_b)$ or $b = (s_b)$)

$$a \approx b \Longleftrightarrow s_a = s_b$$ $\square$

A profile associates with each object a set of mutually dissimilar attributes, which represents the type, state, and identifier information about the object that is used in binding names to the object. A *representation* is a set of profiles, one per object, that captures a complete set of name-to-object bindings (see *binding rule*).

Definition: Let $A = S \cup (S \times V)$ be the set of all attributes obtainable from S and V. A *profile* is a finite set of attributes $\{a_0, ..., a_k \in A\}$ such that:

$$\forall i, j : 0 \leq i, j \leq k :: i \neq j \Longrightarrow a_i \not\approx a_j$$ $\square$

Definition: Let O be a set of objects, and P be the set of all valid profiles. A *representation* is a set $\Omega = \{(o_0, p_0), ... \in O \times P\}$, such that:

$$\forall o \in O :: \exists (o, p) \in \Omega \wedge$$
$$\forall (o_i, p_i), (o_j, p_j) \in \Omega :: i \neq j \implies o_i \neq o_j \qquad \square$$

The distinction between simple and compound attributes, and indeed the notion of a *set* of attributes, is syntactic sugar that simplifies the manipulation of names and bindings by programmers and by the naming system. One could map all the possible combinations of object characteristics onto distinct values in V (i.e. a Gödel numbering), and thereby represent the characteristics of an object with a single complex attribute. However, this would make profiles difficult to specify, and would require considerable decoding on the part of an implementation. We therefore define a more convenient format. The complex form is intended to represent characteristics that take on a value from some set on which problem-specific relations are defined (see below).

It is possible to represent any of a finite set of values using relatively small number of simple attributes; specifically, a group of n pairs of simple attributes can be used to represent 2^n values. The group of simple attributes may be likened to an n-bit word, with the presence of one member of each pair corresponding to a 1, and the other to a 0. However, given the naming convention that we will define, treating such a group of attributes as a value from an ordered set would be extremely cumbersome. Given that complex attributes are necessary, one might conclude that all attributes should be of this form. However, there are many object characteristics for which such a representation would be meaningless. A unique identifier, for example, is just a symbol—it makes no sense to associate a value with it. One would thus have to define a special “null” value that was always ignored.

4.2.1 Naming Convention

Given a set of profiles that capture all the significant characteristics of a corresponding set of objects, we can now define a naming convention that allows the construction of names that consist precisely of the criteria designating a target set of objects, and a simple binding rule that determines how such a name is bound to its target set. We do this by allowing names to be propositional expressions over attributes, as follows:

Definition: A *name* is a propositional formula of the form $[t_0 \diamond ... \diamond t_r]$[1], where $\{t_0, ..., t_r\} \subset T$, and $\diamond \in \{\vee, \wedge\}$. $\square$

where T is a set of *terms*, which may be binary relations on V:

Definition: Let $\Phi = \{\{(v, w), ... \in V \times V\}, ..., \{(x, y), ... \in V \times V\}\}$ be a set of binary relations on V. The set of *terms* is
$T = S \cup \{s\phi v | (s \in S) \wedge (\phi \in \Phi) \wedge (v \in V)\}$. $\square$

The naming convention does not specify a name generation mechanism. This is because the set of names defined by the naming convention is

[1]Groups of terms may be be parenthesized to override the usual precedence of $\wedge$ over $\vee$. This has been omitted from the specification for the sake of clarity.

simply the set of propositional formulas over the set of possible attributes chosen by the programmer. Name generation thus consists of choosing a set of program-specific attribute symbols and a set of relations, both of which are performed at program-definition time.

4.2.2 Binding Rule

A name is *bound* to an object exactly when the name evaluates to *true* with respect to the profile of the object:

Definition: (*binding rule*) Let Ω be a representation, and $(o, p) \in \Omega$. Name n is *bound* to object o **iff** $p \vdash n$ (p satisfies n). $\Box$

An individual term of a name is satisfied by a given profile **iff** the profile contains an attribute for which the term evaluates to *true*:

Definition: (*satisfies*, $\vdash$, w.r.t. attributes)

$$\forall a = (s_a) \in A, t = s_t \in T :: a \vdash t \Longleftrightarrow s_a = s_t$$
$$\forall a = (s_a, v_a) \in A, t = s_t \phi v_t \in T :: a \vdash t \Longleftrightarrow (s_a = s_t \wedge (v_t, v_a) \in \phi) \qquad \Box$$

Each term in a name is either a symbol or a symbol, relation, and a value. The former evaluates to *true* if a given profile contains the symbol, while the latter evaluates to *true* if the profile contains the symbol, and the attribute's value satisfies the relation with the term's value. The entire name is evaluated by applying the usual rules of the propositional calculus.

The set of names that is bound to an object is completely determined by the object's profile, that set being all those names (propositional expressions) that are satisfied by the profile. Conversely, the target set to which a name is bound is exactly that set of objects whose profiles satisfy the name.

The definition of the binding rule, in conjunction with the earlier definition of a representation, indicates what a context is in the case of Associative Broadcast. Recall that in section 3.2.5 we characterized a context as the set of bindings that exist at a given consistent cut of a computation. Since a set of bindings is determined by a representation (set of profiles), the representation that exists at a consistent cut uniquely determines a context.

The act of evaluating a name with respect to a representation at a consistent cut is therefore exactly the name resolution mechanism (see below).

4.2.3 Propagation Rule

Objects may learn new names by receiving a message containing values or attribute symbols previously unknown to the recipient. Propagation is thus specified by the following message format:

Definition: The set of messages is $M = N \times D$ where, N is the set of names and D is a set of data value tuples $\{(v_0, ..., v_k) | (v \in S) \vee (v \in V)\}$. $\square$

That is, a message is of the form (*name, data*), and the data may contain elements of the set of symbols, S, or the set of values, V used to construct attributes.

4.2.4 Design Choices

The design of the naming convention and binding rule defined above was influenced by the following considerations:

- *Names must be evaluable in finite time.* An implementation of the resolution mechanism must be able to determine in finite time whether a particular name is bound to a particular object, in order to ensure finite delay of messages. In fact, we will require the function evaluation to be executed atomically with respect to the behavior of each object, so computational complexity should be minimized.

- *Names must be compactly representable.* Since the expression itself will be included in a message, it must be short enough to be transmitted efficiently.

- Given a representation in which all m profiles are distinct, it must be possible to specify each of the 2^m possible subsets as the target set of a single name.

While the specification language for names is conceptually that of first-order logic, we have not found it necessary to make use of quantification in any of the problems to which Associative Broadcast has been applied. Restricting names to propositional expressions improves the efficiency of the matching operation. It is also desirable to disallow negation in names to guard against inadvertent communications. We wish to disallow names of the form

$[\neg x]$, where x is a simple attribute that does not appear in any profile. Such a name allows the sender to send a message to all objects in the system, even though it may not know any of the attributes appearing in the profiles of any given object. Note that we can always replace the negation $\neg x$ with $notx$, and include the attribute $notx$ in a profile whenever x is removed from the profile. Similarly, the term $\neg(s\phi v)$ may be replaced by $(s\phi' v)$, assuming every relation ϕ has an inverse ϕ'.

4.3 Mechanisms

Having defined what names and bindings are in Associative Broadcast, we can now discuss how bindings are established by the binding mechanism and used by the resolution mechanism to support the communication mechanism. The binding and communication mechanisms consist of primitive actions that may be executed by objects. We define the semantics of these primitives by defining the set of computations that may be constructed using these actions. These computations are defined as partial orders of events, where an event is an execution of one of the primitives (or a "resolution event"), in a manner similar to the definition of the naming system properties in section 3.4. This also defines the operation of the resolution mechanism.

4.3.1 Primitives

Since all binding information is contained in the profiles of objects, the binding mechanism consists of a primitive that alters an object's profile. Each object controls its own profile, and the execution of the primitive *profile* := *pexpr*

sets the profile of the object to *pexpr*, but does not affect the profiles of other objects.

Communication is specified by the primitive

broadcast m

where m is a message taken from M. **broadcast** implements the name propagation mechanism, and implicitly invokes the resolution mechanism, as defined below.

4.3.2 Semantics

In the following definitions, O is the set of objects and P is the set of profiles.

Definition: The set of *events* is $E = B \cup L \cup R \cup C$, where

$B = O \times M$ is the set of *broadcast* events,

$L = O \times P \times M$ is the set of *message arrival* events,

$R = O \times M$ is the set of *message receipt* events,

$C = O \times P$ is the set of *profile alteration* events $\square$

A computation contains four types of events: broadcast of a message, arrival of a message at an object, receipt of a message by an object, and alteration of an object's profile. Each event is labeled with the object participating in the event, the object's profile at the time the event occurs (for events in $L \cup C$ only,) and the message involved in the event (with the exception of events in C.) We distinguish between the arrival and receipt of a message because not every message that arrives at an object is received by the object.

Notation: We denote the components of an event e as $e.o$, $e.p$, and $e.m$, and the components of a message m as $m.n$ and $m.d$ $\square$

Definition: A *computation* is a pomset $(\Pi = \langle e, ... \in E \rangle, \mapsto)$ such that (1)–(8) below hold. Let $\Pi_B = B \cap \Pi, \Pi_L = L \cap \Pi, \Pi_R = R \cap \Pi$, and $\Pi_C = C \cap \Pi$.

$$\forall e, f \in \Pi :: (e.o = f.o) \Longrightarrow (e \mapsto f) \vee (f \mapsto e) \tag{1}$$

$$\begin{aligned} &\forall e, f \in \Pi :: (e.o \neq f.o) \wedge (e \mapsto f) \Longrightarrow \exists b \in \Pi_B, l \in \Pi_L :: \\ &\qquad (b.o = e.o) \wedge (l.o = f.o) \wedge (b \mapsto l) \wedge (b.m = l.m) \\ &\qquad \wedge ((e = b) \vee (e \mapsto b)) \wedge ((l = f) \vee (l \mapsto f)) \end{aligned} \tag{2}$$

$$\begin{aligned} &\forall l \in \Pi_L :: \exists c \in \Pi_C :: (c.o = l.o) \wedge (c.p = l.p) \wedge (c \mapsto l) \\ &\qquad \wedge (\nexists d \in \Pi_C :: (c.o = d.o) \wedge (c \mapsto d \mapsto l)) \end{aligned} \tag{3}$$

$$\forall b \in \Pi_B, w \in O :: \exists l \in \Pi_L :: (l.o = w) \wedge (l.m = b.m) \wedge (b \mapsto l) \tag{4}$$

$$\begin{aligned} &\forall k, l \in \Pi_L : k \neq l : k.o = l.o :: \exists a, b \in \Pi_B : a \neq b :: \\ &\qquad (a \mapsto k) \wedge (b \mapsto l) \wedge (a.m = k.m) \wedge (b.m = l.m) \end{aligned} \tag{5}$$

$$\forall a, b \in \Pi_B, w \in O : a \neq b :: \exists k, l \in \Pi_L : k \neq l : k.o = l.o = w ::$$

$$(a \mapsto k) \wedge (b \mapsto l) \wedge (a.m = k.m) \wedge (b.m = l.m) \qquad (6)$$

$$\forall r \in \Pi_R :: \exists l \in \Pi_L :: (l.o = r.o) \wedge (l.m = r.m) \wedge (l.p \vdash l.m.n)$$

$$\wedge (\nexists e \in \Pi :: l \mapsto e \mapsto r) \qquad (7)$$

$$\forall l \in \Pi_L :: l.p \vdash l.m.n \Longrightarrow \exists r \in \Pi_R :: (l.o = r.o) \wedge (l.m = r.m)$$

$$\wedge (l \mapsto r) \wedge (\nexists e \in \Pi :: l \mapsto e \mapsto r) \qquad (8)$$

□

The structure of a computation based on E and $\mapsto$ is constrained by the eight conditions given above. Condition (1) specifies a total order on the events at each object, that is, all events involving a particular object are atomic relative to one another. This total order is not required, but weaker conditions complicate the definitions considerably and do not contribute anything of interest. A crucial property that any weaker set of conditions would have to preserve is the atomicity of profile alteration events relative to message arrival events and receipt events. Without this restriction, the name resolution mechanism would no longer be well defined.

Condition (2) states that two events at two distinct objects are related by $\mapsto$ only if a message passes "between" the two events. The third condition ensures that the profile of an object only changes at profile alteration events. (4) requires a broadcast message to arrive at every object in finite time. The fifth and sixth conditions enforce a correspondence between broadcast and arrival events by requiring the arrival of distinct messages at an object to have resulted from distinct broadcast events (5), and conversely, requiring distinct broadcast events to result in distinct message arrivals at every object (6).

Finally, conditions (7) and (8) specify the name resolution mechanism. The arrival of a message at an object is followed immediately by the receipt of that message by the object **iff** the object's profile at the arrival event satisfies the name contained in the message. The synchronization requirements are extremely weak. Both broadcast and profile alteration are *spontaneous* events—they are generated by the internal behavior of an object—while message arrival and receipt events occur in direct response to other events. Thus the model does not prohibit executions in which the profile of an object changes while messages are in transit. If an object changes its profile such that it either joins or leaves the group specified by the name in a message that is in transit, then the membership of the group is determined by nondeterministic ordering of profile alteration and message arrival events at each object.

This nondeterminism is not inherently any more useful or detrimental than any other form that is allowed by other concurrent systems. The effect of nondeterminism is entirely dependent on the semantics of a particular problem. Some problems have solutions whose executions are naturally deterministic, other problem solutions are unaffected by nondeterminism and may therefore benefit from increased parallelism, and still others require group membership to be deterministic.

Although it is not a primary goal of this research to address problems requiring atomic[21] or "virtually synchronous"[12] broadcast, there is a straightforward extension to the models of names and communication that integrates the concept of virtual time and its associated synchronization mechanism, time warp[38], to provide deterministic group membership. To sketch this approach briefly: The time warp mechanism maintains multiple versions of an object, each corresponding to the state of the object at a virtual time that lies between "global virtual time" and the current value of the object's local virtual clock. We simply include in each of these versions the profile of the object as it exists at that virtual time. Each message contains a "virtual receive time" in addition to a name. The message arrival event (i.e. the evaluation of the name on the object's profile) is then defined to occur when the value of the object's local virtual clock equals the virtual receive time of the message. This extension imposes a virtual synchronization on the resolution mechanism that gives the *appearance* of a name being resolved against the profiles of all objects in one globally atomic step. Alternately, one may apply the technique described in [53].

Conditions (7) and (8) above state that a message arrival event for which the object's profile satisfies the name contained in the message is followed immediately by the receipt of that message. It might thus appear that an object must always be prepared to receive a message that has just arrived, without generating any intervening spontaneous events. Viewing message arrivals in this way as events that occur unpredictably relative to the behavior of objects would lead to an unreasonable requirement that an object abort an operation in progress whenever a message arrived. However, note that an implementation of the communication system is free to define the arrival of a message at an object as occurring when the object is prepared to receive a message. This means that at least some messages that physically arrive while an object is "busy" must be queued until the object is ready to participate in the logical arrival event.

It is sufficient but not necessary to queue every physically arriving message at every object. The message need only be queued if the name contained in the message is satisfied by the object's profile, because the profile may not have changed when the message reaches the head of the queue.

4.3.3 Object Behavior

The following definitions specify how objects react to the receipt of a message and generate broadcast and profile alteration events.

Definitions The following definitions refer to D, M, and E as previously defined.

An object consists of a set of variables whose values comprise the object's state, and a set of operations that transform the state of an object in response to the receipt of messages.

Definition: An *object* is specified by $o = (X, I, F, H, p_0)$, where

X is a set of variables
I is a set of initial values for X
F is a set of operations
H is a set of attribute-operation bindings
p_0 is an initial profile □

The definition of an object includes a set of attribute-operation bindings. This set, which may change during the course of a computation, associates simple attributes in the object's profile with its operations. An operation may have zero or more simple attributes associated with it, but each simple attribute is associated with at most one operation. Collectively, these attribute-operation bindings make up the type specification of the object, and we refer to these attributes as *type attributes*.

Definition: Let p be the object's profile. The set of *attribute-operation bindings* is $H = \{(a_0, f_0), ..., (a_k, f_k)\}$ such that

$$\forall (a, f) \in H :: (a \in S \cap p) \wedge (f \in F) \wedge$$
$$\forall i, j : 0 \leq i, j \leq k :: i \neq j \Longrightarrow a_i \neq a_j \qquad \Box$$

Definition: An *operation definition* is of the form $\texttt{mname}(y_0, ..., y_n)\{body\}$, where the y_i are formal parameters and *body* is a program that computes the operation's function. *body* may contain one or more instances of the following statements:

broadcast m;
$profile := pexpr$;
$H := mexpr$;

where m is a message, $pexpr$ is and expression that evaluates to a valid profile, and $mexpr$ is an expression that evaluates to a valid set of attribute-operation bindings. $\Box$

A well-formed message contains a name that refers to exactly one type attribute. Furthermore, a well-formed message contains a tuple of data values for which the function (i.e. operation) associated with that type attribute is defined. Practically speaking, this means that the tuple of data values must supply the correct number and type of arguments to the operation.

Definition: A message $m = ([t_0 \diamond ... \diamond t_r], d) \in M$ is *well-formed* relative to o **iff**

$$\exists i, (a, f) : 0 \leq i \leq r : (a, f) \in H :: a \vdash t_i$$

$$\wedge (\nexists j, (b, g) : i \neq j : a \neq b :: b \vdash t_j)$$

$$\wedge f(d) \text{ is defined.} \qquad \Box$$

For the sake of simplicity and ease of analysis we have adopted a single-thread task model of execution, in which an object interacts with the outside world only at the beginning and end of a finite sequence of actions, rather than a process model because a task model imparts more structure to a computation. One need only reason about the state of an object at the beginning and end of a local computation, rather than keeping track of the state at every atomic step.

The behavior of an object consists of a sequence of local computations, each of which is a finite sequence of events. Every local computation begins with the receipt of a message, which is followed by a finite number of broadcast and profile alteration events. The state of the object at the end of a local computation is a function of its state at the beginning of the local computation and of the set of data values carried by the message. Broadcast and profile alteration events are generated as side-effects of the evaluation of the function.

Definition: A *state* is an assignment of values to X. □

Definition: Let Σ be the set of states. An *operation* is a function: $\Sigma \times D \rightarrow \Sigma$ □

Definition: A *local computation* is a sequence $\langle (r, \sigma_0) \mapsto (q_1, \sigma_1) \mapsto \cdots \mapsto (q_i, \sigma_i) \mapsto \cdots \mapsto (q_n, \sigma_n) \rangle$ such that

$$r \text{ is well-formed relative to } o \wedge$$
$$(r \in R) \wedge (\exists f \in F :: f(\sigma_0, r.m.d) = \sigma_n) \wedge$$
$$\forall i : 1 \leq i \leq n :: q_i \in B \cup C$$
□

It might appear that object behavior does not conform to a task model because an object may broadcast messages before it completes the execution of a operation. However, this behavior is exactly equivalent to one in which all outgoing messages are delayed until the local computation completes, since the object does not receive any more messages until the start of the next local computation. Also note that although we allow multiple profile alteration events during a local computation, the state of the profile is significant only at the end of the local computation, when the next message arrival event takes place.

The definition of a local computation specifies that at most one operation is active at any time. We do not allow multiple execution threads within objects in order to avoid issues of mutual exclusion on the variables of an object. Any computation that relies on multiple simultaneously active operations can be equivalently specified as separate objects that synchronize via messages instead of by mutual exclusion on shared variables.

4.3.4 Implementation Strategy

We assume that each object resides at one of a set of host *sites* that are interconnected by a communication network that supports asynchronous, unordered broadcast. Such networks include broadcast-bus (Ethernet), n-cube, ring, and shared-memory architectures. The implementation of Associative Broadcast is divided between the compiler, the *send* runtime system (send RTS), and the *receive* runtime system (receive RTS). The send RTS and receive RTS at each site form the interface between objects and the network hardware. They maintain the data structures that implement profiles, provide the *broadcast* primitive and the profile-modification mechanism, and perform selector/profile matching.

To allow the matching operation to be performed efficiently, the selectors should be converted to a normal form for which the matching algorithm can be optimized. This conversion may be performed by the compiler in cases where selectors are constant expressions, but may have to be left to the send RTS if selectors are generated by objects at runtime (i.e. if we allow variables of type `selector`). Selectors should be converted to conjunctive normal form expressions in which the conjuncts are arranged in order of increasing complexity of evaluation. That is, singleton conjuncts that refer to simple attributes should appear first. Since the matching of a selector against a particular profile terminates as soon as any conjunct is not satisfied, simpler conjuncts should be evaluated first to reduce the average time for failed matches. E.g.,

$$[(\texttt{ITEMTYPE} = \mathit{red} \vee \texttt{ITEMTYPE} = \mathit{blue}) \wedge \texttt{INSERT} \wedge \texttt{NOTFULL}]$$

should be converted to

$$[\texttt{INSERT} \wedge \texttt{NOTFULL} \wedge (\texttt{ITEMTYPE} = \mathit{red} \vee \texttt{ITEMTYPE} = \mathit{blue})]$$

The send RTS implements the **broadcast** primitive, possibly performs selector normalization, and initiates message transmission on the network. Most of the functionality of Associative Broadcast is contained in the receive RTS. The receive RTS at each site maintains the profiles of the local objects ("local profiles"), and implements the profile-modification mechanism. It matches the selectors of incoming messages, and delivers messages to objects upon successful matches. The matching operation is performed in two stages, called the *primary filter* and the *secondary filter*. This division is based upon the assumption that sites will receive many messages whose selectors are not satisfied by *any* local profiles. Such "sparse broadcasts" occur when the average size of a target set is small compared to the number of sites, or when objects are grouped so that those that are likely to need to communicate frequently are collocated to minimize network traffic.

The purpose of the primary filter is to quickly discard those messages that have no possibility of matching a local profile. This is done by maintaining a hash table of the *union* of the sets of attribute symbols appearing in the local profiles. When a message arrives at a site, the primary filter scans the selector, and attempts to find each attribute symbol appearing in the selector in the hash table. If none of the attribute symbols in the selector are found, then no local profiles satisfy the selector, and the message can be discarded. Furthermore, if any simple attributes that are singleton conjuncts in the selector are not hashed successfully, the message may be discarded as well.

Once a message has passed the primary filter, it must be matched against each local profile by the secondary filter. This operation also takes advantage of the selector's normal form, terminating as soon as an unsatisfiable term is encountered. Several profiles can be matched concurrently with associative or parallel hardware support.

4.3.5 Matching Time Complexity

A selector can be scanned by the primary filter in $O(1)$ time per attribute symbol appearing in the selector, and a hash-table lookup takes $O(1)$ time. Since the lookup of the first conjunct may cause the message to be discarded, the best-case complexity for the primary filter is $O(1)$, and the worst case is $O(n)$ for a selector of length n. Similarly, a secondary-filter match may fail in $O(1)$ time, yielding a best-case complexity of $O(p)$ (all matches fail) and worst case of $O(pn)$ (all matches succeed), when there are p profiles that are matched sequentially.

Previous work on "packet filters"[47] has shown that low-level per-object filtering is practical, and may in fact improve efficiency by reducing the number of context switches needed to process an incoming message[47]. Since packet filters consist of arbitrary user-supplied code, we expect that on average the matching of a selector to a single profile is considerably more efficient than the execution of a single packet filter. In the algorithms we have developed using Associative Broadcast, selectors contain a small, bounded number of attribute symbols. Finally, it is possible for a sophisticated compiler or send RTS to identify those conjuncts that have a low probability of being satisfied, and rearrange selectors accordingly to optimize the matching operation.

Chapter 5

Related Work

We are fundamentally concerned with three issues: broadcast-based systems, communication abstraction, and the analysis of naming systems (naming theory). In this chapter we discuss representative research in each of these areas.

The development of broadcast-oriented interconnection networks such as bus and ring networks has motivated a search for ways in which the capabilities of these networks might be exploited. Rather than viewing broadcast as a general abstraction tool for the specification of distributed programs, as we do, it is often used only to enhance the performance of a particular implementation[1]. In some other cases the broadcast mechanism is optimized to solve only certain specific problems. Within this category, consistency protocols for replicated objects is a typical application. The development of broadcast communication has thus been driven by specific requirements from below, rather than by general principles from above. Nevertheless, some models have

[1]The implementation of the Linda tuple space suggested in [31] is a case in point.

been proposed that are not necessarily limited to the solution of specific problems, and are thus the ones most closely related to our research. We review some representative models that seek to provide a generally-applicable broadcast facility.

Description is a form of communication abstraction that is related to certain database concepts, particularly ones appearing in relational database models. We summarize the relationship of the Associative Broadcast model to that of a relational database system, and discuss two models of computation whose naming systems support description. In addition, we review a proposed programming technique for adding communication abstraction to existing languages.

Finally, we survey the research in analysis of naming systems. All the work in naming theory of which we are aware deals exclusively with name resolution, and does not address the communication semantics of naming systems.

5.1 Broadcast-Based Systems

5.1.1 The ISIS System

ISIS [10, 11, 12] is a programming environment for distributed systems that supports the implementation of *resilient* objects. A resilient object is one that can tolerate a certain number of *site failures* and still remain operational. This is accomplished by keeping identical copies of the object at several sites in the system. For each operation on the resilient object, one such copy is designated the *coordinator*, and actually executes the operation. The other copies, known as *cohorts*, act as passive backups to the coordinator. Should the coordinator fail while the operation is being executed, one of the cohorts takes over as coordinator and completes the operation.

The central problem in maintaining these resilient objects is to keep the states of the cohorts consistent with the state of the coordinator, so that a cohort is prepared to assume the role of coordinator at any time. In order to accomplish this, the copies of the object execute what amount to variations of a transaction commit protocol. These protocols logically order the events of an ongoing computation (site failures and message transmissions.) The communication subsystem of ISIS is designed specifically to support efficient and fault-tolerant execution of these protocols.

The communicating entities in ISIS are *processes*. Processes may be gathered into *process groups*. The intended use of the group abstraction is that a process group corresponds to a resilient object, where each process in the group implements one of the copies of the object. Each process then runs at a distinct site in the system, so that a resilient object implemented as k processes can tolerate $k-1$ site failures. The processes in a process group execute the consistency protocols using the *group broadcast* communication primitives provided by ISIS.

To accommodate the requirements of a variety of consistency protocols, ISIS defines four broadcast primitives: FBCAST, CBCAST, ABCAST, and GBCAST. These primitives provide a tradeoff between performance and the amount of logical ordering they impose on the message transmission events of a computation. FBCAST is the least expensive (in terms of the amount of synchronization among processes that it requires) and guarantees only local ordering: two messages broadcast by one process are received in the same order they were sent. CBCAST is more expensive, and adds a transitivity guarantee: if process A broadcasts a message, and process B broadcasts a message after it receives the broadcast from A, then if process C receives both these broadcasts, it receives the message from A before it receives the message from B. ABCAST goes a step further. Besides guaranteeing transitivity, it ensures that the messages broadcast using two concurrent instances of ABCAST are received in the same order by every process that receives both messages. Finally, the most expensive form of broadcast, GBCAST, guarantees this global ordering with respect to messages sent using any of the other broadcast primitives. For example, if a GBCAST occurs concurrently with an FBCAST, all processes that receive both messages will receive them in the same order.

While these communication primitives are available at the user level, their primary purpose is to support the abstraction of resilient objects, which in turn supports a collection of fault-tolerant programming tools. Thus the designers of ISIS found it advantageous to build complex synchronization mechanisms into their model of communication that are not necessarily useful for the specification of concurrent programs, but rather for their robust execution.

In contrast, Associative Broadcast treats broadcast communication as the fundamental structuring tool for concurrent computations, but imposes no ordering upon events except that implied by causality.

The difference between the goals of the ISIS system and those of Associative Broadcast are reflected in their respective naming systems.

The Associative Broadcast concept of a target set is related to that of an ISIS process group. However, a target set is an implicitly-defined grouping of objects that is an artifact of the logic used to specify names, while process groups are explicitly maintained entities, designated by a unique, arbitrary name that is bound to all of the members of the group. When an Associative Broadcast object alters its profile, it may be joining or leaving infinitely many target sets. Since a process joining or leaving a process group typically involves at least one operation per group, emulating target sets using process groups is potentially very expensive. It requires the definition of a process group for every possible combination of attributes, causing the number of process groups to be exponential in the global number of attributes. This is extremely impractical, and the ISIS naming system was never intended for this purpose.

5.1.2 Broadcasting Sequential Processes (BSP)

An early recognition of some of the advantages of using broadcast in distributed programs appears in [30]. This paper describes an extension to Hoare's CSP [36] that adds several kinds of broadcast primitives, and contains persuasive arguments in favor of using broadcast primitives to structure concurrent computations. However, the BSP model allows only system-wide broadcasts, and leaves it to individual recipients to filter out unwanted messages. Similar message filtering is performed by the *selective accept* statement of Concurrent C[28], although that language has no built-in broadcast primitives.

The semantics of the BSP primitives are defined only informally at a very high level. We include this model here primarily because it is the one that most closely approximates the synthesis of broadcast and description in the Associative Broadcast model.

Briefly, the additional communication primitives introduced in BSP are as follows:

- The statement **all**$\odot m$[2] is an *unbuffered broadcast.* The message m is sent to every *other* process in the system. If a process has previously executed a receive statement when the message arrives at the site where the process is executing, the process receives the message, otherwise the message is "lost."

- The statement **all**$\otimes m$ is a *buffered broadcast.* m is sent to every other process, and placed in a FIFO buffer associated with the process, unless that buffer is empty and the process is currently blocked at a receive statement.

- The statement **any**$\oslash m$ receives a message m from the buffer associated with the process if it is not empty, otherwise it blocks until the message arrives.

- A variation of the broadcast statements are those that have the form $[p_1, p_2, ..., p_n] \otimes m$, which multicast the message m to a specified set of processes.

- A receive statement of the form **any**(info $= i$, sender $= [p_1, p_2, ..., p_n]$) $\oslash$ m waits for a message from one of the specified processes, whose text contains the specified "information," i. Any messages that arrive ahead of the desired message are discarded.

[2] We use symbols here that differ slightly from those in the original paper.

The author lists several advantages of "broadcast programming" using these primitives, and of broadcasting in general: Processes may interact without having explicit knowledge of each other. Processes can be dynamically added and deleted while a distributed program is running. The activity of the processes can be monitored without having to modify the monitored processes, since a process can be added to monitor all broadcasts. The author also notes that broadcasting is an appropriate mechanism for efficient use of an underlying broadcast interconnection network. Finally, special emphasis is placed upon the suitability of broadcast for programs consisting of an *administrator* process that delegates work to a set of *worker* processes. The stated advantage over CSP is that the administrator and workers need not name each other explicitly, thus the administrator can delegate its work to varying numbers of workers, as needed.

This system differs from Associative Broadcast in several significant ways. It is described only very informally, and mixes a variety of communication primitives together. Along with buffered and unbuffered broadcast, which are asynchronous, the model retains the synchronous communication of CSP. BSP allows only system-wide broadcasts, and leaves it to individual recipients to filter out unwanted messages. Similar message filtering is performed by the *selective accept* statement of Concurrent C[28], although that language has no built-in broadcast primitives. The (info $= i$) construct places the burden of resolution on the programmer, who has to take into account the format of every possible message generated by possibly unrelated computations, instead of on the underlying communication system, where it properly belongs. Lastly, the usefulness of unbuffered broadcast is unclear.

5.1.3 Other Broadcast-based Systems

Still another approach to broadcasting is taken by Gueth, et al. in their description of "broadcasting source-addressed messages" (BSM) [34]. In this scheme, a unique identifier is associated with each process. This identifier is included in every message broadcast by the process. Each process may "listen" for messages broadcast by a *single* source, that is, every message received by a process originates at the same sender. This model is peculiar in that it only allows a "tree" of dependencies among processes, with possibly some "back-edges" to the "root process." The purported advantage of this is that the computation is free from side-effects. A process never has to address the possibility of messages arriving from multiple sources in some unknown order. The system is intended mainly for application to real-time process control, in which such deterministic behavior can be of importance.

A number of other models use a statically-named channel, port, or socket approach. In [3], a sender broadcasts a message by placing it in one of several named channels. Each process in the system may read the message from the channel at most once. The message remains in the channel until it is explicitly removed by the sender. This resembles the communication mechanism of Linda [31], but in this case the semantics of adding and removing messages to/from the globally accessible space are not as well defined, nor do they provide for selecting particular messages from a channel using a pattern-matching scheme.

The models proposed in [43] and [4] are almost as inflexible as BSM, and are essentially equivalent in that they implement broadcast using a set of statically defined "network ports" or "network sockets," respectively. A sender broadcasts a message by specifying a port (socket) name to which the message is to be delivered. Each process that has *imported* the port receives the message. Importation is not a dynamic process—the ports imported by a process are declared statically at compile time.

5.2 Communication Abstraction

5.2.1 Script

An early contribution by Francez and Hailpern was their recognition of a need for communication abstraction mechanisms to complement the existing spectrum of control, data, and synchronization abstractions. They state that the purpose of the abstraction mechanism they introduce in [26] is to "allow the hiding of low-level details concerning sequencing of communications," restrict the patterns of communication ... to well-structured patterns," and to "enable a single definition of frequently used patterns."

Rather than defining a specific model of computation that incorporates this mechanism, the approach they take is to present a general strategy for implementing communication abstraction using the existing constructs and communication primitives of a language that supports concurrent processing. In fact, they specifically require that their abstraction mechanism, called a *script*, "should not add functional power to the host language." They also explicitly avoid issues of "dynamic concurrency," meaning, in essence that they rely on static binding of names to processes. Since scripts do not add any expressiveness to a language, this is a necessary assumption if the technique is to be widely applicable. Specific examples are given of adding scripts to CSP, Ada, and shared-variable language that uses monitors.

A script has three components that are specified using the constructs of the host language: a set of *roles*, which are formal parameters to which the names of processes are bound when the script is executed, a set of formal data parameters representing the data to be transmitted, and a *body*, which is a concurrent program that implements the function of the script using the communication primitives of the host language. A script is thus to communication primitives as an abstract data type is to the constructs of a conventional sequential programming language.

A process *enrolls* in a script by naming the role to which its name is to be bound, naming the other processes with whom it wants to communicate, and possibly supplying some values for the data parameters of the script. This is known as *partners-named enrollment*. An alternative is *partners-unnamed enrollment*, in which processes communicate anonymously with each other. This is reminiscent of description, except that the host language must be capable of supporting this type of enrollment. The authors note that this cannot be applied to CSP, for example. The body of the script executes when the required roles have been filled.

The overall concept of communication abstraction is one that we have embraced in defining Associative Broadcast. However, our approach is quite different. Where scripts are a technique for adding abstraction by hiding the communication primitives of an existing naming system, the abstraction in Associative Broadcast is realized directly by the communication primitive itself. Also, Associative Broadcast concerns itself directly with dynamic concurrency. Finally, scripts do not alter the underlying naming system of the host language, so any implementation of an abstraction is subject to the restrictions of that language.

5.2.2 Relational Databases

The central idea of the Associative Broadcast naming system is the representation of objects by sets of attributes (profiles.) The representation referred to by names and relied upon by the resolution mechanism is thus a set of sets of attributes, much the way a relation in a relational database system is a set of *ordered* sets of attribute *values* (tuples). A profile in a representation may change as the result of the execution of a method by its associated object, while a tuple in a relation may be replaced by another as the result of an update. There is thus a certain correspondence between the Associative Broadcast naming system and the relational database model. However, there are some significant differences as well.

One may draw the correspondence between the two models as follows. For any representation (set of profiles) it is possible to construct a corresponding relation. Let k be the number of distinct symbols appearing in the union of the profiles that comprise the representation. Let p be the number of profiles in the representation. Then the corresponding relation is a set of p k-tuples, such that there is a 1–1 mapping from profiles to tuples. The attributes of the relation (i.e. the labels on the fields of each tuple) are the k distinct symbols. The values contained in each tuple are as follows. Let a be an attribute of the relation, t a tuple, and q the profile corresponding to t. If the simple attribute a appears in q, then the field of the tuple labeled a contains the special value *True*. If the complex attribute (a, val) appears in q, then the field of the tuple labeled a contains *val*. Finally, if no attribute whose symbol is a appears in q, then the field of the tuple labeled a contains the special value *False*.

The name resolution mechanism in the Associative Broadcast naming system thus corresponds to the execution of a *select* query in a relational database. A name is a boolean function on profiles. Similarly, the specification of a select query is a boolean function on tuples. The name resolution mechanism delivers a message to each object whose profile satisfies the name, while the execution of the select query returns all those tuples that satisfy its specification. An individual term in a name is satisfied by an attribute in a profile **iff** the corresponding field of the tuple that corresponds to the profile contains either *True* (if the term is just a symbol), or if it contains the value specified by the term. It is thus straightforward to specify an equivalent select query for every valid name in the Associative Broadcast model of names.

Several differences between the two models are also readily apparent. First, the objects in a relational database are entirely passive and can only change in response to an update transaction originating outside the database system. The database objects *are* the tuples themselves, while the objects in our model are active, and can change their profiles spontaneously (as well as those of other objects, by sending them messages.) An Associative Broadcast computation is not necessarily composed of transactions that each have a well-defined beginning and end, as is a database computation. Partly because of these differences, there is no *a-priori* requirement for maintaining the consistency of objects that receive overlapping concurrent broadcasts, as there is for overlapping database transactions. A database typically contains many relations, while a representation consists of only one "relation." A name is a very specialized instance of the general concept of a select query, which may be much more powerful. This is also the only equivalent of a database operation in associative broadcast—we are not concerned with joins, etc.

5.2.3 Linda and the AP3-based OS

The concept of description also appears in two other models. However, unlike a relational database or an Associative Broadcast computation, in which objects are (or are represented by) tuples of attribute values, these models rely on a database of tuples whose existence is independent of that of the objects that actually carry out the computation. The objects communicate with each other through the database, by explicitly examining and modifying it.

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.

The naming system of Linda at this level is simple, yet quite powerful. It is also unique in that tuples are names and data simultaneously. If one views a tuple as a name, then that name is bound to the process(es) that execute a matching *in* or *read* operation. Data is passed wherever there is a formal parameter in a pattern.

Linda differs from Associative Broadcast mainly in its communication semantics and its implementation requirements. The same Linda "message" may be received multiple times by the same process, if the process executed multiple *read*s with the same pattern. Another difference is that there is no way to specify in a pattern that an element of a matching tuple should satisfy some relation with a corresponding value in the pattern. Thus the Associative Broadcast naming system is slightly more expressive in this respect. Linda also supports broadcast in the sense that several *read* operations by distinct processes may return the same tuple. However, the tuple may remain in TS forever, so the causal relationship between events in Linda may be quite complex.

While distributed implementations of *TS* have been proposed[16], Linda is better suited to a tightly-coupled multiprocessor with shared storage than to a distributed system. In contrast, the Associative Broadcast naming system has the same properties under our taxonomy as that of Linda, without requiring a logically shared data structure like the tuple space.

AP3 OS For the sake of completeness, we mention here another model that is less well-defined than Linda, but that is similar in concept. [7] describes an operating system in which processes communicate via a centralized database. The behavior of a process is specified by patterns and corresponding actions. When the database enters a state that satisfies a pattern, the associated action is performed by the process. Actions may modify portions of the database atomically. This model is in effect a production rule system applied at the operating system level. Since it relies on a centralized database, the model is not well suited to distributed architectures.

5.3 Naming Theory

Even though there have been some theoretical treatments of naming in general, the subject has not been extensively studied, especially with respect to communication. Certainly there is no theory that claims to formalize all the properties of a majority of existing naming systems. As we mentioned earlier, what work has been done in this area is concerned with formally describing resolution mechanisms, rather than the communication semantics of naming systems. This is not entirely surprising, because, like the development of broadcast communication, research in naming has been motivated from below by problems at the syntactic level. That is, whether or not there exists a clear understanding of the expressive power of naming systems, it is always necessary to be able to resolve names to objects. Thus, in some ways, understanding how naming systems work has been a more urgent problem than understanding what they can do. Our purpose in developing the taxonomy is to take the first steps toward understanding the latter. We review previous work in naming theory here mainly to show how it differs from our approach.

Early contributions in this area were made by Saltzer [55], whose lengthy and expansive treatise is one of the first recognitions of naming as a problem worthy of being studied independently. Although it does so informally, this work defines many of the terms and concepts important to understanding naming systems. Specifically, Saltzer clarifies the essential properties of names, bindings, contexts, and resolution, by stating that a name is a symbol that is used to refer to an object, a binding is "a mapping from a name to a particular object," a context is "a particular set of bindings," and that resolution is the process of locating an object, given a name and a particular context. While these definitions may seem trivial, many of these terms had previously only been defined with respect to particular implementations of naming systems.

Based on these terms, the remainder of the paper illustrates three important concepts. The first is that it is advantageous to approach naming problems abstractly, so that the only important characteristics of an object are the names that are bound to it. This distinguishes between naming and other issues. The second is the concept that bindings may be grouped into contexts, and that different resolution mechanisms may be applicable in each of a collection of contexts. This gives an indication of how naming systems might be structured and modularized. Finally, it is pointed out that names might be bound not only to objects, but to contexts as well, thus providing multiple levels of indirect reference. This last concept is in fact the focus of the majority of subsequent work, as it introduces many potential problems to the resolution process.

The notion of naming indirection was first formalized in [50], which models relationships between objects, names, and contexts as directed graphs. Each node in such a graph represents an object or a context. Each edge is labeled with exactly one name. The set of bindings that make up a context are thus represented by the set of outgoing edges of the corresponding node. Since objects may have outgoing edges there is an implicit "local context" for each object. The indirect name used by one object to refer to another is the concatenation of names labeling a path from the first object to the second. A requirement on the structure of these graphs is that each outgoing edge of a node has a unique label.

This model is then used to describe two types of graphs: *relative* and *absolute*. A relative graph is one in which there is at most one uniquely labeled edge from each node to each other node. A name in this graph might be globally ambiguous, but may be disambiguated by indicating the "source" node relative to which the name is to be interpreted. An absolute graph has a distinguished node called the "root," from which exactly one uniquely labeled edge leads to every other node. This is a representation of a "flat" name space. The authors discuss the relative merits of these two properties with respect to "The Clearinghouse," a distributed name server which is the main point of the paper. No other possible types of graphs are discussed, nor are any resolution mechanisms formalized.

More recently, a general strategy for formally describing resolution mechanisms was discussed in [19], and extended in [14] . This approach used function composition to define both the set of bindings and the resolution mechanism. In this model, a context is a function from names to (context, name) pairs, where the context portion may be null, in the case that the name portion uniquely identifies an object. The *universal resolution function*, f_R, maps (context, name) pairs to objects, while the *closure mechanism*, f_C, maps the current state of a computation to a context. Finally, the resolution mechanism is the composition $f_R((f_C(state), name))$. That is, the current state is mapped to a context, and the name to be resolved is mapped to an object using that context. The process of formalizing resolution thus amounts to formally defining these three types of functions. Several examples are given, applying the model to the Internet[46] addressing scheme, among others. However, no attempt is made to state general properties of naming systems and the communication patterns representable by contexts. In fact, the authors explicitly state that this is a separate and complementary part of the "naming problem," which they deliberately avoid.

The beginnings of a comprehensive theory of names were outlined in [25]. This paper also represents objects, contexts, and names using a directed graph. A naming system consists of such a *name graph*, plus a *naming convention*, a *resolution function*, and a set of *naming invariants*. The naming convention is a rule for adding and deleting names and bindings, while the resolution function maps names to objects. The naming invariants are assertions that describe the structure of the name graph, and hold both before and after the addition or deletion of a name or binding. For example, the relative and absolute properties described above are typical naming invariants in this model. While this approach has merit, it is not intended to describe naming relationships among elements of a concurrent computation, but rather sequences of graph transformations, such as might describe the maintenance of the symbol table in a compiler.

Finally, our hierarchical taxonomy of naming systems is reminiscent of the hierarchy of synchronization mechanisms presented in [58]. An extensive survey of distributed programming languages, which served as the starting point for our classification of naming systems, may be found in [5].

Chapter 6

Classifications

The taxonomy defined in chapter 3 provides a framework for the systematic classification and comparison of naming systems. We now use the taxonomy to rank according to their expressiveness the naming systems of a range of representative concurrent systems from the literature. The first part of this chapter consists of a traversal of one of the paths through the hierarchy from the category **bdmasr**, ($\perp$), to **BDMASR** ($\top$). This traversal illustrates the successive increases in expressiveness denoted by the *subsumes* relation, $\sqsupseteq$.

At each step in the path traversal we show that the naming system property by which the current category dominates its predecessor is required to solve a particular problem. That is, for each such category, C, and problem, P, no solution to P exists under any naming system that is a member of a category properly subsumed[1] by C. Put another way, C is the least expressive category of naming systems along the traversed path under which a solution exists. The naming system requirements of the problems we discuss arise because of

[1]Proper subsumption is the irreflexive form of $\sqsupseteq$.

constraints that we place on the form of the solution. These constraints are not artificial, but ensure that the solution has such characteristics as symmetry, minimal communication, and absence of polling.

The path traversal ends with the classification of Associative Broadcast. In the balance of the chapter we classify several additional naming systems. While the naming system of every concurrent system is a member of exactly one category of the taxonomy, and we classify a diverse range of well-known systems, we do not aspire to an exhaustive classification of existing naming systems. However, the examples presented here are sufficient to illustrate the following points:

- *Depth:* There exists a continuous range of naming systems from category $\bot$ to $\top$.

- *Breadth:* There exist naming systems in categories I and J such that neither $I \sqsupset J$ nor $J \sqsupset I$. That is, the hierarchy is not sparse.

- *Significance of properties:* The distinction between categories that differ in a given naming system property, p, is meaningful because there exists a class of problems whose solution requires p.

- *Systematic choice:* The taxonomy systematizes the process of choosing a naming system that allows communication to be specified at a level of abstraction appropriate to a given problem. Having classified a set of naming systems and found the naming system properties required by the problem, we may immediately identify within the hierarchy the sublattice(s) of categories containing naming systems that have the required properties.

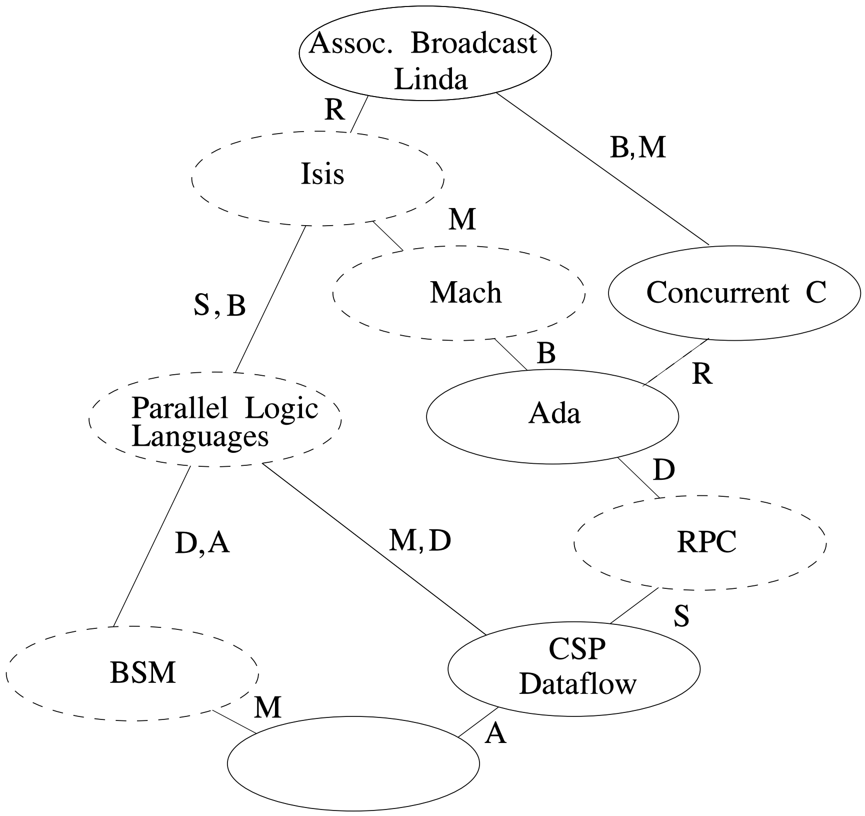

Figure 6.1: A portion of the hierarchy

- *Classification of Associative Broadcast:* Associative Broadcast resides in category $\top$, subsuming the properties of all categories in the taxonomy.

Figure 6.1 shows the portion of the hierarchy containing the naming systems discussed in this chapter. Lines between categories are labeled with the properties by which the categories differ. Solid outlines indicate the categories visited in the path traversal.

6.1 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.

6.2 Classification of CSP

Communicating Sequential Processes[36] is a concurrent system 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.

6.2.1 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 $\{(P, d, R), (Q, e, R)\}$.

6.2.2 Properties

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

- *Static binding:* A name $n = (i, c, j)$ appears in the binding of an event, e, only if 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.

- *Static domains:* For any two events e, f, if $e.o = f.o$, then $e.d = f.d$, because all identifiers are static.
- *Singularity:* 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.
- *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 $n = (P, c, Q)$, $m = (P, d, Q)$. That is, P may refer to Q using two distinct constructors.
- *Private names:* For any name $n = (i, c, j)$, $n \in e.d$ only if e is an event of the process whose identifier is i. Therefore, for any two communications (e, n, f) and (g, n, h), $e.o = g.o$.
- *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.

6.3 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 concurrent systems in the literature whose naming systems reside in $\perp$. 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. As the following example illustrates, aliased names are necessary to support nondeterminism in communication.

6.3.1 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 according to the following conditions:

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[29]. 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. The analysis in [29] attributes a "polling bias" to arbitrary restrictions on the Ada rendezvous mechanism. However, as we show here and in a subsequent example, polling is an undesirable behavior that is often attributable to fundamental naming system properties, of which, for example, the CSP or Ada naming systems are only specific manifestations. Thus, when attempting to identify the source of a restriction on the specification of communications, we should look beyond the syntax and semantics of a particular concurrent system to the fundamental naming system properties that it implements.

6.3.2 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.

6.3.3 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. The complementary view is that a more expressive naming system raises the level of abstraction at which the communication requirements of the algorithm are specified, thereby hiding the details of their implementation.

6.4 Classification of Ada

Ada[24] is a language designed from the beginning with concurrent programming in mind. It addresses some of the perceived limitations of CSP. Ada allows *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.

6.4.1 Elements

Communication in Ada is by means of the rendezvous[2], 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 is to execute the *accept* action for the specified entry.

- *Objects:* Each object is a distinct Ada task.
- *Names:* Let I be the set of possible task identifiers, and E the set of possible entry identifiers. Then an Ada name is a pair, (*destination-task-id, entry-id*) from the set $I \times E$. This differs from CSP in that the source task is not identified.

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

- *Communications:* A communication models the execution of an entry call action and a corresponding accept action. Let c be the execution of a call to entry e in task T, and let a be the execution of the corresponding accept action for e by T. Then the computation contains the communication (c, n, f) **iff** n is the identifier pair (T, e).

- *Domains:* 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, f, of task S **iff** e is an entry-id fixed by the program text of S, and T is a declared-task-id fixed by the program text of S, or T is a created-task-id assigned to a local variable of S at the occurrence of f.

- *Bindings:* 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 $\{(T, d), (T, e), \ldots\}$, where T is the identifier of the task at which the event occurs, and $d, e, \ldots$ are the entries specified in the accept action.

6.4.2 Properties

The Ada naming system is a member of category **bDmASr**:

- *Static binding:* If a name $n = (T, e)$ appears in the binding of an event, a, then the identifier of $a.o$ is T (see *Bindings*, above). Since task identifiers are unique, all events a where $n \in a.b$ are collocated.

- *Dynamic domains:* The identifiers of created tasks may be assigned to local variables. Therefore, upon the assignment of a created task identifier, T to a variable at event g, it may be that $(T, e) \in g.d$ for some entry identifier e of task T, but $(T, e) \notin f.d$ for all events $f \mapsto g$, $f.o = g.o$. 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.

- *Singularity:* From *Static binding*, above, we have $f.b \cap g.b = \emptyset$ for two events f and g, if $f.o \neq g.o$. Therefore each name is bound to at most one object.

- *Aliased names:* For a task, T containing entries d and e, there exists a computation containing two communications (c, n, a) and (x, m, y), such that $n = (T, d)$, $m = (T, e)$, a and y are executions of actions of T, and $c.o = x.o$. That is, one task may call another using two distinct entries ((T, d) and (T, e) may coexist in a domain).

- *Shared names:* For a task, T containing entry e, there exists a computation containing two communications (c, n, a), and (x, n, y), such that $n = (T, e)$, a and y are executions of actions of T, and $c.o \neq x.o$. That is, two distinct tasks may call a third using the same entry.

- *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 accept action specifies a finite set of entries. Therefore, in a behavior $e_0 \ldots$, $\bigcup_i e_i.b$ is finite.

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

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 private names property is realized. Examples of systems in which communication among objects is specified using private names include dataflow languages[2] (streams), parallel logic languages[61] (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.

6.6 Classification of Concurrent C

Concurrent C[28] is an extension of the C language[39] that supports distributed programming. It resembles Ada in that a computation consists of a dynamic set of processes that communicate using the rendezvous mechanism[3] through *transactions* (equivalent to entries in Ada).

6.6.1 Elements

A process initiates a communication by executing a *transaction call*. A transaction call specifies two static identifiers that indicate the called process and a transaction within that process respectively. The transaction call is completed when the destination process executes the *accept* action for the given transaction. 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.

- *Objects:* Each object is a distinct process.
- *Names:* 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. Let I be the set of possible process identifiers, T the set of possible transaction identifiers, and A the set of possible parameter lists. Then a Concurrent C name is a triple, (*process-id, transaction-id, parameter-list*) from $I \times T \times A$.

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

- *Communications:* A communication models the execution of a transaction call action and a corresponding accept action. Let c be the execution of a call to transaction t in process P, and let a be the execution of the corresponding accept action for t by P. Then the computation contains the communication (c, n, f) **iff** n is the identifier pair (P, t).

- *Domains:* A name (P, t, a) is in the domain of an event, e, of process Q **iff** P and t are fixed by the program text of Q or assigned to local variables of Q at the occurrence of e, and a is a list of values of local variables of Q, at the occurrence of e.

- *Bindings:* 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** $n = (P, t, a)$, where a is a parameter list for which *expr* evaluates to true.

6.6.2 Properties

The Concurrent C naming system is a member of category **bDmASR**:

- *Static binding:* If a name $n = (P, t, a)$ appears in the binding of an event, e, then the identifier of $e.o$ is P (see *Bindings*, above). Since process identifiers are unique, all events e where $n \in e.b$ are collocated.

- *Dynamic domains:* The identifiers processes and transactions may be assigned to local variables, and names have a component consisting of variable values (parameters). Therefore, upon the assignment of a variable at event f, it may be that a name $n \in f.d$ but $n \notin e.d$ for all events

$e \mapsto f$, $e.o = f.o$. 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.

- *Singularity:* 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.

- *Aliased names:* For a process P containing transactions t and s, there exists a computation containing two communications (c, n, e) and (x, m, f), such that $n = (P, t, a)$, $m = (P, s, b)$, e and f are executions of actions of P, and $c.o = x.o$. It may also be the case that $s = t$ and $a \neq b$. That is, one process may call another using two distinct transactions, or may call a single transaction using distinct parameter lists.

- *Shared names:* For a process, P containing transaction t, there exists a computation containing two communications (c, n, e), and (x, n, f), such that $n = (P, t, a)$, e and f are executions of actions of P, and $c.o \neq x.o$. That is, two distinct tasks may call a third using the same entry.

- *Descriptive reference:* In the action *accept t(...) suchthat (expr)*, *expr* may refer to the local variables of the process executing the action. Therefore the set of names that is bound to an object at an event may depend upon the local state of the object. This local state may be at least countably infinite. Therefore, in a behavior $e_0 \ldots$, $\bigcup_i e_i.b$ may be countably infinite. Furthermore, references to *expr* may occur intermittently at a unique object.

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 6.8 we discuss naming systems that combine descriptive reference with dynamic binding and multiplicity.

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

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[29]. 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. We illustrate this distinction with the following example.

6.7.1 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 6.3.1, 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:

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.

6.7.2 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 6.3.3: 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[29] of 99 entries in Ada, one for each possible value of x.

6.8 Classification of 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[31] (described in section 5.2.3) and Associative Broadcast, 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. This shows that the set of properties that constitutes the category $\top$ can be implemented in the absence of logically shared data structures.

6.8.1 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.

6.8.2 Properties

$\top$ differs significantly from the categories discussed earlier in that it combines the descriptive reference property with 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.

- *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 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 set of tuples that match t. Therefore, the set of names bound at the occurrence of such an action is infinite. Also, actions may uniquely and intermittently refer to pattern t. 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. Furthermore, a given set of attributes may exist intermittently in the profile of a unique object.

6.9 Category $\top$ (Associative Broadcast) vs. bDmASR (Concurrent C)

While the Concurrent C naming system has the descriptive reference property, it is insufficient for describing *sets* of objects, and for allowing the membership of these sets to vary. In the following section we give an example of a problem that requires multiplicity and dynamic binding in addition to descriptive reference.

6.9.1 A Distributed Cycle-Detection Problem

The manager object specified in section 6.7.1 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 member-

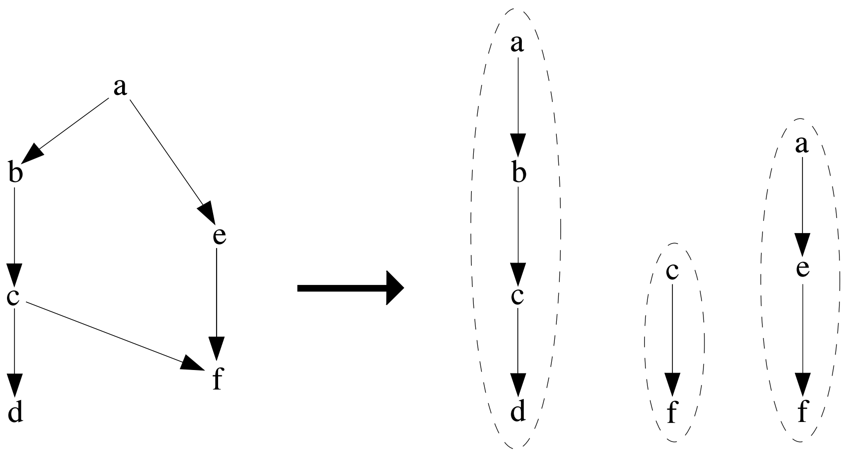

Figure 6.2: Partitioning of edges among objects

ship 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 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 6.2). 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[23]. 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.

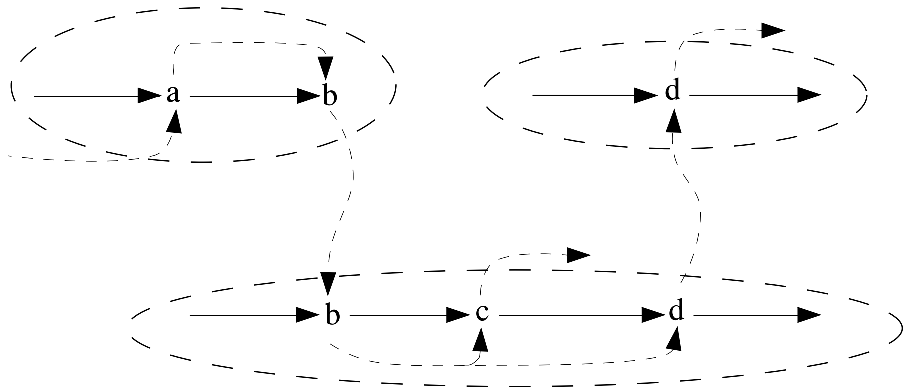

Figure 6.3: Probe propagation

1. There are two types of communications: (**add**, (u, v)) is a communication from the environment to 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 6.3). 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.

This problem requires all the properties of category $\top$. Descriptive reference is necessary because of the requirement for minimal communication: Each object maintains a subset of an infinite set of possible nodes. The proof of this requirement is similar to the proof given in section 6.7.2. 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. Dynamic binding is necessary 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). The need for dynamic domains arises from the dynamic structure of the graph. Aliased names are required because one object may need to send several distinct probes ("entering" at distinct nodes) to another object. Similarly, shared names are required because several objects may need to send identical probes to a given object.

6.9.2 Algorithm

To show that the properties of $\top$ are also sufficient to solve the cycle detection problem, we specify a solution using 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)], (probe, n, p));
        }
    }
}
```

Here again the principle of expressiveness applies. The descriptive reference, dynamic binding, and 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.

We revisit the cycle detection problem in a more complex form in section 7.2.

6.10 Additional Classifications

Several of the categories bypassed in our earlier traversal of the hierarchy contain the naming systems of some important and well-known concurrent systems. While the list of systems discussed in this chapter is by no means exhaustive, the following classifications, in combination with those presented earlier, cover a majority of the important concurrent programming paradigms, and illustrate the breadth of the hierarchy. Where naming systems exhibit properties similar to those systems discussed above, we omit some of the details of the classification.

6.10.1 BSM (bdMasr)

It is perhaps significant that we have encountered only one concurrent naming system in the literature, called Broadcasting Source-Addressed Messages (BSM, described in section 5.1.3), whose naming system has the unaliased names property. Without aliased names, the naming system is unable to support nondeterminism in communication between a single source/destination pair, a fundamental requirement for general-purpose concurrent systems.

- The BSM naming system is distinguished from all others that we have encountered by the unaliased names property. The unaliased names property in BSM is due to the requirement that each sender/receiver pair carry out communication over a single, globally unique name.

- The structure of a computation is further restricted by the fact that each receiver "listens" for messages from a single sender throughout the entire computation. Therefore, BSM not only has the static binding property, but there exists only a single name for each object. This limits the naming system to nondescriptive reference. The set of communication dependencies among objects forms a tree.

- The naming system differs from $\perp$ in the multiplicity property. For each name there may be several listeners; such a name is therefore bound to a set of destination objects.

- Domains in BSM are static because the name associated with the sender is fixed.

- Names are private because each is associated with exactly one sender.

6.10.2 Dataflow Languages (bdmAsr)

In a dataflow language[15], a computation is specified by a set of *operators* interconnected by *data streams*. Each such stream transmits data values produced on an *output* of an operator to the *input* of another operator, so that exactly one stream is connected to each output and each input. The action of an operator is governed by its *firing rule*, which is a predicate over the state of the inputs and outputs of the operator. When the firing rule is satisfied, the operator consumes values from its inputs and produces values on its outputs. Operators may execute in parallel. In such a computation, the operators are the communicating objects, and a communication is the transmission of a data value from an output to an input. A dataflow name is an (output, input) identifier pair that specifies a stream. A name (o, i) is in the domain of an object **iff** o is an output of the object. Similarly, (o, i) is bound to an object **iff** i is an input of the object.

- Domains and binding in dataflow computations are both static because each input and each output is statically associated with its operator, and new (output, input) pairs are not created.
- Because each output or input identifier appears in exactly one pair, the singularity and private-names properties hold.
- Names are non-descriptive because the binding of an event is limited to the fixed set of names corresponding to the operator's inputs.
- Aliased names are allowed because there may exist two names (o, i) and (p, j) such that o and p are outputs of a single operator, and i and j are inputs of a single, distinct operator.

While dataflow operators are typically less powerful objects than CSP processes, the dataflow naming system occupies the same category as that of CSP. This allows us to conclude that any advantages in expressiveness that CSP may have over dataflow languages do not derive from its naming system.

6.10.3 Remote Procedure Call (bdmASr)

The remote procedure call communication mechanism[13] is commonly used in distributed operating systems where server processes execute operations in response to requests from client processes. Operations are defined as procedures within the server processes, similar to entries in Ada and transactions in Concurrent C. RPC names consist of the procedure identifiers. RPC differs from CSP and Ada, respectively, in that it permits shared names (several clients may call a single procedure), but allows only static domains (references to remote procedure identifiers are fixed by the program text).

6.10.4 Mach (BDmASr)

Mach[1] is an operating system kernel for distributed systems that supports interprocess communication based on *message ports*. A port is a data structure associated with a single process through which the process receives messages. In the Mach naming system, names consist of port identifiers.

- Ports may be created dynamically, and port identifiers may be passed among processes in messages. Therefore the dynamic domains property holds.
- Dynamic binding holds because Mach allows ports to be detached from a process and assigned to another process.
- Multiple ports per process are permitted, and several processes may send messages to a single port. Therefore Mach allows both aliased names and shared names. However, each port is associated with exactly one port, therefore singularity holds.
- While the set of ports associated with a process during a computation is potentially infinite, and a port may be intermittently bound to a process, once a port is no longer attached to any process it is discarded, and

cannot be recreated for the duration of the computation. This violates the intermittence condition of the descriptive reference property, which therefore does not hold.

6.10.5 Parallel Logic Languages (bDMAsr)

Logic programs are sets of clauses of the form

$$A(x_0, \ldots, x_n) : -B(x_i, \ldots, x_j), C(x_k, \ldots, x_l), \ldots$$

where A is known as a *goal* and $B, C, \ldots$ are known as the subgoals of A. Clauses are evaluated by *unifying* (setting equal the values of) the arguments of the goal with the like-named variables of each subgoal. Where an argument contains a value (is *instantiated*), that value is assigned to corresponding variables in each subgoal. If an argument is *uninstantiated*, the corresponding variables are left uninstantiated as well. Subgoals containing uninstantiated variables then become goals that are recursively evaluated using the existing set of clauses. This process *succeeds* when all variables in a clause become instantiated, or fails when no matching clause is found.

In a parallel logic language[61], subgoals are evaluated in parallel. Each such evaluation is logically a distinct process, although in practice one process may carry out a chain of recursive evaluations of a particular clause. Since each variable in a logic program execution becomes instantiated (gets a value) at most once, they are known as *single-assignment* variables. Instantiation of single-assignment variables constitute the communication mechanism in the execution of parallel logic languages. For example, in the parallel evaluation of subgoals $B(x), C(x, y)$, the evaluation of B may cause x to be instantiated.

In that case value of x is read by the process evaluating C. The variable x is thus "shared" by the evaluator of B and the evaluator of C. To implement a *stream* of values from the evaluator of B to the evaluator of C, the value of x may be set to a pair, (v, x'), where v is a value and x' is the name of a new, uninstantiated variable. The subsequent evaluation of B now instantiates x' as (w, x''), the evaluation of C reads x', and so on.

In the execution of a parallel logic program, names are variable identifiers, and objects are the clause evaluators. For each pair of events, e, f, where e is the instantiation of a variable and f is a use of the instantiated variable, there exists in the computation a communication (e, f). A name is in the domain of an event **iff** the corresponding variable may be instantiated at that event. A name is in the binding of an event **iff** the corresponding variable is read at that event.

- For each set of communications arising from the instantiation of a variable, there exists exactly one source event, because the variable is instantiated at most once. Therefore there do not exist two communications over a given name having distinct source events. Thus the naming system has the static binding property because names are never "reused."

- The dynamic domains property holds because new variable names may be created (see *streams*, above), and passed among objects to be used in subsequent communications.

- An instantiated variable may be "read" by more than one destination object. Therefore the multiplicity property holds.

- An object may instantiate several variables, all of which are subsequently read by a single, distinct object. Thus the naming system has the aliased names property.

- a variable that has been instantiated by one object may not be instantiated by any other object. Thus variable names are not shared among objects.

- While an object may read a potentially infinite set of variables during a computation, an object reads a given variable at most once. Therefore, the intermittence condition of the descriptive reference property is not satisfied, and nondescriptive reference holds.

6.10.6 ISIS (BDMASr)

Communication in the ISIS system (described in section 5.1.1) is based upon the concept of the process group. A process may join or leave any of a set of process groups during an execution. A process may send a message to a process group, or to an individual process. In the former case, the message is delivered to each member of the group. The objects in the ISIS naming system are processes, and names are either process identifiers or group identifiers. A name that is a group identifier is bound to a an object exactly when the object is a member of the group.

- The naming system has the dynamic binding property because the membership of a process group may change. Therefore, successive communications over a given group name may have distinct sets of objects as their destinations.

- The dynamic domains property holds because process identifiers may be passed among processes in messages.

- A single source event may originate a set of communications over a process group name. Therefore multiplicity holds.

- A process may be a member of several groups concurrently. A single source object may originate communications over process group names that have members in common. Therefore, an object may be the destination of several communications over distinct names from a single source object, and the aliased names property holds.

- Names are shared because distinct source objects may originate communications over both process and group names.

- Nondescriptive reference holds because the number of process groups is fixed, and process identifiers are statically bound names. Therefore the number of names that may be bound to an object over a behavior is bounded.

Chapter 7

Associative Broadcast Applications

We originally motivated the design of Associative Broadcast by noting the existence of a class of problems that require dynamic, *1–N* patterns of communication if their solutions are to be algorithms that are distributed, symmetric, and in which communication occurs only between those objects among which dependencies exist. We can now characterize this class more precisely as consisting of those problems that, under conditions of symmetry, decentralization, and minimal communication, have solutions only under naming systems that are members of category $\top$. We call these *category-*$\top$ problems.

In the preceding chapter we discussed a number of problems (*category-*$\overline{\top}$ problems) that each required one or two of the constituent properties of $\top$. If we remove one or more of the conditions of symmetry, decentralization, or minimal communication, category-$\top$ problems are then solvable under under some less expressive naming system. However, the difficulty of specifying such a solution is typically much greater than experience with category-$\overline{\top}$ problems would suggest. This is because in removing the symmetry, decentralization, or minimal communication conditions and choosing a less expressive naming system, we are often forced to specify as part of our algorithm a protocol that performs the functions that would otherwise be performed by the naming system. Therefore, by imposing these conditions, we not only obtain solutions that have desirable properties, but we are forced to express our solutions using a naming system that supports a set of communication abstractions that make the solutions easier to specify.

We briefly introduced a category-$\top$ problem, cycle-detection in a dynamic graph, in section 6.9.1. We now specify as solutions to four additional problems the following algorithms, using Associative Broadcast:

- A distributed truth-maintenance system, in which objects communicate to maintain the global consistency of a set of equations of the form $X = expr$. The algorithm allows dynamic modification, repartitioning, and replication of equations.
- A distributed optimistic protocol for maintaining consistency in replicated databases in the presence of network partition failures. This is a particular application of the generic cycle-detection algorithm discussed earlier, and eliminates the global freezing required by its centralized[22] counterpart.
- A distributed constraint solver for arbitrary sets of constraints, which may contain cyclic dependencies. This generalization of the truth-maintenance algorithm also allows allows dynamic modification, repartitioning, and replication of constraints.
- A virtual-time synchronization protocol that does not suffer from the transitivity delay and incremental rollback of Time Warp[38] protocol.

One surprising property of several of these algorithms is that they continue to behave correctly when components of the program are replicated. A generalization of this "side-effect" of using Associative Broadcast leads to the result we present in chapter 8: design criteria for replicating program components without introducing additional synchronization.

7.1 A Truth Maintenance System

Consider a set of equations, each of which has the form

$$var = \langle expr \rangle$$

where *var* is a single variable, and $\langle expr \rangle$ is an expression over the variables appearing on the left side of an equation. We say that the equation *defines* *var*, and *depends upon* the set of variables appearing in $\langle expr \rangle$. The equations may be of any length or form, subject to the following conditions:

- Each variable appearing in $\langle expr \rangle$ appears on the left hand side of exactly one equation.
- There does not exist a set of equations $\{z_0, ..., z_{n-1}\}$ such that z_i depends upon $z_{i+1 \,\mathbf{mod}\, n}$

The first condition ensures that all variables that are *used* (appear in an expression) are defined, and that variables are not multiply defined. The second condition disallows cyclic dependencies. A set of equations that satisfies these conditions is said to be *consistent*.

The purpose of the truth maintenance system (TMS) is to allow this set of equations to be modified and extended incrementally, while keeping the values of the variables consistent. A typical example of a truth maintenance system is a spreadsheet program. One can envision a distributed implementation in which a user at one site may enter or modify equations that depend upon equations at other sites. The task of the TMS program is then to obtain the necessary variable values from the remote sites to bring the values of the locally-defined variables up to date.

We implement the TMS by partitioning the set of equations among a set of objects. The system of equations is manipulated by user objects, which send messages to the TMS objects instructing them to add new equations or modify existing ones. Initially, the TMS is in an "empty" state where no equations have yet been added. Subsequently, the user objects send a finite number of instructions that add equations or modify ones that have previously been added in such a way that new dependencies are created. It is assumed that user objects send only valid instructions. That is, no instruction is sent that creates a cyclic dependence, and after all instructions have been sent, there exists exactly one equation that defines each of the variables that is used in an expression. Note however that we do not disallow "forward" references—an instruction may be sent that uses a variable before[1] an instruction that defines that variable. We only require that a message defining the variable is eventually sent. It would be straightforward to extend the TMS to allow detection of cyclic dependencies. However this would complicate the example unnecessarily.

[1]in the sense that the arrival of the former message may precede the arrival of the latter, in accordance with the definition of $\mapsto$.

7.1.1 Object Specification

In the following object definition, we adopt the convention that `v` refers to the value of variable `v`, while `"v"` refers to the variable name itself. The state variable of an object is a set, X, of equations that it maintains, which is initially empty. The initial profile of each object is

$$\{self, \texttt{CHANGE}, \texttt{REQUEST}, \texttt{UPDATE}, (\texttt{DEFINES}, \emptyset), (\texttt{DEPENDS}, \emptyset)\}$$

where `CHANGE`,`REQUEST`, and `UPDATE` are type attributes bound to the corresponding operations defined below. The value of `DEFINES` is the set of TMS variables defined by the equations maintained by the object whose values are currently known, and the value of `DEPENDS` is the set of all TMS variables appearing on the right hand side of these equations. Both are initially the empty set, $\emptyset$. *self* is a unique identifier referred to only by user objects.

```
change(v, expr) {
    add or replace v = expr in X;
    if value of v can be computed {
        DEFINES := DEFINES∪ {"v"};
        broadcast ([UPDATE∧ v ∈ DEPENDS], (v, value_of(v)));
    }else
        DEFINES := DEFINES− {"v"};
    foreach new dependence "x" in expr {
        DEPENDS := DEPENDS∪ {"x"};
        broadcast ([REQUEST∧ "x" ∈ DEFINES], ("x"));
    }
}

request(v) {
    broadcast ([UPDATE∧ v ∈ DEPENDS], (v, value_of(v)));
}
```

```
update(v, val) {
    foreach equation "x" = ⟨expr⟩ {
        if ⟨expr⟩ depends on v {
            reevaluate ⟨expr⟩ using val;
            if value of x can be computed
                DEFINES := DEFINES ∪ {"v"};
            if x has changed²
                broadcast ([UPDATE ∧ "x" ∈ DEPENDS],("x",x));
        }
    }
}
```

The computation proceeds as follows: When an object receives a message instructing it to change one of the equations that it maintains, or to add an equation to the system, it makes the necessary local changes. If the value of the variable defined by the equation is known, the variable is included in the object's DEFINES attribute and an update message containing the value of the variable is broadcast, otherwise it is excluded. For each new dependence that is created, the object broadcasts a request to be informed of the value of the variable on which a local equation depends, after adding that variable to its DEPENDS attribute.

Upon receiving a request message, the object broadcasts the value of the requested variable to all those objects that contain equations depending on it.

When an update message is received, the object reevaluates each of its equations that depend on the variable contained in the message. If this results in a new value for any of the variables defined by the object's equations, it broadcasts each newly discovered value to all those objects that depend on it.

7.1.2 Correctness

We state the correctness condition that our program must satisfy as follows:

Theorem: Given a finite number of messages sent by user objects in finite time, the computation terminates in finite time with a consistent set of equations. $\square$

A terminating computation is one in which there exists an event at each object that is preceded by all other events at that object. In order to prove the above correctness condition, we need the following lemma:

Lemma: Let c be the last message receipt event at object o, such that $c.m$ adds or modifies the equation $v = \langle expr \rangle$. For each variable x in $expr$, if the value of x is not known to o, then there exists a message receipt event u such that $u.o = c.o = o$, $c \mapsto u$, and $u.m$ updates o with the value of x. $\square$

Proof: (by induction) We define a *maximal dependence chain* as a set of equations $\{z_0, ..., z_n\}$ such that z_i depends upon z_{i-1}, $0 < i \leq n$, and z_0 does not depend upon any other equations. Since the system of equations is acyclic, for each dependence in an equation, the equation is a member of at least one such chain.

Basis: The lemma holds trivially for z_0.

Induction: Assume the lemma holds for z_i. We show that it holds for z_{i+1}.

- Let h be the object that defines x in equation z_i. When o receives $c.m$, if the value of x is not known to o, it adds x to its `DEPENDS` attribute and broadcasts a request for the value of x.

- Let a be the profile alteration event, b the broadcast of the request, and r the arrival of the request at h. If h receives the request, then it broadcasts an update message containing the value of x. Let v be the broadcast of this message and w its arrival at o. Since $a \mapsto b \mapsto r \mapsto v \mapsto w$, $a \mapsto w$ by transitivity of $\mapsto$. Therefore o receives the update.

- If h does not receive the request, then either it has not yet received a message from a user object that defines x, or there exist at h at least one unsatisfied dependence in the equation defining x. If the former is true, then h will eventually receive a message defining x (by the problem definition), and will broadcast an update containing the value of x if there are no unsatisfied dependencies in the definition of x. This update will be received by o. By the induction hypothesis, for each unsatisfied dependence there exists a receipt of an update message that satisfies the dependence. Let d be the receipt of the last such update. Since h did not receive the request sent by o, $r \mapsto d$. Upon receiving $d.m$, h broadcasts an update message containing the value of x. Let e be the broadcast of this message and f its arrival at o. Then $a \mapsto b \mapsto r \mapsto d \mapsto e \mapsto f$, and o receives the update.

The lemma states that all dependencies are eventually satisfied. We now proceed with the proof of the theorem.

Proof:

1. At each object there exists an event that is the arrival of the last user message at that object. By the lemma, any dependencies created by this last message are eventually satisfied. Thus there exists at each object an event that is the last broadcast of a request by that object.

2. Since an object only broadcasts one update in response to a request, there is an event at each object that is its last response to a request.

3. Each object maintains a finite number of equations. Therefore an object broadcasts a finite number of updates in response to updates it receives. Since there are no cyclic dependencies, there is an event at each object that is its last response to an update.

4. By 1.–3. the computation terminates.

5. By 2. and 3., for each variable there is a last broadcast of an update containing the value of that variable. By the lemma, all dependencies on the variable at an object are created before the arrival of this last update. Therefore, all objects whose equations depend on this variable terminate having received this last value. Hence the system of equations is consistent. □

7.1.3 Properties of the TMS Algorithm

Several significant characteristics of specifying a distributed computation using Associative Broadcast are evident in this example. First, the solution is symmetric and distributed—all objects have the same specification, and no objects are distinguished as controllers or maintainers of a global state during execution. Second, the specification and communication behavior of all objects is independent of the partitioning of the equations among the objects. Each object may maintain a single equation, or all equations may be the responsibility of a single object, in which case it is both the sender and sole recipient of every message. This partitioning independence may be attributed to the separation of the specification of object behavior from the specification of the communication dependencies. Furthermore, it is possible to dynamically repartition the equations among objects to achieve load balancing in a distributed implementation.

Finally, the solution as given behaves correctly even if equations are replicated among objects, i.e., a copy of an equation may be maintained by each of several objects. This means that if each equation is replicated among k objects, and each of these objects resides at a unique site in a distributed system, the TMS will continue to behave correctly in the presence of up to $k-1$ site failures, assuming the non-failed sites remain mutually connected.

In addition to fault tolerance, replication may provide improved performance if equations upon which many other equations depend are replicated at several sites, assuming faster responses from objects residing at the same site as a requester.

7.2 A Consistency Protocol for Replicated Databases

In a replicated database, there exist several copies of each data item, with individual copies residing at distinct sites throughout the system. Correct operation of the database requires that the values contained in all the copies of all the data items remain mutually consistent in the presence of concurrent updates. This is usually achieved by implementing accesses (read and write operations) as atomic transactions and executing these under the control of a transaction commit protocol. There are several notions of consistency, the most common being that of *serializability*: ensuring that the database state after some set of transactions is executed is equivalent to the state obtained by executing the same transactions in some totally-ordered sequence.

When the interconnection network fails in such a way as to prevent communication between some subsets of sites (a "partition failure"), then it may be impossible to both maintain serializability and allow all transactions to proceed while the network remains partitioned. This is because copies of a data item may reside in distinct partitions, and there is no way to bring these copies into agreement if a transaction originating in one partition attempts to write a new value to the data item.

Solutions to this problem take one of two forms: either transactions that attempt to update data items, some of whose copies are inaccessible, are prevented, thus maintaining consistency (pessimistic protocols), or such transactions are permitted, thus potentially creating inconsistencies that must later be detected and eliminated (optimistic protocols). In section 7.2.3 we present a distributed version of the well-known optimistic protocol of Davidson [22] that relies upon Associative Broadcast to eliminate the centralized control and global freezing of transactions of the original algorithm, while employing minimal communication.

7.2.1 The Original Optimistic Protocol

The Optimistic Protocol is based upon the idea of constructing a precedence graph that captures the partial temporal order of all the transactions in the system between the occurrence of a partition failure and the reconnection of the network. The database is consistent **iff** this precedence graph contains no cycles. The graph is constructed as follows: a log is maintained in each partition that records the order of reads and writes on the data items, and the transaction to which each operation belongs. The set of all transactions is the set of vertices of the precedence graph. An edge $T_{xi} \rightarrow T_{yj}$ indicates that transaction T_{xi} occurred before transaction T_{yj}, where x and y indicate the partition in which the transactions were executed. Such an edge exists **iff** one of the following holds:

- *Dependence edge*: T_{yj} read a data item that was previously written by T_{xi}, and $x = y$.
- *Precedence edge*: T_{xi} read a data item that was later written by T_{yj}, and $x = y$.
- *Interference edge*: T_{xi} read a data item that was written byT_{yj}, and $x \neq y$.

The first two types of edges relate transactions within a common partition, while the third captures a conflict between transactions that access copies of a data item in distinct partitions. Since the read transaction in one partition did not read the value written by the write transaction in the other partition, the only way that the database can possibly be consistent is to consider the read to have occurred before the write. It is assumed that set of data items written by a transaction is contained in the set of items

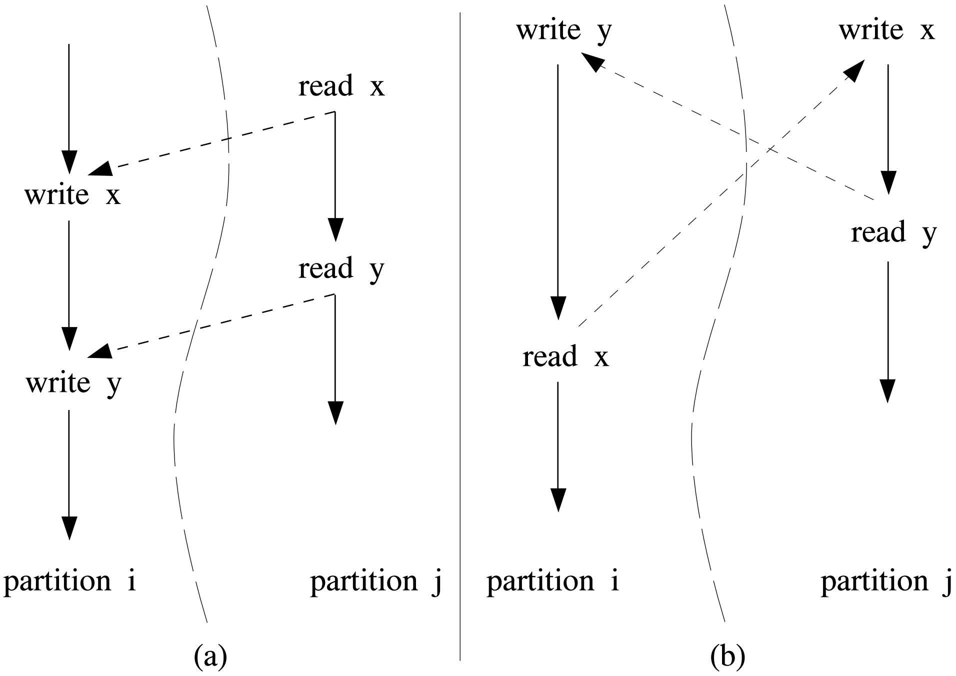

Figure 7.1: (a) Interference edges (b) A cycle

read by the transaction. Therefore, a write-write conflict across partitions is manifested as a pair of interference edges. The existence of interference edges does not imply that the database is inconsistent. However, in order for the precedence graph to contain a cycle, there must exist at least two interference edges, since it is assumed that the commit protocol prevents cycles within a partition (figure 7.1).

7.2.2 Disadvantages

The original Optimistic Protocol is centralized, relying upon a *co-ordinator* to collect the transaction logs upon reconnection. This has several disadvantages:

- *Asymmetry*: A coordinator must be chosen to construct the graph. This may require an additional leader-election protocol, create communication bottlenecks, make the system less fault tolerant, and make the algorithm more difficult to specify.

- *Log transfer*: Either during the execution of transactions or at the time the algorithm is executed, log information for each data item must be transferred to the coordinator. This transfer may be expensive. If log information is kept only at the coordinator, it may be lost due to further network partitioning or failure of the coordinator.

- *Global freezing*: While the precedence graph is being constructed, no additional transactions may be executed, since these might add cycles that are not reflected in the logs. Transaction processing must therefore be frozen at the time network reconnection is detected, and persists while the log information is first collected and then analyzed.

The above disadvantages might be avoided by distributing the graph construction algorithm. Specifically, the problems of asymmetry and expensive log transfers would not be present if the log information were stored with the data items and evaluated in place. Furthermore, freezing transactions globally while the logs are evaluated runs counter to the basic optimistic assumption that inconsistencies are rare. A truly optimistic protocol would allow transactions to proceed concurrently, so that, if no inconsistencies are eventually discovered, transaction processing is never suspended.

7.2.3 The Distributed Optimistic Protocol

We propose a new protocol based on the following observation: If each copy of a data item is maintained by a distinct object, and each such object logs only the transactions that access the object, then the set of dependence edges, as well as the set of precedence edges, is partitioned among the set of all objects. That is, each object is aware of some subset of these edges (figure 6.2). There is also sufficient information to determine the set of interference edges. We call the objects that maintain the individual copies of a particular data item *cohorts*, and the partition in which an object resides its *local partition*.

Cycles in the graph are discovered by passing tokens, or "probes," along the edges of the graph, and thus from object to object. If the graph contains a cycle, then at least one probe will reach some object twice. When a probe reaches a particular vertex at an object, it must be sent to all other objects that also contain that vertex, i.e. that were touched by the corresponding transaction. Since transactions may be in progress as the protocol executes, the set of objects that must receive the probe may be growing concurrently. We propagate probes using Associative Broadcast by adding a vertex (transaction ID) to the profile of an object when the object is touched by a transaction, and specifying the destination vertex in the selector of an outgoing probe message.

The protocol effectively operates on a set of "distributed snapshots" of the system state, even as the global state is changing, in a manner similar to the general algorithm described in [17]. The global state examined by the protocol consists of the local states of the objects at the time each first receives a probe. This global state is consistent, but need not exist at any instant. It is this characteristic that allows transaction processing to proceed concurrently

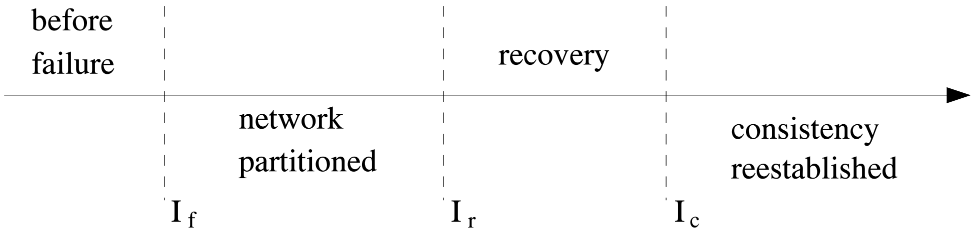

Figure 7.2: Time intervals bounded by partition failure, reconnection, and termination of recovery

with construction of the precedence graph. The notion of consistency referred to here is the one defined in [17]: A snapshot of a local state may record the receipt of a message only if there exists another snapshot that records the sending of the message. That is, messages must not appear to have been received before they are sent. This differs from database consistency, which requires transactions to be executed in serializable order.

For the expository purposes of this example, we make the following assumptions about objects, transactions, and network partition failures (figure 7.2).

- Let I_f be the time at which a failure occurs, and I_r the time of the earliest reconnection, such that $I_f < I_r$. Then there exists no distinct failure at time I_x such that $I_f < I_x \leq I_r$.

- Let I_c be the earliest time at which the database is consistent, such that $I_r \leq I_c$. Then there exists no failure at time I_x such that $I_r < I_x < I_c$.

- The failure at time I_f is detected by each object in the system during the interval $I_f \ldots I_r$.

- The reconnection at time I_r is detected by each object in the system during the interval $I_r \ldots I_d$, where $I_d - I_r$ is positive and finite.
- Each object knows the total number of its cohorts.
- During the interval $I_f \ldots I_r$, all objects in a partition agree upon a globally unique *partition ID*, and each object determines the number of its cohorts in the local partition.
- Transactions and objects have globally unique IDs.

7.2.4 Object Specification

Initially,

$\{(\mathtt{ITEM}, i), (\mathtt{READS}, \emptyset), \mathtt{PROPAGATE}, \mathtt{MERGE}\}$

is the profile of each object, where i identifies the data item contained in the object, and `PROPAGATE` and `MERGE` are type attributes bound to the corresponding operations defined below. The value of `READS` is the set of IDs of transactions that have read or written[3] data item i, initially the empty set. The simple attribute `WRITTEN` is added to the profile when data item i is modified. The `PARTITION` attribute identifies the partition in which the object resided prior to reconnection.

In the following code, the variables *mcount*, t_0, t_w, t_n, D, and P represent the number of `MERGE` messages received by the object, the first logged transaction, the first logged write transaction, the last logged transaction, the local copy of the object's data item, and the object's unique ID, respectively.

---

[3]Recall that the readset of a transaction contains its writeset.

```
initially
    mcount = 0;
    merged = FALSE;

Upon_partitioning() {
    x := local_partition_id;
    profile := profile ∪ {(PARTITION, x)};
}

Upon_committing_transaction(t) {
    log t;
    READS := READS ∪ {t};
    if t modifies D then profile := profile ∪ {WRITTEN};
}

Upon_detecting_reconnection() {
    m := (WRITTEN ∈ profile)
    broadcast([MERGE ∧ ITEM = i ∧ PARTITION ≠ x], D, m);
    foreach t ∈ READS {
        label t with P;
        broadcast([PROPAGATE ∧ ITEM ≠ i
            ∧PARTITION = x ∧ t ∈ READS], t, P);
    }
    if (READS ≠ ∅ ∧ ¬merged) then {
        broadcast([PROPAGATE ∧ ITEM = i
            ∧PARTITION ≠ x ∧ WRITTEN], t_n, P);
    }
    if (¬merged) then delay subsequent transactions;
}
```

```
propagate(r,p) {
    if (∃ u ∈ READS labeled with p ∧ (r → u
        ∨r = u ∨ (r ∉ READS ∧(t_w → u ∨ u = t_w)))) then
        declare inconsistency;
    else if ( ∄ u ∈ READS labeled with p) then {
        foreach t′ :: t′ ∈ READS ∧ (r → t′ ∨ t′ = r
            ∨(r ∉ READS ∧(t′ = t_w ∨ t_w → t′))) {
            label t′ with i;
            broadcast([PROPAGATE ∧ ITEM ≠ i
                ∧PARTITION = x ∧ t′ ∈ READS], t′, p);
        }
        if (¬merged) then {
            broadcast([PROPAGATE ∧ ITEM = i
                ∧PARTITION ≠ x ∧ WRITTEN], t_n, p);
        }
    }
}

merge(val, mod) {
    if (mod) then
        if (WRITTEN ∈ profile ∨ currently locked for write) then
            declare inconsistency;
        else D := val;
    mcount := mcount + 1;
    if (mcount = total cohorts − cohorts in local partition) then {
        ensure all cohorts participate in subsequent transactions;
        allow delayed transactions to proceed;
        merged := TRUE;
}
```

The computation proceeds as follows. When an object is notified that a partition failure has occurred, it adds an attribute to its profile whose value identifies the partition in which it resides. When the object subsequently commits a transaction, it logs the transaction and adds the transaction ID to its READS attribute. If the transaction modifies the data item, the attribute WRITTEN is added to the object's profile.

When an object detects that the network has been reconnected, it broadcasts the value of its data item copy to all other copies in partitions other than its own, and begins an instance of the graph construction algorithm. For each local transaction, the object labels the transaction with the unique ID of the object, and traverses the precedence edges reachable from the transaction as follows: It broadcasts the transaction ID and the unique identifier of the object (the probe). This probe is received at each object in the local partition that contains an edge originating at that transaction (figure 6.3). The traversal of interference edges is effected by broadcasting a probe to all objects that contain copies of the same data item as the sender, reside in partitions that differ from that of the sender, and whose copies have been modified.

Subsequent transactions involving the object's copy of the data item are delayed until the object has received a `MERGE` message from all other copies of its data item that reside in partitions other than its own. This is necessary to prevent subsequent transactions from creating cycles. Such cycles might not be detected, since the object might not receive any more probes.

Upon receiving a probe message, an object declares a database inconsistency if it has propagated the probe before, and the transaction at which the probe enters precedes or equals that at which it previously exited. The declaration of an inconsistency causes all transaction processing to stop and a separate recovery algorithm to be run. Otherwise, if the probe has not been propagated before, the object labels all reachable local transactions with the probe, and propagates it along all those edges reachable from the vertex (transaction ID) at which the probe entered, as above.

Receiving a MERGE message causes the object to declare an inconsistency if both its copy and the received copy of the data item have been modified. It is necessary to consider a write lock as a modification, because once the transaction commits, an inconsistency will exist[4]. If the local copy has not been modified and the received copy has been, then the local copy is updated to reflect the received value. When the required number of *merge* messages have been received, any delayed transactions, as well as subsequent transactions, are allowed to proceed. We say that a set of cohorts are *merged* when the variable *merged* has the value TRUE at each cohort. Since all cohorts now maintain identical copies and participate in subsequent transactions, these transactions do not create any new interference edges between the cohorts.

In the absence of inconsistencies, the database is eventually restored to a state in which all cohorts corresponding to a given data item maintain identical copies. Since the database is consistent, there exists at most one modified copy of each data item. The unmodified copies are updated upon receipt of the MERGE message broadcast by the modified copy.

7.2.5 Correctness

The algorithm must satisfy the following three conditions:

- An inconsistency is declared **iff** a cycle exists in the precedence graph.
- The database becomes consistent within finite time after reconnection occurs ($I_c - I_r$ finite).

[4]This is because the transaction may have already read the local copy of the data item at the time the MERGE message arrives. An optimization would be to merely abort the transaction instead of declaring an inconsistency.

- No transactions remain blocked forever.

To ensure that the first condition holds, it suffices to prove the following two lemmata:

Lemma: If an inconsistency is declared, then there exists a cycle in the precedence graph. □

Proof: An inconsistency is declared in one of two ways:

- **case:** A `MERGE` message is received from a cohort, indicating that the cohort's copy has been modified. In addition, the receiving object's copy has been modified.

  In this case thee exists an interference edge from each write transaction in one object to each write transaction in the other. Therefore, there exists at least one cycle.

- **case:** A probe is received that has been propagated previously, and the transaction at which the probe enters the object precedes the transaction at which it previously exited the object.

  Since probes are propagated only along edges of the graph, the probe must have traversed a cycle. □

Lemma: If a cycle exists in the precedence graph, then an inconsistency is declared. □

Proof: Let C be a set of edges that form a cycle, and let $e \in C$ be the last edge in C added to the graph. It suffices to show that at least one probe is propagated along e. There are two cases:

- **case:** e is an interference edge. Let r be the read transaction that is the source of e, and w the write transaction that is the destination of e. Let m be an object that is read by r, and o an object that is written by w. By definition of an interference edge, m and o are cohorts.

  When an object participates in a transaction, either $merged =$ TRUE and all its cohorts participate as well, or $merged =$ FALSE and the object has not yet detected reconnection. Since no interference edges are created among merged cohorts, neither m nor o has detected reconnection at the time it participates in r or w, respectively. Hence neither m nor o has broadcast any probes. Therefore, both will eventually broadcast initial probes that visit r and w, respectively. If o receives the probe from m, then e has been traversed. If o does not receive the probe from m, then w has not yet been logged at o when the probe arrives. Therefore, r has already been logged at m when o broadcasts its initial probe. Since e is the last edge added to form a cycle, there exists a path from w to r. Therefore m will eventually receive the probe broadcast by o, and will propagate it along e.

- **case:** e is not an interference edge. Let t be the transaction that creates e at object o.

  Since e creates a cycle, there exists an interference edge, f, originating in the local partition of o, that is reachable from t. Let u be the transaction that is the source of u. Since u is reachable from t, t precedes u. By the argument given in the previous case, a probe traverses f. Since e creates a cycle and t precedes u, o will receive the probe and propagate it along e. □

We now show that the second condition holds:

Proof: If the computation terminates, then, by the first condition, either no inconsistencies exist, or at least one inconsistency is detected. If an inconsistency is detected, the separate recovery algorithm restores consistency. It therefore remains to be shown that the computation terminates when no inconsistencies exist. Each object broadcasts exactly one `MERGE` message, and originates a finite number of probes. Each object propagates each received probe exactly once to a finite number of recipients. Therefore a finite number of messages are received, and the computation terminates. □

Finally, we show that no transactions remain blocked forever:

Proof: For a transaction to remain blocked forever, a participating object would either have to block transactions repeatedly, or for at least one infinite time interval. However, each object blocks transactions at most once, and each object receives a `MERGE` message from each of its non-local cohorts in finite time. Therefore, each object blocks transactions for a single, finite interval. □

7.2.6 Properties of the Distributed Optimistic Protocol

Unlike a centralized implementation of the Optimistic Protocol, our distributed protocol allows transaction processing to continue during the interval $I_r \ldots I_c$, so long as no inconsistencies are detected. In particular, transactions that are in progress at time I_r are not aborted, as they necessarily would be if the protocol were not distributed. In addition, because transaction logs are evaluated in place, and cycle detection runs concurrently with transaction processing, the overhead of log transfer is avoided, and transaction processing is never globally frozen.

7.3 A Distributed Constraint solver

In this example we consider the application of Associative Broadcast to the problem of determining the values of a set of finite-domain variables, subject to a set of *constraints*. A constraint is a relation on the values of the variables, e.g. $x > y$ or $A \wedge \neg B$. A *state* is an assignment of values to the set of variables. A variable is *fixed* **iff** there is exactly one value of the variable such that all constraints hold. This characterization of constraint programming is an abstraction of the structure of more practical concurrent constraint languages, such as one discussed in [6, 44]. We adopt a method of propagating information among constraints similar to the "local propagation" technique used by Consul [48, 59].

The purpose of the constraint reduction system (CRS) is to allow the set of constraints to be extended incrementally, while fixing definite values for as many variables as possible. We construct a concurrent algorithm to solve this problem by partitioning the set of constraints among a set of objects. We call the set of constraints maintained by an object its *local constraints*. We say that an object *depends* upon a variable **iff** the variable appears in at least one local constraint. Each object determines a set of *locally inconsistent* states—the set of states that fail to satisfy the constraints that it maintains. We call a set of (*variable, value*) pairs that violate at least one local constraint an *inconsistency*.

Initially, the set of local constraints at each object is empty. The set is extended dynamically by user objects, which periodically send messages to the CRS objects instructing them to add local constraints. The computation terminates (no more messages are broadcast among CRS objects) when the number of locally consistent states at each object is either 0 or 1. In the former case the system is overspecified—no assignment of values to variables satisfies all constraints, while in the latter the value of each variable is fixed (each locally consistent state satisfies the global set of constraints).

7.3.1 Object Specification

The initial profile of each object is

$$\{\mathit{self}, \texttt{ADD}, \texttt{REQUEST}, \texttt{PRUNE}, (\texttt{DEPENDS}, \emptyset), (\texttt{RESTRICTS}, \emptyset)\}$$

where `CHANGE`,`REQUEST`, and `UPDATE` are type attributes bound to the corresponding operations defined below.

The value of the `DEPENDS` attribute is the set of variables upon which the object depends, and the value of the `RESTRICTS` attribute is the set of variables that belong to at least one inconsistency. Both are initially the empty set, $\emptyset$. *self* is a unique identifier referred to only by user objects.

Each object has the following specification:

```
add(constraint) {
    find new locally inconsistent states using constraint;
    DEPENDS := DEPENDS− {fixed variables};
    foreach new dependence v in constraint {
        DEPENDS := DEPENDS∪ {v};
        broadcast([REQUEST ∧ v ∈ RESTRICTS], v);
    }
    foreach new inconsistency i = {(v_0, val_0), ..., (v_n, val_n)} {
        RESTRICTS := RESTRICTS ∪ {v_0, ..., v_n};
        broadcast([PRUNE ∧ {v_0, ..., v_n} ∩ DEFINES ≠ ∅], i);
    }
}

request(v) {
    foreach inconsistency i :: v ∈ {(v_0, val_0), ..., (v_n, val_n)}
        broadcast([PRUNE ∧ {v_0, ..., v_n} ∩ DEFINES ≠ ∅], i);
}

prune(inc_val_set) {
    find new locally inconsistent states using inc_val_set;
    DEPENDS := DEPENDS− {fixed variables};
    foreach new inconsistency i = {(v_0, val_0), ..., (v_n, val_n)} {
        RESTRICTS := RESTRICTS ∪ {v_0, ..., v_n};
        broadcast([PRUNE ∧ {v_0, ..., v_n} ∩ DEFINES ≠ ∅], i);
    }
}
```

The computation proceeds as follows: when an object receives a message instructing it to add a new local constraint, it searches for new locally inconsistent states. Any variables that are fixed by the new constraint are removed from the `DEPENDS` attribute. New dependencies are added to the `DEPENDS` attribute, and requests for corresponding inconsistencies are broadcast. Subsequently, each new inconsistency is broadcast. Each such broadcast

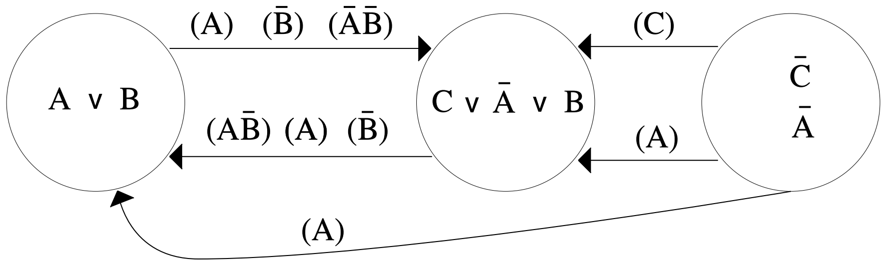

Figure 7.3: A constraint solver

is received only by those objects that depend upon at least one of the variables in the inconsistency.

Upon receiving a request message, the object broadcasts each inconsistency that contains the requested variable. When a message containing an inconsistency is received, the object searches for new locally inconsistent states, proceeding as above. Figure 7.3 shows a boolean constraint solver consisting of three components and a possible sequence of inconsistencies passed between objects.

7.3.2 Correctness

The proof of correctness is similar to our earlier proof for the truth-maintenance system. Once the last constraint has been added, termination is assured because the set of locally inconsistent states at each object can only grow smaller: Since only *in*consistencies are communicated among objects, eventually no new inconsistent states are discovered. Furthermore, any inconsistent state by definition violates at least one local constraint. The object that maintains the violated constraint eventually eliminates that state, and broadcasts the inconsistency to all objects that depend upon it. Therefore, at termination the variable values at all objects are globally consistent.

7.3.3 Properties of the Constraint Solver

The constraint solver retains the symmetry, decentralization, dynamic repartitionability, and replicability properties of the TMS algorithm discussed earlier, while allowing cyclic dependencies among arbitrary constraints. [6] describes a parallel constraint solver in which objects broadcast their entire local state to all other objects whenever a variable is fixed. In contrast, our constraint solver requires communication only between those objects that maintain constraints among which dependencies exists. As variables become fixed, the set of dependencies among constraints is reduced.

The constraint solver in [54] uses a phased state-space search algorithm to evaluate a set of boolean constraints in parallel. The value of an arbitrarily chosen "pivot" variable is fixed (to the value *true* or *false*). The objects then recursively search the subtree of states defined by the fixed variable, choosing a new pivot variable at each level of the tree and evaluating their constraints using this new value. If a contradiction is reached, the previously chosen pivot variable is inverted, and the search moves into the corresponding subtree. This algorithm requires extensive synchronization among objects, since each object must wait for all the others to complete their previous constraint evaluation before a new pivot value can be chosen (by a central controller). In our constraint solver, objects search their local states independently, and exchange intermediate results asynchronously.

7.4 A Virtual Time Synchronization Protocol

In many distributed computations it is necessary to ensure that the order in which events occur conforms to a total order that is derived from a correctness constraint on the computation. Examples of such constraints are the serializability of database transactions, or the monotonicity of timestamps in a discrete-event simulation. However, as Lamport noted in [40], the causal relationship of events (e.g. $a \mapsto b$ because a is the sending of a message and b is the receipt) yields only a partial order, because of the absence of a global clock. He proposed the *logical clocks* mechanism as a method of extending the inherent partial order to a total order. Subsequently, Jefferson[38] observed that the converse approach is to specify a total order on the events, and then to provide a mechanism that causes the events of a computation to be executed in a partial order that conforms to the specified total order. He proposes a mechanism called Time Warp that allows the programmer to specify the *virtual time* at which an event (e.g. message receipt) must occur, and then synchronizes processes to ensure that the specified order is observed.

The original Time Warp mechanism is subject to two sources of inefficiency, which we call *transitivity delay* and *incremental rollback*. In section 7.4.3 we use Associative Broadcast to specify a new virtual time synchronization protocol that overcomes these disadvantages.

7.4.1 Time Warp

Time Warp synchronizes a set of processes that communicate using asynchronous message passing. Each process maintains a local "virtual clock" that records the *local virtual time* of the process. When a process sends a message, it labels the message with a *virtual send time* equal to the local virtual time when the message was sent, and a *virtual receive time*, the virtual time at which the message is to be received. The virtual receive time must be at least the virtual send time. Each process has an *input queue* and an *output queue*. When a message arrives at a process, it is placed in its input queue. When a process sends a message, a copy of the message is placed in the output queue of the process. A process consumes messages in its input queue by choosing a message whose virtual receive time is equal to the local virtual time of the process. It then acts upon the message (possibly causing the process to send messages of its own), and sets its local virtual clock to the next smallest virtual receive time among the messages in its input queue, or ∞ if the queue is empty.

It is possible for a message to arrive in the input queue of a process whose virtual receive time is earlier (less) than the local virtual time of the process. In this case the message has arrived "in the virtual past." All of the work done by the process between its current virtual time and the virtual receive time of the new message is potentially incorrect, since it did not take the new message into account. To allow the process to undo this work, it checkpoints its state each time it increases its local virtual clock. Upon receiving a message in the virtual past, the process rolls back to its most recent checkpoint taken at or before the virtual receive time of the new message. The process completes its rollback by sending an *antimessage* for each of the messages in its output queue whose virtual send time is later than the current local virtual time. The antimessage has virtual send and receive times identical to the original, "positive" message. An antimessage has the effect of annihilating with its positive counterpart in the input queue of the receiving process, as well as with the copy of the positive message in the output queue of the sending process. Antimessages arriving in the virtual past are treated the same as positive messages.

Finally, processes execute a separate protocol that periodically updates *global virtual time* (GVT). GVT is the minimum of the set of local virtual times and the virtual send times of all unreceived messages. This allows processes to discard all checkpoints and input messages older than GVT.

7.4.2 Disadvantages

In the execution of the Time Warp protocol, there may occur two phenomena that increase both its communication and processor overhead during rollback. The first of these is *transitivity delay*, in which a process is at the end of a long chain of antimessages that are sent when the process at the head of the chain initiates a rollback. Consider the sequence of events shown in figure 7.4. At local virtual time 5, process A sends the positive message $+m$ to process B, with virtual receive time 5. Upon receiving $+m$, B sends $+n$ to process C, and C subsequently sends $+p$ to process D. After sending $+m$, process A receives a message whose virtual receive time is 4, and must therefore roll back and send an antimessage, $-m$. This results in B and C sending antimessages $-n$ and $-p$, respectively. Thus, once A rolls back to time 4, there is a delay of three messages before D initiates its rollback, even though it is certain before the first antimessage is sent that D must roll back, since its receipt of $+p$ is at the end of a chain of messages that begins with $+m$. D should begin its rollback as soon as possible after A rolls back, so that it may begin executing forward again sooner.

The second phenomenon, *incremental rollback*, is due to the receipt of

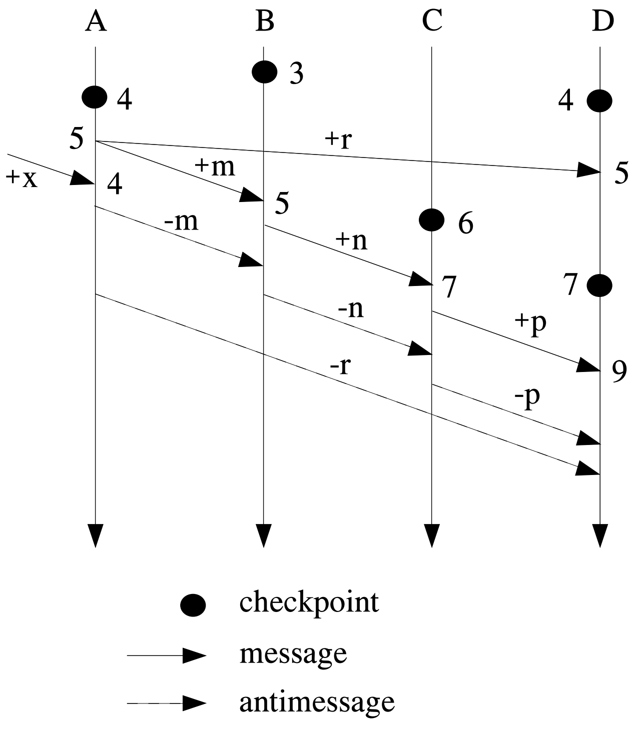

Figure 7.4: A Time Warp execution

multiple antimessages. Consider message $+r$, which A sent directly to D at the same virtual time it sent $+m$. When A rolls back, it sends antimessage $-r$ in addition to $-m$. When D receives $-p$, it rolls back to its checkpoint at virtual time 7. When it subsequently receives $-r$, it must roll back a second time to its checkpoint at virtual time 4. Thus a single rollback at A has resulted in multiple rollbacks at D. Multiple rollbacks consume additional processor time, and may cause more antimessages to be sent than a single, "deep" rollback. A rollback by A to a virtual time preceding 5, should cause only a single rollback (also to a time preceding 5) at each process that has received messages from A.

Both transitivity delay and incremental rollback could be avoided if it were possible for process A to notify every process along the chain of messages beginning with $+m$ (and $+r$) that it must roll back to a local virtual time that precedes the virtual receive time of a message that is part of the chain. In the next section we describe an alternative to the Time Warp protocol that replaces the functionality of a chain of antimessages with a single *kill* message. Associative Broadcast is used to deliver a kill message from a process that is rolling back to all those processes that depend upon the rollback.

7.4.3 Broadcast Time Warp (BTW)

The BTW protocol is based upon the following observation: When a process (e.g. A) rolls back to a checkpoint at virtual time t because it received a positive message in the virtual past (a *spontaneous rollback*), the need for another process (e.g. D) to roll back (an *induced rollback*) is completely determined by the existence of a chain of messages between the first process and the second process, such that the virtual send time of the first message in the chain is greater than t. Therefore, to determine whether a process, p, needs to roll back, it is sufficient to record at p the virtual time of the most recent message from each other process, q, that begins a dependence chain from q to p. To make use of this information, the protocol must perform these functions:

- Include in each message the virtual send time(s) that begin the dependence chain(s) of which the message is a member.
- Provide a mechanism for process q to inform process p that q has rolled back to time t **iff**[5] there exists a chain of messages from q to p beginning at a time greater than t.
- Remove messages that are part of the affected dependence chain(s) from the input queue of p.

To perform the second function, the profile of each object (process) includes a set of *dependence attributes*. A dependence attribute is of the form (*id, (vt, count)*), where *id* is a process identifier, *vt* is a virtual time, and *count* indicates the number of times that process *id* has previously executed events at local virtual time $>$ *vt*, or, equivalently, the number of times *id* has rolled back to time *vt*. Such an attribute in the profile of p denotes that the most recent dependence chain of messages from *id* to p is the $count^{th}$ such chain beginning at virtual time *vt*.

A *message* is of the form (*alist, rt, data*), where *alist* is a list of dependence attributes, *rt* is the virtual receive time of the message, and *data*

[5]To satisfy our universal requirement for minimal communication, p should be informed of q's rollback only if the dependence exists.

is the message's data portion. We refer to the components of message m as $m.l$, $m.r$, and $m.d$, and the components of a dependence attribute, a, as $a.i$, $a.v$, and $a.c$. A *kill message* consists of a single dependence attribute, indicating the origin of a chain of messages that is to be killed. To simplify our algorithm, we assume that any two messages from a single sender are received in the order they were sent (FIFOness w.r.t senders).

7.4.4 Object Specification

The initial profile of each object is

$$\{self, \texttt{QUEUE}, \texttt{KILL}, \texttt{GVT}\}$$

where `QUEUE`,`KILL`, and `GVT` are type attributes bound to the corresponding operations defined below. `queue(`m`)` is executed in response to the arrival of message m at an object. `kill()` processes kill messages, and `gvt()` responds to updates of global virtual time (assumed to be performed by a separate algorithm). In the following code, *lvt* refers to the local virtual time of the object, Q is the input queue, r is the virtual time to which a process must roll back upon receiving a kill message, and *count*(t) is the number of times the object has rolled back to time t.

```
queue(m) {
    if ∃ kill message k in input queue
        ∧(∃a ∈ m.l :: a.i = k.i ∧ a.v > k.v ∧ a.c ≤ k.c) then
        discard m;
    if m.r < lvt then {
        restore checkpoint taken at t, t ≤ m.r (including profile);
        broadcast([KILL ∧ self.v > t], (self, (t, count(t))));
        foreach t', t ≤ t' ≤ lvt
            count(t') := count(t') + 1;
    }else {
        place m in Q;
        foreach a ∈ m.l {
            if ∃(a.i, (v, c)) ∈ profile
                profile := profile − (a.i, (v, c));
            profile := profile ∪ {(a.i, (a.v, a.c))};
        }
    }
}

kill(k) {
    place k in Q;
    r := +∞;
    foreach m ∈ Q, ∃a ∈ m.l :: a.i = k.i ∧ a.v > k.v {
        if m.r < r then
            r := m.r;
        remove m from Q;
    }
    if r ≤ lvt
        restore checkpoint taken at t, t < r (including profile);
        foreach t', t ≤ t' ≤ lvt
            count(t') := count(t') + 1;
}

send(p, d) {
    m := ({(self, (lvt, count(lvt)))}, lvt, d);
    foreach (i, (v, c)) ∈ profile
        if (i, (v, c)) differs from previous value sent to p
            m.l := m.l ∪ {(i, (v, c))};
    broadcast([p], m);
}
```

```
gvt(t) {
    discard checkpoints older than t;
    foreach m ∈ Q, m.r < t
        remove m from Q;
    foreach kill message k ∈ Q, k.v < t
        remove k from Q;
}
```

Objects maintain an input queue of messages as in the original Time Warp protocol, but need not keep a queue of output messages. Furthermore, unlike antimessages, which annihilate with only one positive message, kill messages remain in the input buffer until removed by the GVT algorithm. When an object receives a message, it examines its input queue for any previously received kill message that kills a chain of messages to which the new message belongs. If a matching kill messages is found, the new message is discarded. Otherwise, the virtual receive time of the new message is examined. If it is less than the current local virtual time (i.e. the message arrived in the virtual past), then a rollback is initiated to the checkpoint taken immediately prior to the virtual receive time of the message. After a rollback to time t, the object broadcasts a kill message containing its unique identifier, the rollback time, and a count that distinguishes the current kill message from previous kill messages for the same virtual time. This kill message will be received only by those objects that have added a corresponding dependence attribute to their profiles. If the new message has arrived in the virtual future, it is placed in the input queue. The dependence attributes contained in the message are added to the profile of the object.

When a kill message, $(i, (v, c))$ from object i is received, the receiving object removes from its input queue all those (previously received) messages that are part of a chain that begins at i after virtual time v. The kill message remains in the input queue to allow removal of any messages that are part of such a chain, but arrive after the kill message itself. However, messages that belong to subsequent chains originating at object i and at time v will not be killed by this kill message, since their *count* value will exceed c. Subsequent to receiving a kill message, the object must roll back **iff** its local virtual time exceeds the virtual receive time of any of the killed messages, that is, if it has already processed a message that is now killed. The checkpoint to which it must roll back is the one taken immediately prior to the minimum of the virtual receive times of the killed messages. A rollback that occurs in response to the receipt of a kill message does not cause any additional kill messages to be sent, since the original kill message will induce any necessary rollbacks at other messages.

A message that is to be sent to object p is constructed by including in the *alist* of the message a dependence attribute that contains the identifier of the sending process, the current local virtual time (i.e. the virtual send time[6]), and the *count* value for that virtual time. In addition, any dependence attributes that have changed since the last time a message was sent to p are included in the *alist*. A message from q to p extends to p any dependence chains that affect q. The information contained in the *alist* allows p to receive any subsequent kill messages affecting those chains.

Finally, when an object receives a message informing it of a change in global virtual time, it discards any checkpoints that are older than GVT, as well as any (kill) messages whose virtual receive times precede GVT.

[6]We have arbitrarily set the virtual receive time of each message equal to its virtual send time.

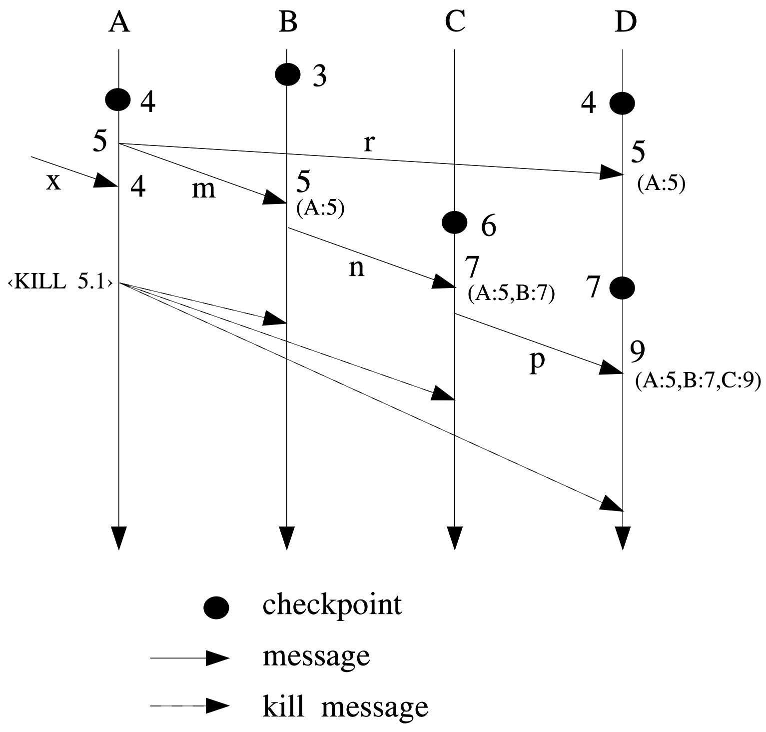

Figure 7.5: An execution of Broadcast Time Warp

Figure 7.5 shows an example execution. Messages r, m, n, and p have been sent as in figure 7.4. For each of B, C, and D there is a dependence chain of messages that originates at object A at virtual time 5. At C and D there are also chains beginning at $B : 7$, and at D there is a chain beginning at $C : 9$. When A receives message x, it must roll back to virtual time 4. This causes it to send a kill message, killing those chains beginning at $A : 5$ whose *count* is 1. Since each of B, C, and D are on at least one of these chains, they each receive the kill message, which removes messages r, m, n, and p from their input queues. Upon receiving the kill message, B rolls back to its checkpoint at time 3, C to 6, and D skips the intermediate checkpoint at 7 and rolls back to 4, since the kill message kills the chains containing both p and r.

The kill message remains in the input queues, however any subsequent chains originating at $A:5$ will have a *count* greater than 1, and will therefore not be killed by this kill message.

7.4.5 Properties of BTW

BTW eliminates the transitivity delay of Time Warp by killing all messages in a dependence chain that is no longer valid with a single broadcast message. Processes needing to roll back are thus delayed for only constant (one message interval) time, instead of a time that is linear in the length of dependence chain. Although it is still possible for processes to roll back several times in a row, a single rollback at one process will never cause multiple rollbacks at another process, as it may in Time Warp. The number of messages sent per rollback is substantially reduced. Exactly one broadcast message is sent per spontaneous rollback, and no additional messages are generated by induced rollbacks. In Time Warp, a single rollback may give rise to an unbounded number of antimessages. This reduction in the number of messages is achieved at the cost of increased message size. The size of a BTW message depends upon the size of its *alist* component. A chain of messages that touches n processes may contain messages whose *alist* has up to n elements. However, the *alist* of most messages in the chain will be smaller.

BTW also eliminates the need for keeping previously sent messages in an output queue. This reduction in storage requirements is offset by an increase in the checkpoint size (storing the object's profile), and the need to retain kill messages in the input queue. The output queue may also be retained in order to implement the "lazy cancellation" mechanism described in [27].

Lazy cancellation delays the sending of antimessages after a rollback until it is certain that upon reexecution, the process will send a different set of messages than it did before the rollback. In the same way, BTW may delay sending a kill message until it is certain that at least one of the messages in the output buffer belongs to a chain that must be killed.

Chapter 8

Replicable Programs

Distributed systems offer the possibility of tolerating processor and communication faults through replication of programs and data. Replication increases the availability of the system by allowing the execution of a program to continue in the presence of faults as long as a sufficient number of copies of its components remain intact. Various replication schemes aimed at increasing availability have been proposed. These are typically based on voting [32, 35], or the "primary copy" technique [10, 20, 49, 60]. These techniques have the advantage of allowing programs to be written without concern for how their parts will be replicated during execution, but have the disadvantage of requiring extensive synchronization among the copies of replicated program components beyond that which is inherent in an unreplicated execution, in order to prevent inconsistencies that arise from nondeterminism. Synchronization may take such forms as enforcing a global ordering on communication [11], checkpointing the execution state, or executing a rollback to recover from failures. Much effort must be devoted to implementing these mechanisms efficiently.

Synchronization is the price one pays for placing the burden of implementing replication completely on the programming language, compiler, and operating system. Since it is not known how a given program will behave, conventional replication schemes must ensure that no possible behavior, in combination with the asynchronous nature of distributed systems, will lead to inconsistencies in a replicated execution. This subjects the execution of all programs to the synchronization required by the worst-behaved program.

We present an alternative approach to replication by investigating the conditions under which all extraneous synchronization may be avoided, while retaining the desirable properties of conventional replication techniques. There are known techniques (based on sequence-numbering of communications) for executing strictly deterministic distributed programs in a replicated manner, without using primary-copy or voting schemes. We have found that by choosing an appropriate model of computation and adopting a *symmetric* replication scheme, we can state two simple *replicability conditions* under which no extra synchronization among replicated components of the program is needed, even when components behave nondeterministically. The conditions constrain only the *local* behavior of components, and are thus easy to implement and verify.

8.1 Programs

We consider a dataflow-like class of programs, called *partitionable programs*, in which a set of *components* communicate by asynchronously reading and writing streams of data values, where the *input streams* of a component are taken from one or more *output streams* of other components. Where multiple instances of a component exist, their respective output streams are arbitrarily *interleaved* to form the corresponding input streams. Each component instance may execute on any of a set of communicating processors in a distributed system.

8.2 Replicability

We consider a program to be replicable if the output streams produced by components in a replicated execution are equivalent (in a sense that we formally define) to the corresponding streams produced in an unreplicated execution. This notion of equivalence leads to a fundamental distinction between deterministic and nondeterministic programs. In a program where nondeterminism exists, there are by definition several possible behaviors for a given set of initial conditions, whereas the behavior of deterministic programs is strictly a function of the initial conditions. The behavior of nondeterministic programs in the presence of replication is much less predictable than that of deterministic programs.

The replicability conditions for nondeterministic programs thus require that components be insensitive to both the order of values in a stream and the number of instances of any given value. Whereas the behavior of a deterministic program is identical with and without replication, in a replicated execution of a nondeterministic program, we can know only that each stream contains a correct set of values. Although this is a much weaker assertion, by constructing an example program that implements a concurrent constraint solver, we show that it provides sufficient information to allow any algorithm expressible as a constraint program to be executed in a fault-tolerant manner.

8.3 Properties of Replicable Programs

Programs whose components satisfy the replicability conditions have the following properties:

- *Replication transparency:* The program specification is independent of the degree of replication during execution. There may exist a distinct, finite number of copies of each component.
- *Failure transparency:* Failure and recovery events are not taken into account in the program, and execution continues correctly without rollback after a failure. Executions remain correct in the presence of up to $k-1$ failures of each k-replicated stream. Failures may cause loss, duplication or reordering of stream values.
- *Freedom from global synchronization:* The replication scheme introduces no additional synchronization among components.
- *Architectural independence:* Correct replicated execution does not rely upon a specific underlying architecture or interconnection network topology. Copies of components may be arbitrarily assigned to processors. Streams may be implemented using asynchronous message passing.
- *Compositionality:* A program formed by composing two replicable programs is itself replicable.

- *Completeness:* The class of algorithms that can be implemented as replicable programs is at least as large as the class of algorithms that can be implemented as a concurrent constraint program.

- *Performance:* Since the copies of a component never synchronize and replication is symmetric, increasing the number of copies of a component may improve performance by reducing communication delays.

8.4 Definitions

8.4.1 Partitionable Programs

Partitionable programs are a generalization of dataflow programs [2] in that they consist of a set of *components*, each of which encapsulates code and data, and which communicate by asynchronously reading and writing streams of data values. Unlike dataflow programs, values written to an output stream may appear in more than one input stream, and input streams may consist of an arbitrary interleaving of the elements of several output streams. We call these programs partitionable because each copy of a component may be arbitrarily assigned to one of a set of communicating processors.

Definition: A *component* consists of a set of labeled *input ports*, a set of labeled *output ports*, a set of local variables, a *guard*, and an *action*, such that:

- There exists one input stream for each input port, and the component reads the values of the stream from the port in FIFO order.

- There exists one output stream for each output port. Values are appended to the output stream when the component writes a value to the output port.
- The guard is a predicate over the local variables and the input ports. The action of the component is executed when the guard evaluates to *true*. Once a component's guard becomes true, it remains so until the action is executed, which occurs within finite time.
- The action is a set of statements that may read values from input streams, write values to output streams, and read and write the local variables of the component.
- The *initial state* of the component consists of initial values for the local variables and a finite initial sequence of values at each input port. These initial sequences represent input from the environment of the program. □

Each value written to an output port is subsequently read from the corresponding input ports. We call the entire sequence of values written to an output port an *output stream* and the entire sequence read from an input port an *input stream*. Let P be the label of an output port. Then a value written to that output port will appear at *each* input port that is labeled P. Both input ports and output ports may have non-unique labels. If there exists more than one output port labeled P, then the input stream appearing at each input port, P, is an arbitrary *interleaving* of the output streams emanating from the output ports labeled P (figure 8.1).

It is not necessarily the case that two distinct input streams formed by the interleaving of a set of output streams are identical. The order in

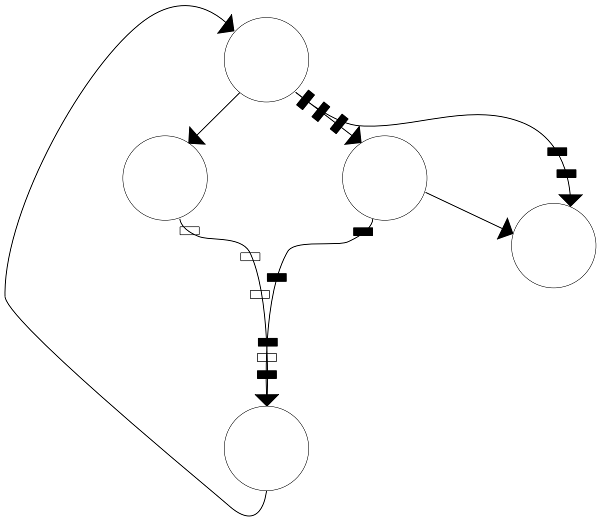

Figure 8.1: A partitionable program with interleaved output streams

which values appear within the streams may differ. Each value appearing at an output port is a message that is broadcast (using Associative Broadcast) to all the components having corresponding input ports. Another possibility is to use a logically-shared data structure such as the Linda[31] *tuple space* to implement the streams. We discuss this further in section 8.6.2.

8.4.2 Replication

A *replicated program* is one which contains several components that are identical. Components are identical **iff** all the elements mentioned in section 8.4.1 are identical, *including* the labels of the ports and the initial states. In an *unreplicated program*, all components differ from each other. Replicated programs are created from unreplicated ones simply by arbitrarily duplicating each component. A *replicated execution* is the execution of a replicated program. In a replicated program, we refer to each of a set of identical components

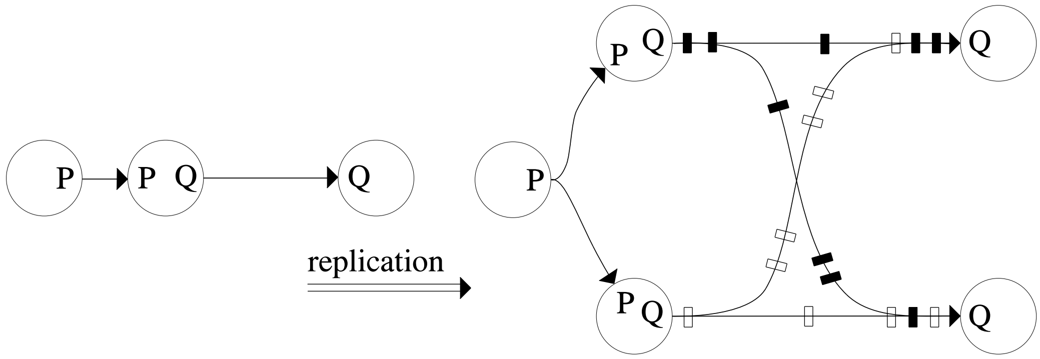

Figure 8.2: A replicated program

as *instances* of the component.

It is important to note that the creation of a replicated version of an unreplicated program may cause interleaving of output streams into input streams where none exists in the unreplicated execution. Figure 8.2 illustrates this effect. Since interleaving is arbitrary, the respective input streams of two instances of a component may differ. This is a source of nondeterminism that is due entirely to replication. We discuss nondeterminism further in section 8.4.5.

8.4.3 Failure and Continuation

As we shall see, the correct operation of a component depends upon the information content of the entirety of each of its input streams. Information content is a notion that is defined with respect to the semantics of a particular program, and thus cannot be fully defined in the abstract. Because of this, the exact set of failures that can be tolerated differ for each program. However, we can define a set of failures that is guaranteed to be tolerated by every program:

- The execution of a program is correct if there exists at least one non-failed instance of each component, and each of the input streams of the component is *complete*.

- Let i be an input stream that contains an interleaving of k output streams produced by k instances of a component. Let V_j be the set of *unique values* appearing in stream j, one of the k output streams, in an execution without failures. Let V_i be the set of unique values appearing in input stream i. Then i is complete **iff** there exists j such that $V_j \subseteq V_i$.

We call a stream from which some unique values are missing due to a failure a *fragmented stream*. A correct input stream is then one that consists of at least one unfragmented instance of each of the distinct output streams that comprise the input stream. Since no instance of an output stream is distinguished as a "primary" instance, there is no explicit recovery procedure to be performed upon failure. The execution of a component continues correctly without interruption as long as all its input streams are complete. It is in fact possible for an execution to be correct even when no complete input streams exist. However, it is difficult to state a general set of conditions for this to be true.

The set of failures tolerated by replicated programs includes those failures where processors behave in a fail-stop[56] manner, which may cause one or more instances of a component to stop producing their output streams prematurely. Also, any stream value, once produced, may be finitely duplicated, and stream values may be reordered. However, Byzantine failures[41] in which components produce incorrect values are not tolerated.

8.4.4 Correctness

In order for replication to be useful, it must be known in what way a replicated execution behaves "the same as" an unreplicated execution. There are several alternative notions of correctness that one may adopt. We choose to compare the output streams produced in a replicated execution with the corresponding output streams in an unreplicated execution. A correct replicated execution is one that preserves the input/output relation (i.e. the relation from initial states to sets of output streams) of the unreplicated execution. Our reason for this choice is twofold: firstly that expressing correctness in terms of an input/output relation is a familiar and widely accepted idea, and secondly that it ensures that replicable programs may be *composed*. We discuss compositionality further in section 9.

Definition: A program is *replicable* **iff** the output streams produced by each component instance in a replicated execution are equivalent to the corresponding output streams in an unreplicated execution, given the same initial conditions. □

The notion of equivalence is dependent upon the semantics of the program. Two output streams that are equivalent are not necessarily identical—specifically, where nondeterminism exists it is not possible to guarantee this identity. We define equivalence formally in section 8.5.1

8.4.5 Nondeterminism

A partitionable program is nondeterministic **iff**, for a given initial state, there exists at least one output port for which it is impossible to determine *a priori* which of several alternative output streams will be produced. Nondeterministic programs result either from nondeterministic behavior of the components (e.g. a component reads from whichever of its input ports next becomes nonempty), or from the arbitrary interleaving of nonidentical output streams. Nondeterminism is useful in solving certain types of problems, and is in fact a necessary property in some cases [15].

In a deterministic program it is possible to ensure that all non-failed instances of a component produce identical output streams. When an instance of a component fails, the remaining instances continue to produce output streams identical to those the failed component would have produced. It is therefore valid to attempt to extract from an interleaving of identical streams one of the original streams. One commonly used method is to assign sequence numbers to the stream elements and use these numbers to eliminate duplicate input values. However, in a nondeterministic program, even two identical instances of a component may produce non-identical output streams. It would therefore be futile to select one of a set of interleaved, *non*identical streams and extract it from the interleaving, since, should the originator of the chosen stream fail, the execution could not successfully continue using elements of the remaining streams.

Nondeterminism limits what we can predict about the composition of a given stream in a given execution. One thing that is in general entirely unpredictable is the order in which values will appear in input stream. Since this order is unpredictable, we cannot impose an ordering requirement on output streams, unless we are willing to permit a component to read in all of its input before producing any output. This is because the input values a component needs to produce its first outputs may be last in its input streams. In a program with cyclic dependencies among objects, this would clearly present a paradox. Replication has the additional effect of creating arbitrary numbers of duplicates of each value in an input stream.

This leads to the following observations:

- Any correctness condition that a replicable program is to meet may not specify the order in which output values are to be produced.
- A component must be prepared to accept the values of its input streams in any order.
- A component must be prepared to accept any number of duplicates of an input value.
- Any one input stream of a component may be arbitrarily longer than any other. Therefore a component may never wait for two or more inputs to be nonempty, since it might wait forever while one input is nonempty, leading to deadlock. Thus a component must execute its action as soon as any of its inputs becomes nonempty, i.e. its guard must be of the form, "there exists a nonempty input[1]."

[1]In dataflow terminology, this is a "pure OR" firing rule.

8.5 Replicability

In an unreplicated execution of a nondeterministic program, the output stream written to a given output port is one of a set of *possible* streams. We consider these possible streams as being semantically equivalent, since each one is a valid output for the given initial condition. In a replicated execution, each instance of a component may produce (at least) any one of these alternative output streams. A corresponding input stream therefore contains an interleaving of equivalent, but not necessarily identical, output streams.

Since not much can be deduced from the arbitrary order of elements in an interleaving of nonidentical streams (where some streams may be fragmented due to failures), we confine our attention to the set of unique values appearing in a stream, called the *value set* of the stream. Consider an input stream that is an interleaving of k output streams in which at most $k-1$ of the streams are fragmented. The value set of such a stream contains at least all the unique values appearing in the unfragmented stream, and may additionally contain subsets of the value sets of each of the other $k-1$ output streams. Therefore, a program is replicable if its components produce correct outputs when their input streams are arbitrarily-ordered value sets of this form.

It remains to define what we mean by "correct outputs," that is, to make precise the definition of correctness given in section 8.4.4. This is the subject of the next section.

8.5.1 Value-Set Correctness

A *value-set correctness constraint* specifies the set of unique values that may appear in an output stream, but does *not* specify their order. In section 8.5.2 we define the conditions under which a replicated execution satisfies such a constraint, given that it is satisfied by a corresponding unreplicated execution.

Definition: A value-set correctness constraint is a pair: (E, R), where

- E is an equivalence relation on value sets such that:

  $$(u, v) \in E \Longrightarrow (u, u \cup v) \in E \wedge (u, u \cup w) \in E, \text{ where } w \subset v.$$

- R is a relation such that for $(D, V) \in R$:
  - D is an initial state of the execution
  - V is a tuple of n value sets, one corresponding to each of the n output ports in the program
  - $\forall (v_j, v_a) \in E :: \exists (D, V) \in R ::$

    $$V = (v_0, \ldots, v_{j-1}, v_j, v_{j+1}, \ldots, v_n) \Longrightarrow$$
    $$\exists (D, V') \in R :: V' = (v_0, \ldots, v_{j-1}, v_a, v_{j+1}, \ldots, v_n)$$

    (i.e. there exists V with v_j replaced by v_a) $\Box$

The equivalence relation, E, specifies those value sets that are considered to be semantically equivalent with respect to the program; that is, two streams whose value sets are equivalent may be substituted one for another in an execution, and the execution remains correct. We call such streams *equivalent streams*. Note that the equivalence relation must be defined with some care. For a given initial condition and a given output, there should exist an equivalence class that includes at least all the value sets of the streams that may be written to the output in an unreplicated execution[2]. In many cases there exist value sets that are semantically equivalent to these, but that can never be produced in an unreplicated execution. However, it is possible for these value sets to be produced in a replicated execution because there exist input streams (due to interleaving) that do not exist in an unreplicated execution. Therefore the equivalence relation should be constructed so that each equivalence class it defines includes all those values sets that are in fact semantically equivalent, not only those that it is possible for an unreplicated execution to produce. In section 8.6 we give examples of defining the equivalence classes for some specific problems.

Any equivalence class that contains value sets u and v also contains the union of the two, $u \cup v$, and $u \cup w$ ($w \subset v$), the union of one set with any subset of the other. The first represents the interleaving of equivalent streams, while the second includes interleavings that contain fragmented streams. All of these are in the same equivalence class because any of them may be an input to a component in a replicated execution, and the component must produce outputs that are equivalent in each case if the program is to be replicable (see next section).

The second relation, R, in the correctness constraint specifies the set of possible correct output value sets for each output port in the program, for

[2]If this is not the case, then even the unreplicated execution will not satisfy this correctness constraint.

a given initial condition. If R holds for a tuple containing one member of an equivalence class, then it holds equally for all tuples containing another member of the equivalence class in the same position. That is, if one value set is correct then all value sets equivalent to it are also correct.

8.5.2 Replicability Conditions

In stating the replicability conditions we refer to the following auxiliary definitions:

Definition: An execution is *convergent* **iff** the shortest initial subsequence of each (input or output) stream that contains the entire value set of the stream is finite. Otherwise, the execution is *divergent*: e.g., some non-failed component executes forever without producing a complete value-set on at least one of its output ports. $\square$

Definition: *I/O tuples:* An *input tuple* (*output tuple*) of a component is a tuple of value sets, one for each input port (output port) of the component. $\square$

Definition: *Tuple equivalence:* Let $\Omega = (E, R)$ be a value-set correctness constraint, and $J = (j_0, \ldots, j_n)$ and $K = (k_0, \ldots, k_n)$ (input or output) tuples. Then $J \equiv K$ (J and K are equivalent with respect to Ω) **iff** $\forall 0 \le i \le n :: (j_i, k_i) \in E$. $\square$

Definition: *I/O relation:* Let C be a component. Then $\rho(C)$ is the *I/O relation* of C that maps initial states and input tuples to output tuples, such that for every $(J, d, F) \in \rho(C)$, F is an output tuple produced by C upon executing with initial state d and input tuple J. $\rho(C)$ specifies exactly the behavior of C. □

Definition: *Span:* Let v be the value set of stream s. Then the *span* of s, $\lambda(s)$, is the length of the shortest initial subsequence of s in which each element of v appears at least once. □

In order for the replicated execution of a nondeterministic program to satisfy a value-set correctness constraint, it must meet the following two conditions. The first condition requires that components behave correctly when their input streams contain interleavings of (possibly fragmented) equivalent streams. This ensures that components are insensitive to the order in which values appear at their input ports, as well as the number of duplicates of an element that it receives. The second condition requires that a replicated execution be *convergent* if the corresponding nonreplicated execution is convergent. This ensures that the program will still make progress when replicated.

1. *Equivalence:* $(J, d, Q) \in \rho(C)$
   $\wedge\, (K, d, T) \in \rho(C) \wedge J \equiv K \Longrightarrow Q \equiv T$

2. *Convergence:* Let $i_0, \ldots, i_n$ be the input streams of C and $o_0, \ldots, o_m$ the output streams of C. If $\lambda(i_k)$ is finite for $0 \leq k \leq n$, then $\lambda(o_k)$ is finite for $0 \leq k \leq m$.

Informally, condition 1 states that, for a fixed initial state, a component that reads one of a set of equivalent input tuples will write one of a set of equivalent output tuples. Condition 2 states that the span of each output stream is finite if the span of each input stream is finite.

Claim: Given that an unreplicated execution of a program satisfies a value-set correctness constraint, Ω, a replicated execution of the program satisfies Ω if the replicability conditions hold. $\square$

Proof: Assuming that a nonreplicated execution satisfies a value-set correctness constraint, we must show first that none of the output streams of the replicated execution violates the correctness constraint, and second, that the replicated execution makes progress.

Output correctness Assume that there exists an incorrect output stream. Since the nonreplicated execution is correct, it must be the case that the value set of the incorrect stream is not equivalent to the value set of the corresponding stream in the nonreplicated execution (by definition of value-set correctness). The value set of each incorrect stream must differ by at least one element from the value set of a correct stream. By condition 1, the only way for a component instance to produce an incorrect value is for it to first receive an incorrect value on one of its inputs. Therefore, there exists a chain or cycle of component instances, each of which produces an incorrect value. In any such chain or cycle, there must exist one component instance that produces the first incorrect value. By condition 1, that first incorrect value cannot be spontaneously produced. Contradiction. The case for omitted values is similar.

Progress By an argument similar to **Output correctness**, no component instance may spontaneously produce a stream whose span is infinite without violating condition 2[3]. □

8.6 Examples

8.6.1 Global Minimum Finder

The following example illustrates the order-insensitive behavior of components in a replicable program. The purpose of the program is to find the global minimum among several sets of integers. We present this example to exhibit a simple replicable program and a value-set correctness constraint satisfied by that program.

Program The program consists of several MIN components and one MERGE component. Each MIN component searches the set of integers arriving at its inputs for a new minimum value. When it discovers such a value, it sends it on to the MERGE component, which receives the local minima from each MIN component and determines the global minimum. After the input stream of each MIN component has been exhausted, the execution should eventually terminate with the global minimum assigned to a local variable of the MERGE component. In order to reduce communication among components, whenever the MERGE component find a new global minimum, it sends it back to all the MIN components. These then send to the MERGE component only local minima

[3]We have not guaranteed that a replicated execution will terminate if a nonreplicated execution terminates. However, after finite time, no new values will be produced on any output stream in the replicated execution.

that are smaller than the latest global minimum they have received.

Correctness, Replicability and Fault-Tolerance The values appearing in streams are simply integers. The output stream of a MIN component is correct **iff** it contains the minimum of the set of integers appearing at the input of the component. The equivalence relation, E, that holds among the value sets of such streams is simply that $a \equiv b$ **iff** $min(a) = min(b)$, where a and b are value sets. The entire execution is correct (specified by the relation R) **iff** the value set of at least one of the MIN output streams contains the global minimum.

Clearly, this value-set correctness constraint is satisfied even by MIN components that do nothing more than copy their inputs to their outputs. If we wish to constrain more precisely the local behavior of components, e.g. to ensure that each MIN component heeds the global minima it receives from the MERGE component, we are free to add such constraints. Such constraints represent improvements on the "minimally correct" behavior of the overall program, which is what is specified by the value-set correctness constraint.

The program is replicable because an interleaving of any two equivalent streams contains the same minimum value as each of the streams individually, and is therefore equivalent to each of the streams alone. Similarly, if at least one of a set of interleaved streams is nonfragmented, the minimum value of the interleaving remains unchanged. Therefore, the execution survives $k-1$ failures of k-replicated output streams.

8.6.2 Constraint Solver

In the following example we discuss the replicability of a restricted version of the constraint solver specified in section 7.3. The purpose of this example is to show that there exists a large class of algorithms that can be implemented as replicable programs. By implementing a concurrent constraint solver, it is possible to achieve a replicated execution of any algorithm that is expressible as a constraint program [44]. This is a large, Turing-complete class of programs. Thus the class of replicable programs is at least as large as those that can be executed by concurrent constraint solvers such as Consul [6]. The efficiency characteristics of our constraint solver are similar to these, in that it achieves the same degree of parallelism, and must rely to the same extent upon suitable heuristics in the satisfaction of arbitrary constraints.

Let K be a set of constraints assigned to a component, and V the set of variables appearing in elements of K. Then the component has the following form:

- For each set of variables v such that $v \subseteq V$, there exists an input port labeled "v" and an output port labeled "v."
- Let I be the set of input ports of a component, O the set of output ports, and X the set of previously determined inconsistencies. Then the behavior of the component is specified by the following guard and action:

$\exists i \in I :: i$ nonempty $\longrightarrow$
    $< \forall i \in I$: read i;
    find maximal set of new inconsistencies, N,
    using X and values read from I;
    $X = X \cup N$;
    $\forall n \in N$:
        write n to output o such that
        (vars appearing in n) = (label of o);
    $>$

- Initially, $X = \emptyset$ and $\forall i \in I :: i = \perp$, where $\perp$ is the "null inconsistency."

Initially, the guard of each component is *true* by virtue of the special value $\perp$ at the head of each input stream. Each component reads its input streams, finds a maximal set of new inconsistencies (using a constraint-satisfaction algorithm appropriate to its local constraints), and writes each inconsistency to the output whose label corresponds to the set of variables appearing in the inconsistency.

Each component reads a set of inconsistencies, finds a new set of inconsistencies, and propagates these to all components whose local states might be affected by this discovery. The computation terminates because the set of locally consistent states at each component can only grow smaller, so eventually no new inconsistencies can be discovered. The final set of locally-consistent states at each component is also globally consistent: Any inconsistent state by definition violates at least one constraint. Therefore, at some point in the execution an input value is read that eliminates each such state.

The value set of each stream, i.e. the union of all the inconsistencies written to a particular output port, describes (by its complement) a set of states that is consistent with the local constraints of the component, and is also globally consistent.

Replicability and Fault-Tolerance Replicated and unreplicated executions of the above program are equivalent under a correctness constraint that we state as follows:

- The relation R specifies that the value set of each output stream describes a globally-consistent state.
- The equivalence relation, E, is such that if value sets u and v describe an identical set of states, then $(u, v) \in E$.

It is straightforward to show that the program satisfies the replicability conditions and the correctness condition, given the above definition of the equivalence relation. Informally, the program is replicable because an interleaving of two or more streams whose value sets each describe a set of inconsistent states, S, also describes exactly S. Therefore, no matter how many instances of a component exist, the "downstream" components will always receive the same information. Figure 8.3 shows a the boolean constraint solver of figure 7.3 with one of its components replicated.

Similarly, if at least one of a set of interleaved streams is nonfragmented, the value set of the interleaving describes the same set of inconsistent states. Therefore, the execution survives $k-1$ failures of k-replicated output streams.

Implementation We have designed an implementation of the replicable constraint solver for boolean constraints using the C-Linda[57] programming language, a version of the Linda[31] model based on C[39]. Processes in Linda communicate using a logically-shared data structure, the *tuple space* (TS),

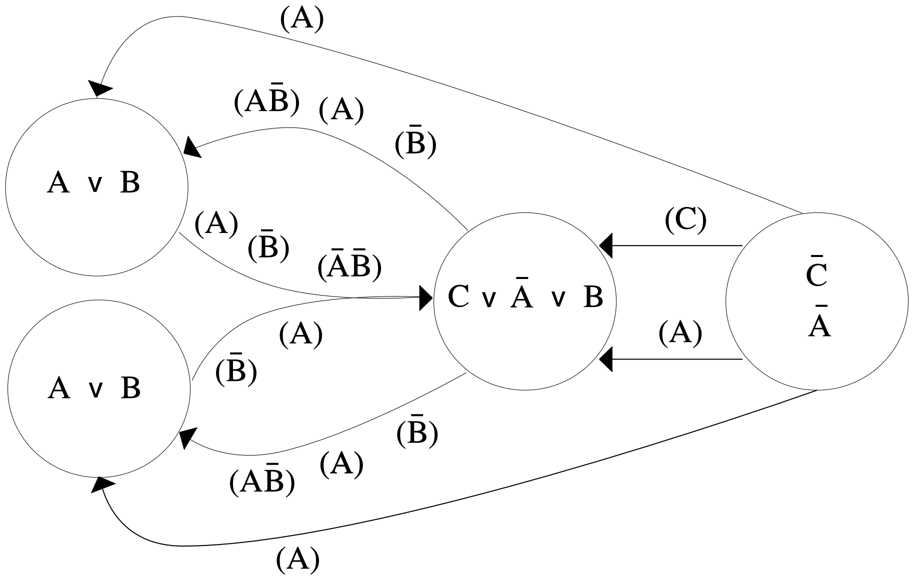

Figure 8.3: A replicated constraint solver

whose implementation is architecture-dependent. TS supports the *out*, *in*, and *read* operations, which respectively insert, remove, and copy *tuples* to/from TS, possibly by specifying a *pattern* of formal parameters that retrieves a matching tuple from TS.

The major task in creating a Linda version of the constraint solver is to implement the communication model assumed by partitionable programs using TS. TS does not specifically support streams, however, since stream values may be reordered, it suffices to implement each stream value as a tuple in TS. The first element of each tuple indicates the stream to which the value was written, the second element contains the value itself, and the third element is a bit-string that is used to remove the tuple from TS after its value has been input by each component that reads from the stream to which the tuple belongs. Each process has a unique index into the bit string that it uses to mark the tuple after reading it.

Each component of the constraint solver is a Linda process. Upon initialization, the process creates a fixed number of replicas of itself. The components then behave as specified by the algorithm, using *out* to write values to streams and *in* to read values from streams. When a tuple is removed from TS by *in*, the process first checks the bit string to see if it has read the tuple before. If not, it copies the value field to a local variable. If the bit string is full (all relevant processes have read it), the tuple is discarded. Otherwise the tuple is put back in TS using *out*.

Omission failures may be simulated by an extra process that randomly removes tuples from TS. Reordering and duplication are inherent in the semantics of TS itself.

We have found that the Linda model is not the ideal environment for implementing the constraint solver, nor partitionable programs in general. Too much programming effort is required to implement the stream semantics, and communication is less efficient than it could be, because each process may retrieve a tuple many times from TS before the tuple is finally removed. Processes should see each value exactly once. Message-passing systems such as Associative Broadcast would be more suitable, since the semantics of a message are closer to those of stream values—each message arrives at most once at each process.

Chapter 9

Concluding Remarks

9.1 Summary of Contributions

We began our investigation of the role of naming systems by observing that the composition and structure of a concurrent computation are manifest in its patterns of communication, and that the specification of these patterns relies upon the naming system used to indicate the objects participating in a communication. The ability of a naming system to express particular communication patterns (its *communication semantics*) are determined by properties of its name generation, propagation, and binding mechanisms. The patterns of communication that may be specified using a given naming system indicate the level of *communication abstraction* provided by the naming system. More expressive naming systems raise the level of abstraction at which patterns of communication may be specified, and allow these communication patterns to be implemented at a lower level, hiding the details of their implementation from the user of the naming and communication systems.

We have defined a taxonomy of naming systems that isolates six ordered pairs of naming system properties, which determine the ability of naming systems to express a set of fundamental patterns of communication. The pairs of properties are ordered by the *subsumes* relation, $\sqsupset$. Categories of naming systems are formed by conjoining one property from each of the six orthogonal pairs. Extension of $\sqsupset$ to categories yields a hierarchy (complete lattice) of categories that may be used to classify naming systems and rank them according to expressiveness. We have used the taxonomy to classify and compare a range of representative naming systems of concurrent programming languages and distributed operating systems from the literature. These classifications demonstrate that both the depth (differences in expressive power among naming systems) and breadth (incomparability of some pairs of naming systems) of the hierarchy are reflected by existing naming systems. Furthermore, it is shown that naming systems with identical properties may differ substantially in their implementation, and that distinct concurrent systems may be based on identical naming models. We show that the distinctions among categories of naming systems are meaningful by giving examples of problems that require specific properties. Finally, we show that the naming system of Associative Broadcast is a member of category $\top$, subsuming the properties of all naming systems in the taxonomy.

A high level of communication abstraction may be required when constraints such as symmetry, decentralization, and minimal communication are placed on the form of an algorithm that is the solution to a particular problem. For the class of category-$\top$ problems, these constraints require the specification of dynamic, *1–N* patterns of communication. These constraints not only yield an algorithm with desirable properties, but may in fact make the algorithm easier to specify by forcing us to use a naming system that provides a level of communication abstraction appropriate to the problem.

Associative Broadcast enables the specification and implementation of distributed solutions to category-$\top$ problems. The Associative Broadcast naming system represents each communicating object by an abstraction of its local state (the object's profile). Names are propositional expressions that identify dynamic *target sets* of objects. Both binding (profile modification) and resolution (profile-name matching) are local operations, and require senders to have only static global knowledge. The **broadcast** communication primitive unifies 1–1 communication (the special case in which the target set is of cardinality 1), with 1–N communication. Attributes uniformly represent the types, local states, and identifiers of objects.

We have used Associative Broadcast to obtain new solutions to four category-$\top$ problems: Truth maintenance (later generalized to constraint reduction), consistency of replicated databases in the presence of network partition failures, and virtual-time synchronization. In addition to the common characteristics of symmetry, decentralization, and minimal communication the algorithms presented exhibit such desirable properties as dynamic replicability and repartitionability (constraint solver), absence of global freezing of transactions (distributed optimistic protocol), and absence of transitivity delay and incremental rollback (broadcast Time Warp). These additional properties result from the communication abstractions provided by Associative Broadcast. Finally, we generalize the observed replicability property of the constraint solver into design criteria for replicable programs requiring minimal synchronization.

9.2 Future Work

We have omitted discussion and analysis of resolution because the soundness of naming systems (section 1.1) ensures that characteristics of the resolution mechanism that affect the specification of communications are taken into account in classifying the naming system. That is, the maximal computation set allowed by a naming system contains no computations in which unresolvable bindings are established: In order for a communication (e, n, f) to exist in a valid computation, it must be the case that name n is resolvable to object $f.o$ in the namespace of object $e.o$. Therefore, classification of naming systems within the taxonomy takes resolution into account.

However, the taxonomy does not *distinguish* naming systems according to their resolution properties, since all the naming system properties we define have to do with binding and knowledge of names. We have omitted such properties because we have not found them to be significant in determining the patterns of communication that naming systems are capable of expressing. However, it is possible to augment the taxonomy with an orthogonal set of resolution properties. Previous work[19, 50] has already produced very useful analytical models of partitioned, hierarchical name spaces, which are the source of most resolution complexity. This work would serve as a starting point for determining resolution properties.

One way to view the naming system properties we have defined is as a set of "progress" properties: They define the ability of a naming system to express necessary communication patterns. It may be useful to take a similar approach in defining "naming safety" properties of individual computations. For example, we can define the set of *concurrent bindings* as $\{e.b|e$ are concurrent events}, and similarly for the set of *concurrent domains*. Then we may define, e.g., the "no orphans" property of a program: at least one name of each object in a concurrent set of bindings exists in a concurrent set of domains, i.e. all objects remain reachable.

Future work on Associative Broadcast entails the design of efficient profile-name matching mechanisms for specific distributed architectures. We have outlined an implementation for large-granularity, broadcast-based systems. It remains to investigate the implementation requirements of matching mechanisms for shared-memory systems and point-to-point networks, and for small-granularity parallel systems. In such systems it is likely to be useful for the **broadcast** primitive to coexist, for reasons of efficiency, with conventional communication primitives. Furthermore, Associative Broadcast can be augmented with primitives for acknowledged broadcast and atomic broadcast. For some problems (e.g. n-body simulations), it is useful for a sender to be able to wait for an acknowledgment that ensures that the name contained in a broadcast message has been matched against the profiles of all potential recipients. Atomic broadcast ensures still stronger synchronization: Messages are received in the same order by all objects.

BIBLIOGRAPHY

[1] M. Accetta, et al., Mach: a new kernel foundation for UNIX development, *Carnegie Mellon Univ. Comp. Science Dept. Tech. Rep.*, 1987

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

[3] 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

[4] M. Ahamad and A. Bernstein, Multicast communication in UNIX 4.2BSD, *Proc. 5th Int. Conf. on Distributed Computing Syst.*, May 1985

[5] H. Bal, et al., Programming languages for distributed computing systems, *ACM Comp. Surv.*, vol. 21, no. 3, Sep. 1989

[6] D. Baldwin, Consul: a parallel constraint language, *IEEE Software*, July 1989

[7] R. Balzer, Living in the next generation operating system, *Proc. 10th World Comp. Congress*, Sept. 1986

[8] B. Bayerdorffer, An analytical taxonomy of naming systems, *Univ. of Texas Dept. of CS Tech. Report TR-92-48*, Dec. 1992, submitted to *ACM Trans. on Comp. Sys.*

[9] B. Bayerdorffer, Distributed programming with associative broadcast, submitted to *27th HI Int. Conf. Syst. Sci.*, Jan. 1994

[10] K. Birman, et al., Implementing fault-tolerant distributed objects, *IEEE Trans. Software Eng.*, vol SE-11, no. 6, June 1985

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

[12] K. Birman and T. Joseph, Exploiting virtual synchrony in distributed systems, *Proc. 11th ACM Symp. Operating Systems Principles*, Nov. 1987

[13] A. Birrel and B. Nelson, Implementing remote procedure calls, *ACM Trans. Comp. Syst.*, vol. 2, no. 1, Feb. 1984

[14] M. Bowman, et al., Reasoning about naming systems, *to appear*, 1993

[15] J. Brock, A formal model of non-determinate dataflow computation, *PhD. thesis, Dept. EE and CS, MIT*, Aug. 1983

[16] N. Carriero, The implementation of tuple space machines, *Ph.D. Thesis, Dept. of Computer Science, Yale Univ.*, Dec. 1987

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

[18] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988

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

[20] E. Cooper, Replicated distributed programs, *UCB/CSD 85/231, UC Berkeley*, May 1985

[21] F. Cristian, Atomic broadcast: from simple message diffusion to byzantine agreement, *Proc. 5th Int. on Fault-Tolerant comp.*, June 1985

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

[23] S. Davidson and H. Garcia-Molina, Consistency in partitioned networks, *ACM Computing Surveys*, vol. 17, no. 3, Sept. 1985

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

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

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

[27] A. Gafni, Space management and cancellation mechanisms for time warp, *Ph.D. Thesis, Dept. of Computer Science, Univ. of Southern California TR-85-341*, Dec. 1985

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

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

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

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

[32] D. Gifford, weighted voting for replicated data, *Proc. 7th ACM Symp. Operating Syst. Principles*, Dec. 1979

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

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

[35] M. Herlihy, A quorum-consensus replication method for abstract data types, *ACM Trans. Computer Syst.*, vol. 4, no. 1, Feb. 1986

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

[37] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985

[38] D. Jefferson, Virtual time, *ACM Trans. Prog. Languages and Systems*, vol. 7, no. 3, July 1985

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

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

[41] L. Lamport, et al., The byzantine generals problem, *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 3, July 1982

[42] L. Lamport, On interprocess communication, *DEC Syst. Res. Cent. Report*, no. 8, 1985

[43] T. LeBlanc, An analysis of language models for high-performance communication in local-area networks, *Proc. SIGPLAN '83 Prog. Lang. Issues in Software Sys.*, 1983

[44] W. Leler, Constraint programming languages, *Addison-Wesley, New York*, 1988

[45] R. Metcalfe and D. Boggs, Ethernet: distributed packet switching for local computer networks, *Communications of the ACM*, vol. 19, no. 7, July 1976

[46] P. Mockapetris, Domain names—implementation and specification, *RFC 1035, USC Inf. Sci. Institute*, Nov. 1987

[47] J. Mogul, et al., The packet filter: an efficient mechanism for user-level network code, *Proc. 11th ACM Symp. Operating Systems Principles*, Nov. 1987

[48] J. Mulac and D. Baldwin, Local propagation as a constraint-satisfaction technique, *Tech. Rep. 265, CS Dept., Univ. of Rochester*, 1989

[49] B. Oki and B. Liskov, Viewstamped replication: a new primary copy method to support highly-available distributed systems, *Proc. 7th ACM Symp. Principles Dist. Computing*, Aug. 1988

[50] 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

[51] A. Pnueli, The temporal logic of programs, *18th IEEE Symp. on Foundations of Computer Sci.*, 1977

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

[53] D. Reed, Implementing atomic actions on decentralized data, *ACM Trans. on Comp. Sys.*, vol. 1, no. 1, Feb. 1983

[54] D. Russell, Parallelism through broadcast actors, *MCC Technical Report*, no. PP-149-87, May 1987

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

[56] F. Schneider, Fail-stop processors, *Proc. 26th IEEE Computer Soc. Int. Conf.*, Mar. 1983

[57] Scientific Computing Associates, A Linda user's manual, New Haven, CT, 1987

[58] A. Singh, Ranking in distributed systems, *Ph.D. thesis, Dept. of Computer Sciences, Univ. of Texas at Austin*, Dec. 1989

[59] G. Steele, The definition and implementation of a computer programming language based on constraints, *Tech. Rep. AI-TR-595, Artificial Int. Lab., MIT*, 1980

[60] M. Stonebraker, Concurrency control and consistency of multiple copies of data in distributed INGRES, *IEEE Trans. Software Eng.*, vol. 5, no. 3, May 1979

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

[62] P. Wegner, Dimensions of object-based language design, *Proc. ACM OOPSLA Conference*, Oct. 1987

VITA

Bryan Bayerdorffer was born on December 10, 1962, in New York City, the son of Ira and Herman Bayerdorffer. Upon graduation from Blind Brook High School in Rye Brook, New York, he entered Cornell University in the Fall of 1981, where he studied Computer Science and Electrical Engineering at the College of Engineering. He received the Bachelor of Science degree in 1985, and was admitted to the Ph.D. program in Computer Sciences at the University of Texas at Austin in the same year. In 1988, he married Margaret Elizabeth Kohn of Saratoga Springs, New York.

Permanent address: 301 Cove Road

Riva, MD 21140

Internet Mail: bryan@scs.usna.navy.mil

This dissertation was typeset with LaTeX by the author.