HOW PROCESSES LEARN

K. Mani Chandy & Jayadev Misra
Department of Computer Sciences University of Texas Austin, 78712

1. Introduction

Processes in distributed systems communicate with one another exclusively by sending and receiving messages. A process has access to its state but not to the states of other processes. Many distributed algorithms require that a process determine facts about the overall system computation. In anthropomorphic terms, processes "learn" about states of other process in the evolution of system computation. This paper is concerned with how processes learn. We give a precise characterization of the minimum information flow necessary for a process to determine specific facts about the system.

The central concept in our study is that of *isomorphism* between system computations with respect to a process: two system computations are isomorphic with respect to a process if the process behavior is identical in both. In anthropomorphic terms, "system computations are isomorphic with respect to a process" means the process cannot distinguish between them.

Many correctness arguments about distributed systems have the following operational flavor: "I will send a message to you and then you will think that I am busy and so you will broadcast . . . ". Such operational arguments are difficult to understand and error prone. The basis for such operational arguments is usually a "process chain": a sequence of message transfers along a chain of processes. We advocate nonoperational reasoning. The basis for nonoperational arguments is isomorphism; we relate isomorphism to process chains. Algebraic properties of system computations under isomorphism provide a precise framework for correctness arguments.

It has been proposed [3,6] that a notion of "knowledge" is useful in studying distributed computations. In earlier works, knowledge is introduced via a set of axioms [4]. Our definition of knowledge is based on isomorphism. Our model allows us to study how knowledge is "gained" or "lost". One of our key theorems states that knowledge gain and knowledge loss both require sequential transfer of information: if process q does not know fact b and later, p knows that q knows b, then q must have communicated with p, perhaps indirectly through other processes, between these two points in the computation; conversely, if p knows that q knows b and later, q does not know b then p must have communicated with q between these two points in the computation. In the first case, the effect of communication is to inform p of q's knowledge of b. Analogously, in the second case, the effect of communication is to inform q of p's intention of relinquishing its knowledge (that q knows b). Generalizations of these results for arbitrary sequences of processes are stated and proved as corollaries of a general theorem on isomorphism.

We use the results alluded to in the last paragraph

©1985 ACM 0-89791-167-9/1985/0800-0204 $00.75

for proving lower bounds on the number of messages required to solve certain problems. We show, for instance, that there is no algorithm to detect termination of an underlying computation using only a bounded number of overhead messages.

2. Model of a Distributed System

A distributed system consists of a finite set of processes. A process is characterized by a set of process computations each of which is a finite sequence of events on that process. Process computations are prefix closed, i.e. all prefixes of a process computation are also process computations (of that process). An event on a process is either a *send*, a *receive* or an *internal event*. A *send* event on a process corresponds to sending of a message to another process. A *receive* event on a process corresponds to reception of a message by the process. There is no external communication associated with an *internal* event. For a set of processes P, a send event by P is a send event by some component process of P to a process outside P; similarly a receive event by P denotes receipt by some process in P of a message sent from outside P. Communication among processes in P are internal events of P. We use "e is on P", for event e and process set P, to denote that e is an event on some process in P. We rule out processes which have no event in any computation. We assume that all events and all messages are distinguished; for instance, multiple occurrences of the same message are distinguished by affixing sequence numbers to them.

Let z be any sequence of events on component processes of a distributed system. The *projection* of z on a component process p, denoted by z_p, is the subsequence of z consisting of all events on p. A finite sequence of events z is a *system computation* of a distributed system means (1) for all processes p, z_p is a process computation of p and, (2) for every receive event in z, say receipt of message m by process p, there is a send event, of sending m to p, which occurs earlier than the receive in z: this send event will be called the send event *corresponding* to the receive. We leave it to the reader to show that system computations are prefix closed.

In this paper we consider a single (generic) distributed system. For instance, when we say "z is a computation" we mean that z is a computation of the distributed system considered here. We use *computation* to mean *system computation* when no confusion can arise.

Notation: We use x, y, z to denote computations, p, q for processes and P, Q for process sets; these symbols may be used with subscripts or superscripts. The concatenation of two sequences y and z will be denoted by $(y;z)$. For sequences y and z, $y \leq z$ denotes that y is a prefix of z; in this case (y, z) denotes the suffix of z obtained by removing y from z. The empty sequence will be denoted by *null*. The symbol $=$ is used to denote equalities among sets and among predicates. The symbol $\equiv$ is used for definitions. The set of all processes in the system will be denoted by D and for any process set P, $\overline{P} = D - P$.

3. Isomorphism

We define the relation $[p]$ on the set of system computations as follows.

Definition: For system computations x,y:
$x\,[p]\,y \;\equiv\; (x_p = y_p)$.

In other words, $x\,[p]\,y$ means p's computation is the same in system computations x and y. In this case, we say x, y are *isomorphic with respect to p*. For a process set P, define relation $[P]$, on the system computations, as follows.

Definition: $x\,[P]\,y \;\equiv\;$ for all p in P, $x\,[p]\,y$.

Thus $x\,[P]\,y$ means that, given only the computations of processes in P we cannot distinguish x from y. From definition, $x\,[\{\ \}]\,y$, for all computations x, y where $\{\ \}$ denotes the empty set. Observe that $[P]$ is an equivalence relation.

It is convenient to represent all such isomorphism relations by an *isomorphism diagram*: an undirected labelled graph whose vertices are computations and there is an edge labelled $[P]$ between vertices x, y if P is the largest set of processes for which $x\,[P]\,y$. Observe that every vertex has a self loop labelled $[D]$ where D is the set of all processes in the system. Note that $x\,[D]\,y$, $x \neq y$, implies y is a permutation of x.

Example 1: Consider a system with two processes, p *and* q, for which part of the isomorphism diagram, showing the relationships among four system computation, is given below.

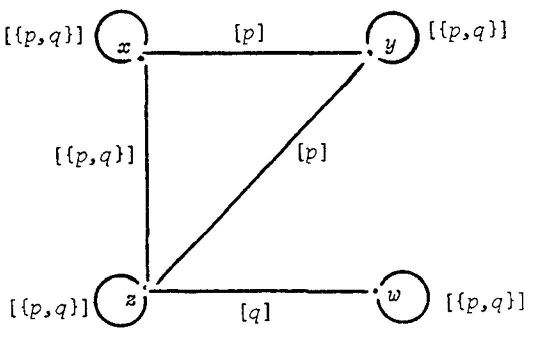

Figure 3-1: An Isomorphism Diagram

From the diagram $x\,[p]\,y$, but not $x\,[q]\,y$. This means p has the same computations in both x and y, whereas q's computations in x and y differ. Computations x and z have the same computations for both p and q; hence one is a permutation of the other. There is no direct relationship between y and w; neither $y\,[p]\,w$ nor $y\,[q]\,w$ holds. However, there is an indirect relationship between y and w because $y\,[p]\,z$ and $z\,[q]\,w$. We explore such indirect relationships next.

□

Definition: Let $n > 0$ and P_i be process sets, $0 \leq i \leq n$.

$x\,[P_0 \ldots P_n]\,z \equiv x\,[P_0 \ldots P_{n-1}]\,y$ *and* $y\,[P_n]\,z$, for some computation y.

Hence, $[P\,Q] = [P] \circ [Q]$ where "$\circ$" is the relational composition operator. This operator is associative (from properties of relations). In terms of the isomorphism diagram, $x[P_0 \ldots P_n]\,z$ means there is a path from x to z whose edges are labelled with $Q_0, \ldots, Q_n$, respectively, where $Q_i \supseteq P_i$, for all i.

Example 1 (contd.): We have $y\,[p\;q]\,w$ and $w\,[q\;p]\,y$. Also, trivially, $y\,[q\;p]\,z$, $y\,[q\;p\;q]\,z$, etc.

□

We note some properties of isomorphism relations. In the following, $P, P_1, \ldots, P_n, Q$, denote arbitrary process sets and x, y, z denote arbitrary computations.

1. $[P]$ is an equivalence relation.
2. (Substitution) $([\beta] = [\delta])$ *implies* $([\alpha\,\beta\,\gamma] = [\alpha\,\delta\,\gamma])$ for arbitrary sequences of process sets α, β, γ, δ.
3. (Idempotence) $[P\,P] = [P]$
4. (Reflexivity) $x\,[P_1 \ldots P_n]\,x$
5. (Inversion) $x\,[P_1 \ldots P_n]\,y = y\,[P_n \ldots P_1]\,x$
6. (Concatenation) For $0 < m < n$,

   $\exists y\colon\; x\,[P_1 \ldots P_m]\,y,\; y\,[P_{m+1} \ldots P_n]\,z = x\,[P_1 \ldots P_m\,P_{m+1} \ldots P_n]\,z$
7. $[P \cup Q] = ([P] \cap [Q])$
8. $(Q \supseteq P) = ([Q] \subseteq [P])$
9. $(P = Q) = ([P] = [Q])$
10. $Q \supseteq P$ *implies* $([Q\,P] = [P] = [P\,Q])$

These properties follow from properties of relations and our model. We only sketch a proof of one part of property 8:

$([Q] \subseteq [P])$ *implies* $(Q \supseteq P)$.

If $Q \not\supseteq P$ then there is a process p in $P - Q$. From our model, p has an event e in some computation $(x;e)$. Then $x\,[Q]\,(x;e)$ and $\sim x\,[P]\,(x;e)$. Hence $[Q] \not\subseteq [P]$.

3.1. Process Chains

As noted in the introduction, the basis for many operational arguments are process chains: process p informing q which in turn, informs r etc. One of our goals is to replace such concepts by algebraic properties of system computations. In this section, we show how process chains are related to isomorphism. We first define process chains; this definition is along the lines suggested by Lamport [5].

Definition: For events e, e' in a computation z, $e \xrightarrow{z} e'$ means:

1. e' is a receive and e is the corresponding send, or
2. events e, e' are in the same process computation and ($e = e'$ or e occurs earlier than e'), or
3. there exists an event e'' such that $e \xrightarrow{z} e''$ and $e'' \xrightarrow{z} e'$.

For brevity we write $e \rightarrow e'$ when the computation z is understood from context. We will write $e_0 \rightarrow e_1 \rightarrow \ldots e_{n-1} \rightarrow e_n$, as shorthand for $e_0 \rightarrow e_1$ and $\ldots$ and $e_{n-1} \rightarrow e_n$. Observe that $e \rightarrow e$ for every event e in z. A computation z has a *process chain* $<P_0\ P_1\ \ldots\ P_n>$ means there exist events $e_0, e_1, \ldots e_n$, not necessarily distinct, in z such that event e_i is on P_i, for all $0 \leq i \leq n$, and $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$.

Observation 1: Any occurrence of "P " in a process chain may be replaced by "$P\ P$", or vice versa, since for any event e on P, $e \rightarrow e$.

Observation 2: Let x be a sequence consisting of a subset of events from a computation y. Suppose that for every event e in x, every e', where $e' \xrightarrow{y} e$, is also in x and $e' \xrightarrow{x} e$. Then x is a computation.

3.2. Relationship Between Isomorphism and Process Chain

Theorem 1: (Fundamental Theorem of Process Chains)

Let z be a computation and x a prefix of z. Let P_1, $P_2\ \ldots\ P_n$, $n \geq 1$, be sets of processes. Then $x\,[P_1\, P_2 \ldots P_n]\, z$ or there is a process chain $<P_1\, P_2 \ldots P_n>$ in (x,z).

□

Proof: Omitted

□

3.3. An Application of Isomorphism: How To Construct A Computation By Fusing Separate Ones

In this section, we show an application of isomorphism: we give a construction to "fuse" two computations to obtain a new computation, provided certain types of paths exist in the isomorphism diagram. We motivate the discussion by the following observations. Suppose $(x;E)$ and $(x;\overline{E})$ are computations where all events in E are on a process set P and all events in $\overline{E}$ are on $\overline{P}$. Then, from definition, $(x;\overline{E};E)$ and $(x;E;\overline{E})$ are also computations, because events in $E,\overline{E}$ are independent and hence may be fused in arbitrary order. A similar result appears in Fischer, Lynch and Paterson [2]. The following lemma is a generalization of this observation.

Lemma 1: Let x, y, z be computations where $x \leq y$ and $x \leq z$. Let P, Q be such that $P \cup Q = D$, $x\,[P]\,y$ and $x\,[Q]\,z$. Then there exists a computation w where $x \leq w$, $y\,[Q]\,w$ and $z\,[P]\,w$.

□

The relationships among x, y, z and w are represented by the following commutative isomorphism diagram.

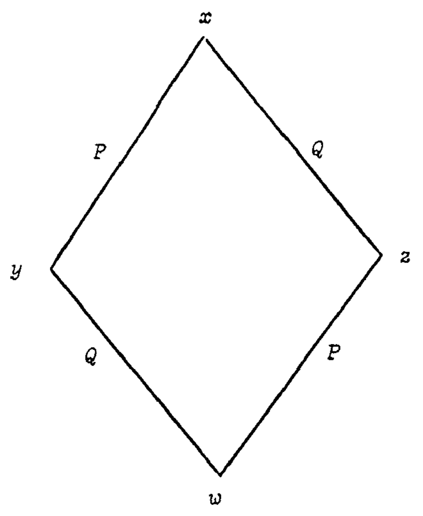

Figure 3-2: Isomorphism Diagram Depicting Fusion

Proof of the Lemma:
Let $w = x; (x,y); (x,z)$.

From the condition of the lemma, (x, y) has events only on $\overline{P}$ and (x, z) has events only on $\overline{Q}$. Since $P \cup Q = D$, $\overline{P} \cap \overline{Q} = \{\ \}$ and hence no process has events in both (x,y) and (x,z). It follows, from definition of computations, that w is a computation. Also $y\,[\,Q\,]\,w$, $z\,[P]\,w$ and $x \leq w$, as required for proof of the lemma.

□

Note that, in the construction of lemma 2, all events from E and $\overline{E}$ were present in the fused computation. We prove a far more general result below. We show that for any two arbitrary computations y and z, the projected computations, y_P and $z_{\overline{P}}$, may be fused to form a new computation provided there is a computation x which is a prefix of both y and z, and no message sent by $\overline{P}$ in (x,y) is received by P in (x,y) and no message sent by P in (x,z) is received by $\overline{P}$ in (x,z). This makes intuitive sense: processes in P can execute all events in y given only that processes in $\overline{P}$ execute all events up to x and similarly for executions of events on $\overline{P}$ up to z. However, the statement and proof of this result are difficult without the notion of isomorphism. We note that the result may be easily generalized to fusions of arbitrary numbers of computations under similar constraints.

Theorem 2: (Fusion of Computations): Consider system computations x, y, z where $x \leq y$ and $x \leq z$. Let P be a set of processes such that there is no process chain, (1) $<P\,\overline{P}>$ in (x, y) and (2) $<\overline{P}\,P>$ in (x, z). Then there is a computation w where, $x \leq w$, $y\,[\overline{P}]\,w$ and $z\,[P]\,w$. That is, w consists of all events on $\overline{P}$ from y and all events on P from z.

□

Proof of the Theorem: According to theorem 1, absence of process chains as given in this theorem means that, $x\,[\,P\,\overline{P}\,]\,y$ and $x\,[\overline{P}\,P]\,z$.

Consider the isomorphism diagram in Fig. 3-3. Label the intermediate point between x, y as u and between x, z as v in this figure. Now we apply lemma 1 to x, u, v to obtain w. Note that, $u\,[\overline{P}]\,y$ and $u\,[\overline{P}]\,w$; hence $y\,[\overline{P}]\,w$. Similarly $z\,[P]\,w$. This proves the theorem.

□

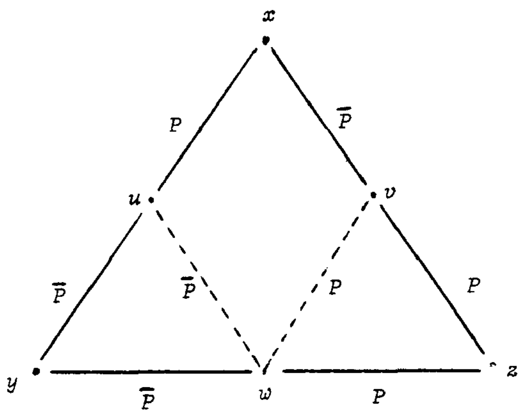

Figure 3-3: Isomorphism Diagram Depicting Proof of Fusion Theorem

3.4. Semantics Of Event Types In Terms Of Isomorphism

We now use isomorphism to state and derive some important facts about various types of events. First, note that a process carries out an internal event or sends a message depending on its own computation alone. Therefore, if a process takes such a step in a computation x, it will also do so in y, if x, y are isomorphic with respect to this process. An analogous result holds for internal and receive events. The following principle, which states these facts formally, may be proven from the definition of system computation.

Principle of Computation Extension::

Let e be an event on P.

1. e is an internal or send event:
   ($x\,[P]\,y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation

2. e is an internal or receive event:
   $(x;e)\,[P]\,y$ *implies* $(y-e)$ is a computation, where $(y-e)$ is the sequence obtained by deleting e from y.

□

Note: In (1), $(x;e)\,[P]\,(y;e)$ and in (2), $x\,[P]\,(y-e)$.

Corollary: Let e be a receive event on P and let the corresponding send event be on Q.

($x\,[P \cup Q]\,y$ *and* $(x;e)$ is a computation) *implies* $(y;e)$ is a computation.

□

Proof: e is an internal event of $P \cup Q$.

□

Following theorem follows from the principle of computation extension.

Theorem 3: Let $(x;e)$ be a computation where e is an event on P.

Case 1: e is a receive:

for every z: $(x;e)\,[P\overline{P}]\,z$ *implies* $x\,[P\overline{P}]\,z$

Case 2: e is a send:

for every z: $x\,[P\overline{P}]\,z$ *implies* $(x;e)\,[P\overline{P}]\,z$

Case 3: e is an internal event:

for every z: $(x;e)\,[P\overline{P}]\,z = x\,[P\overline{P}]\,z$

□

Proof: We will prove only Case 2; other cases are similarly proven.

$x\,[P\overline{P}]\,z$ *implies* there exists y, $x\,[P]\,y$ *and* $y\,[\overline{P}]\,z$. From principle of computation extension, $(y;e)$ is a computation and $(x;e)\,[P]\,(y;e)$.

Also, $(y;e)\,[\overline{P}]\,y$.

Hence, $(x;e)\,[P\overline{P}\,\overline{P}]\,z$ and therfore, $(x;e)\,[P\overline{P}]\,z$.

□

This theorem captures the intuitive notion that the set of possible computations, isomorphic with respect to P, can only shrink in size as a result of a reception as computations which do not include the corresponding send are ruled out. Similarly, the set of possible computations, isomorphic with respect to P cannot shrink as a result of a send: after the send, additional computations which accept the message sent are isomorphic while all prior isomorphic computations remain isomorphic. An internal event can neither expand nor shrink the set of isomorphic computations.

4. Knowledge

As we have remarked earlier, predicates of the type P *knows* b *at* x may be defined using isomorphism. We explore properties of such predicates in our model. We show that they satisfy the "knowledge axioms" as given in [3,6]. We prove a general result which shows that certain forms of knowledge can only be gained or lost in a sequential fashion along a chain of processes. That is, if b is false for a computation and later, P_1 *knows* P_2 *knows* ... P_n *knows* b (this represents knowledge gain), then there is a process chain $<P_n\, P_{n-1}\, \ldots\, P_1>$ between these two points of the computation. Conversely, if P_1 *knows* P_2 *knows* ... P_n *knows* b and later, b is false (this represents knowledge loss), then there is a process chain $<P_1\, P_2\, \ldots\, P_n>$ between these two points of the computation.

Crucial to our work is the notion of *local predicates*: a predicate local to p can change in value only as a result of events on p. We show that local predicates play a key role in understanding knowledge predicates.

4.1. Knowledge Predicates

Let b denote a predicate on system computations and "b *at* x" its value for computation x. Our predicates are *total*, i.e. for each x, b *at* x is either *true* or *false*. We furthermore assume that $x\,[D]\,y$ *implies* (b *at* $x = b$ *at* y) for every predicate b. Thus predicate values depend only upon computations of component processes and not on the way independent events are ordered in a linear representation of the computation. A predicate c is a *constant* means c *at* $x = c$ *at* y, for all computations x, y. We now define (P *knows* b) *at* x.

Definition: (P *knows* b) *at* $x =$ for all y: $x\,[P]\,y : b$ *at* y

Note that b may itself be a predicate of the form Q *knows* b' in the above definition. We next note some facts about knowledge predicates. In the following, x, y are arbitrary computations, b, b' are arbitrary predicates and P, Q are arbitrary sets of processes. All facts are universally quantified over all computations. We use the convention that P *knows* Q *knows* b *at* x is to be interpreted as (P *knows* (Q *knows* b)) *at* x.

1. P *knows* b *at* $x =$ for all y: $x\,[P]\,y$: P *knows* b *at* y
2. $x\,[P]\,y$ *implies* [P *knows* b *at* $x = P$ *knows* b *at* y]
3. (P *knows* b) *implies* ($P \cup Q$ *knows* b)
4. (P *knows* b) *implies* (b)
5. (P *knows* b) *or* ($\sim P$ *knows* b)
6. (P *knows* b) *and* (P *knows* b') $= P$ *knows* (b *and* b')
7. ((P *knows* b) *or* (P *knows* b')) *implies* (P *knows* (b *or* b'))
8. (P *knows* $\sim b$) *implies* ($\sim P$ *knows* b)
9. ((P *knows* b) *and* (b *implies* b')) *implies* (P *knows* b')
10. P *knows* P *knows* $b = P$ *knows* b
11. P *knows* $\sim P$ *knows* $b = \sim P$ *knows* b
12. P *knows* c, for any constant c.

These facts are easily derivable from the definition of *knows*. We give a proof of (11), whose validity in other domains have been questioned on philosophical grounds [3].

Lemma 2: P *knows* $\sim P$ *knows* $b = \sim P$ *knows* b □

Proof: P *knows* $\sim P$ *knows* b *at* x
= for all y: $x\,[P]\,y$: $\sim P$ *knows* b *at* y, from definition
= for all y: $x\,[P]\,y$: there exists z: $y\,[P]\,z$: $\sim b$ *at* z, from definition
= there exists z: $x\,[P]\,z$: $\sim b$ *at* z, since $[P]$ is an equivalence relation
= $\sim P$ *knows* b *at* x

□

There are situations where multiple levels of knowledge such as, P *knows* Q *knows* b, are useful. For instance, consider a *token bus* which is a linear sequence of processes among which a token is passed back and forth; processes at the left or right boundary have only a right or left neighbor to whom they may pass the token; other processes may send it to either neighbor. There is only one token in the system and initially it is at the leftmost process. Consider a token bus with five processes labelled p, q, r, s, t from left to right. When r holds the token,

r *knows* ((q *knows* (p does not hold the token)) *and* (s *knows* (t does not hold the token)))

Relations of the form $[P\,Q]$, with multiple process sets arise from predicates with multiple occurrence of *knows*;

For instance:
p knows q knows b at z
$=$ for all y: $x\,[\,p\,]\,y$: *q knows b at y*
$=$ for all y: $x\,[\,p\,]\,y$: (for all z: $y\,[\,q\,]\,z$: *b at z*)
$=$ for all z: $x\,[\,p\ q\,]\,z$: *b at z*

4.2. Local Predicates

Let b be a predicate on system computations, and P a set of processes. We define a predicate *P sure b* as follows.

Definition: (*P sure b*) *at x* $\equiv$ [(*P knows b*) *at x or* (*P knows* $\sim$*b*) *at x*]

In other words (*P sure b*) *at x* means that P knows the value of *b at x*.

We define *unsure* as negation of *sure*.

Definition: *P unsure b* $\equiv$ $\sim$*P sure b*

Hence, (*P unsure b*) *at x* $=$ [($\sim$*P knows b*) *at x and* ($\sim$*P knows* $\sim$*b*) *at x*]

Definition: b is *local* to P $\equiv$ for all x: (*P sure b*) *at x*.

That is, the value of b is *always* known to P. Local predicates capture our intuitive notion of a predicate whose value is controlled by the actions of processes to which it is local.

We note the following facts about local predicates; in the following, b is an arbitrary predicate and *P, Q* are arbitrary sets of processes.

1. (*b is local to P and* $x\,[P]\,y$) *implies* (*b at x* $=$ *b at y*)
2. *b is local to P implies* (b $=$ *P knows b*)
3. *b is local to P* $=$ ($\sim$*b*) *is local to P.*
4. *b is local to P implies* [*Q knows b* $=$ *Q knows P knows b*]
5. (*P knows b*) *is local to P.*
6. *b is local to P and b is local to Q and P,Q* are disjoint *implies b* is a constant.
7. b is a constant *implies b is local to P.*
8. (*P sure b*) *is local to P.*

Proof of (1) follows from definition of knowledge and local predicates. (2) and (3) follow trivially. (4) follows from *Q knows b at x* $=$ for all y: $x\,[\,Q\,]\,y$: *b at y* $=$ for all y : $x\,[\,Q\,]\,y$: *P knows b at y* (since b is local to P) $=$ *Q knows P knows b at x.* (5) follows from, (*P knows P knows b* or *P knows* $\sim$*P knows b*) $=$ (*P knows b* or $\sim$*P knows b*) $=$ *true.* Proof of (6) is important and hence is given below as a lemma. (7) and (8) are trivially proven from definition.

Lemma 3: *b is local to* disjoint sets *P, Q implies* b is a constant

□

Proof: We show that *b at x* $=$ *b at null*, for all x. Proof is by induction on length of x.

b at null $=$ *b at null.*
b at (x;e) $=$ *b at x*, because event e is not on P or e is not on Q, and hence
$(x;e)\ [\,P\,]\ x$ or $(x;e)\ [\,Q\,]\ x$;
then the result follows from property (1).

□

For a system of processes, *b is common knowledge* is defined as the greatest fix point of the following equation.

b is common knowledge $\equiv$ *b and* (*p knows b*) *is common knowledge*, for all processes p. Intuitively, *b is common knowledge* means b is *true*, every process *knows b*, every process *knows* that every process *knows b*, etc.

Halpern and Moses [3] have shown that common knowledge cannot be gained, if it was not present initially, in a system which does not admit of simultaneous events. The following corollary to lemma 3 shows that common knowledge can *be neither gained nor lost* in distributed systems.

Corollary: In a system with more than one process, for any predicate *b, b is common knowledge* is a constant.

□

Proof: For any process p, *b is common knowledge* $=$ *p knows (b is common knowledge)*. Hence, *b is common knowledge* is local to every p. Applying lemma 3, *b is common knowledge* is a constant.

□

It is possible to show that even weaker forms of knowledge cannot be gained or lost in our model of distributed systems. Process sets P, Q have *identical knowledge* of b means,

P knows b $=$ *Q knows b*

Corollary: If P, Q are disjoint and have identical knowledge of b then *P knows b* (and also *Q knows b*) is a constant.

□

Proof: *P knows b* is local to P and *Q knows b* is local to Q. From *P knows b* $=$ *Q knows b*, they are also local to Q and P respectively. The result follows directly from lemma 3.

□

Corollary: If P,Q are disjoint and *P sure b* $=$ *Q sure b*, then *P sure b* (and also *Q sure b*) is a constant.

□

4.3. How Knowledge Is Transferred

We show in this section that chains of knowledge are gained or lost in a sequential manner.

Theorem 4: For arbitrary process sets $P_1 \ldots, P_n$, $n \geq 1$, predicate b and computations x, y,

(P_1 *knows* $\ldots P_n$ *knows b at x and* $x\,[P_1 \ldots P_n]\,y$) *implies* (P_n *knows b at y*)

□

Proof: Proof is by induction on n. For $n = 1$, P_1 *knows b at x*, $x\,[P_1]\,y$ *implies* P_1 *knows b at y*, trivially.

Assume the induction hypothesis for some $n-1$, $n > 1$, and assume

P_1 *knows* $\ldots P_n$ *knows b at x* and $x\,[P_1 \ldots P_n]\,y$.

We shall prove P_n *knows b at y*.

From $x\,[P_1 \ldots P_n]\,y$, we conclude that there is a z such that,

$x\,[P_1 \ldots P_{n-1}]\,z$ and $z\,[P_n]\,y$.

From $x\,[P_1 \ldots P_{n-1}]\,z$ and P_1 *knows* $\ldots$ P_{n-1} *knows* (P_n *knows b*) *at x*, we conclude, using induction, P_{n-1} *knows* P_n *knows b at z*. Hence, P_n *knows b at z*.

Since $z\,[P_n]\,y$, P_n *knows b at y*.

□

Corollary: For arbitrary process sets $P_1 \ldots P_n$, $n \geq 1$, predicate b and computations x, y,

(P_1 *knows* $\ldots P_{n-1}$ *knows* $\sim P_n$ *knows b at x and* $x\,[P_1 \ldots P_n]\,y$) *implies* $\sim P_n$ *knows b at y*

□

Note: For $n = 1$ antecedant is, $\sim P_n$ *knows b at x*.

Corollary: Theorem 4 holds with *knows* replaced by *sure*.

Theorem 4 can be applied to (1) $x \leq y$ (knowledge is lost) and (2) $y \leq x$ (knowledge is gained). Using theorem 1, we can deduce that there is a process chain $< P_1 \ldots P_n >$ in the former case and $< P_n \ldots P_1 >$ in the latter case. We first prove a simple lemma about the effect of receive or send on knowledge: we show that certain forms of knowledge cannot be lost by receiving nor gained by sending.

Lemma 4: (How events at a process change its knowledge)

Let b be a predicate which is local to $\overline{P}$ and (x;e) a computation where e is an event on P.

1. e is a receive: {knowledge is not lost}
   (*P knows b at x*) *implies* (*P knows b at* (x;e))

2. e is a send: {knowledge is not gained}
   (P *knows* b *at* $(x;e)$) *implies* (P *knows* b *at* x)

3. e is an internal event: {knowledge is neither lost nor gained}
   (P *knows* b *at* x) = (P *knows* b *at* $(x;e)$)

□

Proof: We prove only (1). Consider any z such that $(x;e)\ [P]\ z$. We will show b *at* z and hence it follows that P *knows* b *at* $(x;e)$.

Since $z\ [\overline{P}]\ z$, we have $(x;e)\ [P\overline{P}]\ z$.

From theorem 3, since e is a receive, $x\ [P\overline{P}]\ z$. Since b is local to $\overline{P}$,

P *knows* b = P *knows* $\overline{P}$ *knows* b.

From theorem 4,

(P *knows* $\overline{P}$ *knows* b *at* x, $x\ [P\overline{P}]\ z$) *implies*
($\overline{P}$ *knows* b *at* z)
($\overline{P}$ *knows* b *at* z) *implies* (b *at* z)

This completes the proof.

□

Corollary: (b is local to $\overline{P}$, $\sim P$ *knows* b *at* x, P *knows* b *at* y, $x \leq y$) *implies* (P receives a message in (x, y)).

□

Corollary: (b is local to $\overline{P}$, P *knows* b *at* x, $\sim P$ *knows* b *at* y, $x \leq y$) *implies* (P sends a message in (x, y)).

□

Theorem 5: (How Knowledge Is Gained:) Let x, y be computations where $x \leq y$, $\sim(P_n$ *knows* $b)$ *at* x and (P_1 *knows* ... P_n *knows* b) *at* y, for arbitrary process sets $P_1 \ldots P_n$, $n \geq 1$. Then there is a process chain $<P_n \ldots P_1>$ in (x, y).

Furthermore, if b is local to $\overline{P}_n$ then P_n has a receive event in (x, y) such that b *at* z holds for every prefix z of y which includes the corresponding send event.

□

Theorem 6: (How Knowledge Is Lost:)

Let x, y be computations where $x \leq y$, P_1 *knows* ... P_n *knows* b *at* x and $\sim P_n$ *knows* b *at* y, for arbitrary process sets $P_1 \ldots P_n$, $n \geq 1$. Then there is a process chain $<P_1 \ldots P_n>$ in (x, y). Furthermore, if b is local to $\overline{P}_n$ then P_n has a send event in (x, y).

□

Observe that the statements of the two theorems are not entirely symmetric for receive and send events. The reason is that every computation including a receive must also include the corresponding send, but not conversely.

Theorems 4, 5, 6 and their corollaries hold with *knows* replaced by *sure*.

5. Applications Of The Results

We sketch a few applications of the theory developed so far. A full treatment of these results may be found in [8].

We show that it is impossible for process P to track the change in value of a local predicate of $\overline{P}$, exactly at all times; P must be unsure about the value of this predicate while it is undergoing change. We also show that necessary condition for changing a local predicate b of $\overline{P}$, is that $\overline{P}$ *knows* P *unsure* b, at the point of change.

Traditional techniques for process failure detection based on time-outs assume certain execution speeds for processes and maximum delays for message transfer. It is generally accepted that detection of failure is impossible without using time-outs, a fact that we prove formally. We use the fact that failure of a process is local to the process and the process does not send messages after its failure; hence other processes remain unsure at all points about a process failure.

We show that any algorithm, which detects termination of an underlying computation, requires at least as many overhead messages, in general, for detection as there are messages in the underlying computation. We first show that in order for termination to be detected, an overhead message is sent by some process, without its first receiving a message, after the underlying computation terminates; this fact is proven directly from the theorem of knowledge gain, because detecting termination amounts to gaining knowledge.

Next we show that a process is sometimes required to send an overhead message even when the underlying computation has not terminated, because the computation may be isomorphic (with respect to this process) to a computation in which the underlying computation has terminated. Using these two results, we construct a computation, in which the number of overhead messages is at least as many as the number of underlying messages.

6. Discussion

We have shown that isomorphisms between system computations with respect to a process is a useful concept in reasoning about distributed systems. Isomorphism forms the basis for defining and deriving properties about knowledge. "Scenarios" have been used [7] to show impossibility of solving certain problems; in our context, a scenario is a computation, and isomorphism is the formal treatment of equivalence between scenarios. Theorems on knowledge transfer provide lower bounds on numbers of messages required to solve certain problems. We have used isomorphism as the basis of fusion theorem and related isomorphism to semantics of send, receive and internal events.

A number of generalizations of this work are possible: we can define isomorphism based on states of processes, rather than computations; we can introduce the notion of time into computations; we can define *belief* in terms of isomorphism. Most of the results in this paper are applicable in the first case but not in the other two cases.

Acknowledgement: We are indebted to Shmuel Katz, Joe Halpern, E.W. Dijkstra and Bengt Jonsson for their comments. Particular thanks go to Ernie Cohen for a careful reading of the manuscript and insightful comments.

This work was supported in part by a grant from the Office of Naval Research under N00014-85-K-0057.

REFERENCES

1. K. M. Chandy & J. Misra: "Drinking Philosophers Problem", *TOPLAS*, October 1984.

2. M. J. Fischer, N. Lynch & M. Paterson, "Impossibility of Distributed Consensus with one Faulty Process", *Journal of the ACM*, April 1985.

3. J. Y. Halpern & Y. Moses: "(Knowledge And Common Knowledge In A Distributed Environment", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Vancouver, Canada, August 1984.

4. J. Hintikka: "*Knowledge and Belief*", *Cornell University Press*, 1962.

5. L. Lamport:, "Time, Clocks and the Orderings of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, pp. 558-564, July 1978.

6. D. Lehmann, "Knowledge, Common Knowledge, and Related Puzzles", *ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, Vancouver, Canada, August 1984.

7. N. Lynch & M. Fischer, "A Lower Bound for the Time to Assure Interactive Consistency", *Information Processing Letters*, Vol. 14, No. 4, June 1982.

8. K. M. Chandy & Jayadev Misra, "How Processes Learn", Distributed Computing, Vol. 1, No. 1, January 1986, (Published by Springer Verlag).