



National Library  
of Canada

Bibliothèque nationale  
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada  
K1A 0N4

## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.



National Library  
of Canada

Bibliothèque nationale  
du Canada

Canadian Theses Service    Service des thèses canadiennes

Ottawa, Canada  
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59147-1

Distributed Global State Detection System -  
Specification and Design

Dimitrios Livas

A Thesis  
in  
The Department  
of  
Computer Science

Presented in Partial Fulfillment of the Requirements for  
the Degree of Master of Computer Science at  
Concordia University  
Montréal, Québec, Canada

August 1990

© Dimitrios Livas, 1990

## Abstract

### Distributed Global State Detection System - Specification and Design

Dimitrios Livas

Global state is an important concept in distributed systems where a common clock is not available because in it lies the key to solving many problems: from program design to program analysis. In this thesis, a formal model, for specifying a distributed system and its global state using a partial-ordered model, is given. The requirements for local state recording and global state compilation for a periodically recording system are formulated as well, and algorithms are proposed. We show that global time, a refinement of logical time is helpful in compiling global states. The compilation algorithm guarantees that there is no avalanche of rolling forward of local states retained at the compilation process even though global coordination of recording is purposely removed. An experimental study confirms our results and demonstrates various aspects of such a system. Subsequently an object-oriented design is proposed of a distributed kernel that provides user and system processes with global state information.

**Key Words:** *Global state, periodic recording, global time, logical time, partial-order, state compilation, distributed computing system, object-oriented distributed system.*

Dedicated to my parents, Antonia and Georgo.

*Αφιερωμένο στους γονείς μου, Αντωνία και Γιωργό.*

## Acknowledgments

I would like to express my deepest appreciation to my supervisor, Dr. H. F. Li, who was always available when I needed him. His guidance, insight and knowledge gave me the motivation to complete this work. I would also like to thank him for his financial support.

Special thanks to my friend, Rick Clark, for proofreading and helping me to improve this thesis.

I would also like to thank my friends and colleagues in the Department for all the good and bad times.

Last but not least I thank *“τον πιστοτερο μου φιλο”*, Γιασμινά.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Modelling Distributed Computing Systems (DCS) . . . . .	2
1.2	DCS Specification in Pomsets . . . . .	4
1.3	Synchronization in Distributed Systems . . . . .	5
1.4	Distributed Global State Recording and Compilation . . . . .	6
1.5	The Global State Detection Kernel . . . . .	7
<b>2</b>	<b>Pomset Model of a Distributed Computing System</b>	<b>8</b>
2.1	Pomsets . . . . .	8
2.1.1	Pomset Algebra . . . . .	9
2.2	Modelling Distributed Systems Using Pomsets . . . . .	12
2.3	Process States and Consistency . . . . .	19
<b>3</b>	<b>Pomset Model of the Global State Detection System</b>	<b>22</b>
3.1	Recording Process . . . . .	22
3.2	A Complete State Recording and State Compilation System . . . . .	24
3.3	Compilation Algorithm . . . . .	26
3.4	Refinement of the Model . . . . .	26
<b>4</b>	<b>Logical and Global Time</b>	<b>27</b>

4.1	Logical and Global Time and their Properties . . . . .	28
<b>5</b>	<b>State Recording and State Compilation Algorithms</b>	<b>35</b>
5.1	A State Recording Algorithm . . . . .	35
5.2	State Compilation Algorithms . . . . .	37
5.2.1	Using Logical Time . . . . .	37
5.2.2	Using Global Time . . . . .	43
5.3	Implementation and Testing of the LTCA and GTCA Algorithms . . . . .	45
<b>6</b>	<b>The Design of the Global State Detection Kernel (GSDK)</b>	<b>55</b>
6.1	GSDK Objectives . . . . .	55
6.2	Object Oriented Distributed Systems . . . . .	59
6.3	Overview of GSDK . . . . .	68
<b>7</b>	<b>GSDK Processes</b>	<b>75</b>
7.1	Process Structure . . . . .	75
7.2	User Defined Program . . . . .	77
7.2.1	Dining Philosophers in GSDKL . . . . .	82
7.3	Process State Variables . . . . .	90
7.4	Process Code . . . . .	93
7.5	Site Port . . . . .	95
7.6	Implementation Issues . . . . .	97
<b>8</b>	<b>GSDK Site Control Processes</b>	<b>100</b>
8.1	Site Controller Structure . . . . .	100
8.2	Port Structure . . . . .	103
8.3	Process Table Structure . . . . .	105
8.4	Site Global Clock Structure . . . . .	106
<b>9</b>	<b>GSDK Central Sites</b>	<b>108</b>
9.1	General Central Site Structure . . . . .	108



9.2	Leaf Central Site Structure . . . . .	109
9.3	Central Site Structure . . . . .	111
9.4	Global State Database . . . . .	111
<b>10 Concluding Remarks and</b>		
	<b>Future Directions</b>	<b>113</b>
10.1	Use of Pomsets for DCS Modelling . . . . .	113
10.2	DCS Specification in Pomsets . . . . .	114
10.3	Logical and Global Time Properties . . . . .	114
10.4	State Recording and State Compilation Algorithms . . . . .	115
10.5	The GSDK Distributed Programming Environment . . . . .	115
10.6	Future Directions . . . . .	116
	<b>Bibliography</b>	<b>118</b>

# List of Figures

1.1	Expressiveness of concurrent specification models. . . . .	3
2.1	The basic pomset $p_{(n,t)}$ . . . . .	14
2.2	A pomset of the Distributed Computing System $DCS_2$ . . . . .	17
2.3	A Transition System $TS(p)$ made out of a pomset $p$ . . . . .	20
3.1	A recording algorithm of pomset $(a  b)ca$ . . . . .	24
3.2	A complete state recording and state compilation system. . . . .	25
4.1	Logical time labelling of events of pomset $p$ . . . . .	29
4.2	Global time labelling of pomset $p$ and Logical time labelling derived from global time labelling and the Recording Algorithm $LSR$ . . . . .	32
5.1	The compilation algorithm $LTCA$ . . . . .	41
5.2	The compilation algorithm $GTCA$ . . . . .	46
5.3	Distance (number of recordings) vs. recording interval for fully connected processes. . . . .	48
5.4	$GT$ average case, $GT$ worst case, $LT$ average case and $LT$ worst case vs. recording interval for fully connected processes (12 processes). . . . .	49
5.5	$GT'$ average case, $GT'$ worst case, $LT'$ average case and $LT'$ worst case vs. recording interval for fully connected processes (12 processes). . . . .	49
5.6	$GT'$ average case, $GT'$ worst case, $LT'$ average case and $LT'$ worst case vs. recording interval for ring connection (12 processes). . . . .	50
5.7	$GT$ average case, $GT$ worst case, $LT$ average case and $LT$ worst case vs. window size for fully connected processes (12 processes, recording interval = 12). . . . .	50

5.8	<i>GT average case, GT worst case, LT average case and GT worst case</i> vs. number of processes for fully connected processes (12 processes, recording interval = 6).	53
6.1	The structure of the GSDK.	69
6.2	The structure of the <i>process</i> class.	70
6.3	The structure of the SCP class.	71
6.4	The structure of the LCS class.	72
6.5	The structure of the CS class.	73

# List of Tables

5.1	Test results of 12 processes, fully connected where all processes have almost equal execution delays (number of recordings and number of events). . . . .	51
5.2	Test results (number of recordings and number of events) of 12 processes, fully connected where each communicating process is twice as slow as the previous one (assuming the processes are numbered in a sequence). . . . .	52

# Chapter 1

## Introduction

The need for more computing power from many applications as well as the rapid progress of microprocessor technology and communication systems have made *distributed computing systems* (DCS) a strong focus of interest.

A DCS consists of a number of processor/memory pairs, called nodes or sites, connected through a communication subsystem. It is characterized by the absence of a common memory and a common clock. Synchronization is achieved via message-passing.

A DCS provides high performance and reliability. Since an application may be decomposed into a number of concurrent computations executed in parallel, the throughput of a DCS may be much higher than that of a centralized system. Moreover, when a node crashes, its computations may be restarted at other functioning nodes.

However, despite the above mentioned advantages, distributed programs are difficult to develop. Software engineering of DCS and distributed algorithm design face the following problems:

- A well accepted model for specifying distributed computations is still not available.
- The lack of a common clock and memory makes instantaneous global state a non-existing concept.

## 1.1 Modelling Distributed Computing Systems (DCS)

The development of DCS requires their specification, design, implementation and testing. Several models have been proposed for the specification of distributed and parallel computations. Eventually these models will lead to the development of programming languages. Thus, distributed applications will be built by specifying their behaviour in terms of simple primitives provided, rather than dealing with how these specifications are mapped to an underlying architecture.

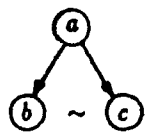

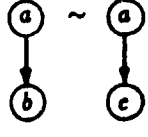
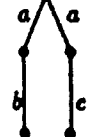

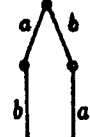
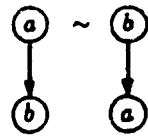

The most widely studied models are:

**Linear Set Models:** In the linear set models the behaviour of a system is defined as the set of all its possible execution linearizations. Concurrency is not a primitive notion; it is simulated with sequentiality and non-determinism. Moreover, nondeterministic choice is not distinguished from deterministic choice and nondeterministic choice may occur only at the initial state of the system.

**Branching Tree Models:** In branching tree models, like the linear set models, concurrency is not a primitive notion. However, nondeterministic choice is distinguished from deterministic choice and where the choice and nondeterminism may occur can be explicitly defined. The most successful representatives of the branching tree models are CCS [30] and CSP [14] models.

**Partial Order Event Structure Models:** In partial order event structure models [4], [48] concurrency is considered a primitive notion distinct from nondeterminism. Moreover where concurrency and nondeterminism occur may be defined explicitly.

Some examples of distributed computations expressed in the above described models are given in figure 1.1 from [35]. These examples show the expressiveness of these models. The letters  $a$ ,  $b$ ,  $c$  are used to identify actions to be executed. The program  $a(b + c)$  specifies that either  $b$  or  $c$  is executed after  $a$  has been executed.  $ab + ac$

Approach Behavior Program	Partial Order Event Structure	Branching Tree	Linear Set
$a(b + c)$			$\{ab, ac\}$
$ab + ac$			$\{ab, ac\}$
$a \parallel b$			$\{ab, ba\}$
$a! + ba$			$\{ab, ba\}$

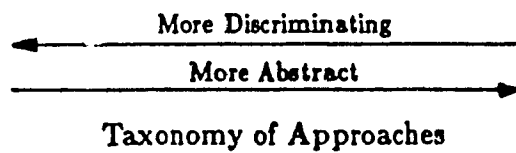


Figure 1.1: Expressiveness of concurrent specification models.

specifies that either  $a$  followed by  $b$  is executed or  $a$  followed by  $c$  is executed.  $a\parallel b$  specifies that  $a$  is executed in parallel with  $b$ .

We can observe that from the linear set to partial order event structure, the models become more discriminating, and following in the opposite direction the models become more abstract. None of the above models is ideal for specifying all distributed or parallel computations. The model in which a particular system is specified should be chosen according to the system needs. A more discriminating model may be avoided in case the extra primitives provided are not used.

## 1.2 DCS Specification in Pomsets

The *partially ordered multi-set* or *pomset* model of Gischer [13] and Pratt [36] provides the same primitives as the partial order event structure model except that nondeterminism may only be specified at the initial state of the system. In this thesis Pratt's pomsets are used to create a general model for DCS specification due to their ability to distinguish between concurrency and nondeterminism. Subsequently, this model is used for the specification of some distributed algorithms useful in many applications. These algorithms are not sensitive to where nondeterminism occurs, however, the model is general and it could be used for the specification of algorithms and systems that are concerned with this aspect. For this purpose, the model can be easily ported to partial order event structures. In chapter 2 a brief introduction to pomset theory is given.

Partial order models have been characterized as monolithic. That is, systems specified using these models cannot be expressed in terms of component subsystems. However, in software engineering monolithic system specification and design is not acceptable. Systems must be defined in terms of reusable, expandable and independent modules.

In this thesis, a method to specify distributed computations, called process behaviours, using pomsets by inheriting the properties of other process behaviours, is introduced. Thus systems may be specified in terms of their component modules. The specification primitive used for this purpose is introduced in chapter 2 and is



called the *construction rule*.

Using *construction rule* we define process composition and decomposition by synchronizing the communication events of the component processes. The communication events of a process model the sending and receiving messages to/from other processes.

*Construction rule* and *process composition* are the basic tools that we use to specify a general model of a DCS. This model can be used for the specification of distributed algorithms and distributed applications where correctness may be proved in an easy formal way.

Our DCS pomset model is ideal for distributed algorithms that are concerned with the global state of a DCS. The model provides definitions of component (local) states of the system and composite (global) states of the system, as well as operations for state composition and decomposition.

### 1.3 Synchronization in Distributed Systems

Synchronization is an essential concept in concurrent and distributed computing. The cooperation of several concurrent processes, in an attempt to achieve a common goal, requires some means of synchronization. In centralized systems the common clock and memory serve the purpose. The events of these processes can be uniquely time-stamped and ordered according to these time labels.

However, synchronization in a distributed environment is much more tricky. Without a common clock and memory, conventional methods and tools are not useful. Lamport introduced a synchronization construct called logical time [21] which defines a *happened-before* relationship among the system events. The partial order of the occurrence of the events produces their logical time labeling.

In chapter 4, we further explore the properties of logical time. We prove that logical time cannot be used for some applications due to its weakness in identifying the partial ordering of events in a distributed computation. Subsequently, we show that *global time*, a refinement of logical time [25], [26], [40], uniquely identifies the events of a distributed computation and the partial order of the event execution is

revealed by their global time labels. Moreover we prove that global time also satisfies the logical time axiom, introduced in chapter 4. Thus, all properties of logical time are properties of global time as well.

## 1.4 Distributed Global State Recording and Compilation

An instantaneous global state of a DCS is defined by the states of the component sites and channels of the DCS at a given time instance. Due to the lack of a common clock, the observation of an instantaneous global state is impossible. However, the use of consistent global states of a DCS may be as useful as instantaneous states. Although a consistent global state  $S$  may never be observed instantaneously during the execution of a distributed computation, it is a state reachable from the initial state and can further lead to the current state of the system. A formal definition of consistent global state is given in chapter 2. Numerous global state detection algorithms have been proposed [5], [7], [10], [20], [23]. These algorithms share a common characteristic: local snapshots are taken with coordination throughout the system so that each invocation of the algorithm involves  $O(n^2)$  to  $O(e)$  messages passed among the  $n$  processes of a system with  $e$  channels. Moreover, these algorithms assume the recording of one global state so they are not concerned with global state compilation.

In chapter 5 we present a periodic recording algorithm based on logical time. Recording of local states is performed without explicit coordination among processes. The worst case time complexity of the algorithm is  $O(1)$ . Subsequently, we propose two compilation algorithms; one using logical time and the other using global time. Compilation of local states into global states makes use of known properties of logical time [6], [31] and its refinement, global time [25], [40]. The compilation algorithms maintain a small database of local states while guaranteeing the availability of a recent global state of the system. Periodically recorded global states may find application in the following areas:

**Fault tolerance:** Rollback and recovery is performed at the most recent global state

of the system [11], [17], [18], [16], [43], [45].

**Stability detection:** Continual detection of deadlock or termination etc. can be performed [6], [27], [28], [39].

**Distributed Debugging:** Periodic recording is useful for tracing program execution in debugging [22], [24], [49].

**Programming tool:** Interesting application algorithms that make use of global system information can be easily supported [41], [46].

The algorithms proposed in this thesis assume a reliable and FIFO communication subsystem. Protocols that ensure reliable and FIFO delivery, e.g. TCP/IP, buffer the messages in transit at the sender site until the delivery is acknowledged. Thus, a channel state that is defined by the messages in transit, can be revealed by the states of the sender and receiver sites.

Thus, we assume that the state of a DCS is revealed by the states of its component sites and ignore channel states in our theory.

## 1.5 The Global State Detection Kernel

The Global State Detection Kernel (GSDK) is a distributed programming environment that provides primitives for the development of distributed applications. It makes process creation, deletion and communication transparent. However, what makes GSDK special is that it provides applications with global or partial state information at their request. GSDK also provides a language in which distributed applications may be specified. It is a primitive programming language whose statements correspond to the programming constructs provided by the kernel.

The design of the GSDK is presented in chapters 6 to 9. It is based on the model and algorithms specified in chapters 2 to 5.

GSDK is an object oriented system. It is easy to expand, port, and modify. The objectives of the system are introduced in chapter 6.

# Chapter 2

## Pomset Model of a Distributed Computing System

The formalism followed in this thesis is based on *pomsets* (partially ordered multisets) as used by Gischer [13] and Pratt [36]. A brief introduction of the relevant parts is given in this chapter (more details are given in [13], [36]). Subsequently a DCS is modeled using pomsets.

### 2.1 Pomsets

A *labelled partial order* (lpo) is a 4-tuple  $(V, \Sigma, \Gamma, \mu)$  consisting of

1. a vertex set  $V$  for modelling events,
2. an alphabet  $\Sigma$  for modelling actions,
3. a partial order  $\Gamma$  on  $V$ ,

$$\Gamma \subseteq V \times V [(a, b), (b, c) \in \Gamma \Rightarrow c \neq a \wedge (a, c) \in \Gamma],$$

$$(e, f) \in \Gamma \iff e \rightarrow f \text{ (} e \text{ occurs before } f\text{),}$$

4. a labelling function  $\mu : V \rightarrow \Sigma$  assigning symbols (actions) to vertices (events), each labelled event represents an occurrence of the corresponding action; the same action may occur many times.

A pomset (partially ordered multiset) is the isomorphism class of an lpo, denoted  $[V, \Sigma, \Gamma, \mu]$ . There are several types of pomsets classified according to their characteristics :

- Multiset : a pomset where  $\Gamma = \emptyset$  (minimal order)
- Tomset : a pomset with a total (maximal) order  
 $(\forall a, b \in V [a \rightarrow b \vee b \rightarrow a])$
- String : a finite tomset
- Poset : a pomset where  $\mu$  is one to one and onto
- Set : a poset that is also a multiset
- Atom : a singleton pomset (both a set and a string)
- Unit : an emty pomset  $\epsilon$  (empty string and empty set)

A *process* is a set of pomsets, in the same manner that a language is a s t of strings. A set of pomsets specifies the nondeterministic behaviour of the process, where each member of the set is a possible behaviour.

### 2.1.1 Pomset Algebra

The following pomset operations are used in this thesis:

- *Concurrence*  $p \parallel q$  of two pomsets  $p, q$  is defined as

$$[V_p, \Sigma_p, \Gamma_p, \mu_p] \parallel [V_q, \Sigma_q, \Gamma_q, \mu_q] = [V_p \uplus V_q, \Sigma_p \cup \Sigma_q, \Gamma_p \cup \Gamma_q, \mu_p \cup \mu_q].$$

Where  $\uplus$  is used to denote the union of two disjoint sets.

- *Concatnation*  $p; q$  of two pomsets  $p, q$  is defined as

$$[V_p, \Sigma_p, \Gamma_p, \mu_p]; [V_q, \Sigma_q, \Gamma_q, \mu_q] = [V_p \uplus V_q, \Sigma_p \cup \Sigma_q, \Gamma_p \cup \Gamma_q \cup (V_p \times V_q), \mu_p \cup \mu_q].$$

Thus every event of  $p$  precedes every event of  $q$  in the resulting pomset.

- *Projection*  $proj(p, S)$  is a pomset  $r$  derived from pomset  $p$  restricted to events labelled with actions in a set  $S$  such that the causality among these events is

preserved. Formally,

$$r = \text{proj}(p, S) \iff \begin{cases} \Sigma_r = S \cap \Sigma_p, \\ V_r = \{ e \mid e \in V_p \wedge \mu_p(e) \in \Sigma_r \}, \\ \Gamma_r = (V_r \times V_r) \cap \Gamma_p, \\ \mu_r = (V_r \times \Sigma_r) \cap \mu_p. \end{cases}$$

- *Orthoccurrence*  $p \times q$  of two pomsets  $p, q$  is defined as

$$[V_p, \Sigma_p, \Gamma_p, \mu_p] \times [V_q, \Sigma_q, \Gamma_q, \mu_q] = [V_p \times V_q, \Sigma_p \times \Sigma_q, \Gamma, \mu]$$

where,

$$\Gamma = \{ ((a, b), (c, d)) \mid (a, c) \in \Gamma_p \wedge (b, d) \in \Gamma_q \},$$

$$\mu = \{ ((v, u), (a, b)) \mid (v, a) \in \mu_p \wedge (u, b) \in \mu_q \}.$$

Orthoccurrence is to concurrence as cartesian product is to disjoint union.

- *Prefix closure*  $\pi(p)$  of a pomset  $p$  is not a pomset but a process (set of pomsets) consisting of the set of prefixes of  $p$ .  $q$  is a prefix of  $p$  written  $q \leq_\pi p$ , when  $q$  is derived from  $p$  by deleting a subset of events of  $p$ , provided that if event  $u \in V_p$  is deleted and  $(u, v) \in \Gamma_p$  then  $v$  is also deleted.

**Example :**  $\pi(01 \parallel 2) = \{01 \parallel 2, 01, 0 \parallel 2, 0, 2, \epsilon\}$

- *Suffix closure*  $\sigma(p)$  of a pomset  $p$  is not a pomset but a process (set of pomsets) consisting of the set of suffixes of  $p$ .  $q$  is a suffix of  $p$  written  $q \leq_\sigma p$ , when  $q$  is derived from  $p$  by deleting a subset of events of  $p$ , provided that if event  $u \in V_p$  is deleted and  $(v, u) \in \Gamma_p$  then  $v$  is also deleted.

**Example :**  $\sigma(01 \parallel 2) = \{01 \parallel 2, 01, 1 \parallel 2, 1, 2, \epsilon\}$

- *Augment closure*  $\alpha(p)$  is the set of augments of  $p$ .  $q$  is an augment of  $p$  written  $p \leq_\alpha q$ , when  $q$  differs from  $p$  only in its partial order, which must be a superset of that of  $p$ .

**Example :**  $\alpha(01 \parallel 2) = \{01 \parallel 2, 012, 021, 201, 0(1 \parallel 2), (0 \parallel 2)1\}$

- *Linearization*  $\lambda(p)$  is the set of all linear augments of  $p$ .

**Example :**  $\lambda(01 \parallel 2) = \{012, 021, 201\}$ .

- *Partial linearization*  $\lambda_X(p)$  is the set of all augments of  $p$  where the events of  $p$  labelled  $X$  are totally ordered.

**Example :**  $\lambda_X(X1\parallel X) = \{X1X, XX1, X(X\parallel 1)\}$ .

- *Colocation* : The next two operations are defined only for alphabets of the form  $\Sigma = A \times B$ . We say that two events are *colocated* when each of their labels is a pair and the two labels have the same second component.

**Example :** consider the pomset  $p = (A, B)\parallel(A, C)\parallel(B, C)$ , the events labelled  $(A, C)$  and  $(B, C)$  are colocated because their labels have the same second component.

- *Local linearization*  $\Delta(p)$  is the set of linearizations of the colocated events of  $p$ .

**Example :**

$$\Delta((0, 1)\parallel(0, 0)\parallel(2, 1)) = \{ (0, 1)(2, 1)\parallel(0, 0), (2, 1)(0, 1)\parallel(0, 0)\}.$$

- *Local partial linearization*  $\Delta_X(p)$  where  $(X, Y) \in \Sigma_p$  ( $\Sigma_p$  is the labelling set of  $p$ ) is the set of linearizations of the colocated events of  $p$  where their first label component is  $X$  and they agree in their second label component.

**Example :**

$$\Delta_A((A, B)\parallel(A, B)\parallel(A, C)\parallel(B, C)\parallel(B, C)) =$$

$$\{(A, B)(A, B)\parallel(A, C)\parallel(B, C)\parallel(B, C)\}.$$

$$\Delta_{(A, B)}((A, B)\parallel(A, B)\parallel(A, C)\parallel(B, C)) =$$

$$\{(A, B)(A, B)\parallel(A, C)(B, C), (A, B)(A, B)\parallel(B, C)(A, C)\}.$$

- *Set operations* like union, intersection and set difference are used to define new processes ( sets of pomsets ).
- *Pomset homomorphism*  $h(p)$  replaces each vertex  $v$  of  $p$  with a pomset  $q$  determined by  $h$ . It is a size preserving homomorphism iff  $|V_q| = 1$  and it is used for the relabelling of events, else it is an expanding homomorphism.

- *Process homomorphism*  $h(p)$  transforms  $p$  to a set of pomsets  $Q$  where each member of  $Q$  is created by the replacement of any event  $v$  of  $p$  by a set of pomsets determined by  $h$ .

## 2.2 Modelling Distributed Systems Using Pomsets

We model a DCS as a set of communicating sites. Each site is represented by its computational behaviour specified by a process (in pomset context).

A process  $P$  is specified by a set of pomsets (behaviours)  $\{p_1, p_2, \dots\}$ . In the specification, only the *necessary* causality among the events in  $p_i$  is modeled and the following process rule applies:

**Rule 1 (PR (Process Rule))** *Let  $P$  be a process.*

$$p, q \in P \Rightarrow p \not\prec_a q$$

□

Intuitively, *PR* asserts that no behaviour in  $P$  is an augment of another behaviour in  $P$ , else the additional causality introduced by the augment is not necessary.

To complete our model of a distributed computation consisting of  $n$  processes, we introduce:

**Basic pomset**  $p_{(n,i,k_i[n],\ell_i[n])}$ : The basic pomset  $p_{(n,i,k_i[n],\ell_i[n])}$  specifies a maximal concurrent behaviour of a computation  $i$  that executes infinite number of internal events in parallel with the execution of a sequence of  $k_i[j]$  sending events, that model the sending of messages to a computation  $j$ , and a sequence of  $\ell_i[j]$  receiving events, that model the receiving of messages from the computation  $j$ .

**Basic process**  $P_{(n,i)}$ : Is the set of all possible computations  $p_{(n,i,k_i[n],\ell_i[n])}$  for a given  $i$ . Any two members of  $P_{(n,i)}$  differ in the arrays  $k_i$  and  $\ell_i$ . That is, they send or receive a different number of messages to or from at least one computation  $j$ .



**Construction rules:** They are logical expressions defined using pomset operations.

They are used to define process behaviours using basic pomsets and processes as basic building blocks.

**Definition 1 (Basic pomset  $p_{(n,i)}$  and Basic process  $P_{(n,i)}$ )**

$$p_{(n,i,k_i[n],\ell_i[n])} = (\|_{\infty}(I, i\beta\gamma)) \parallel (\|_{(j,i,n)}h(S \times i \times j \times \Delta_{k_i})) \parallel (\|_{(j,i,n)}h(R \times j \times i \times \Delta_{l_i}))$$

where,

$$\|_{\infty}X \quad = \quad X \parallel X \parallel \dots \parallel X \parallel X \parallel \dots$$

*(concurrent occurrence of infinite number of X's)*

$$\|_{(j,i,n)}f(j) = f(1) \parallel f(2) \parallel \dots \parallel f(i-1) \parallel f(i+1) \parallel \dots \parallel f(n)$$

$$\Delta_n \quad = \quad 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow (n-1) \rightarrow n$$

$h$  : is a size preserving pomset homomorphism which replaces each label  $(X, i, j, \ell)$  with  $(X, ij\ell)$

$\beta, \gamma$  : dummy integer values ( say always 0 ). They are introduced so that events are homogeneously labelled.

The basic process  $P_{(n,i)}$  is the set of all basic pomsets, i.e.  $P_{(n,i)} = \{p_{(n,i)} \mid k_{ij}, l_{ij} \in \omega \cup \infty, i \neq j, 1 \leq j \leq n\}$  ( $\omega$  is the set of non-negative integers).

□

**Example :**

$$S \times i \times j \times \Delta_3 = (S, i, j, 1) \rightarrow (S, i, j, 2) \rightarrow (S, i, j, 3)$$

$$h(S \times i \times j \times \Delta_3) = (S, ij1) \rightarrow (S, ij2) \rightarrow (S, ij3)$$

□

A pictorial representation of  $p_{(n,i)}$  is shown in Figure 2.1. Conceptually, a basic pomset  $p_{(n,i)}$  is a behaviour of the basic process  $P_{(n,i)}$  and contains a set of internal events labelled  $(I, i\beta\gamma)$ , a set of sending events labelled  $(S, ij\ell)$  and a set of receiving

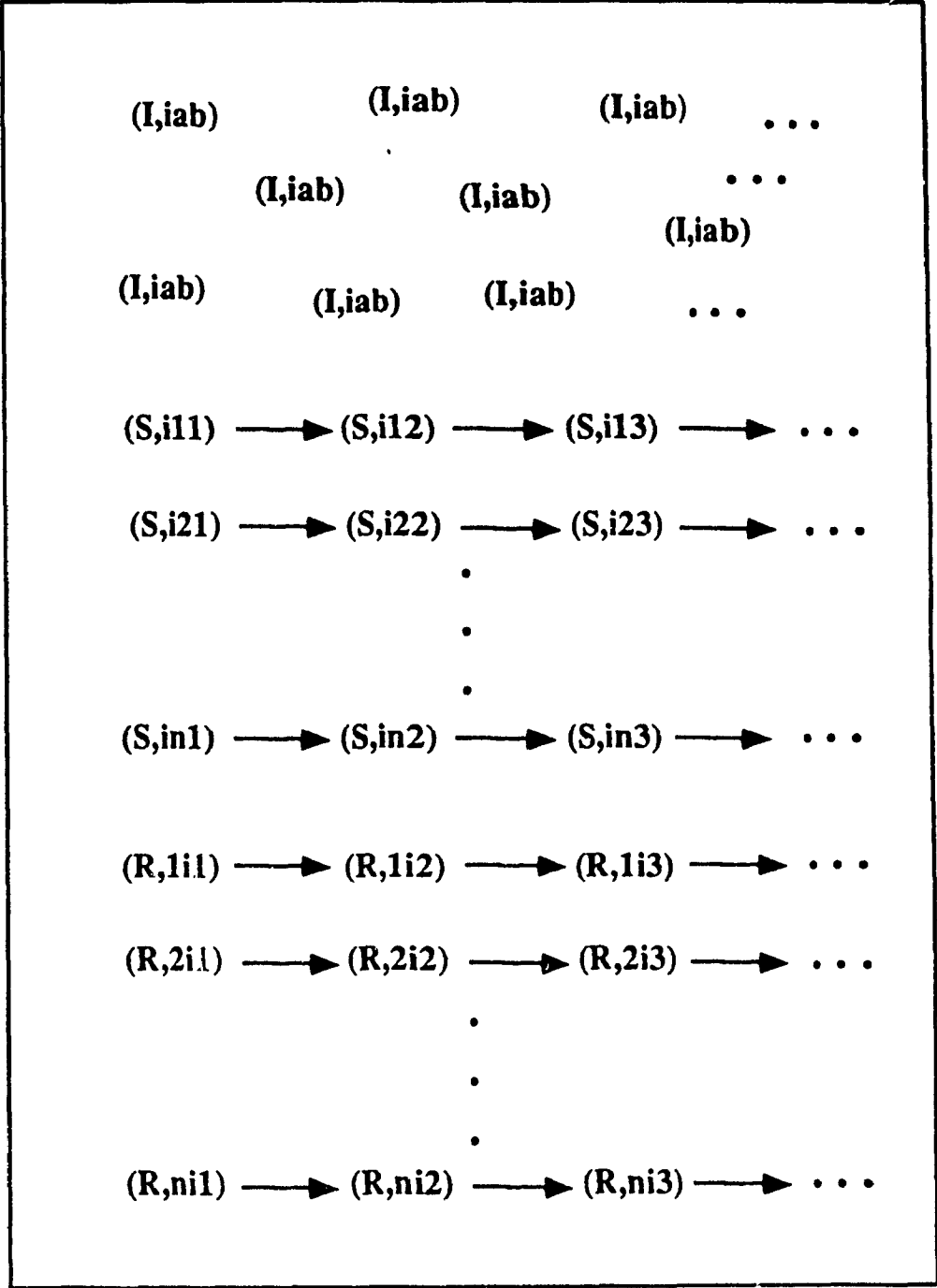


Figure 2.1: The basic pomset  $p_{(n,i)}$

events labelled  $(R, j\ell)$ . Semantically,  $(S, ij\ell)$  represents the  $\ell^{\text{th}}$  sending event from  $P_{(n,i)}$  to  $P_{(n,j)}$  and these events are totally ordered for each combination of  $i$  and  $j$ . Thus  $(S, ij\ell) \rightarrow (S, ij(\ell+1))$ . Similarly  $(R, j\ell)$  denotes the  $\ell^{\text{th}}$  receiving event from  $P_{(n,j)}$  by  $P_{(n,i)}$ .

Process  $DP_i$  models the concurrent execution of events on a node  $i$  of a DCS. The formal definition of this process is as follows:

**Definition 2** ( $DP_i$ ) *A distributed process  $DP_i$  contains a subset of augments of behaviours of the basic process  $P_{(n,i)}$ . Formally,*

$$DP_i \subseteq \alpha(P_{(n,i)}) [\bar{\Delta} p, q \in DP_i [p \leq_a q]], \quad 1 \leq i \leq n$$

□

We can apply pomset operations (defined in section 2.1) on the behaviours of  $\cup_i DP_i$  to obtain the behaviours of a new process. Different *construction rules* correspond to different uses of these operations. In general, we denote these rules by  $Const\_Rule_{P_n}(p, p_1, \dots, p_n)$ :  $p \in P_n$  is constructed by applying some specific rules to  $\{p_1, \dots, p_n\}$ ,  $p_i \in DP_i$ . Formally

$$P_n = \{ p \mid \forall i (1 \leq i \leq n) \exists p_i \in DP_i [Const\_Rule_{P_n}(p, p_1, \dots, p_n)] \}.$$

$DCS_n$  models a DCS of  $n$  nodes. Formally:

**Definition 3** ( $DCS_n$ ) *A distributed computing system  $DCS_n$  is formed by a set of distributed processes with linearization of colocated events (sending and receiving events). Formally,*

$$DCS_n = \{ p \mid \forall i (1 \leq i \leq n) \exists p_i \in DP_i [Const\_Rule_{DCS_n}(p, p_1, \dots, p_n)] \}$$

where,

$$\begin{aligned} Const\_Rule_{DCS_n}(p, p_1, \dots, p_n) \equiv & p \in (\Delta_{(R,S)}(\parallel, p_i)) \wedge \\ & (v \in V_{p_i} [\mu_p(v) = (S, ij\ell)] \Rightarrow \\ & (\exists u \in V_{p_j} [\mu_p(u) = (R, ij\ell)] \wedge (v, u) \in \Gamma_p)) \end{aligned}$$

$p = [V_p, \Sigma_p, \Gamma_p, \mu_p]$  and  $\Delta_{(R,S)}(\|, p_i)$  is the set of local partial linearizations of events labelled  $(S, x), (R, y)$  of pomset  $(\|, p_i)$ .

□

An example of a behaviour of a  $DCS_2$  is drawn in Figure 2.2. FIFO communication is enforced by including  $((S, ij\ell), (R, ij\ell))$  in  $\Gamma_p$ . The following properties of  $DCS_n$  follow immediately from the definition.

**Properties of  $DCS_n$ :**

**PR1 :**

$$p \in DCS_n [Const\_Rule(p, p_1, \dots, p_n)] \iff [V_p = \bigsqcup_i V_{p_i}, \Sigma_p = \bigsqcup_i \Sigma_{p_i}, \mu_p = \bigsqcup_i \mu_{p_i}, \Gamma_p = tr((\bigsqcup_i \Gamma_{p_i}) \bigsqcup \Gamma_{pRS})]$$

where,

$$\Gamma_{pRS} = \{(u, v) \mid u \in V_{p_i}, v \in V_{p_j}, \mu(u) = (S, ij\ell), \mu(v) = (R, ij\ell)\}.$$

$tr(\Gamma)$  is the transitive closure of the set of elements of  $\Gamma$ , that is,

$$((a, b) \in \Gamma \Rightarrow (a, b) \in tr(\Gamma)) \wedge ((a, b), (b, c) \in tr(\Gamma) \Rightarrow (a, c) \in tr(\Gamma)).$$

**PR2 :** All members of  $DCS_n$  are infinite pomsets. This is due to the infinite number of internal events of each  $p_i \in DP_i$ . We avoid to deal with finite computations so that termination is not an issue in our compilation algorithms and it can be easily handled in implementation.

□

The following theorem asserts that if  $Const\_Rule_{DCS_n}(p, p_1, \dots, p_n)$  holds then  $p$  uniquely identifies  $p_1, \dots, p_n$  and  $p_1, \dots, p_n$  uniquely identifies  $p$ .

**Theorem 1**  $\forall i p_i, q_i \in DP_i, p, q \in DCS_n,$

$$Const\_Rule_{DCS_n}(p, p_1, \dots, p_n) \Rightarrow ((Const\_Rule_{DCS_n}(p, q_1, \dots, q_n) \iff \forall i [q_i = p_i]) \wedge (Const\_Rule_{DCS_n}(q, p_1, \dots, p_n) \iff p = q)).$$

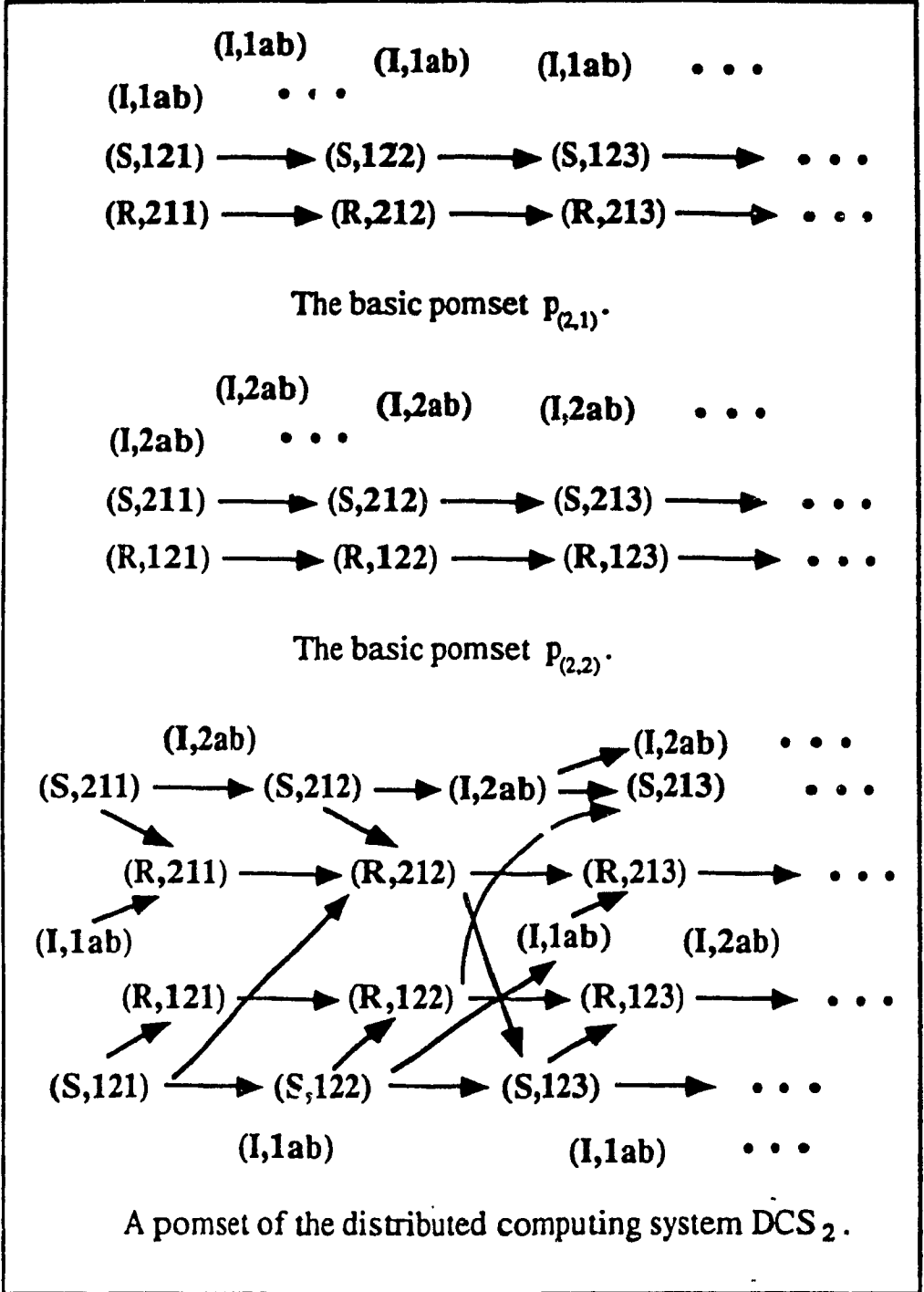


Figure 2.2: A pomset of the Distributed Computing System  $DCS_2$

**Proof:** (by contradiction)

Assuming  $Const\_Rule_{DCS_n}(p, p_1, \dots, p_n)$  holds while

1. the predicate  $\neg(Const\_Rule_{DCS_n}(p, q_1, \dots, q_n) \iff \forall i [q_i = p_i])$  is satisfied.

Then

either  $\neg Const\_Rule_{DCS_n}(p, q_1, \dots, q_n) \wedge \forall i [q_i = p_i]$  holds which contradicts

$$\neg Const\_Rule_{DCS_n}(p, p_1, \dots, p_n)$$

or  $Const\_Rule_{DCS_n}(p, q_1, \dots, q_n) \wedge \exists i [q_i \neq p_i]$  holds

$$\Rightarrow \exists q_i \neq p_i \wedge A = \Delta_{(R,S)}(\|i, p_i) \cap \Delta_{(R,S)}(\|i, q_i) \neq \emptyset \quad (\text{Definition 2})$$

$$\Rightarrow \forall r \in A [\Gamma_r = (\uplus_i \Gamma_{p_i}) \uplus \Gamma_{rRS} = (\uplus_i \Gamma_{q_i}) \uplus \Gamma_{rRS}] \quad (\text{Property PR1})$$

$$\Rightarrow \forall i \Gamma_{p_i} = \Gamma_{q_i}.$$

$$\text{But } p_i, q_i \in DP_i = \alpha(p_{(n,i)}) \Rightarrow \Gamma_{p_i} \neq \Gamma_{q_i} \quad (\text{Contradiction}).$$

2. (Conversely) the predicate  $\neg(Const\_Rule_{DCS_n}(q, p_1, \dots, p_n) \iff p = q)$  is satisfied. Then

either  $\neg Const\_Rule_{DCS_n}(q, p_1, \dots, p_n) \wedge p = q$  holds which contradicts

$$\neg Const\_Rule_{DCS_n}(p, p_1, \dots, p_n)$$

or  $Const\_Rule_{DCS_n}(q, p_1, \dots, p_n) \wedge p \neq q$  holds

$$\Rightarrow V_p = V_q = \uplus_i V_{p_i}, \Sigma_p = \Sigma_q = \uplus_i \Sigma_{p_i}, \mu_p = \mu_q = \uplus_i \mu_{p_i}, \quad (\text{Property PR1})$$

$$\Gamma_p = (\uplus_i \Gamma_{p_i}) \uplus \Gamma_{pRS}, \Gamma_q = (\uplus_i \Gamma_{p_i}) \uplus \Gamma_{qRS},$$

$$\Gamma_{pRS} = \Gamma_{qRS} = \{(u, v) \mid u \in V_{p_i}, v \in V_{p_j} \ (i \neq j),$$

$$\mu(u) = (S, ij), \mu(v) = (R, ij)\}$$

$$\Rightarrow p = q \quad (\text{Contradiction}).$$

□

**Definition 4 (Construction Operator of  $DCS_n$ )** The following notation is defined for future use.

$$Const\_Rule_{DCS_n}(p, p_1, \dots, p_n) \iff p = p_1 \circ \dots \circ p_n$$

□

## 2.3 Process States and Consistency

Under the assumption of a reliable and FIFO communication subsystem the state of a DCS can be defined by the states of its component sites. Moreover a state of a site is identified by the computation that lead the site from the initial state to the current one. We denote  $STATES_P$  the set of states of a site or a DCS identified by the computational behaviour  $P$ .

### Definition 5

1. A process  $P$  has one and only one initial state  $S_{init_P}$ .
2.  $STATES_p$ ,  $p \in P$ , is the set of states of process  $P$  defined by all prefixes of  $p$ .  
Formally,

$$q \leq_{\pi} p \iff \exists S_q \in STATES_p [S_{init_P}qS_q].$$

3.  $STATES_P = \bigcup_{p \in P} STATES_p$ .
4.  $p, q \in \pi(P) \wedge (p \leq_{\alpha} q \vee q \leq_{\alpha} p) \iff \exists S \in STATES_P [S_{init_P}pS \wedge S_{init_P}qS]$ .

□

Given a pomset  $p \in P$  there is a one to one mapping from  $\pi(p)$  to  $STATES_P$  such that for each prefix  $q \leq_{\pi} p$  there is a state  $S_q \in STATES_P$  and  $S_{init_P}qS_q$  (the occurrence of  $q$  moves process  $P$  from its initial state to state  $S_q$ ). If  $p \leq_{\alpha} q$  then the states  $S_p$  and  $S_q$  ( $S_{init_P}pS_p$ ,  $S_{init_P}qS_q$ ) are considered one and the same state. A pomset  $r$  could lead process  $P$  from state  $S_{\beta}$  to state  $S_{\gamma}$  if and only if  $\beta$  is a prefix of  $\gamma$  and  $(\beta; r)$  is an augment of  $\gamma$ . The following transition system results:

### Definition 6 ( $TS(P)$ (Transition System of process $P$ ))

$$TS(P) = (STATES_P, \pi(\sigma(P)), TR(P))$$

$$TR(P) = \{S_{\beta}rS_{\gamma} \mid r \in \pi(\sigma(P)), \beta \leq_{\pi} \gamma \in \pi(p) \wedge \gamma \leq_{\alpha} \beta; r\}$$

□

An example is illustrated in Figure 2.3. The paths of the transition system  $TR(P)$  of a process  $P$  define the possible executional observations of  $P$ . A snapshot, called

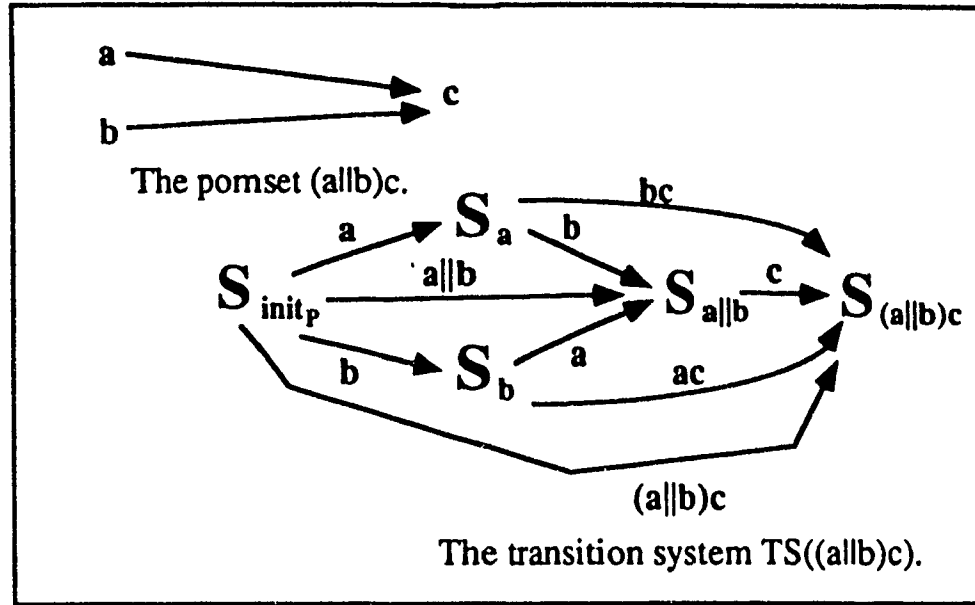


Figure 2.3: A Transition System  $TS(p)$  made out of a pomset  $p$

recording, of the state  $S_r$  of process  $P$  may be taken during an observation defined from a path  $f = \{S_x, \dots, S_y\}$  if  $S_r \in f$ . However, a recorded state  $S_r$  of process  $P$  during an observation defined from a path  $f$  starting at state  $S_x$  and ending at state  $S_y$  is considered consistent if there exists any path  $\varphi$  in  $TR(P)$  that starts at state  $S_x$  and ends at state  $S_y$  and  $S_r \in \varphi$ . A consistent recorded state of  $P$  is defined formally as:

**Definition 7 (CRS (Consistent Recorded State))** Let  $S_x, S_y \in STATES_P$ ,

$$CRS(P, S_x, S_y) = \{S_r \mid S_r \in STATES_P \wedge \exists p_x \leq_{\pi} p_r \leq_{\pi} p_y \leq_{\pi} p (p \in P) \\ [S_{init_P} p_x S_x \wedge S_{init_P} p_r S_r \wedge S_{init_P} p_y S_y]\}$$

A state  $S_r$  of  $STATES_P$  is a consistent state of process  $P$  recorded between states  $S_x$  and  $S_y$  ( $S_x, S_y \in STATES_P$ ) iff  $S_r \in CRS(P, S_x, S_y)$ .

□

A DCS can be viewed as a computation composed by the computations of the component sites. Moreover, the state  $S$  of a site or DCS is identified by the computation that lead the site or DCS from the initial state to  $S$ . Thus a state of a DCS



identified by computation  $p$  is composed by the states identified by the component computations of  $p$ . A more general definition follows

**Definition 8 (CSCS (Composite State, Component State))**

A state  $S_q \in STATES_p$  ( $q \leq_n p$ ,  $p \in P$ ) is a composite state consisting of component states  $S_{q_1} \in STATES_{p_1}, \dots, S_{q_n} \in STATES_{p_n}$  iff

$$Const\_Rule_P(q, q_1, \dots, q_n)$$

written as  $S_q = S_{q_1} \circ \dots \circ S_{q_n}$ .

□

The construction rule, process composition/decomposition, and state composition/decomposition are general tools for modelling distributed computations.  $DCS_n$  models a distributed system with  $n$  sites.

In the following chapters we use these tools to specify particular distributed computing systems and prove their properties.

## Chapter 3

# Pomset Model of the Global State Detection System

The use of the state of a DCS is necessary for a number of applications. A DCS could provide fault tolerance by saving its states spontaneously. Thus in case of crash the distributed computation could restart from the most recent consistent state. Moreover, stability detection algorithms, e.g. deadlock detection algorithms [28], [27], use the global state of the system.

In this chapter we expand  $DCS_n$  to model a complete state recording and state compilation system. In this system the component processes  $DP_i$  of  $DCS_n$  record their states at a central site process  $CA$ . The  $CA$  process collects the recorded states and compiles them into consistent global state (Definition 7 (CRS)). The model can be used for expressing and proving properties of arbitrary recording and compilation algorithms.

### 3.1 Recording Process

A recording process  $R(P)$  is a process superposed on process  $P$ . Each pomset  $q$  of  $R(P)$  is constructed from a pomset  $p$  of  $P$  in such a way that each event  $e$  added to  $p$  defines a state  $S_a$  of  $P$ .  $S_a$  is defined by the prefix  $a$  of  $p$  and it consists of all events of  $p$  that precede the state definition event  $e$  in  $q$ . All such state definition events are totally ordered. The set of all such pomsets of  $R(P)$  form a compilation process  $CP(P)$ .

**Definition 9**  $R(P), R(p)$

$$R(P) = \{ q \mid \exists p \in P [Const\_rule_{R(P)}(q, p)] \},$$

$$Const\_Rule_{(R(P))}(q, p) \equiv q \in R(p),$$

where,

$$R(p) = \{ q \mid V_q = V_p \cup V_r, \Sigma_q = \Sigma_p \cup \Sigma_r, \mu_q = \mu_p \cup \mu_r, \Gamma_q = tr(\Gamma_p \cup \Gamma_r \cup G(p, r)) \},$$

$$\Sigma_r = \{ SD_k \mid k \in \omega \},$$

$$\mu_r : V_r \rightarrow \Sigma_r \text{ (one to one and onto),}$$

$$\Gamma_r = \{ (e_i, e_j) \mid (e_i, SD_{k_i}), (e_j, SD_{k_j}) \in \mu_r \wedge 0 \leq k_i < k_j \},$$

$$G(p, r) \subseteq V_p \times V_r,$$

$tr(\Gamma)$  is the transitive closure of the set of elements of  $\Gamma$ , that is,

$$((a, b) \in \Gamma \Rightarrow (a, b) \in tr(\Gamma)) \wedge ((a, b), (b, c) \in tr(\Gamma) \Rightarrow (a, c) \in tr(\Gamma)).$$

□

It is easy to observe that each member of  $R(p)$  differs from any other only in their  $G(p, r)$  and  $|V_r|$ . Each member of  $R(p)$  is identified with a recording algorithm because it specifies the states of  $p$  being recorded. An event  $e$  of  $R(p)$  labelled  $SD_k$  defines a state recording of  $p$ .

**Definition 10** ( $REC\_ALG(P)$ ) A recording algorithm  $REC\_ALG(p)$  applied on a pomset  $p \in P$  is a member of  $R(p)$ .

□

A deterministic recording algorithm applied on  $P$ ,  $REC\_ALG(P)$ , produces one and only one member of  $R(p)$  for each  $p \in P$ .

**Property (Recording Algorithm (RA))**

It follows immediately from the definition of  $R(p)$  and  $REC\_ALG(p)$  that a recording algorithm  $REC\_ALG(p)$  is specified by the sets  $\Sigma_r$  and  $G(p, r)$ .

□

An example of recording is given in Figure 3.1 where  $SD_1$  is associated with state  $S_{(a||b)}$  and  $SD_2$  with state  $S_{(a||b)c}$ .

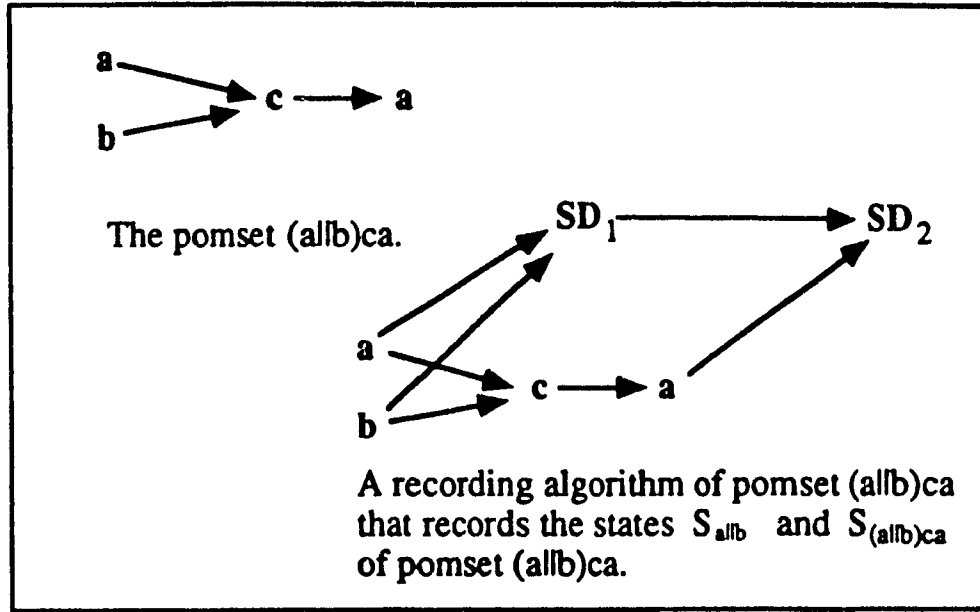


Figure 3.1: A recording algorithm of pomset  $(a||b)ca$

### 3.2 A Complete State Recording and State Compilation System

A complete state recording and state compilation system  $CSys(DCS_n)$  is a process superposed on  $DCS_n$ . It models a system constructed from  $n$  distributed processes  $REC\_ALG(DP_i)$ ,  $(1 \leq i \leq n)$  which record the states of  $DP_i$  processes and forward them to a central site  $CS(DCS_n)$ . The latter composes them to form global states of  $DCS_n$ . The central site process is formed by the linearizations of the state definition events of  $REC\_ALG(DP_i)$ . The state definition events of different processes for each  $i$ ,  $(1 \leq i \leq n)$ , are distinctly labelled with  $(SD, ikl)$ . The integers  $k, l$  are reserved for use by specific recording algorithms that we will develop later (e.g.  $k$  could be the  $k^{th}$  recording of process  $DP_i$ ). An example is given in Figure 3.2.

**Definition 11** ( $CSys(DCS_n)$ )

$$CSys(DCS_n) = \{ q \mid \exists p \in DCS_n [Const\_Rule_{CSys}(DCS_n)(q, p)] \}$$

$$Const\_Rule_{CSys}(DCS_n)(q, p) \equiv$$

$$q \in \lambda_X(q_1 \circ \dots \circ q_n) \wedge q_i = REC\_ALG(p_i) \in R(p_i) \wedge p = p_1 \circ \dots \circ p_n \in DCS_n$$

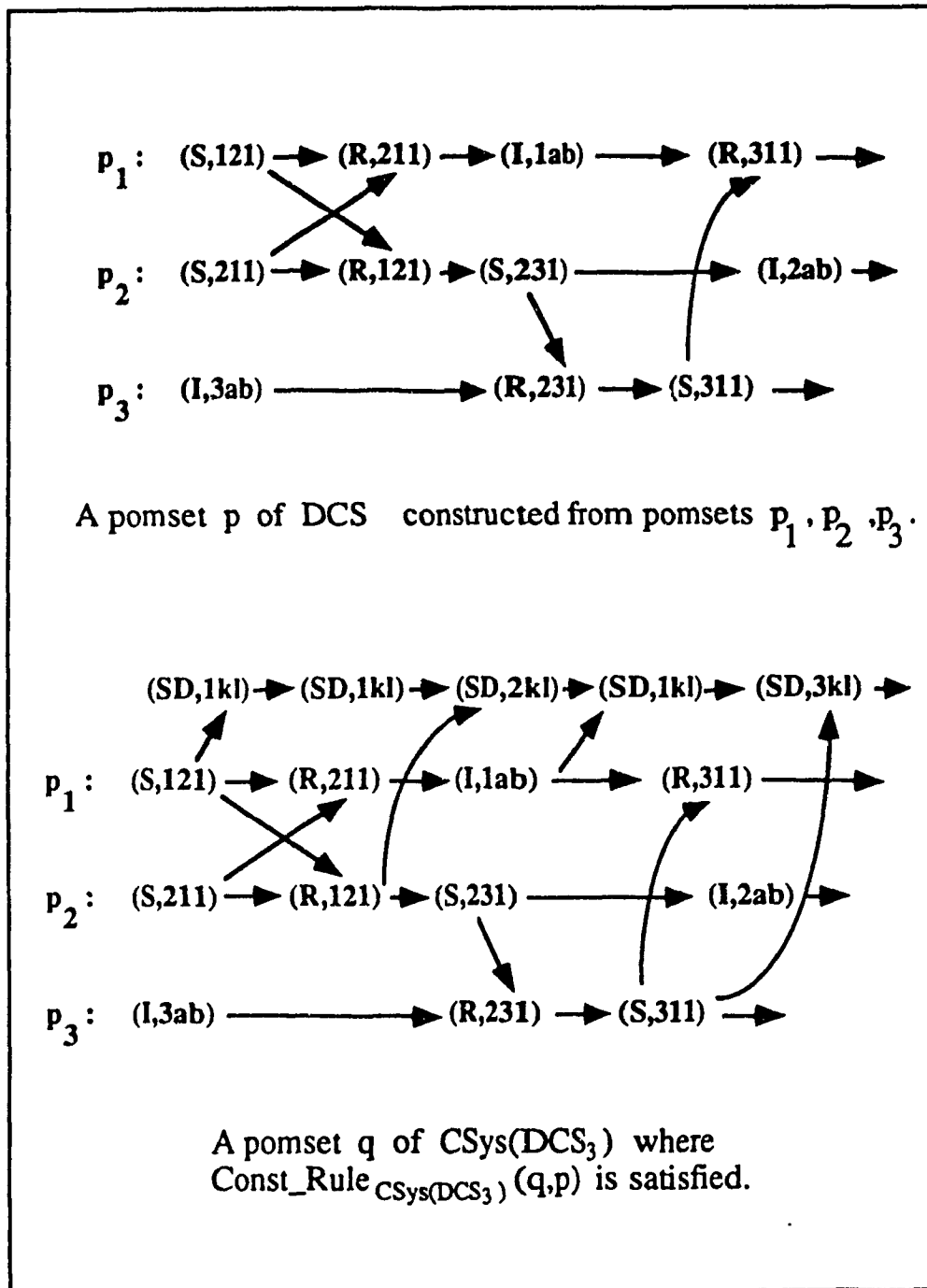


Figure 3.2: A complete state recording and state compilation system.

where  $X = (SD, ikl)$  and  $\lambda_X(p)$  is the set of all partial linearizations of events labelled  $X$  in  $p$ .

□

### 3.3 Compilation Algorithm

The central site process  $CA(DCS_n)$  contains events of  $CSys(DCS_n)$  labelled with  $(SD, ikl)$ , ( $1 \leq i \leq n$ ), formally defined as,

**Definition 12** ( $CA(DCS_n)$ )

$$CA(DCS_n) = \{ q \mid \exists p \in CSys(DCS_n) [ Const\_Rule_{CA(DCS_n)}(q, p) ] \}$$

$$Const\_Rule_{CA(DCS_n)}(q, p) \equiv q = proj(p, \{(SD, ikl) \mid (1 \leq i \leq n), k, l \in \omega\}).$$

□

A Compilation Algorithm  $CALG$  is invoked by the events of  $CA(DCS_n)$ . It updates a database of component states and can be viewed as an expanding homomorphism that replaces all events labelled  $(SD, i\alpha\beta)$  in  $CSys(DCS_n)$  with algorithm  $CALG$ .

Objectively, the compilation algorithm maintains a database consisting of at most  $m$  component states of each component process while guaranteeing a recent global state of  $DCS_n$  in the database. These will be developed in chapter 5, together with two compilation algorithms based on logical and global time.

### 3.4 Refinement of the Model

The state recording and compilation algorithms presented in the following sections assume an architecture in which a distributed system  $DCS_n$  is constructed from  $n$  processes  $DP_i$  where all  $p \in DP_i$  are tomsets. That is the sites the  $DP_i$  processes model are uniprocessors executing sequences of actions. Thus  $DP_i$  is redefined to be

$$DP_i = \lambda(p_{(n,i)}).$$

All the other definitions and theorems remain valid.

## Chapter 4

# Logical and Global Time

The following quote is taken from graffiti seen on a bar wall at Austin, Texas [3]:

Time is nature's way of keeping everything from happening all at once.

Time is a *happened-before* relationship defined on a set of events. A clock is a tool for labelling these events in such a way that the order of occurrence of these events is defined by their labels.

Real clocks are not the appropriate tools for labelling the events of a distributed computation, since they are not common to all processes in the system. Real clocks are never perfectly synchronized thus an event labeled with time 2:50 on site  $S_1$  could happen before an event labeled with time 2:00 on site  $S_2$  if the clocks of these sites have a value difference of 1 hour.

For this reason an instantaneous global state cannot be recorded in a distributed environment. Suppose that the above two sites record their states when their clocks have value 1:00. The clock of site  $S_2$  will have this value an hour later than the clock of site  $S_1$ , thus the compiled state composed of the states of the two sites may show that a message not yet sent from site  $S_1$  is already received from site  $S_2$ .

In this chapter we explore some properties of *logical time*. We show that logical time is not powerful enough to identify the partial ordering of the events of a distributed computation. Subsequently we introduce some properties of *global time*, which is an improved version of logical time. Global time uniquely identifies the events of a distributed computation and reveals the temporal ordering among these events.

## 4.1 Logical and Global Time and their Properties

For the purpose of revealing temporal relationships among events in a distributed system, the logical time system introduced by Lamport [21] could be used. Consider a time stamp function  $LC$  (logical clock) for process events that satisfies the following axiom

### Axiom 1 (Logical Time Axiom (LTA))

Consider  $e_1, e_2 \in V_p$  and  $LC(e_1), LC(e_2) \in \omega$  then,

$$e_1 \rightarrow e_2 \Rightarrow LC(e_1) < LC(e_2)$$

□

The events of the pomset in Figure 4.1 are labelled according to the algorithm reported in [21].  $LC$  could be applied as a homomorphism on all events of a process  $P$ . Unfortunately in LTA “ $<$ ” is not isomorphic to “ $\rightarrow$ ”.

**Theorem 2 (LT1) Independence (absence of causality) among the events of a distributed computation cannot be revealed by examining the logical clock labels of the events except in case they are equal.**

**Proof :** We need to prove only the exception:

$$\begin{aligned} LC(e_1) = LC(e_2) &\Rightarrow LC(e_1) \not< LC(e_2) \wedge LC(e_2) \not< LC(e_1) \\ &\Rightarrow e_1 \not\rightarrow e_2 \wedge e_2 \not\rightarrow e_1 && \text{(Axiom LTA)} \\ &\Rightarrow e_1 \text{ and } e_2 \text{ are independent.} \end{aligned}$$

□

A stronger time-stamp function is necessary in order to determine if two arbitrary events  $e_1$  and  $e_2$  have temporal dependency. This can be accomplished by means of a global time label (defined later), as used in a number of applications previously [24], [40].

The following definitions ( $last_i, next_i$ ) are used in theorem proving.



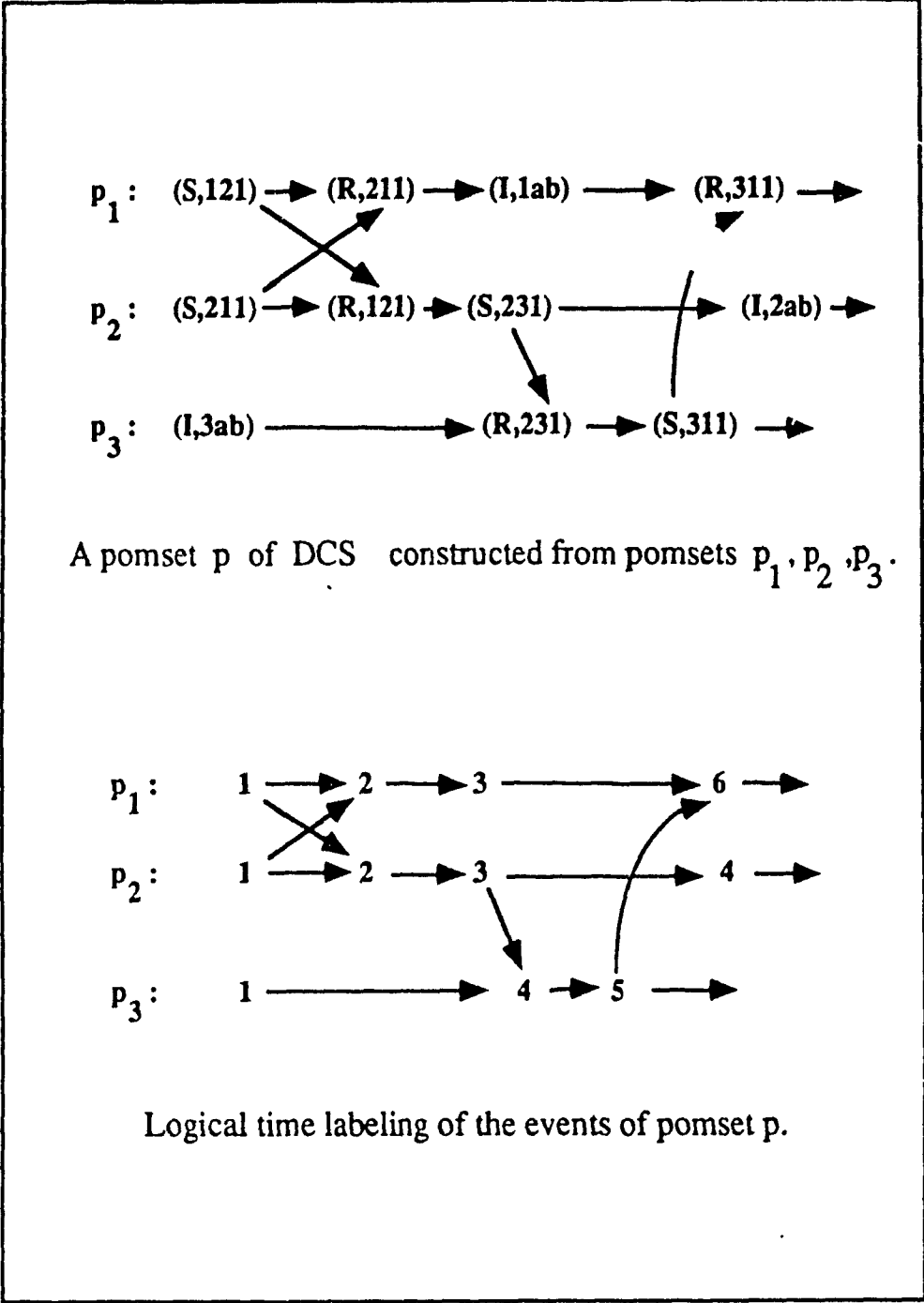


Figure 4.1: Logical time labelling of events of pomset  $p$ .

**Definition 13** ( $last_i$ )  $last_i(q)$  of a prefix  $q$  of  $p \in DCS_n$  is the last event of tomset  $p_i$  that is also event of  $q$ . Formally, given that  $q \leq_\pi p \in DCS_n$ , ( $p = p_1 \circ \dots \circ p_n$ ) then

$$last_i(q) = u \in V_{p_i} \iff \nexists v \in V_{p_i} [(u, v) \in \Gamma_q]$$

□

**Definition 14** ( $next_p$ )  $next_p(e)$  of an event  $e$  of a tomset  $p$  is the first event of tomset  $p$  that follows event  $e$  in  $p$ . Formally, given that  $e, \varepsilon \in V_p$  then

$$next_p(e) = \varepsilon \iff e \rightarrow \varepsilon \wedge \nexists a \in V_p [e \rightarrow a \rightarrow \varepsilon]$$

□

**Theorem 3 (LTPR)**

Given that  $p \in DCS_n$ ,  $p = p_1 \circ \dots \circ p_n$ ,  $\forall i [e_{r_i} \in V_{p_i}]$  then

$$\exists x \forall i [LC(e_{r_i}) \leq x < LC(next_{p_i}(e_{r_i}))] \Rightarrow \exists r \leq_\pi p [\forall i last_i(r) = e_{r_i}]$$

**Proof :** Let  $A \equiv \exists x \forall i [LC(e_{r_i}) \leq x < LC(next_{p_i}(e_{r_i}))]$  then

$$\begin{aligned} A &\Rightarrow \nexists e_{r_i}, e_{r_j}, \alpha [\alpha \in V_{p_i} \wedge e_{r_i} \rightarrow \alpha \rightarrow e_{r_j}] \quad (\text{else } LC(e_{r_j}) > x) \\ &\Rightarrow \exists r \leq_\pi p [\forall i last_i(r) = e_{r_i}] \quad (\text{definition of prefix}) \end{aligned}$$

□

The above theorem leads to a recording algorithm where the  $n$  processes record their states right after the occurrence of the event whose logical clock label is the largest logical clock label that does not exceed a specific  $x$ . Then these  $n$  local states can be composed to form a (global) state of  $DCS_n$ .

**Definition 15** ( $V_{p_i}$ )  $V_{p_i}(e)$ , ( $e \in V_{p_i}$ ,  $p = p_1 \circ \dots \circ p_n$ ,  $p \in DCS_n$ ), is the set of events of  $V_{p_i}$  of  $q \leq_\pi p$  where  $q$  is formed by deleting from  $p$  with the exemption of  $e$  all the events that do not precede  $e$  in  $p$ . Formally,

$$\forall e \in V_{p_i} \quad V_{p_i}(e) = \{ \alpha \mid \alpha \in V_{p_i} \wedge (\alpha \rightarrow e \vee \alpha = e) \}$$

□

**Lemma 1** Given that  $e_i \in V_{p_i}$ ,  $e_j \in V_{p_j}$ ,  $p = p_1 \circ \dots \circ p_n$ ,  $p \in DCS_n$  then

$$V_{p_i}(e_i) \subseteq V_{p_i}(e_j) \iff (e_i \rightarrow e_j \vee e_i = e_j)$$

**Proof :**

$$\begin{aligned} V_{p_i}(e_i) \subseteq V_{p_i}(e_j) &\iff e_i \in V_{p_i}(e_j) \\ &\iff e_i \in \{\alpha \mid \alpha \in V_{p_i} \wedge (\alpha \rightarrow e_j \vee \alpha = e_j)\} \text{ (Definition } V_{p_i}(e)) \\ &\iff e_i \rightarrow e_j \vee e_i = e_j \end{aligned}$$

□

**Definition 16 (GCL)** A global clock label  $GC$  of the events of a pomset  $p \in DCS_n$  is a one to one mapping  $GC : V_p \rightarrow \omega^n$  applied as a size preserving homomorphism on the events of  $p$  such that : Given  $e \in V_p$ ,

$$GC(e) = (\ell_1(e), \dots, \ell_n(e)) \iff \forall i [\ell_i(e) = |V_{p_i}(e)|]$$

□

An example of global clock labelling of events is given in Figure 4.2.

**Theorem 4 (Global Time Property (GTP))**

Given  $p \in DCS_n \wedge p = p_1 \circ \dots \circ p_n \wedge \forall i, j [e_i \in V_{p_i}, e_j \in V_{p_j} (e_i \neq e_j)]$ .

$$\ell_i(e_i) \leq \ell_i(e_j) \iff e_i \rightarrow e_j.$$

**Proof :**

$$\begin{aligned} \ell_i(e_i) \leq \ell_i(e_j) &\iff |V_{p_i}(e_i)| \leq |V_{p_i}(e_j)| \text{ (Definition GCL)} \\ &\iff V_{p_i}(e_i) \subseteq V_{p_i}(e_j) \\ &\iff e_i \rightarrow e_j \quad \text{(Lemma 1 (} e_i \neq e_j \text{))} \end{aligned}$$

□

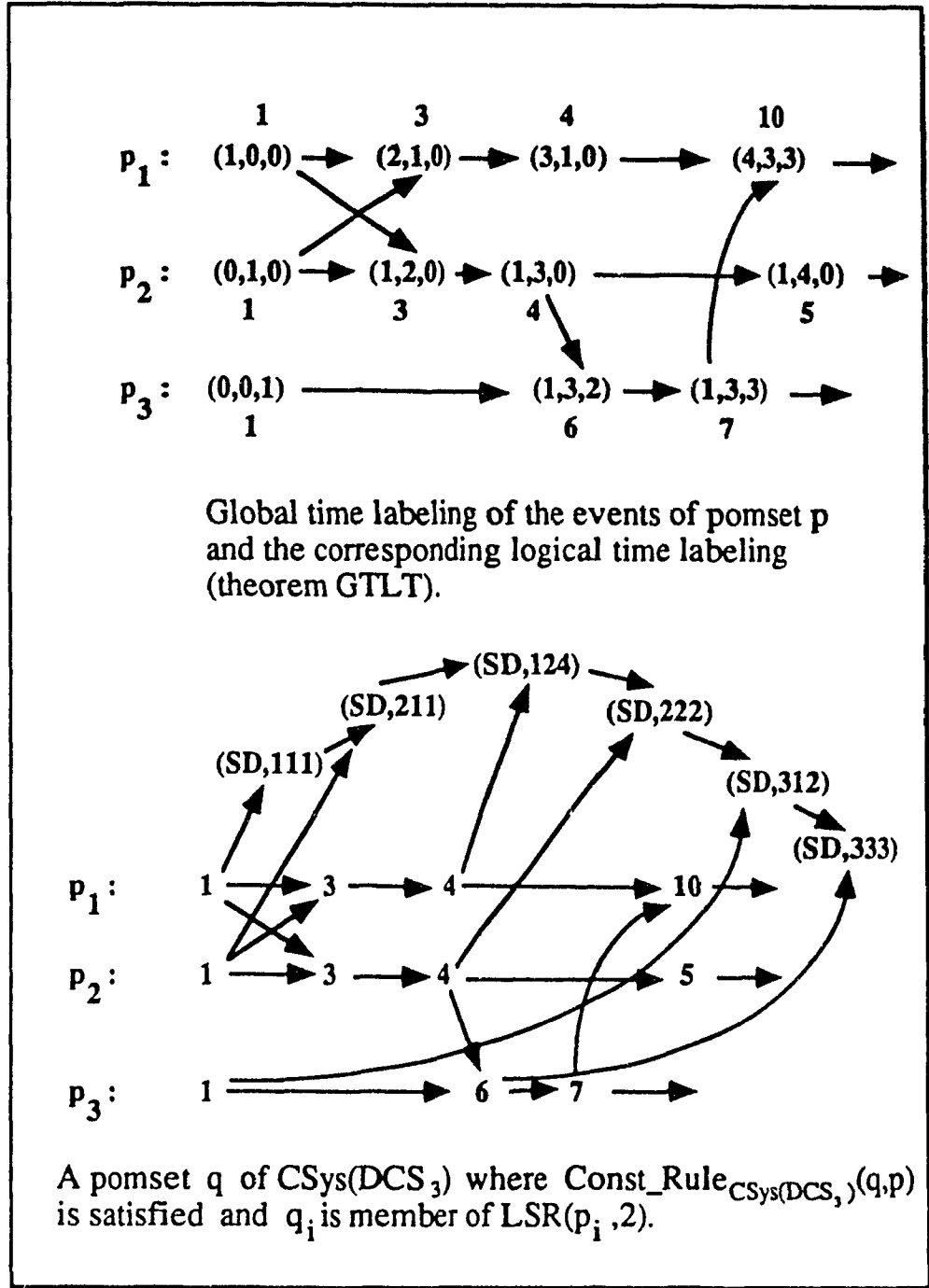


Figure 4.2: Global time labelling of pomset  $p$  and Logical time labelling derived from global time labelling and the Recording Algorithm  $LSR$ .

According to theorem (GTP) the global clock labels of two events reveal the temporal causality between them. This property of global clock labelling is used in the construction of simple recording and compilation algorithms.

**Theorem 5 (Global Time and Prefix (GTPR))**

Given  $p \in DCS_n$ , ( $p = p_1 \circ \dots \circ p_n$ ),  $e_i \in V_{p_i}$ , ( $1 \leq i, j \leq n$ ).

$$\exists r \leq_{\pi} p [\forall i \text{ last}_i(r) = e_i] \iff \forall i \neq j [\ell_i(e_i) \geq \ell_i(e_j)]$$

**Proof :**

$$\begin{aligned} \forall i \neq j [\ell_i(e_i) \geq \ell_i(e_j)] &\iff \nexists i \neq j [\ell_i(e_i) < \ell_i(e_j)] \\ &\iff \nexists i \neq j [V_i(e_i) \subset V_i(e_j)] \quad (\text{Definition GCL}, V_{p_i}(e)) \\ &\iff \nexists i \neq j [\exists a \in V_i [e_i \rightarrow a \rightarrow e_j]] \\ &\hspace{15em} (\text{Lemma 1, Definition } V_{p_i}(e)) \\ &\iff \exists r \leq_{\pi} p [\forall i \text{ last}_i(r) = e_i] \quad (\text{Definition last}_i, \text{prefix}) \end{aligned}$$

□

Theorem GTPR shows that given any state  $S_i$  of each site of a distributed system  $DCS_n$ , global time identifies if the state of  $DCS_n$  composed of these  $n$  states is a consistent recorded (global) state of the system. Theorem GTPR asserts that state  $S_r$  of the  $DCS_n$ , identified by the prefix  $r$ , is a consistent recorded state between the initial state of the system and state  $S_p$ , identified by prefix  $p$ , if and only if the  $i^{\text{th}}$  element of the global time label of the last event of the state  $S_i$  of site  $i$  (prefix of  $p_i$ ) is greater than or equal to the  $i^{\text{th}}$  element of the global time labels of all the other states  $S_j$  of the sites  $S_j$ , where  $i \neq j$ .

Thus theorem GTPR identifies a compilation algorithm where the received states of  $n$  sites are composed to form consistent global states of the system.

**Theorem 6 (Global Time and Logical Time (GTLT))**

Given  $p \in DCS_n$ , ( $p = p_1 \circ \dots \circ p_n$ ),  $e_i \in p_i$ ,  $e_j \in p_j$  and  $f(e) = \sum_{i=1}^n \ell_i(e)$ ,  $e \in p$ .

$$e_i \rightarrow e_j \Rightarrow f(e_i) < f(e_j)$$

Thus the function  $f$  on the global clock labels satisfies the logical time axiom LTA.

**Proof :** From definitions ( $V_{p_i}$ ) and (GCL)

$$\begin{aligned} e_i \rightarrow e_j &\iff \forall k [\ell_k(e_i) \leq \ell_k(e_j)] \wedge \ell_j(e_i) < \ell_j(e_j) \\ &\Rightarrow \sum_{k=1}^n \ell_i(e_i) < \sum_{k=1}^n \ell_i(e_j) \end{aligned}$$

□

Theorem GTLT asserts that all results obtained from logical time can also be obtained via this transformation of global time. An example of logical time label derived from global time label is given in Figure 4.2.

**Definition 17 (Partial Global Time (PGT))** Given  $r \leq_\pi p \in DCS_n$ , ( $p = p_1 \circ \dots \circ p_n$ ),  $\forall i \text{ last}_i(r) = e_i$ .

$GC(r) = (\ell_1(r), \dots, \ell_n(r))$  represents the global time value of the state identified by  $r$  where,

$$\ell_i(r) = \max(\ell_i(e_1), \dots, \ell_i(e_n))$$

□

**Theorem 7 (Global Time and Partial States (GTPS))**

Given  $p \in DCS_n$ , ( $p = p_1 \circ \dots \circ p_n$ ),  $e_i \in V_{p_i}$ , ( $1 \leq i, j \leq n$ ),  $q_1 = p_1 \circ \dots \circ p_k$ ,  $q_2 = p_{k+1} \circ \dots \circ p_n$ .

$$\exists r_1 \leq_\pi q_1, \exists r_2 \leq_\pi q_2, \exists r \leq_\pi p [\forall i \text{ last}_i(r) = e_i \wedge r = r_1 \circ r_2]$$

$$\iff$$

$$\exists r_1 \leq_\pi q_1, \exists r_2 \leq_\pi q_2 [\forall i \in \{1..k\} [\ell_i(r_1) \geq \ell_i(r_2)] \wedge \forall i \in \{k+1..n\} [\ell_i(r_2) \geq \ell_i(r_1)]]$$

**Proof :** The proof come directly form theorem 1, theorem 5 (GTPR) and definition PGT.

□

Theorem GTPS shows that state composition may be done in a hierarchical structure. The states of the processes are first composed to partial states of the system and subsequently the partial states are composed to a global state of the system. Thus the composition process could be distributed to more than one site.

## Chapter 5

# State Recording and State Compilation Algorithms

In this chapter a state recording algorithm and two compilation algorithms are presented. These algorithms are based on the properties of logical and global time introduced in chapter 4. They are specified using the pomset model of a DCS introduced in chapters 2 to 3.

The algorithms are implemented and tested on a network of Sun workstations. Implementation information and the test results are given in section 5.3. A wider set of test results is given in [26].

### 5.1 A State Recording Algorithm

The following recording algorithm is based on theorem *LTPR*. A process instance (pomset)  $p_i$  of  $DP_i$  records its state when the last event of the prefix of  $p_i$  that defines that state has logical clock label that is a multiple of a given number  $d$ .

According to property RA a recording algorithm  $REC\_ALG(p_i) \in R(p_i)$ ,  $p_1 \circ \dots \circ p_n \in DCS_n$  is identified by  $\Sigma_r$ , and  $G(p_i, r_i)$ .  $LSR(p_i, d) \in R(p_i)$  is a recording algorithm presented next. For  $e \in V_r$ ,  $\mu(e)$  is denoted by  $(SD, ik_e l_e)$ .

**Definition 18 (base, offset)** For  $a \in V_{p_i}$ ,  $d \in \omega$

$$\text{offset}(a, d) \triangleq \lceil \frac{LC(\text{next}_d(a)) - LC(a)}{d} \rceil - 1$$

$$\text{base}(a, d) \triangleq \lceil \frac{LC(a)}{d} \rceil$$

□

**Algorithm 1 (Local State Recording Algorithm (LSR))**

*Input* : a tomset  $p_i$  and  $d \in \omega$ .

*Output* : a member of  $R(p_i)$  satisfying the following.

Let  $W = \{a \mid a \in V_{p_i}, [\text{offset}(a, d) \neq 0 \vee LC(a) \bmod d = 0]\}$ . The members of  $W$  are the last events of prefixes of tomset  $p_i$  that correspond to recorded states of  $p_i$ .

$$\Sigma_{r_i} = \{(SD, k_e l_e) \mid a \in W [k_e = \text{base}(a, d) \wedge l_e = k_e + \text{offset}(a, d)]\}$$

$$G(p_i, r_i) = \{(a, e) \mid a \in V_{p_i}, e \in V_{r_i}, [k_e = \text{base}(a, d) \wedge l_e = k_e + \text{offset}(a, d)]\}$$

□

**Definition 19 (pref)** Consider  $p_1 \circ \dots \circ p_n \in DCS_n$ ,  $e \in V_{q_i}$  and  $q_i = LSR(p_i, d) \in R(p_i)$  [ $\mu_{q_i}(e) = (SD, i k_e l_e)$ ].

$$t_{i_e} = \text{pref}(q_i, (SD, i k_e l_e)) \iff t_{i_e} \leq_{\pi} p_i [a \in V_{t_i} \iff (a, e) \in \Gamma_{q_i}]$$

Thus  $t_{i_e}$  is the maximal prefix of  $p_i$  that precedes the state-definition event  $e$  in  $q_i$ .

□

The recording algorithm  $LSR$  has the following properties :

**LSR1:** Every process instance  $p_i$  records its state every time the logical clock is incremented by  $d$  units.

**LSR2:** Let  $q_i = LSR(p_i, d) \in R(p_i)$ ,  $e \in V_{r_i}$ ,  $t_{i_e} = \text{pref}(q_i, (SD, i k_e l_e))$ .  $S_{t_i}$  is the state recorded for the  $k_e^{th}, (k_e + 1)^{th}, \dots, l_e^{th}$  increments of the logical clock by  $d$  units.



**LSR3:** Each state (corresponds to a unique prefix) is recorded at most once.

**LSR4:** There is one and only one pomset  $LSR(p_i, d) \in R(p_i)$  for a given  $d \in \omega$ . This is easily proved considering the uniqueness of logical clock labels of the events of  $p_i$  (assuming that logical clock is implemented using Lamports algorithm [21]).

**LSR5:** Consider a  $CSys(DCS_n)$  where  $REC\_ALG(p_i) = LSR(p_i, d)$ , ( $1 \leq i \leq n$ ),  $p = p_1 \circ \dots \circ p_n \in DCS_n$ ,  $q_i = pref(LSR(p_i, d), (SD, ik_e, l_e))$ , ( $1 \leq i \leq n$ )  $\exists x [\forall i k_e, \leq x < l_e]$ . According to theorem *LTPR* and definitions *CRS*, *CSCS*,

$$S_{q_1} \circ \dots \circ S_{q_n} \in CRS(DCS_n, S_{init_{DCS_n}}, S_p)$$

**LSR6:** Consider  $a, b \in V_r$ , and  $b$  is the event that occurs right after  $a$  in tomset  $r_i$ . Then it is easy to prove (from definition of LSR algorithm) that  $k_b = l_a + 1$  (recording in a process has numbering that span the entire  $\omega$ ).

**LSR7:** *LSR* can be used by  $CSys(DCS_n)$  when the events are labelled with global clock labels according to theorem (GTLT).

An example of the recording algorithm *LSR* is given in Figure 4.2. Property *LSRPR4* enables a central site process  $CS(DCS_n)$  to compose the states of  $DP_i$ s to form states of  $DCS_n$ . It is preferable to label the events with the global time label because the extra global information provided permits more flexible retention and composition of states of  $DP_i$ 's to form states of  $DCS_n$  which logical clock label will fail to reveal.

## 5.2 State Compilation Algorithms

### 5.2.1 Using Logical Time

The compilation algorithm using logical time is defined by a set of rules that specify if a state should be kept in or discarded from the database. The following predicates are first defined.

**Predicate 1 (*less\_than\_m*)**

Let  $a \in V_r$ ; *less\_than\_m*(( $SD, i_a k_a l_a$ ),  $j$ ) is TRUE if there are fewer than  $m$ , ( $m \geq 2$ ), states of process  $DP_j$  in the database right before the occurrence of event  $a$ .

□

**Predicate 2 (*in\_data\_base*)**

Let  $a, b \in V_r$ ; the predicate *in\_data\_base*(( $SD, i_a k_a l_a$ ), ( $SD, i_b k_b l_b$ )) is satisfied iff the state  $S_{(SD, i_b k_b l_b)}$  is in the database right before the occurrence of event  $a$ .

□

**Definition 20 (*not\_deleted*)**

$$\begin{aligned} &not\_deleted((SD, i_a k_a l_a), (SD, i_b k_b l_b)) \iff \\ &in\_data\_base((SD, i_a k_a l_a), (SD, i_b k_b l_b)) \vee a \rightarrow b \vee a = b \end{aligned}$$

□

**Predicate 3 (*can\_be\_used*)**

Let  $a, b \in V_r$ . *can\_be\_used*(( $SD, i_a k_a l_a$ ), ( $SD, i_b k_b l_b$ )) is TRUE iff,

$$\begin{aligned} &(b \rightarrow a \vee a = b) \wedge \exists x \forall i \neq i_b \exists e_i \in V_r \\ &[k_b \leq x \leq l_b \wedge k_{e_i} \leq x \leq l_{e_i} \wedge not\_deleted((SD, i_a k_a l_a), (SD, i k_r l_r))] \end{aligned}$$

□

Informally *can\_be\_used*(( $SD, i_a k_a l_a$ ), ( $SD, i_b k_b l_b$ )) is satisfied if and only if the state  $S_{(SD, i_b k_b l_b)}$  can be used to form a state of  $DCS_n$  at or after the occurrence of event  $a$ . According to theorem *LTPR* and algorithm *LSR*,  $S_{(SD, i_b k_b l_b)}$  can be used to form a state of  $DCS_n$  at or after the occurrence of event  $a$  if there exists  $x$  ( $k_b \leq x \leq l_b$ ) such that for all other processes  $DP_i$  ( $i \neq i_b$ ) the states  $S_{(SD, i k_r l_r)}$  have not been deleted from the database (either already in the database or yet to be received at the central site) and  $k_{e_i} \leq x \leq l_{e_i}$ .

**Predicate 4 (*form\_new\_state*)**

Let  $CGS_{DCS_n} = S_{(SD,1k_1l_1)} \circ \dots \circ S_{(SD,nk_nl_n)}$  right before the occurrence of an event  $a \in V_r$ .

$$\begin{aligned} & form\_new\_state((SD, i_a k_a l_a), x) \iff \\ & x > \min(l_1, \dots, l_n) \wedge (\forall j, (j \neq i_a, 1 \leq j \leq n), \exists e_i \in V_r \\ & [k_{e_i} \leq x \leq l_{e_i} \wedge k_a \leq x \leq l_a \wedge in\_data\_base((SD, i_a k_a l_a), (SD, ik_{e_i} l_{e_i}))]) \wedge \\ & \nexists y > x [form\_new\_state((SD, ik_i l_i), y)] \end{aligned}$$

□

Informally  $form\_new\_state((SD, i_a k_a l_a), x)$  is satisfied if and only if at the occurrence of event  $a$  there is in the database a state  $S_{(SD, ik_e l_e)}$  for each  $i \neq i_a$  that can be composed together with  $S_{(SD, i_a k_a l_a)}$  to form a state of  $DCS_n$ . The choice of this predicate is justified from theorem LTPR.

The following algorithm is invoked atomically by each event  $e_i \in V_r$ . Initially the initial state  $S_{init_{DP_i}}$  for each process  $DP_i$ , is stored in the database. According to the following algorithm a recent state of  $DCS_n$  is always retained in the database. The recency of this state will be established in Corollary 1. This state is named  $CGS_{DCS_n}$ . Initially

$$CGS_{DCS_n} = S_{init_{DP_1}} \circ \dots \circ S_{init_{DP_n}}$$

**Algorithm 2 (Logical Time Compilation Algorithm)**

Upon receipt of  $S_{(SD, ik_e l_e)}$  **do**

**R1** : if  $\exists x [form\_new\_state((SD, ik_e l_e), x)]$  then

1. Delete from the database all states  $S_{(SD, jk_j l_j)}$  where  $l_j < x$ .
2. Insert  $S_{(SD, ik_e l_e)}$  in the database.
3.  $CGS_{DCS_n} = S_{(SD, 1k_1 l_1)} \circ \dots \circ S_{(SD, nk_n l_n)}$  ( $\forall i k_i \leq x \leq l_i$ ).

**R2** : if  $\neg can\_be\_used((SD, ik_e l_e), (SD, ik_e l_e))$  then discard  $S_{(SD, ik_e l_e)}$ .

**R3** : if  $\nexists x [form\_new\_state((SD, ik_e l_e), x)]$  and  $less\_than\_m((SD, ik_e l_e), i)$  and  $can\_be\_used((SD, ik_e l_e), (SD, ik_e l_e))$  then insert  $S_{(SD, ik_e l_e)}$  in the database.

**R4** : if  $\exists x$   $[form\_new\_state((SD, ik_e, l_e), x)]$  and  $\neg less\_than\_m((SD, ik_e, l_e), i)$   
and

$can\_be\_used((SD, ik_e, l_e), (SD, ik_e, l_e))$  then

1. Discard  $S_{(SD, ik_e, l_e)}$ .
2. Delete from the database all states  $S_{(SD, jk_e, l_e)}$  which do not satisfy  $can\_be\_used((SD, ik_e, l_e), (SD, jk_e, l_e))$ .

□

An example of algorithm LTCA is given in Figure 5.1.

**Definition 21** (*state\_num*) Given that

$$CGS_{DCS_n} = S_{(SD, 1k_{e_1}, l_{e_1})} \circ \dots \circ S_{(SD, nk_{e_n}, l_{e_n})}$$

then

$$state\_num(CG S_{DCS_n}) = \max(k_{e_1}, \dots, k_{e_n}).$$

This means that  $CGS_{DCS_n}$  is composed of the  $state\_num(CG S_{DCS_n})^{th}$  recorded local states of the component processes.

□

**Definition 22** (*next\_state\_num*) Given that  $e_i \in V_\tau$ , occurs between the compilation of states  $CS_1$  and  $CS_2$  at the central site.

$$next\_state\_num((SD, ik_e, l_e)) = state\_num(CS_2)$$

□

**Theorem 8** (LTCA)

Given two consecutive global states of  $DCS_n$  detected by LTCA,

$$CS_1 = S_{(SD, 1k_{e_1}, l_{e_1})} \circ \dots \circ S_{(SD, nk_{e_n}, l_{e_n})}$$

and

$$CS_2 = S_{(SD, 1k_{e_1}, l_{e_1})} \circ \dots \circ S_{(SD, nk_{e_n}, l_{e_n})}.$$

Consider that the LTCA is invoked by the state definition events of pomset  $q$  in figure 7. The maximum number  $m$  of states kept for each process is 2.

Initially			
	$P_1$	$P_2$	$P_3$
1st state (CGS)	$S_{initDP1}$	$S_{initDP2}$	$S_{initDP3}$
2nd state	-	-	-
$d_i$	1	1	1
Right after the occurrence of (SD,111) (R3 is satisfied)			
	$P_1$	$P_2$	$P_3$
1st state (CGS)	$S_{initDP1}$	$S_{initDP2}$	$S_{initDP3}$
2nd state	$S_{(SD,111)}$	-	-
$d_i$	0	1	1
Right after the occurrence of (SD,124) (R4 is satisfied)			
	$P_1$	$P_2$	$P_3$
1st state (CGS)	$S_{initDP1}$	$S_{initDP2}$	$S_{initDP3}$
2nd state	$S_{(SD,111)}$	$S_{(SD,211)}$	-
$d_i$	-1	0	1
Right after the occurrence of (SD,312) (R1 is satisfied)			
	$P_1$	$P_2$	$P_3$
1st state (CGS)	$S_{(SD,111)}$	$S_{(SD,211)}$	$S_{(SD,312)}$
2nd state	-	-	-
$d_i$	-1	-1	0

Figure 5.1: The compilation algorithm *LTCA*.

Then

$$state\_num(CS_2) = \max(k_{e_1}, \dots, k_{e_n})$$

where  $S_{(SD, ik_e, l_e)}$  is the earliest state that follows state  $S_{(SD, ik_e, l_e)}$  which is not deleted from the database for process  $DP_i$  (for all  $i$ ).

**Proof :** The theorem can be proved by proving that

$$\max(k_{e_1}, \dots, k_{e_n}) = \max(k_{e_1}, \dots, k_{e_n}).$$

This is proved by checking that none of the rules of algorithm LTCA would discard from the database a state  $S_{(SD, ik_a, l_a)}$  where  $k_a \leq \max(k_{e_1}, \dots, k_{e_n}) \leq l_a$ .

□

Consider the database maintained by the central site upon the compilation of  $CS_1$ . Let  $S_{(SD, ik_a, l_a)}$  be the most recent state of  $DP_i$  that does not occur after  $S_{(SD, ik_e, l_e)}$  (state of  $DP_i$  used in  $CS_2$ ) and which has already been received by the central site (though not necessarily kept). Let  $\delta_i$  be the number of states received by the central site between the compilation of  $CS_1$  and  $CS_2$ .

**Corollary 1**

$$\delta_i \leq state\_num(GS_2) - l_a,$$

□

**Corollary 2**

$$next\_state\_num((SD, ik_e, l_e)) = \max(k_{e_1}, \dots, k_{e_n}).$$

As it is shown in theorem LTCA  $\max(k_{e_1}, \dots, k_{e_n})$  is known upon the compilation of  $CS_1$ .

□

## 5.2.2 Using Global Time

It is proved in theorem GTLT global time provides more information than logical time and this enables detection of consistent states of  $DCS_n$  in cases where LTCA fails.

**Definition 23** ( $glabel_i$ )  $glabel_i((SD, jk_e, l_e))$  is the  $i^{th}$  element of the global time label of the last event of  $pref(LSR(p_j, d), (SD, jk_e, l_e))$ . Formally,

$$glabel_i((SD, jk_e, l_e)) = \ell_i(last(pref(LSR(p_j, d), (SD, jk_e, l_e))).$$

□

**Predicate 5** (*consistent*)

Let  $a, b \in V_r$ . The predicate  $consistent((SD, i_a k_a l_a), (SD, i_b k_b l_b))$  is satisfied iff

$$glabel_i((SD, i_a k_a l_a)) \geq glabel_i((SD, i_b k_b l_b))$$

□

Let  $E$  be the set of sets of events of  $V_r$  each of which contains exactly one event of each  $V_r$ . Formally,

$$E = \{D \mid D \subseteq V_r \ [|D| = n \wedge \nexists i, a, b [a, b \in D \wedge a, b \in V_r]]\}$$

**Predicate 6** (*form\_new\_state*)

Let  $C(S_{DCS_n} = S_{(SD, i_1 k_1 l_1)} \circ \dots \circ S_{(SD, i_n k_n l_n)})$  right before the occurrence of an event  $e_i \in V_r$ :  $form\_new\_state((SD, i k_e l_e))$  is satisfied iff

$$\exists D \in E [e_i \in D \wedge \forall e_j \in D [not\_deleted((SD, i k_e l_e), (SD, j k_e l_e)) \wedge e_j \rightarrow e_i] \wedge$$

$$\forall a \neq b \in D [consistent((SD, i_a k_a l_a), (SD, i_b k_b l_b))]]$$

□

Informally  $form\_new\_state((SD, i k_e l_e))$  is satisfied if and only if at the occurrence of event  $e_i$  there is in the database a state  $S_{(SD, j k_e l_e)}$  for all  $i \neq j$  that can be composed with state  $S_{(SD, i k_e l_e)}$  to form a new state of  $DCS_n$ .

**Predicate 7 (*can\_be\_used*)**

Let  $a, e_i \in V_r$ . The predicate  $can\_be\_used((SD, i_a k_a l_a), (SD, i_k e, l_e))$  is satisfied iff

$$\exists D \in E [e_i \in D \wedge \forall e_j \in D [not\_deleted((SD, i_k e, l_e), (SD, j_k e, l_e))] \wedge \\ \forall b \neq c \in D [consistent((SD, i_b k_b l_b), (SD, i_c k_c l_c))]]$$

□

Informally  $can\_be\_used((SD, i_a k_a l_a), (SD, i_b k_b l_b))$  is satisfied if and only if the state  $S_{(SD, i_b k_b l_b)}$  can still be used to form a state of  $DCS_n$  upon the occurrence of the event  $a$ .

The following algorithm is invoked by each event  $e_i \in V_r$ . Initially the initial state  $S_{init_{DP_i}}$  for each process  $DP_i$ , is stored in the database. As before, a recent state  $CGS_{DCS_n}$ , is maintained in the database. Initially

$$CGS_{DCS_n} = S_{init_{DP_1}} \circ \dots \circ S_{init_{DP_n}}$$

**Algorithm 3 (Global Time Compilation Algorithm)**

Upon receipt of  $S_{(SD, i_k e, l_e)}$  do

**R1** : if  $form\_new\_state((SD, i_k e, l_e))$  then,

1. For each  $j$  delete from the database all states  $S_{(SD, j_k l_x)}$  where  $l_x < l_j$ .
2. Insert  $S_{(SD, i_k e, l_e)}$  in the database.
3.  $CGS_{DCS_n} = S_{(SD, i_k e, l_e)} \circ \dots \circ S_{(SD, n_k e_n l_{e_n})}$ .

**R2** : if  $\neg can\_be\_used((SD, i_k e, l_e), (SD, i_k e, l_e))$  then  
delete from the database  $S_{(SD, i_k e, l_e)}$ .

**R3** : if  $\neg form\_new\_state((SD, i_k e, l_e)) \wedge less\_than\_m((SD, i_k e, l_e), i) \wedge$   
 $can\_be\_used((SD, i_k e, l_e), (SD, i_k e, l_e))$  then insert  $S_{(SD, i_k e, l_e)}$  in the database.

**R4** : Let  $x = next\_state\_num((SD, i_k e, l_e))$ .  
if  $\neg form\_new\_state((SD, i_k e, l_e)) \wedge \neg less\_than\_m((SD, i_k e, l_e), i) \wedge$   
 $can\_be\_used((SD, i_k e, l_e), (SD, i_k e, l_e))$  then



if  $k_{e_i} \leq x \leq l_{e_i}$ , then

1. Delete the most recent state kept for process  $DP_i$ . That is the state of process  $DP_i$  with the maximum  $k$  kept in the database.
2. Delete from the database all states  $S_{(SD,jk_e,l_e)}$  where the predicate  $can\_be\_used((SD,ik_e,l_e),(SD,jk_e,l_e))$  is not satisfied.
3. Insert  $S_{(SD,ik_e,l_e)}$  in the database.

else 1. Discard  $S_{(SD,ik_e,l_e)}$ .

2. Delete from the database all states  $S_{(SD,jk_e,l_e)}$  which do not satisfy the predicate  $can\_be\_used((SD,ik_e,l_e),(SD,jk_e,l_e))$

□

An example of algorithm GTCA is given in Figure 5.2. This algorithm has the following properties (assume  $x = next\_state\_number((SD,ikl),(SD,ikl))$ ):

**GTCA1** Unlike LTCA, in GTCA a state  $S_{(SD,ik_e,l_e)}$  could be kept in the database for later use even if  $k_{e_i} < x$  (R3). This is due to the difference in the definition of the predicate  $form\_new\_state$  whose validity follows from theorem *GTPR*.

**GTCA2** According to rule R4 a state  $S_{(SD,ikl)}$  is never deleted if  $k_{e_i} \leq x \leq l_{e_i}$ . So in the worst case (where a more recent state cannot be formed) the algorithm has a performance identical to that of LTCA. The same upper bound  $\delta_j$  (Corollary 1) is applicable to both algorithms.

**GTCA3** GTCA can be improved by eliminating more states in the database. Suppose that states  $S_{(SD,ik_1l_1)}$  and  $S_{(SD,ik_2l_2)}$ , ( $k_1 < k_2$ ), are in the database and both can be used to form a new state of  $DCS_n$  with states  $S_{(SD,jk_e,l_e)}$ , ( $\forall j \neq i$ ). Then only  $S_{(SD,ik_2l_2)}$  has to be kept in the database.

### 5.3 Implementation and Testing of the LTCA and GTCA Algorithms

The local state recording algorithm *LSR*, the compilation algorithms *LTCA* and *GTCA* were implemented and tested on two networks of SUN 3.50 workstations (a

Consider that the *GTCA* is invoked by the state definition events of pomset *q* in figure 7. The maximum number *m* of states kept for each process is 2.

Initially			
	<i>P</i> <sub>1</sub>	<i>P</i> <sub>2</sub>	<i>P</i> <sub>3</sub>
1st state (CGS)	<i>S</i> <sub>initDP1</sub>	<i>S</i> <sub>initDP2</sub>	<i>S</i> <sub>initDP3</sub>
2nd state	-	-	-
Right after the occurrence of (SD,111) (R1 is satisfied)			
	<i>P</i> <sub>1</sub>	<i>P</i> <sub>2</sub>	<i>P</i> <sub>3</sub>
1st state (CGS)	<i>S</i> <sub>(SD,111)</sub>	<i>S</i> <sub>initDP2</sub>	<i>S</i> <sub>initDP3</sub>
2nd state	-	-	-
Right after the occurrence of (SD,124) (R1 is satisfied)			
	<i>P</i> <sub>1</sub>	<i>P</i> <sub>2</sub>	<i>P</i> <sub>3</sub>
1st state (CGS)	<i>S</i> <sub>(SD,124)</sub>	<i>S</i> <sub>(SD,211)</sub>	<i>S</i> <sub>initDP3</sub>
2nd state	-	-	-
Right after the occurrence of (SD,312) (R1 is satisfied)			
	<i>P</i> <sub>1</sub>	<i>P</i> <sub>2</sub>	<i>P</i> <sub>3</sub>
1st state (CGS)	<i>S</i> <sub>(SD,124)</sub>	<i>S</i> <sub>(SD,222)</sub>	<i>S</i> <sub>(SD,312)</sub>
2nd state	-	-	-

Figure 5.2: The compilation algorithm *GTCA*.

network of 4 SUN 3.50 and a network of 10 SUN 3.50) and a SUN SparcStation. Up to 12 processes (site control) running in parallel on SUN 3.50's and communicating in several configurations (fully connected, ring, star) using TCP/IP are used in the experiment.

The site control processes are synchronized using global time according to property *LSR7*. The central site receives the states of the site control processes and updates two tables (databases) of states; the first table for *LTCA* and the second for *GTC A*. In order to implement the *LTCA* we convert the global clock labels to logical clock labels (theorem *GTLT*).

The algorithms were tested with the following variations:

1. Number of processes: 3, 6, 9, 12.
2. Recording interval (variable  $d$  in *LSR*): 6, 12, 24, 48, 96, 120, 192, 252, 360, 480.
3. Process communication topology: fully connected, ring, star.
4. Process relative delay (between sending/receiving events):
  - (a) All communicated processes have almost equal delays during execution.
  - (b) Each communicating process is twice as slow as the previous one (assuming the process are numbered in a sequence).
5. Window size (number of states retained for each process at the central site): 3, 6, 9, 12, 15, 18.

For each combination of the above, a test was conducted 10 times and the average values of variables of these executions have been used for the derivation of the following curves. The performance measures used for this study include:

**LT worst case** The average distance (in number of local recordings) between the most recent consistent global state kept in the database and the most recent state received for each process using *LTCA*.

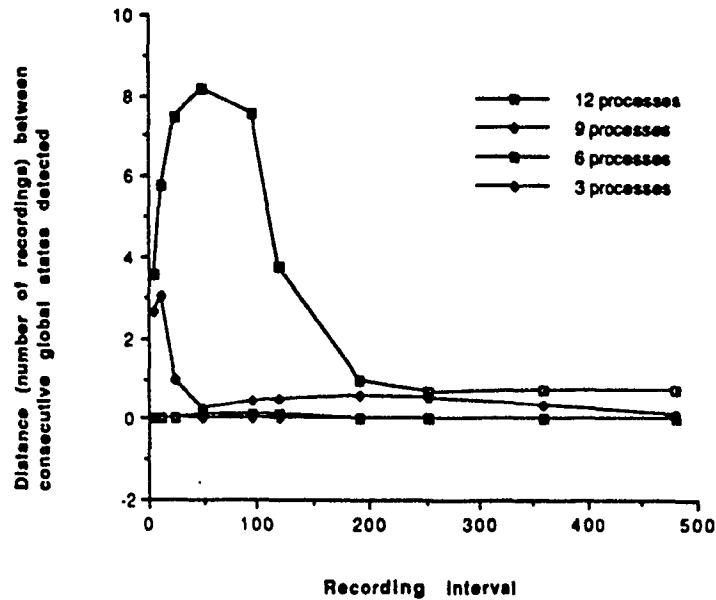


Figure 5.3: Distance (number of recordings) vs. recording interval for fully connected processes.

**LT average case** The average distance (in number of recordings) between the most recent consistent global state kept in the database and the most recent state received for each process using LTCA.

**GT worst case** The same as “LT worst case” but for GTCA.

**GT average case** The same as “LT average case” but for GTCA.

**LT’ worst case** The same as “LT worst case” but the distance is measured in number of events instead of number of recordings.

**LT’ average case** The same as “LT average case” but the distance is measured in number of events instead of number of recordings.

**GT’ worst case** The same as “LT’ worst case” but for GTCA.

**GT’ average case** The same as “LT’ average case” but for GTCA.

For all of the above measures under each combination of test parameters the standard derivation divided by the average (of the 10 executions) observed is much less than 0.1. The following can be observed from the test results.

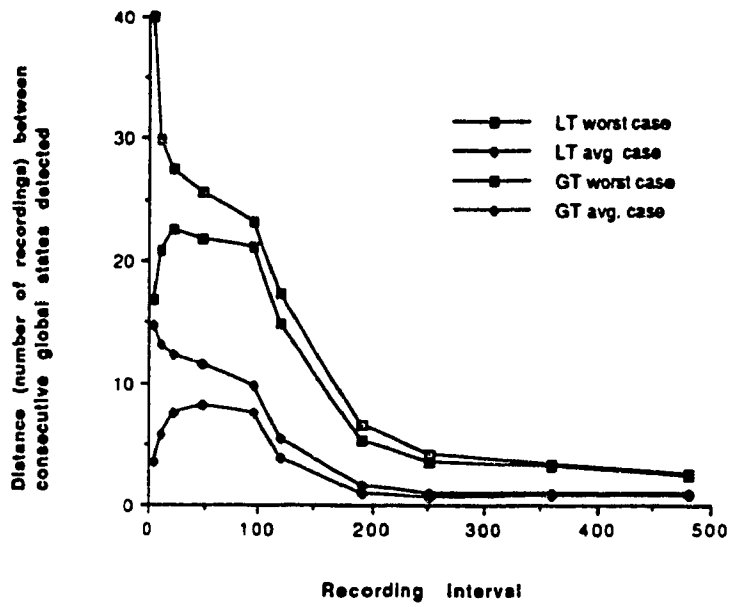


Figure 5.4: *GT average case*, *GT worst case*, *LT average case* and *LT worst case* vs. recording interval for fully connected processes (12 processes).

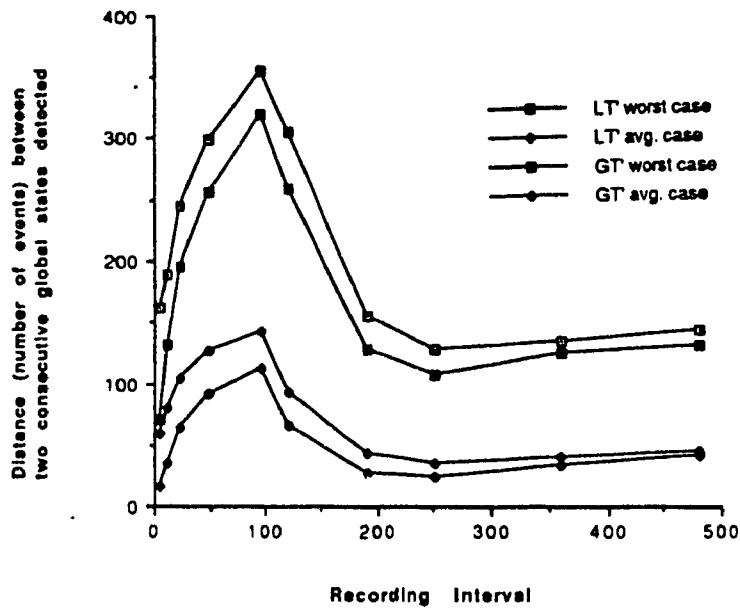


Figure 5.5: *GT' average case*, *GT' worst case*, *LT' average case* and *LT' worst case* vs. recording interval for fully connected processes (12 processes).

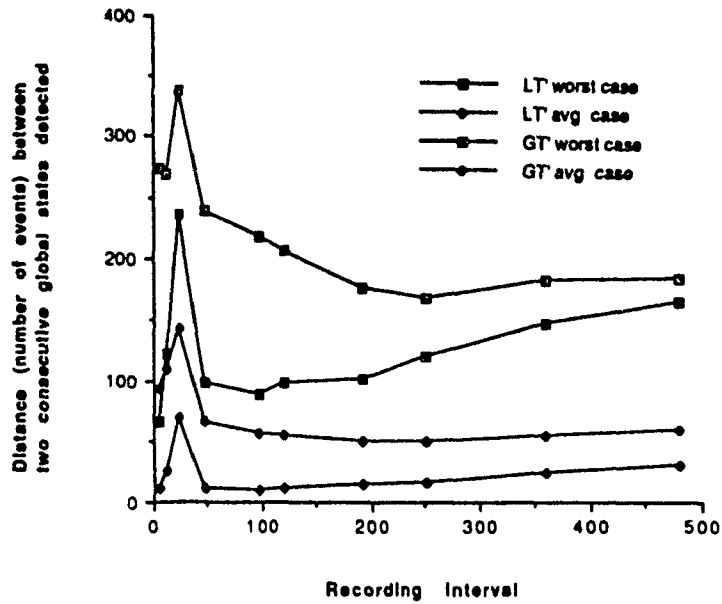


Figure 5.6:  $GT'$  average case,  $GT'$  worst case,  $LT'$  average case and  $LT'$  worst case vs. recording interval for ring connection (12 processes).

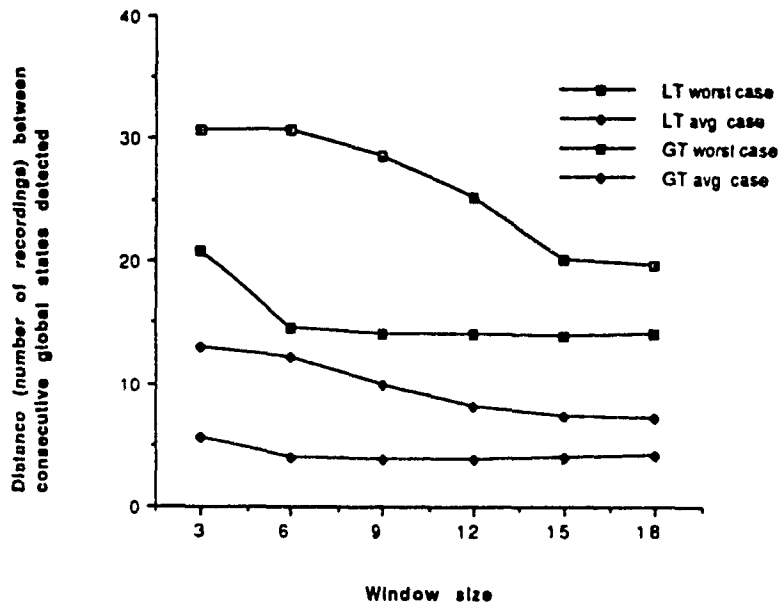


Figure 5.7:  $GT$  average case,  $GT$  worst case,  $LT$  average case and  $LT$  worst case vs. window size for fully connected processes (12 processes, recording interval = 12).

12 processes, fully connected, process relative delay: (a)					
Recording interval	Tot. # of recordings	LT worst case	LT avg. case	GT worst case	GT avg. case
6	924.27	39.99	14.80	16.87	3.55
12	613.33	29.99	13.13	20.85	5.76
24	439.54	27.55	12.36	22.61	7.45
48	336.69	25.61	11.47	21.80	8.17
96	255.24	23.14	9.73	21.15	7.56
120	229.68	17.31	5.51	14.92	3.78
192	178.14	6.63	1.67	5.27	0.98
252	148.36	4.11	1.00	3.53	0.69
360	112.72	3.30	0.98	3.20	0.75
480	87.11	2.62	0.90	2.42	0.73

12 processes, fully connected, process relative delay: (a)					
Recording interval	Tot. # of events	LT' worst case	LT' avg. case	GT' worst case	GT' avg. case
6	3615.38	162.19	58.81	70.71	15.27
12	3630.68	189.05	79.25	131.44	35.52
24	3623.29	244.07	103.78	194.82	63.32
48	3639.88	299.56	126.61	256.65	91.03
96	3625.03	355.52	142.19	319.44	111.42
120	3642.21	305.24	92.38	258.80	66.16
192	3636.48	155.67	43.51	128.40	27.54
252	3627.18	128.67	35.73	107.88	24.71
360	3636.06	133.95	40.05	125.24	34.14
480	3628.62	144.77	44.88	131.22	42.08

Table 5.1: Test results of 12 processes, fully connected where all processes have almost equal execution delays (number of recordings and number of events).

12 processes, fully connected, process relative delay: (b)					
Recording interval	Tot. # of recordings	LT worst case	LT avg. case	GT worst case	GT avg. case
6	927.29	39.92	14.80	16.92	3.54
12	612.70	30.46	13.02	20.88	5.61
24	440.65	28.42	13.03	22.50	8.19
48	334.59	26.27	11.88	22.22	8.47
96	254.98	24.11	10.70	21.64	8.36
120	229.09	22.17	9.45	19.46	7.38
192	178.03	6.15	1.37	5.26	0.68
252	149.05	4.35	1.00	3.68	0.67
360	112.56	3.09	0.97	2.99	0.74
480	87.34	2.49	0.82	2.44	0.57

12 processes, fully connected, process relative delay: (b)					
Recording interval	Tot. # of events	LT' worst case	LT' avg. case	GT' worst case	GT' avg. case
6	3629.74	163.53	58.99	69.15	14.98
12	3634.33	194.53	78.65	129.94	34.68
24	3628.36	250.38	108.67	199.12	69.56
48	3624.42	298.80	131.67	253.75	94.22
96	3621.18	371.09	154.98	327.72	122.68
120	3638.94	378.39	153.82	331.08	121.57
192	3620.69	148.86	36.71	126.77	21.50
252	3633.54	132.27	36.27	115.91	24.93
360	3636.57	130.43	39.64	123.43	33.57
480	3630.92	135.82	43.58	131.10	41.08

Table 5.2: Test results (number of recordings and number of events) of 12 processes, fully connected where each communicating process is twice as slow as the previous one (assuming the processes are numbered in a sequence).



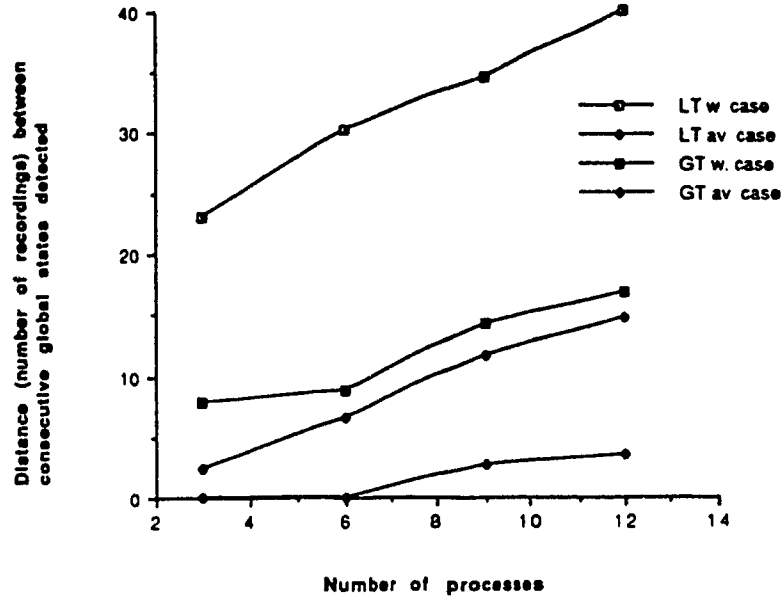


Figure 5.8: *GT average case*, *GT worst case*, *LT average case* and *LT worst case* vs. number of processes for fully connected processes (12 processes, recording interval = 6).

- The average distance (number of recordings) between two consecutive global states detected decreases as the value of recording interval increases. In Figure 5.3 the curves of the average value of *GT average* for fully connected processes is given. Similar observation holds for other process topologies as well as for LTCA.
- In Figures 5.3, we also observe that the smaller the number of processes the better the *GT average* is. Similar results hold for *GT worst case*, *LT average case* and *GT worst case*.
- Figure 5.4 show that *GT' average*, *GT' worst*, *LT' average* and *LT' worst* decrease when the recording interval increases. This does not hold for *GT' average*, *GT' worst*, *LT' average* and *LT' worst* because increase in recording interval results in having more events executed between two consecutive recordings as reflected in Figure 5.5. Figure 5.6 shows the case for a ring connection.
- In Figures 5.4, 5.5 and 5.6 we observe that GTCA always performs better than LTCA.

- Figure 5.7 shows that window size does not affect performance significantly.
- Figure 5.8 reveals that increasing the number of local sites may hurt the effectiveness of GTCA and LTCA: this can be reduced by involving more than one central site organized in an hierarchical structure.
- Tables 5.1, 5.2, show that the relative delay within processes does not affect the test results significantly. However bigger delays may affect the  $LT$  variables for a ring structure.

The algorithms presented in this thesis can be used for applications such as fault tolerance, deadlock detection, termination detection, distributed debugging and program design tools. Although we do not give a detailed analysis of these algorithms it is obvious that the time complexity of LSR is  $O(1)$ . The time complexity (worst case) of our implementation of LTCA is  $O((numberofprocesses) \times (window\ size))$ , and that of the GTCA  $O((numberofprocesses)^3 \times (window\ size)^2)$ . In both cases, the best case is  $O((number\ of\ Processes))$ . Although global time compilation algorithm consumes more cpu time per consistency check, the test results we obtain indicate its performance to be superior to that of the logical time algorithm and is preferred.

Both compilation algorithms share a weakness: the distance between two consecutive global states detected increases with the number of processes. This can be solved by having a hierarchy of compilation sites. Our algorithms do not manipulate the channel states explicitly: we assume a reliable and fifo communication subsystem. Thus messages on transit are always part of the states of the sender processes until they are received. This assumption is reasonable if we consider the communication protocols used by most communication systems. Refinement of the algorithms to take care of channel states is straightforward in our case and is purposely omitted.

## Chapter 6

# The Design of the Global State Detection Kernel (GSDK)

The Global State Detection Kernel (*GSDK*) is an object-oriented distributed system that provides system applications with primitives such as synchronization, global state management, communication and dynamic creation and deletion of active objects (processes). *GSDK* is simple and expansible: the primitives provided can be used to construct a distributed programming environment that provides services such as fault tolerance, load balancing and distributed debugging, among others.

The primitives provided by *GSDK* can be easily ported to existing distributed operating systems such as Mach [1] [2], Demos/MP [34], V kernel [8], Sprite [9], Locus [33].

In this chapter the objectives of the *GSDK*, a brief introduction to the basic concepts of object-oriented distributed systems as well as an overview of the design of the *GSDK* is given. In the following chapters the components of *GSDK* are explained in detail.

### 6.1 GSDK Objectives

Advances in technology have significantly increased the performance of processors and communication systems and have decreased their cost. In the 1990s it would be feasible to have 20 or 50 processors per user [44]. Thus we are led to the conclusion that the key characteristics of computing will be [44]:

- Physically distributed hardware.
- Logically centralized software.

The goal of logically centralized software is to design and implement distributed applications using the concurrency specification of the product and omitting a detailed mapping onto the underlying hardware.

Thus, users will use programming tools for the specification of the execution dependency and security requirements of the parts of their application. These tools will be responsible for the allocation of system resources such as processors, communication systems, etc... which are needed to meet the specific requirements of the application.

Distributed operating systems built on time shared uniprocessor systems like Unix are not the appropriate environment for supporting such programming tools and distributed application development. Unix was designed 20 years ago to support uniprocessor applications. Its simplicity and uniformity made it very popular. However because of the need for distributed computing and network programming it was expanded to a complicated and nonuniform system. For example, communication between Unix processes is not natural because processes are not identified by ports, and it has been added in such a way that it is different for processes running on the same processor and processes running on different processors. A user has to explicitly manipulate system structures to realize inter-process communication.

The need for simple and uniform distributed operating systems is imminent. Systems should be designed with a number of processors and communication subsystems in mind, and should provide user applications with the following services:

- Network process management.
- Uniform process communication.
- Fault tolerance primitives.
- Stability detection primitives.
- Capabilities.

- Application development debugging primitives.
- Application development tools.

Network process management should ensure the transparency of the execution environment of the system. Uniform process communication allows processes to communicate through the same mechanism when they run on the same or different processors. Fault tolerant primitives are used to specify what functions must continue to exist in case of some system failure. Stability detection primitives provide stable state information such as deadlock and termination. Capabilities support system security by defining the access rights of processes. Debugging primitives are used for debugging distributed programs. General application development tools provide users with more abstract constructs that simplify application development.

These are the objectives of the *GSDK*, the Global State Detection Kernel designed in this thesis. Our kernel supports network process management, uniform process communication, fault tolerance primitives, stability detection primitives and debugging primitives. It can also be expanded to provide capabilities. Application development tools may be built using the kernel services provided.

The *GSDK* provides global state information management that can be used for applying fault tolerance and stability detection in two levels:

**System level:** The state of parts of the system is saved periodically. Thus in case of a system failure, e.g. in case of a processor crash, the failing part can be restarted from the last saved state. Deadlock or termination can also be detected applying algorithms such as [27], [28] on the saved consistent global state.

**Application level:** At the application level users may specify the parts of their application that the system should rescue in case of a failure. Thus the state information that the system is managing for fault tolerance support can be decreased significantly. Applications may also use the state information to detect deadlock situations or termination of parts or the entire application.

Global state information can also be used for distributed debugging. For example consider two consecutive states of an application  $S_1$  and  $S_2$  saved by the kernel and the computation  $p$  that leads the application from state  $S_1$  to state  $S_2$ . If  $S_2$  is not an expected state then there is an error in the computation  $p$ . The kernel provides primitives that make the states  $S_1$  and  $S_2$  accessible to the user.

*GSDK* is an object-oriented system. All components of the kernel are well defined independent entities that make the design easy to follow, modify and expand. The system is viewed as a collection of active and passive objects cooperating through well defined interfaces.

An important feature of the *GSDK* is that it includes a programming language, named *GSDKL*. This programming language allows applications to use the kernel primitives without the need for understanding the kernel structure. Although this language is a direct interface to the kernel primitives, it also allows complex structures to be expressed in a sophisticated language such as C++.

For example, at the user level a distributed system of  $n$  processes  $p_1, \dots, p_n$  that cooperate to achieve a common goal, and a controller process  $f$  could be built. The controller process  $f$  obtains the global state of the  $n$  processes from the kernel periodically and checks for faulty conditions which indicate that the goal can not be achieved. If the goal can not be achieved then the process  $f$  will coordinate a re-execution of the application from the previously saved state informing the processes not to follow the same execution path as before. Using the *GSDKL* language, the  $f$  process could ask the system for the state of the  $n$  processes  $p_1, \dots, p_n$  by using the statement

```
getStateInfo(WHOLE_STATE, n, pids, pStates);
```

where *WHOLE\_STATE* is an option that requests the entire states of the  $n$  processes identified by the array of process id's *pid*. The returned states of the  $n$  processes are put in the array of process states *pStates*.

*getStateInfo* is a very powerful primitive provided by the *GSDKL* language that simplifies the design of many distributed algorithms. Such algorithms are developed in the areas of fault tolerance, load balancing, stability detection and distributed

debugging, among others.

## 6.2 Object Oriented Distributed Systems

Practice in software engineering has proved that systems should be composed of well defined, independent, expandible and reusable modules. These requirements lead to object-oriented design. A definition of object-oriented design is given by the following quotes from [29].

Object oriented design is the method which leads to software architectures based on the objects every system or subsystem manipulates (rather than "the" function it is meant to ensure).

Object oriented design is the construction of software systems as structured collections of abstract data type implementations.

In object oriented system literature, an implementation of an abstract data type is called a **class**. Defining a system as a collection of classes implies that classes are units that are meaningful and useful on their own, without consideration of the systems to which they belong.

An **object** is an instance of a class in the same manner that a variable is an instance of a type in structured programming. Classes, like abstract data types, are defined by their **state** and **behaviour specification**. The state of a class is specified by a set of objects. The behaviour of a class is specified by a set of actions that may change and manipulate the class state. These actions are called methods and are analogous to functions and procedures in structured programming.

For example, using C++ [42] notation, a class of an array of integers is defined as:

```
class intArray
{
    int size;
    int *implem;
```

```

public:
    intArray(int s)
    {
        size = s;
        implem = new int[size];
    }
    ~ intArray()
    {
        delete implem;
    }
    int getInt(int position)
    {
        return implem[position];
    }
    void putInt(int elem, int position)
    {
        implem[position] = elem;
    }
};

```

The state of this class is composed of two objects, an integer size that is the size of the array and a reference to an array of integers. These two objects are private, that is, they cannot be accessed directly by a client of class *intArray*. A client class of *intArray* may manipulate the state of *intArray* only by accessing the *public* defined operations which are members of the behaviour specification of the class. An object *x* of class *intArray* with 10 integer elements could be defined as

```
intArray    x = intArray(10);
```

The function *intArray(int)* is the **constructor** of the class *intArray* and is only called at the creation of an object of this class; this function initializes the state of the object. The function *~intArray* is the **destructor** of the class *intArray* and is called to delete an object of this class.



Consider that the object  $x$  of class *intarray* is a member of the state of a class  $C$ . The statement

```
x.putint(10, 3);
```

in a function that is member of the behaviour specification of a class  $C$  is called a **message** from an object of the class  $C$  to the object  $x$ . In this example the message requests  $x$  to put the integer 10 at the third position of the array that the object represents. Class  $C$  is called a **client** of class *intArray* that is a **server** of class  $C$ .

A class is **generic** if it is defined to accept the classes of some of the objects of its state as parameters. For example the class

```
class array(anyClass)
{
    int size;
    anyClass *implem;
public:
    array(anyClass)(int s);
    ~array(anyClass)();
    anyClass getElement(int position);
    void putElement(anyClass elem, int position);
};
```

defines an array of elements of class *anyClass* that is a parameter of the declaration of an object of class *array*. Thus,

```
array(int)    x = array(int)(10);
```

defines an object  $x$  that is an array of integers with 10 elements and

```
array(char)   y = array(char)(10);
```

defines an object  $y$  that is an array of characters with 10 elements.

In object-oriented design, genericity is used for achieving reusability of classes. The same parameterized class definition can be used to specify more than one class

of objects; each of these classes is identified by the parameters passed in the generic class definition.

A class may acquire the structure of other classes for purposes of reusability and structured expansibility through **inheritance**. The following definition of inheritance is taken from [47]:

Inheritance is a relationship between classes whereby one class acquires the structure of other classes in a lattice with a single or multiple parents.

For example,

```
class file
{
    ...
public:
    file(string fileName);
    ~file();
    void open(mode flag);
    void close();
    char getChar();
    void putchar(char ch);
};
```

defines a *file* class whose structure (state and behaviour specification) can be inherited by a class *directory* that is a file with some special characteristics. The *file* class is called the **parent** of the *directory* class. Thus,

```
class directory: file
{
    ...
public:
    directory(string fileName): (filename);
    ~directory();
```

```

    void open(mode flag);
    void close();
    file getEntry();
    void putEntry(file entry);
};

```

defines a class *directory* that is a class *file* with a different (special) behaviour specification.

Another characteristic concept of object-oriented design is **polymorphism**; a definition of polymorphism is given in the following quote taken from [47]:

Polymorphism is the ability of an entity to refer at run time to instances of various classes. Hence, the actual operation performed on receipt of a message depends on the class of the instance.

In *concurrent object oriented systems*, classes may be **passive** or **active** [47], [19]. An active class has a thread of control (a process) while passive classes are used as servers by active ones. A *distributed object-oriented system* is a concurrent object oriented system in which the objects are distributed through a loosely coupled network of processors.

Although the design of the *GSDK* that follows is general, we include some implementation detail issues to make clear how the system, which is object oriented could be implemented in a non object-oriented environment such as Unix, using a static object-oriented language which does not support active objects like C++. Thus we show how active objects as well as **monitor** shared objects [15] could be implemented in such an environment. These issues may be overlooked when a more sophisticated environment is considered like concurrent C++ [12] or the Emerald system [19].

Active objects are objects whose class is an active class. We consider that active objects in a distributed environment may communicate through a shared object (in this document we name such an object a **port**). Active objects may send or receive messages to or from other active objects through the port object. An active object *S* whose main function is receiving requests through a port from other active objects *C*,

and replying to those requests (through the port) is called a **server** of objects  $C_i$ , and objects  $C_i$  are called **clients** of  $S$ . The difference in the server/client relationships between objects that communicate through a port, and objects that are members of the state of other objects, is obvious.

For example, a class *port* could be defined as

```
class port
{
    ...
public:
    port();
    ~port();
    void registerActiveObject(activeObjectId);
    void send(message);
    message receive(activeObjectId);
    message receive();
};
```

An object of class *port* can be shared among several active objects. Consider the class *activeClient* defined as

```
class activeClient
{
    ...
    activeObjectId id;
    port *prt;
    message msg;
    ...
    activeObjectId sid;
    service x;
    servParam parameter;
```

```

...
clientMethod1();
clientMethod2();
...
public:
    activeClient(port* p)
    {
        ...
        prt = p;
        prt->registerActiveObject(id);
        ...
        x.getValue(SERVICE_ID, parameter);
        ...
        msg.insertContents(id, sid, x);
        port->send(msg);
        msg = port->receive(sid);
        x = msg.getContents();
        ...
    }
    ~activeClient();
};

```

and the class *activeServer* defined as

```

class activeServer
{
    ...
    activeObjectId id;
    port *prt;
    message msg;
    service x;
    ...
}

```

```

    serviceMethod1();
    serviceMethod2();
    ...
public:
    activeClient(port* p)
    {
        ...
        prt = p;
        prt->registerActiveObject(id);
        ...
        while (TRUE)
        {
            msg = prt->receive();
            x = msg.getContents();
            switch(x.id)
            {
                case ID1:  serviceMethod1(x);
                    break;
                case ID2:  serviceMethod2(x);
                    break;
                ...
            }
            msg.insertContents(id, x.clientId(), x);
            port.send(msg);
        }
    }
    ~activeServer();
};

```

where both *activeClient* and *activeServer* accept a reference to an object port as a parameter to their constructor. By creating an object of class *activeServer* and several

objects of class *activeClient* and passing as parameters to the constructors of the *activeClient* objects a reference to the same *port*, these objects may use the *port* object to communicate. The constructor of an active object invokes the *registerActiveObject* method of the *port* to request the port to initialize and maintain a queue of messages that other active objects will send to this object. A method of an active object can invoke the *send* and *receive* methods on a port to send a message to another active object or receive a message from an active object. The *receive* method of the port is **overloaded** (form of polymorphism), the method to be invoked is identified by the parameters.

In the above example an *activeClient* object sends a request to an *activeServer* object by invoking the *send* method of the *port* object and then waiting for a reply by invoking the *receive* method of the port object with the *id* of the *activeServer* object as a parameter. The *activeServer* object is executing an infinite loop accepting requests, executing an appropriate *serviceMethod* and then replying to these requests.

Active classes in our design have only two methods public to their clients, a constructor and a destructor. Other methods used by the constructor or the destructor of these objects are declared in the private section of the class and are not accessible to clients. This is due to the constraint that active objects are implemented using Unix processes by making a C++ program (a function *main*) that consists of just a call to the constructor of the active object. A Unix process that executes this program represents an active object. Thus, active clients, as in the above example, request services from active servers not by invoking the methods of those servers but by asking the server to execute a specific service identified by a well known (by both client and server) *id*.

In a Unix network environment (e.g. Sun network), where the interprocess communication interface depend on whether the processes reside on the same or different nodes, port objects provide an abstract interface for communication of objects spread through the network. Port objects are also useful for applications that need to access the state of a port, e.g. the message queues. These applications cannot use the Unix *IPC* interface.

In a Unix network environment, objects residing on different nodes may communicate through a port if the later is implemented as a *server* of the network transparent to all processes of the network. SunOS 4.0 (Unix 4.3bsd compatible) provides high level tools for the implementation of such objects (*RPC/XDR* interface, *rpcgen* pre-processor [32]). These objects can also be built by using low level *IPC* primitives such as sockets or streams provided by the 4.3bsd Unix and Unix V respectively. In a distributed system like Mach where the *IPC* mechanism is not dependent on the location of the communicating processes port objects are not needed, except when the state of the port has to be accessed. In such a system, active objects are identified by ports so they can communicate directly by a well defined uniform interface.

Static objects shared among active objects could be critical sections, since more than one thread of control operates on them at the same time. This can be controlled, if needed, by using *monitors* to implement these objects. An implementation of monitor objects in a Unix environment using *shared memory* and *semaphores* is shown in chapter 7.

### 6.3 Overview of GSDK

A high level specification of *GSDK* is the specification of the *CSys* process in chapter 3. *GSDK* is composed of objects of the following classes: (in the following, we use the name of a class to identify an abstract object of this class).

*GSDK* has an hierarchical structure. The structure of the system as well as the the communication of the active objects through *sitePort* objects and *port* objects is shown in figure 6.1.

**Process:** An active class that specifies a user process. The state of an object of this class includes user defined variables (object of class **stateVar**), user defined code (object of class **code**) that specify a thread of control that manipulate these variables, and a site port (object of class **sitePort**) that is used for communication of active objects that reside on the same node. The user program (variables and code specification) is a parameter of the constructor of an object



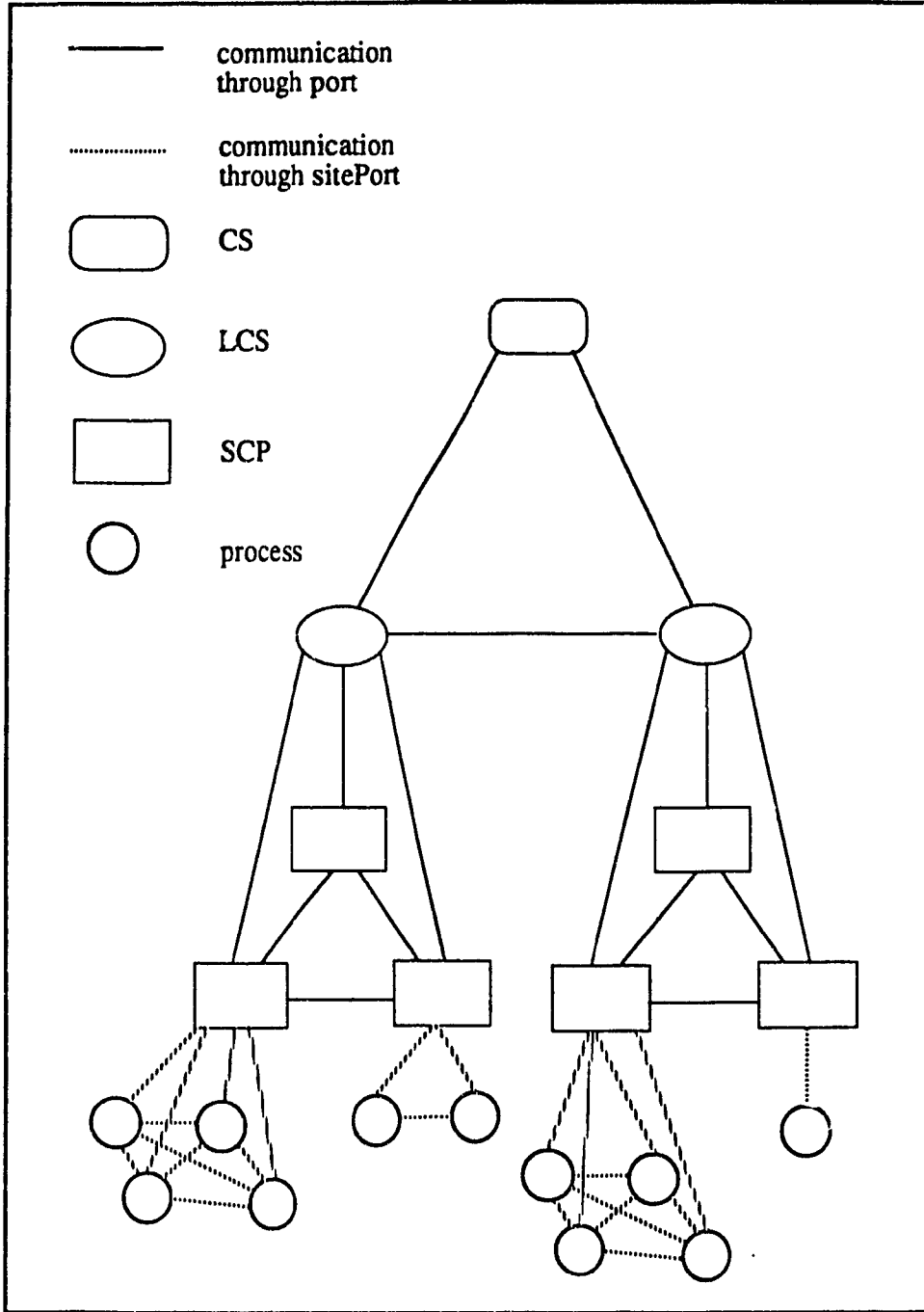


Figure 6.1: The structure of the GSDK.

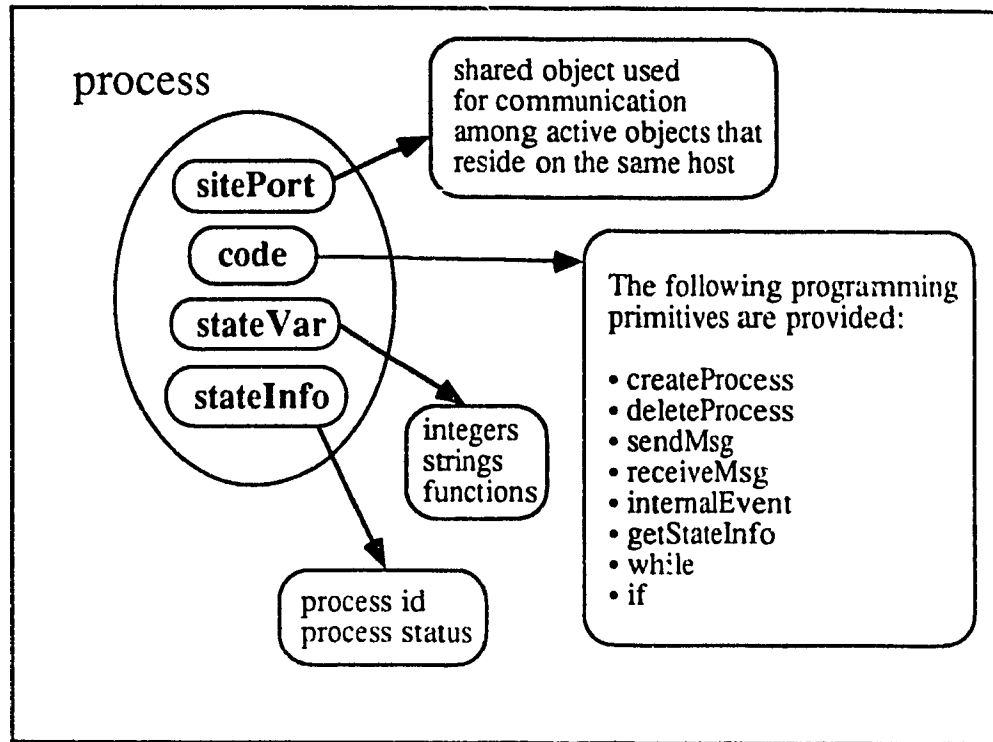


Figure 6.2: The structure of the *process* class.

of class *process*; the variables are used for initialization of the object *stateVar* and the code for the initialization of the object *code*.

The constructor of class *process* initializes the process state and activates the execution of a thread of control that invokes methods of the object *stateVar* specified by the object of class *code*. The structure of the object *process* is shown in figure 6.2.

**Site Control Process (SCP):** An active class that specifies the control of process execution on a node of the network. It controls the global clock (object of class *globalClock*) of the node (that is an implementation of the global time as it is specified in chapter 4), the site port (object of class *sitePort*) that is used for communication between active objects on the same node, the creation and deletion of processes on the node and inter-node communication. It also has access to the state of the system (that is the state of the objects) that resides on the node, and is responsible for recording this state and sending it to a *leaf*

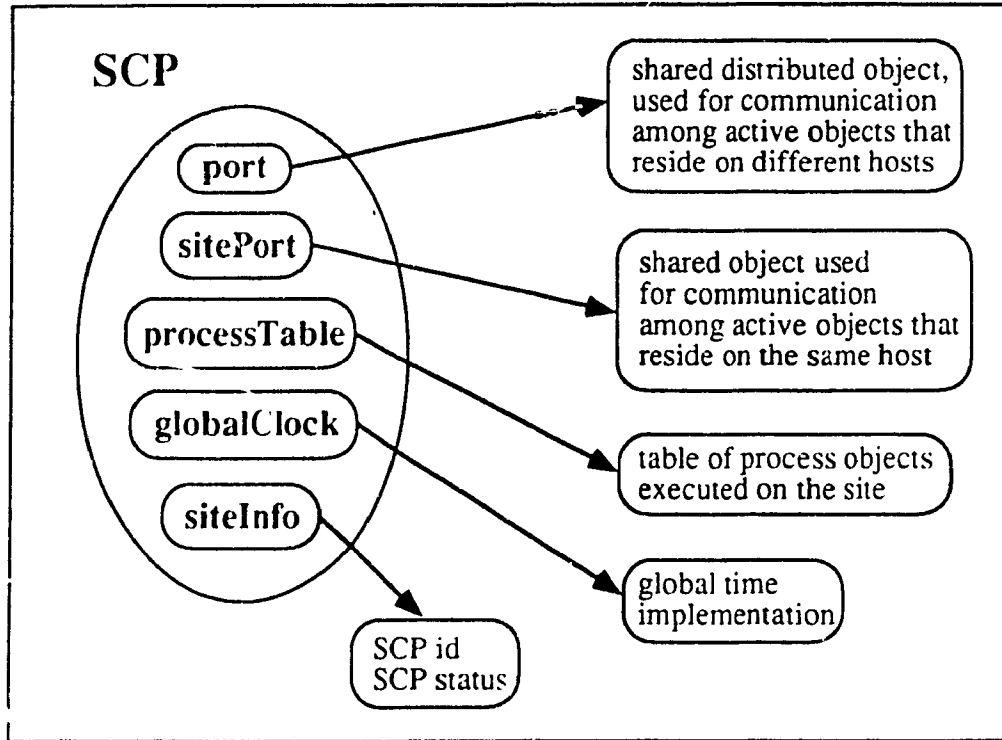


Figure 6.3: The structure of the SCP class.

*central site* object.

The state of an *SCP* object consists of an object **port**, used for communication with other active objects that reside on other nodes of the network, an object *sitePort*, an object **processTable** whose elements are *process* state components, and an object *globalClock*.

The constructor initializes the member objects and activates a thread of control that executes an infinite loop that accepts requests from *process* objects (e.g. communication with processes of other nodes, deletion or creation of a new process) and *leaf central site* objects (e.g. process migration). The destructor deallocates the system resources allocated from the *SCP* at the termination of the system execution. The system may terminate after a special request from the *leaf central site*.

A high level specification of an *SCP* object is given in the chapter 2 by the *DP*, process of the *CSys* in which the events of this process are labeled with global

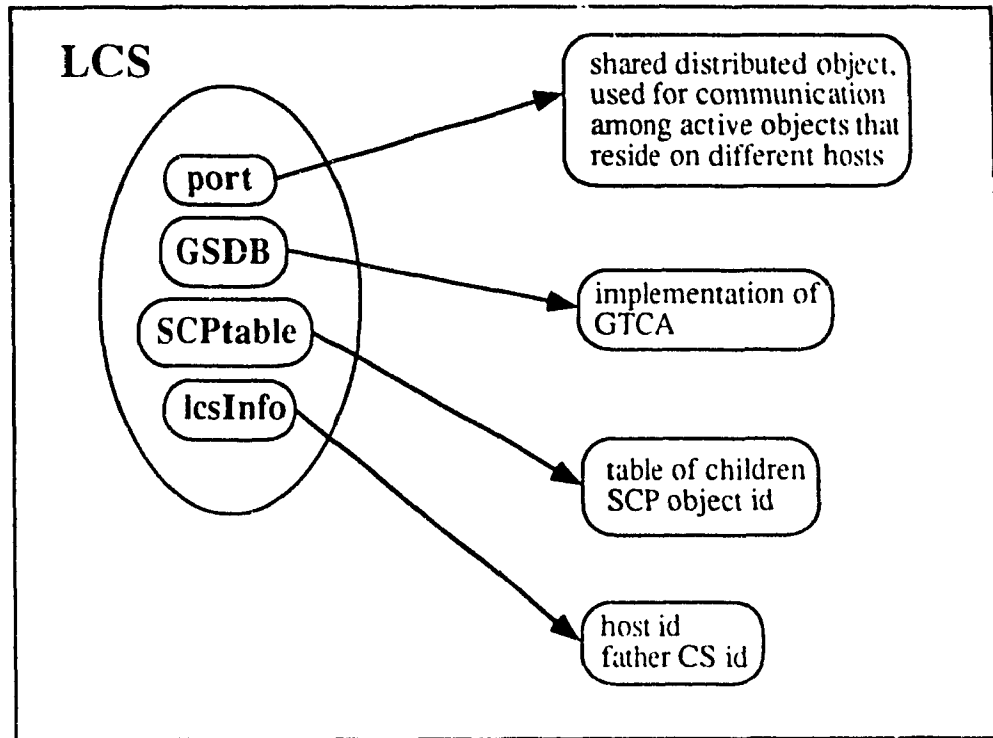


Figure 6.4: The structure of the LCS class.

time. The tomset elements of  $DP_i$  model the observations of the concurrent execution of a number of *process* objects and the *SCP* object executing on the same node. The structure of an object *SCP* is shown in figure 6.3.

**Leaf Central Site (LCS):** A high level specification of the *LCS* is given in chapter 3 by the *CA* process. *LCS* is an active object. The state of this object consists of an object *port* used for communication with *SCP* objects, other *LCS* objects, and **central site** objects, and a **global state database (GSDB)** object.

A *LCS* object is the manager of a number of *SCP* objects. It keeps a database *GSDB* of states of these *SCP* objects and guarantees that at any given time there is a consistent global state in the database, consisting of one state for each *SCP*. *GSDB* is an implementation of the *GTCA* algorithm presented in chapter 5. An *LCS* object sends the compiled consistent global states to a *central site (CS)* object that is responsible for the management of global state information of a number of *LCS* objects.

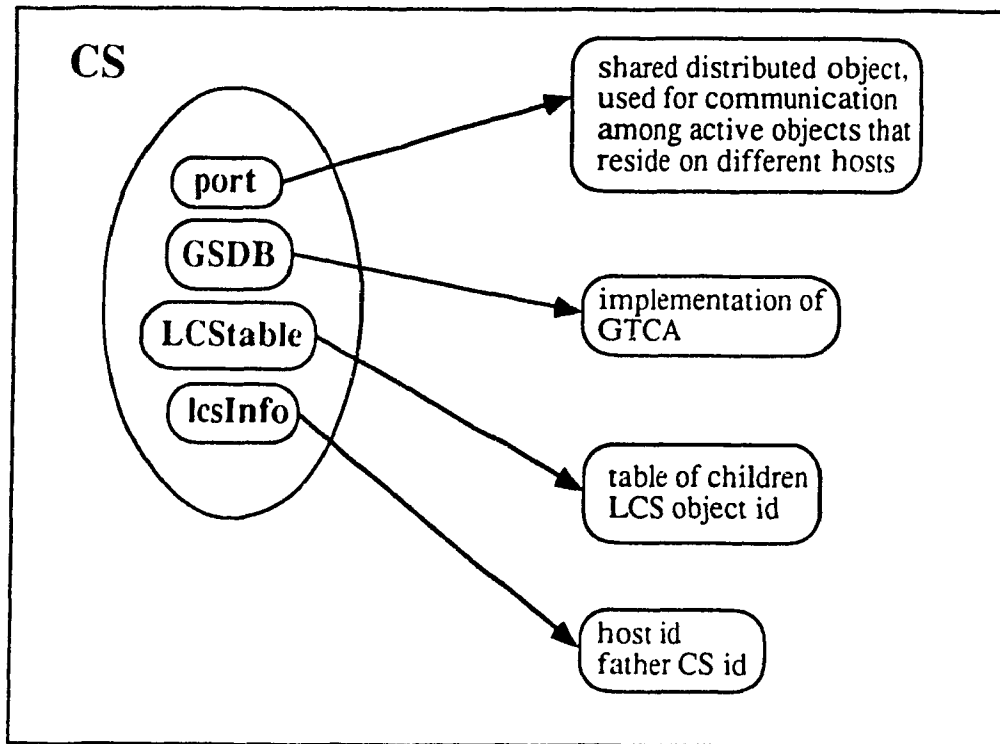


Figure 6.5: The structure of the CS class.

We name the *SCP* objects that are managed by a *LCS* object *children SCP objects* of that *LCS* object.

The constructor initializes the *GSDB* and then executes an infinitely accepts requests and fulfills the following:

- Serves requests of *SCP* objects, controlled by different *LCS* objects, for communication.
- Receives the recording states of its children *SCP* objects and invokes the *GSDB* to be updated.
- Records the consistent states compiled by the *GSDB* to the parent *CS* object (this is not specified in the *CA* process in chapter 3).

The structure of *LCS* object is given in figure 6.4.

**Central Site (CS):** A *central site (CS)* object is the parent (controller) of a number of either *LCS* objects or *CS* objects. It has exactly one parent *CS* object or it

is the root *CS* object. A *CS* object is an active object.

The state of a *CS* object is composed of an object *port* used for communication, an object *GSDB* that collects the local states of the children of this *CS* object and compiles them in consistent global states of the subsystem that has this *CS* object as root.

The constructor of a *CS* object initializes the member objects of *CS* and then executes an infinite loop that accepts requests from the children objects for communication with other children objects of *CS* objects that have the same parent as this *CS* object, receives states of children objects and invokes the *GSDB* object to be updated, and sends consistent global states compiled by *GSDB* to the parent *CS* object (if it is not the root). The destructor, under special request, deallocates the children objects by sending them messages (to deallocate themselves) and deallocates the member objects *port* and *GSDB*. The structure of a *CS* object is given in figure 6.5

The need to distinguish between *sitePort* and *port* objects arises because at the kernel level we need to distinguish between active objects executing on different nodes. Moreover, *sitePort* objects are part of the recording state of a site and are totally accessible by the kernel. The state of *port* objects is not accessible by the kernel when it is implemented in user mode on a Unix like system. However the communication interface of user processes is uniform for processes that run on the same or different nodes.

# Chapter 7

## GSDK Processes

A GSDK *process* object is the basic unit of execution of a user defined program. However, a user defined program may specify the parallel or sequential execution of any number of processes.

A *process* is an active object that, during construction, accepts a *user defined program* as a parameter. A user defined program consists of *state variables* and *code*. *code* is a sequence of operations to be applied to the state variables. A *state variable* may be an integer, a string, or even a *user defined program*. An *operation* may be the request for the execution of a function, the receipt of a message, or even the creation of a *process* using as parameter of its constructor a *user defined program*.

In this chapter we explore the structure of the class *process* and its components as well as the notation definition of a user defined program.

### 7.1 Process Structure

The class *process* in C++ notation is defined as follows:

```
class process
{
    sitePort    port;
    code        events;
    stateVars   vars;
    stateInfo   info;
```

```

public:
    process(int option, pid id, uDefProg prog,
            state pState);
    ~process();
}

```

*port* is the communication subsystem used by *process* to communicate with other active objects. The object *events* is an ordered set of actions to be executed (invoked) on the object *vars*. The object *info* contains process information such as *pid*, *status*, etc. *pid* uniquely identifies a process in the system; *status* identifies if a process is live, terminated, migrating.

The *process* constructor accepts as parameters:

**option:** is an integer object that may take the value of the following descriptions:

**OUT\_OF\_PROG** This option creates a process out of a *user defined program*.

The constructor uses the user defined program *prog* passed as parameter to create and initialize the objects *events* and *vars*. This option is used to create the initial state of a process as well as that process.

**OUT\_OF\_STATE** This option creates a process out of a *state* object. A *state* is a combination of a *code* object and a *stateVars* object. The constructor uses the *pState* object passed as parameter to create and initialize the objects *events* and *vars*. This option is used to create a process that has *pState* as starting state. *pState* is a state reached from an other process of the system. Using the *OUT\_OF\_STATE* option processes may restarted from a previous or current saved state of them. Thus process migration and roll-back and recovery is trivial.

**INHERIT\_VAR** This option creates a process out of the *code* defined in *prog* object and the *stateVars* defined in the *pState* object. This option makes it possible for multiple processes to access and manipulate the same state variables.



**id:** is a part of the unique process id. It contains the *user name* and a user defined *process name*. The user is responsible for not running processes with the same *process name* at the same time.

**prog:** a specification that can be used for the creation of an object of class *code* and an object of class *state Vars*.

**state:** an object that consists of an object of class *code* and an object of class *state Vars*.

The *process* constructor first initializes its *sitePort* to support communication with the parent *SCP*. In a Unix system a *sitePort* is implemented in shared memory and it has a standard well known key, thus processes may be attached to this segment of shared memory. After initialization of the *sitePort*, the constructor creates the *events* and *state Vars* objects and initializes them according to the *prog* parameter. It then informs the parent *SCP* about how these objects can be accessed by the parent *SCP*. In a Unix system the *events* and *state Vars* objects are implemented in shared memory.

After initializing the state of the *process* the constructor invokes the methods on the *events* object to access the actions that this process has to execute. Each action contains a number of variable identifiers on which the action should be performed. The constructor uses these variable identifiers to get these variables from the object *state Vars* and perform on them an operation identified by the action. The constructor continues this procedure until the actions of the *events* object are executed in the order defined.

## 7.2 User Defined Program

The *user defined program* is a specification of:

1. *State variables* such as integers, strings, functions and user defined programs
2. The *ordering of execution* of a set of operations, provided by the class *process* behaviour specification, where these operations operate on the state variables

of this program.

Thus a *user defined program* can be viewed as a pomset whose actions operate on a set of data.

A user defined program can be written in the *GSDKL* (GSDK Language) programming language that is defined by the following BNF:

### GSDKL BNF

```
program      ::= Program( stringIdentifier, stringIdentifier)
              stateVariables
              stateVars
              behaviourSpecification
              commandList
              endProgram

stateVars    ::= identifier : varType ; stateVars |
              identifier : array[n] of varType ; stateVars |
              NULL

varType      ::= process | state | integer | string | function | program

commandList ::= command commandList | condCommand commandList |
              NULL

condCommand  ::= whileCommand | ifCommand

whileCommand ::= while integerIdentifier do commandList endWhile

ifCommand    ::= if integerIdentifier then commandList endif

command      ::= createCommand | deleteCommanad | sendCommand |
              receiveCommand | internalCommand | migrateCommand |
              getSateCommand

createCommand ::= create(integerIdentifier,processIdentifier,
                          programStateIdentifier)

deleteCommand ::= delete(processIdentifier)

sendCommand  ::= sendMsg(processIdentifier, n, ident1, ..., identn)

receiveCommand ::= receiveMsg(processIdentifier, n, ident1, ..., identn)
```

```

internalCommand ::= internal(functionIdentifier, n, ident1, ..., identn)
getStateCommand ::= getStateInfo(integerIdentifier, processIdentifier,
                                     stateIdentifier)

```

The keywords of the language appear in boldface and are language primitives. Literal punctuation marks appear in boldface as well. The vertical bar (|) represents a choice between alternatives. Other primitives are the identifiers (identifier, integerIdentifier, stringIdentifier, functionIdentifier, processIdentifier, stateIdentifier, programIdentifier, programStateIdentifier, ident<sub>i</sub>) and the *n*. An identifier is a string up to 16 characters long that contains no blank. *n* is a string of digits used to represent an integer.

The following description explains the semantics of the language.

**Program**(*stringIdentifier*, *stringIdentifier*): This keyword defines the starting point of a user defined program specification. It is followed by two string identifiers (character strings). The first string is a user identification, called *user name*, and the second a process identification, call *process name*. The process identification is user defined. The pair (user identification, process identification) must be unique in the system. This pair is part of the process id (the process created to execute the program).

**stateVariables**: This keyword defines the starting point of the user defined variables. A variable can be specified as,

identifier : varType;

where *identifier* is a string that uniquely identifies a variable of type *varType* in the *behaviourSpecification* section of the program. A *varType* may be *process*, *state*, *integer*, *string*, *function*, *program* or *array[n]* of *varType*.

**process**: A variable of type *process* represents the id of a process, used for interaction with other processes.

**state:** A variable of type *state* holds the part of the state of a process that is composed of a *stateVars* object and a *code* object. The current state of a process may be saved into a *state* variable and this process may restart execution from that state if it is needed (e.g. crash recovery, migration). Processes can also use the state information of other processes as will be explained later using the *internal* command.

**integer:** An integer variable.

**string:** A variable that represents a character string up to 32 characters long.

**function:** A variable of this type is an identifier of an internal action that may be requested to be executed through the *internal* command. A variable of type *function* is an identifier of a function implemented in C++.

**program:** A variable of this type is an identifier of a program that can be used for the creation of a process. It is implemented as a string that is the name of the executable program out of which a process will be created.

**array [n] of varType:** A variable of this type is an array of *n* elements of variables of one of the above described types.

**behaviourSpecification:** This keyword defines the starting point of the behaviour specification of the process.

**identifier:** In the above BNF, a string up to 16 characters long that contains no blanks.

**xIdentifier:** Where *x* is an *integer*, *string*, *function*, *process*, *state* or *program* represents a variable of type *x*.

**programStateIdentifier:** Represents a variable of type *program* or *state*.

**ident,:** Represents a variable of any type.

**n:** Is an integer variable that is used to define the number of arguments that follow this variable in a command (e.g. *sendMsg*, *receiveMsg* etc).

**while *integerIdentifier* do *commandList* endWhile:** The *while* statement is used for loop construction. The *integerIdentifier* that follows the *while* is the condition variable. While this variable is non zero the list of commands in the body of the *while* statement is executed.

**if *integerIdentifier* then *commandList* endif:** The *if* statement is used for conditional command execution. The *integerIdentifier* that follows the *if* is the condition variable. If this variable is non zero the list of commands in the body of the *if* statement is executed.

**create(*integerIdentifier*, *processIdentifier*, *programStateIdentifier*):** This statement requests the creation of a process. The first parameter is an integer that defines the option for a process creation called *option*. The three options for process creation are defined in the previous section. The second parameter is a *process* variable called *processId*. *processId* is initialized before the *create* statement execution, thus the created process has an id partially defined by *processId*. The *stringIdentifier* values that follow the *Program* statement are members of the state variables of the process and they are initialized with the values of *processId*.

**delete(*processIdentifier*):** This statement deletes a process from the system. The parameter accepted by this statement is the id of the process to be deleted.

**sendMsg(*processIdentifier*, *n*, *ident*<sub>1</sub>, ..., *ident*<sub>*n*</sub>):** This statement sends *n* (second parameter) variables identified by the *ident*<sub>*i*</sub> parameters to the process with *processIdentifier* id.

**receiveMsg(*processIdentifier*, *n*, *ident*<sub>1</sub>, ..., *ident*<sub>*n*</sub>):** This statement assigns to the *n* (second parameter) variables identified by the *ident*<sub>*i*</sub> parameters values sent by the process with *processIdentifier* id, if the *processIdentifier* does not have a null value. If the *processIdentifier* is null then the process waits to receive a message by any process that has sent a message to it.

**internal**(*functionIdentifier*, *n*, *ident*<sub>1</sub>, ..., *ident*<sub>*n*</sub>): This statement calls the function identified by the *functionIdentifier* variable, and passes to it the *n* *ident*<sub>*i*</sub> variables as parameters.

**getStateInfo**(*integerIdentifier*, *processIdentifier*, *stateIdentifier*): This statement assigns to the *stateIdentifier* variable the state of the process defined by the *processIdentifier* variable. The first parameter is an integer that defines the different options that may be requested of a state. Depending on the value of this integer, partial or full state information may be requested. Partial state information includes only the state variables of the process.

There is a one to one mapping between the commands of *GSDKL* and the primitive services provided by the *GSDK*. Although the *GSDK* is an object-oriented system the programs specified in *GSDKL* are not object-oriented programs. The design of a language that provide data-abstraction in a distributed environment is beyond the scope of this work. *GSDK* is designed to support concurrent programming primitives. Thus, the instruction set of *GSDK* includes commands for process creation and communication between processes. The *internal* statement however, permits the invocation of a method on a user defined object in a language like C++. Thus, object-oriented concurrent systems may be implemented on *GSDK* under the restriction that concurrency is distinguished from data-abstraction.

### 7.2.1 Dining Philosophers in GSDKL

In this section the *Dining Philosophers Problem* is presented as an example of how the *GSDKL* can be used for the implementation of concurrent and distributed algorithms on the *GSDK*.

A *philosopher* is an object of class *process* created out of the following *user defined program*:

**Program**(aUser, philos)

**stateVariables**

selfNumber : integer;

```

numOfPhilos : integer;
forks       : process;
true        : integer;
reply       : integer;
getLeft     : integer;
getRight    : integer;
leaveLeft   : integer;
leaveRight  : integer;
gettingForks : integer;
initState   : function;
setReply    : function;
eating      : function;
thinking    : function;
setGetForks : function;
clearGetForks : function;

```

#### behaviourSpecification

```

internal(initState, 10,
          philos, selfNumber, numOfPhilos,
          forks, true, reply, getLeft
          getRight, leaveLeft, leaveRight);

while true do
  internal(setGetForks, 1, gettingForks);
  internal(setReply, 1, reply);
  while reply do
    sendMsg(forks, 2, selfNumber, getLeft);
    receiveMsg(forks, 1, reply);
  endWhile
  internal(setReply, 1, reply);
  while reply do

```

```

        sendMsg(forks, 2, selfNumber, getRight);
        receiveMsg(forks, 1, reply);
    endwhile
    internal(clearGetForks, 1, gettingForks);
    internal(eating, 0);
    sendMsg(forks, 2, selfNumber, leaveLeft);
    sendMsg(forks, 2, selfNumber, leaveRight);
    internal(thinking, 0);
endwhile
endProgram

```

Each *philosopher* has access to the following variables:

**selfNumber:** an integer variable with a unique value for each philosopher. We assume that the *initState* function assigns a unique value to this variable when it is invoked by a *philosopher*.

**numOfPhilos:** an integer variable. It represents the number of *philosopher* objects of the system.

**forks:** a *process* variable. It is the id used for communication with the process *forks* (defined later).

**true:** an integer variable. It is assigned a non-zero value by the *initState* function.

**reply:** an integer variable. It is assigned a non-zero value by the *setReply* function and it is assigned zero by the *receiveMsg(forks, 1, reply)* statement as an indication that a *get fork* request has been granted.

**getLeft:** an integer variable. It represent a *get fork* request. Request to get the left fork.

**getRight:** an integer variable. It represent a *get fork* request. Request to get the right fork.



**leaveLeft:** an integer variable. It represent a *leave fork* request. Request to leave the left fork.

**leaveRight:** an integer variable. It represent a *leave fork* request. Request to get the leave the right fork.

**gettingForks:** an integer variable. It has a zero value while a philosopher is trying to get the forks and a non-zero value while the philosopher is eating or thinking.

**initState:** a function variable. It is invoked to initialize the non-function variables of a philosopher.

**setReply:** a function variable. It is invoked to set the *reply* variable to a non-zero variable.

**eating:** a function variable. It is invoked to simulate an eating philoshopher. It is implemented as a bounded random length delay.

**thinking:** a function variable. It is invoked to simulate a thinking philoshopher. It is implemented as a bounded random length delay.

**setGetForks:** a function variable. It is invoked to set the *gettingForks* variable to a non-zero value.

**clearGetForks:** a function variable. It is invoked to set the *gettingForks* variable to a zero value.

According to the *behaviour specification*, a philosopher first initializes its state variables by invoking the *initSet* function; then it performs an infinite loop executing the following commands:

- Invokes the *setReply* function to set the *reply* variable to a non-zero value; then while the *reply* variable has a non-zero value executes the following:
  1. Sends a message to process *forks* indicating that the *selfNumber* philosopher wants to get the left fork.

2. Receives a message from process *forks* that sets *reply* to zero if the request has granted or to a non-zero value otherwise.

- Repeats the above step for the right fork
- Invokes the function *eating*
- Sends a message to process *forks* indicating that the *selfNumber* philosopher wants to leave the left fork. This request is always granted
- Sends a message to process *forks* indicating that the *selfNumber* philosopher wants to leave the left fork. This request is always granted

*forks* is an object of class *process* created out of the following *user defined program*:

**Program(aUser, forks)**

**stateVariables**

```
numOfPhilos : integer;
fork         : array[10] of integer;
philos       : process;
true         : integer;
request      : integer;
doReply      : integer;
initForks    : function;
setPhilos    : function;
serve        : function;
```

**behaviourSpecification**

```
internal(initForks, 10,
          numOfPhilos, fork, philos,
          true, request, doReply);
while true do
    internal(setPhilos, 1, philos);
    receiveMsg(philos, 2, philosNum, request);
    internal(serve, 3, philosNum, request, doReply);
```

```

        if doReply then
            sendMsg(philos, 1, request);
        endif
    endwhile
endProgram

```

The *forks* process has access to the following variables:

**numOfPhilos:** An integer variable. It represents the number of *philosopher* objects of the system.

**fork:** An integer array variable. It represents the forks that the philosophers share. The philosophers share the 1<sup>st</sup> to *numOfPhilos*<sup>th</sup> forks. If *fork*[*i*] is zero it means that it is free otherwise it is being used.

**philos:** it is a process variable. It is initialized to *null* by the *setPhilos* function. Thus *forks* uses the *receiveMsg* statement to receive messages by any *philosopher*.

**true:** An integer variable. It is assigned a non-zero value by the *initForks* function.

**request:** An integer variable. It is assigned the value of the *getLeft* or *getRight* or *leaveLeft* or *leaveRight* variable of a *Philosopher* process through the *receiveMsg(philos, 2, philosNum, request)* statement. It represents the request of a philosopher.

**philosNum:** An integer variable. It is assigned the value of the *selfNum* variable of a *Philosopher* process through the *receiveMsg(philos, 2, philosNum, request)* statement.

**doReply:** An integer variable. It is assigned a non-zero value by the invocation of the *serve* function if the *request* passed as parameter to *serve* has the value of the *getLeft* or *getRight* variables of a *philosopher*, else it is assigned zero.

**initForks:** A function variable. It is invoked to initialize the non-function variables of the *forks* process.

**setPhilos:** A function variable. It is invoked to set the *philos* variable to *null*.

**serve:** A function variable. It accepts as parameters the *philosNum*, *request* and *doReply* variables. If the request identified by *request* is a *getLeft* or *getRight* request and can be granted for the philosopher identified by *philosNum* then the request is set to zero and the *doReply* is set to a non-zero value.

The *behaviour specification* of the *forks* process is straightforward. The program that specifies a *diner* process that creates and initializes the *philosopher* and *forks* processes, can be defined as follows:

**Program(aUser, diner)**

**stateVariables**

```
numOfPhilos : integer;
philos      : array[10] of process;
forks      : process;
philosProg  : string;
forksProg   : string;
initDinner  : function;
decrease    : function;
```

**behaviourSpecification**

```
internal(initDinner, 5, numOfPhilos
          philos, forks, philosProg, forksProg);
create(OUT_OF_PROG, forks, forksProg);
while numOfPhilos do
    create(OUT_OF_PROG, philos[numOfPhilos], philosProg);
    internal(decrease, 1, numOfPhilos);
```

**endWhile**

**endProgram**

The *philosProg* and *forksProg* variables are strings that identify the files where the *forks* and *philos* programs are stored. The invocation of the *initDiner* function initializes the state variables of the *diner* process. The *create* statement creates a

process using the *user defined programs* accepted as the third parameter to construct the *state* and the *code* objects of this process. The *OUT\_OF\_PROG* option indicates that a process is to be created out of a *user defined program*.

The function variables of the *philosopher* processes, *forks* process and *diner* process are implemented as references to C++ functions implemented by the user. Although these C++ functions accept parameters, they have full access to the state variables of the *process* objects.

It is obvious that the *philos* processes will deadlock when all of them pick up the left fork and start sending requests for picking up the right fork continuously. These requests will never be satisfied and the processes will be in a *deadlock* state. This problem, in any concurrent environment including *GSDL*, can be solved by modifying the *forks* process so that it does not allow more than *numOfPhilos - 1 philos* processes to be served at the same time.

However, in *GSDK*, the *deadlock* state can be detected. When the *philos* processes are deadlocked their *gettingForks* variable is set. Thus, the *forks* process can be modified to periodically get the states of the *philos* processes using the *getStateInfo* statement and determine if the system is deadlocked or not. If the system is deadlocked, *forks* can send a message to one of the *philos* processes to put down its left fork until the rest of the *philos* processes get served.

Thus, the *getStateInfo* primitive provides the optional solution of letting a system react naturally and applying a recovery procedure when an undesirable situation occurs. This is very important for many applications e.g. fault tolerant applications, applications that require safety property checking etc.

## 7.3 Process State Variables

A state variable of a *process* is an object of the following class defined in C++ notation:

```
class stateVariable
{
    varType type;
    varName name;
    varValue value;

public:
    varType getVarType();
    varName getVarName();
    varValue getVarValue();
    void setVarType(varType);
    void setVarName(varName);
    void setVarValue(varValue);
}
```

The *type* object may take a value indicating a *process*, *state*, *integer*, *string*, *function* or *program*. The meaning of these values is the same as the meaning of the values of a *varType* expression, as given in the *GSDKL* BNF in the previous section. The *name* object is a string that uniquely identifies the variable in the program. The *value* object keeps the current value of the variable. The *getVarX* methods, when they are invoked on a *stateVariable* object, return the *X* state member object of this *stateVariable*. The *setVarX* methods set the *X* state member object to the value of the parameter of the method.

An object of class *stateVars* represents the state variables of a *process*. It is an object that controls a variable length array of variables. For each variable defined in the *GSDKL* program, out of which the state of the *process* will be initialized, there is an entry in the array of variables controlled by the *stateVars* object. For each array of cardinality *n* of variables defined in the *GSDKL* program, there are *n* consecutive

entries in the array of variables controlled by the *stateVars* object.

A *stateVars* object is a passive object. The state of a *stateVars* object is composed of an object *numOfVars* of class *integer*, and an object *variable* that is an array of *stateVariable* objects.

The class *stateVars* using *C++* notation is defined as follows:

```
class stateVars
{
    int          numOfVars;
    stateVariable *variable;
public:
                stateVars(int numOfVariables);
                ~stateVars();
    int         getVarIndex(varName name);
    varType     getVarType(int base, int offset);
    varName     getVarName(int base, int offset);
    varValue    getVarValue(int base, int offset);
    void        setVarType(int base, int offset, varType type);
    void        setVarName(int base, int offset, varName name);
    void        setVarValue(int base, int offset, varValue value);
}
```

*numOfVars* is an integer that indicates the cardinality of the array *variable*. The *behaviour specification* functions use a *base* and an *offset* value to access a variable. This makes possible the declaration and manipulation of array variables in languages like *GSDKL* implemented in *GSDK*.

The behaviour specification of the class *stateVars* consists of the methods:

**stateVars(int numOfVariables):** This method is the constructor of the class. It initializes the *numOfVars* to have the value of the parameter *numOfVariables* and then allocates an array of *numOfVars* *stateVariable* objects.

**~stateVars():** This method is the destructor of the class. It deallocates the resources allocated by the constructor at the termination of the *process*.

**setVarType(int base, int offset, varType type):** This method sets the *type* of the  $(base + offset)^{th}$  *stateVariable* to the value *type* passed as parameter. This method is called only at initialization of the *stateVars* objects from the specification of a user defined program in *GSDKL*.

**setVarName(int base, int offset, varName name):** This method sets the *name* of the  $(base + offset)^{th}$  *stateVariable* to the value *name* passed as parameter. This method is called only at initialization of the *stateVars* objects from the specification of a user defined program in *GSDKL*.

**setVarValue(int base, int offset, varValue value):** This method sets the *value* of the  $(base + offset)^{th}$  *stateVariable* to the value *type* passed as a parameter.

**getVarType(int base, int offset):** This method returns the *type* of the  $(base + offset)^{th}$  *stateVariable*, used for type checking.

**getVarName(int base, int offset):** This method returns the *name* of the  $(base + offset)^{th}$  *stateVariable*, used for variable identification.

**getVarValue(int base, int offset):** This method returns the *value* of the  $(base + offset)^{th}$  *stateVariable*, used for accessing the current value of the variable.

**getVarIndex(varName name):** This method returns the *position* in the array *stateVars*, of the variable with name *name*, used for accessing variables given their *name*.

In a Unix environment a *stateVars* object is implemented in shared memory. Thus, it can be shared between a *process* and its parent *SCP*. The *SCP* needs access to the *stateVar* objects of the site since it is responsible for the recording of the state of the site to the parent *LCS* object.



## 7.4 Process Code

A *process* is the basic unit of execution of a user defined program in *GSDK*. It executes statements (actions) in the order defined in the *events* object. The events object controls the set of actions to be executed by a *process*.

The class *action* is defined as follows:

```
class action
{
    actionTypes type;
    actionParam param;
    int         next;
public:
    actionTypes getActionTypes();
    actionParam getActionParam();
    int         getActionNext();
    void        setActionTypes(actionTypes);
    void        setActionParam(actionParam);
    void        setActionNext(int);
}
```

The *type* object may take the value *create* or *delete* or *sendMsg* or *receiveMsg* or *internal* or *getStateInfo*. The meaning of these values is the same as the meaning of the values of the *command* expression, as given in the *GSDKL* BNF in the previous section. The *param* object is a structure that holds and manipulates the parameters of the actions. The parameters of an action are variable *name* objects. These *name* objects identify variables of the *stateVars* object of the process and are used for accessing and manipulation of the indicating variables from the process according to the *type* of the currently executing action. The *next* object is used to identify the position of the next action to be executed in an array of *action objects*. The *getActionX* methods, when they are invoked on an *action* object, return the *X* state member object of this *action*. The *setActionX* methods set the *X* state member object

to the value of the parameter of the method.

An object of class *code* represents an ordered set of actions to be executed by the *process*. It is an object that controls a variable length array of actions. For each statement defined in the *GSDKL* program, out of which the code of the *process* will be initialized, there is an entry in the array of actions controlled by the *code* object.

A *code* object is a passive object. The state of a *code* object is composed of an object *numOfActions* of class *integer*, an object *act* that is an array of *action* objects and an integer object *current* that is the position of the current action to be executed.

The class *code* is defined as follows:

```
class code
{
    int    numOfActions;
    action *act;
    int    current;
public:
    code(int numOfAct);
    ~code();
    action getCurrent();
    void    setCurrent(int);
    void    incCurrent();
    int    endOfCode();
}
```

*numOfActions* is an integer that indicates the cardinality of the array *act*. The behaviour specification of the class *code*

**code(int numOFAct):** This method is the constructor of the class. It initializes the *numOfActions* to the value of the parameter *numOfAct* and then allocates an array of *numOfActions* *action* objects.

**~code():** This method is the destructor of the class. It deallocates the resources allocated by the constructor at the termination of the *process*.

**getCurrent():** This method returns the *current<sup>th</sup>* action of the array *act*.

**setCurrent(int i):** This method sets *current* of *i*.

**incCurrent():** This method increases *i* to indicate the position of the next action in the array *act* to be executed.

**endOfCode():** This method returns a non-zero value if there are more actions to be executed otherwise it returns zero.

In a Unix environment a *code* object like a *stateVars* object is implemented in shared memory. A *code* object is part of the state of the site and it is periodically recorded by the *SCP* to the *LCS* object.

## 7.5 Site Port

A *sitePort* is a shared object used for communication among active objects that are children of the same *SCP*. It is also used for communication among a *process* and its father *SCP*. In a Unix system *sitePort* objects are implemented in shared memory.

The *sitePort* class is defined as:

```
class sitePort
{
    int      numOfProcs;
    msgQueue queue [NUMoFpROCS];
public:
    sitePort();
    ~sitePort();
    void    registerProcess(pid);
    void    send(message);
    message receive(pid sid, pid rid);
    message receive(pid rid);
};
```

*numOfProcs* is the number of active objects that communicate through the *sitePort* object. *queue* is an array of up to *NUMoFpROCS msgQueue* objects. A *msgQueue* object controls a queue of messages to be received by a *process*. It contains the *pid* of the process to which the queue is associated, and a list of messages. The first queue of a *sitePort* object is associated with the *SCP* object. The remaining queues are used by the children processes of this *SCP* object. The messages contain information about the sender and receiver objects.

**sitePort():** This method initializes the state members of of a *sitePort* object and reserves the first queue for the *SCP* object.

**~sitePort():** This method deallocates the resources allocated by the constructor.

**registerProcess(pid):** This method reserves a queue for the process identified by the *pid* passed as parameter. It is called at the creation of new processes.

**unregisterProcess(pid):** This method sets a queue free, that is, the queue is not assigned to any process. It is called at the termination of a process.

**send(message):** This method inserts the *message* passed as parameter to the queue associated with the sender of the message. The *pid* of the sender of the message is contained in the message.

**receive(pid sid, pid rid):** This method deletes the oldest message sent by the process with id *sid* to process *rid* and returns this message to the caller. If there is not any message send from process *sid* to process *rid* then the process *rid* is blocked until a message from process *sid* is inserted in the queue.

**receive(pid rid):** This method deletes the oldest message sent by any process to process *rid* and returns this message to the caller. If there are no messages in the queue of the *rid* process, then the process is blocked until a message is inserted in the queue.

A *sitePort* object is part of the state of the site and is implemented in shared memory. It is recorded periodically by the parent *SCP* object to the *LCS* object.

## 7.6 Implementation Issues

In a Unix environment objects of classes *stateVars*, *code* and *sitePort* must be implemented in shared memory. Thus these objects may be shared among active objects of the site. Unfortunately, available object oriented languages like C++ do not provide primitives for the implementation of shared objects. A C++ object is implemented as a record (*struct*) that contains the state variables of the object and the addresses of the functions that represent the behaviour specification of the object. This makes the implementation of shared objects in shared memory difficult for the following reason:

Assume two Unix processes  $P_1$  and  $P_2$  are attached to the same segment  $SM$  of shared memory and  $P_1$  copies the value of an object  $O_1$  of its state in  $SM$ . The methods of  $O_1$  are represented in the object by the addresses of functions that reside in the code of process  $P_1$ . However, the code of  $P_1$  is not shared among  $P_1$  and  $P_2$ . If  $P_2$  invokes a method of  $O_1$  it accesses the code of another process and the Unix kernel will kill  $P_1$  to protect  $P_2$ .

*GSDK* provides a *generic monitor* class that is a primitive that bypasses the above described difficulties.

A *generic monitor* class is defined as follows:

```
class monitor(stateStruct)
{
    ...
public:
    monitor(stateStruct)(int creatAttach, int shmkey,
                        int semkey, method* op,
                        stateStruct init);
    ~monitor(stateStruct)();
    void* methodInvoke(int methodId, void* param);
};
```

For example the declaration,

```
monitor(stateStruct) x = monitor(stateStruct)(CREAT, SHKEY, SEKEY, op, init);
```

defines a monitor object *x* with state specification *stateStruct* and operations the members of the array *op*. *stateStruct* is a record type that contains no pointer variables.

The state of a *monitor* object is allocated in shared memory and access to it is controlled by semaphores. The methods that specify the behaviour of the object are allocated in the state of the Unix process that owns the *monitor* object. Information is shared between active objects by monitors that belong to different active objects and are attached to the same shared memory region.

The behaviour specification of the *monitor* class consists of the following methods:

**monitor(stateStruct)(creatAttach, shmkey, semkey, op, init):** This is the constructor of the class. *creatAttach* is an option that indicates if the monitor should be attached to a shared memory region already allocated by another process, or it should be attached to a new allocated region of size of *stateStruct*. *shmkey* is the Unix shared memory key to be used. *semkey* is the Unix semaphore key to be used. *op* is an array of operations to be invoked on shared memory. *init* is the initial value of the shared memory region; it is used when the *creatAttach* option indicates creation.

**~monitor(stateStruct)():** This is the destructor of the class. It deallocates the resources of the Unix system allocated by the constructor.

**methodInvoke(methodId, param):** This method invokes the *op[methodId]* function and passes to it the address of the shared memory region the parameter *param* as parameters. *param* is a free type parameter to be used by the invoked function. It returns what the invoked functions returns.

*methodInvoke* locks the shared memory region before call the *op[methodId]* function using the semaphore associated with the monitor and unlocks it before returning the result of the invoked function. Thus, a function may be invoked on the shared

memory region from only one method and from only one process at a time. As a result, *memory* objects are safe from race conditions.

# Chapter 8

## GSDK Site Control Processes

The *GSDK* is a system distributed over a set of processors. On each processor there is an active object that interfaces the user processes (*process* objects) executed on that processor, with the kernel services. This active object is called a *Site Control Process* or *SCP*. *SCP* is responsible for the creation, maintenance and deletion of the system objects that reside on its site. In this chapter we explore the component objects of an *SCP* object.

### 8.1 Site Controller Structure

The *SCP* controls the objects at a site of the system and sends the recorded states of the site to the parent *LCS* object. The objects at a site are a *port*, a *sitePort*, a *globalClock*, a *processTable* and a *siteInfo*. The recordable state of the site consists of all the above objects except the *port* object.

The class *SCP* is defined as:

```
class SCP
{
    port          gPort;
    sitePort      sPort;
    processTable  proc;
    globalClock   gClock;
    siteInfo      info;
```



```

public:
    SCP(scId, portInfo);
    ~SCP();
}

```

The state of an *SCP* object is composed of the following objects:

**gPort:** An object of class *port*. It controls the global communication of the active objects of the site with active objects that reside on other sites. It is accessible only by the *SCP* object.

**sPort:** An object of class *sitePort*. It controls the on-site communication among the active objects of the same site. It is described in detail in section 7.5.

**proc:** An object of class *processTable*. It is an array of *tableEntry* objects. Each *tableEntry* object consists of the *stateVars* object and the *code* object of a *process* of the site. Thus a *process* shares the *code* and the *stateVars* with the *SCP* object. This makes the recording of the process state information to the *LCS* by the *SCP* possible.

**gClock:** An object of class *globalClock*. It is an implementation of the global time concept, introduced in chapter 4.

**info:** An object of class *siteInfo*. It contains information about the *id* of the *SCP*, the *status*, etc. The *id* of a *SCP* is unique in the system and contains information about the host on which the *SCP* resides. The *status* contains information like the number of *process* objects controlled by the *SCP*, how frequently the state of the site should be recorded, the last time that the state of the site was recorded, etc.

The constructor of *SCP* accepts as parameters the *id* of the *SCP* object and an object *portInfo*, that is, information for the construction of the *port* object. *portInfo* contains the network addresses of *port* objects that reside on other nodes. It uses this information to connect to them.

The *SCP* constructor first creates and initializes the site objects *gPort*, *sPort*, *proc*, *gClock* and *info*. Then it controls the following tasks:

**Global clock management:** The *globalClock* object is invoked to increase its global time value when:

- A message is received through the *port* object from another site.
- Each time the site real clock value increases  $x$  units of time, for a given constant  $x$ . All internal events of the site which have occurred during the time interval  $x$  are considered one atomic action. This is a valid assumption given that on a single node the execution of events is totally ordered.

**State recording:** The *LSR* algorithm is used for the recording of the state of the site and sending it to the parent *LCS* object. The state of the site is recorded every  $d$  increments of the logical clock value which corresponds to the value of the global clock (*globalClock* object) of the site.  $d$  is an integer value that is the same for all the *SCP* objects of the system.

**Inter-site communication:** A message sent by a *process* object of the site  $S_1$  to a *process* object of the site  $S_2$  is first sent using the *sitePort* object to the *SCP* of  $S_1$ . Then the *SCP* of  $S_1$  sends the message to the *SCP* of  $S_2$  using the *port* object. Finally the *SCP* of  $S_2$  sends the message to the receiving *process* object through the *sitePort* object at the  $S_2$  site.

**Process creation:** At *process* creation *SCP* is responsible for the following:

- Allocation of the shared memory regions used by the *process* object for the accommodation of *stateVars* and *code* objects.
- Registration of the new *process* to the process maps of the *port* objects at all the sites in the system. Each *port* object of the system has a *map* object that addresses all the active objects of the network.

**Process deletion:** At *process* deletion *SCP* is responsible for the following:

- Deallocation of the shared memory regions used by the *process* object for the storage of *stateVars* and *code* objects.
- Deletion of the *process* network address from the process maps of the *port* objects at all the sites in the system.

**Site shut-down:** At the request of the parent *LCS* object, *SCP* calls the destructor to deallocate all the objects in the system allocated at the site.

In the following sections, the structure of the component objects of an *SCP* object is explained. The structure of the *sitePort* object is given in the previous chapter.

## 8.2 Port Structure

An *SCP* object, like a *process* object communicates with other active objects through the *sitePort* at that site. The *SCP* receives requests from other active objects in the system from the *sitePort* object. The *port* object at the site is used for forwarding messages from the *sitePort* of the sender object to the *sitePort* of the receiver object when these objects reside on different nodes. The need for two-layered communication system comes from the need to access the state of the communication subsystem that is part of the state of the system. The state of a *sitePort* object is accessible in a Unix environment, under the assumption that it is implemented to be executed in user mode. A *port* object state is not accessible if it is implemented using BSD4.3 Unix IPC primitives. The methods of the *port* object can be easily implemented in a Sun network system using the RPC/XDR libraries for creating servers that access the *sitePort* structures of the site. Then the *port* methods will be clients of those servers. The stubs for the clients and servers can be built using the *rpcgen* stub generator provided by the system.

The *port* class is defined as follows:

```

class port
{
    ...
    pidMap map;
public:
    port(portInfo);
    ~port();
    void register(pid);
    void unRegister(pid);
    void send(message);
}

```

The description of the information about how the RPC server will be accessed is considered too low level to be explained in this document. The *map* object is a table that contains the *id* and the corresponding network addresses of the active objects of the system.

The behaviour specification of the system consists of the following methods:

**port(portInfo):** initializes the state of the port according to the information *portInfo* passed as parameter. *portInfo* contains information about the creation of the RPC port server of the site and its naming.

**~port():** Deallocates the resources allocated by the constructor and terminates the RPC port server of the site.

**register(pid):** This method registers a new active object, identified by the *pid* parameter and its network address, to the maps of the ports of all the sites of the system.

**unRegister(pid):** This method removes the *pid* and the network address of the active object identified by the *pid* parameter.

**send(message):** This method invokes the *send* method of the *sitePort* object of the receiver site of the message to send a message passed as a parameter. The

identification of the receiver of the *message* is included in the *message*.

A *port* object contains information about the invocation of the RPC port servers at all the sites in the system. It could also contain information about only the RPC port servers of the parent site and the sites that have the same parent. In this case two ports that don't have the same parent have to communicate through a common ancestor.

### 8.3 Process Table Structure

A *processTable* object can access the *stateVars* and *code* objects at the site. These objects are shared among the *process* objects and the *SCP* object. Thus the *SCP* may record the state of a *process* and send it to the parent *LCS* object.

The *tableEntry* class specifies an entry of the table and is defined as follows:

```
class tableEntry
{
    code      events;
    stateVars vars;
public:
    void      set(code, stateVars);
    code      getCode();
    stateVars getStateVars();
    void      clear();
}
```

The state of the object consists of an object of class *code* and an object of class *stateVars*. The structure of these objects is explained in the previous chapter. The behaviour specification of the object is defined by three methods: *set*, *get* and *clear*. *set* sets the state of the object to the accepting parameter. *get* returns the state of the object. *clear* clears the state of the object.

The *processTable* class is defined as follows:

```

class processTable
{
    tableEntry    *state;
    int           numOfStates;
public:
                    processTable();
                    ~processTable();
    void          set(int, code, stateVars);
    tableEntry    get(int);
    void          clear(int);
}

```

The state of a *processTable* object is composed of an array of *tableEntry* objects and its cardinality. The behaviour specification provide methods for setting, getting and clearing the  $i^{th}$  element of the array.

## 8.4 Site Global Clock Structure

The site's *globalClock* is an implementation of the global time concept introduced in chapter 4. It is used by the *SCP* for the implementation of the *LSR* algorithm.

The *globalClock* class is defined as:

```

class globalClock
{
    int           numOfSites;
    int           *gt;
public:
                    globalClock(int);
                    ~globalClock();
    void          increase(globalClock);
    int*          getValue();
    int           logicalValue();
}

```

}

The state of a *globalClock* object consists of an array of integers *gt* and the cardinality of this array *numOfSites*. The cardinality is equal to the number of *SCP* objects of the system.

The behaviour specification of the class *globalClock* consists of the following methods:

**globalClock(int)**: It allocates an array of integers of cardinality equal to the parameter passed.

**~globalClock()**: It deallocates the resources allocated by the constructor.

**increase(globalClock)**: This method accepts a *globalClock* object as a parameter and applies the global clock algorithm [40] to modify the value of *gt*.

**getValue()**: This method returns the value of *gt*.

**logicalValue()**: This method returns the logical time value of the *globalClock* (Theorem 6 (GTLT)).

# Chapter 9

## GSDK Central Sites

The *GSDK* central sites are responsible for managing the partial and complete global states and communication among the active objects of the system. Central sites are classified into types:

**LCS:** Leaf Central Site objects are responsible for managing the states of their children *SCP* objects. *LCS* objects are also intermediators in the communication among active objects that are descendants of different *LCS* objects.

**CS:** Central Site objects are responsible for managing the consistent states compiled from their children *LSC* objects or *CS* objects. *LCS* objects are also intermediators in the communication among active objects that are descendants of different *CS* objects.

*CS* objects and *LCS* objects share some common functionality and state structure. Thus, a General Central Site *GCS* is first designed to specify the common structure and functionality of a central site. Then the *LCS* class and the *CS* class are defined by inheriting the common *GCS* class and specifying the additional components of their state and behaviour specification.

### 9.1 General Central Site Structure

The *GCS* class defines the common structure and functionality of the central site objects of the system. It is defined as:



```

class GCS
{
    port          gPort;
    sitePort      sPort;
    gsdb          states;
public:
                GSC(gscId, portInfo);
                ~GCS();
}

```

The state of the *GCS* is composed of three objects: *gPort* of class *port*, *sPort* of class *sitePort* and *states* of class *gsdb*. The structure and functionality of the *port* and *sitePort* objects is given in the previous chapters. An object of class *gsdb* is an implementation of the *GSCA* algorithm introduced in chapter 5. It collects the states of *SCP* objects and forms consistent global states of the corresponding *process* objects.

*GCS* defines the structure and functionality of the children classes *LCS* and *CS*. *LCS* objects and *CS* objects are the central sites of the system.

## 9.2 Leaf Central Site Structure

An *LCS* object is a *GCS* object whose children are *SCP* objects. It collects the states of the children *SCP* objects and compiles them into consistent global states using the *GTCA* algorithm. The consistent global states of the corresponding part of the system monitored by a *LCS* object are sent to the parent *CS* object to be used in the compilation of the state of the entire system.

The class *LCS* is defined as:

```

class LCS: GCS
{
    SCPtable      scp;
    lscInfo       info;
}

```

```

public:
                                LCS(gcsId, portInfo);
                                ~LCS();
}

```

The *LCS* class inherits the structure of the *GCS* class. Thus, a *LCS* object is a *GCS* object with additional state components: an object *scp* of class *SCPtable* and an object *info* of class *lscInfo*. The *scp* object controls a table that contains information about the children *SCP* objects. The *info* object contains information about the node on which the *LCS* object is located and the parent *CS* object.

The behaviour specification of the *LSC* class consists of two methods, a constructor and a destructor. The constructor first initializes the component objects of the state of the object and then fulfills the following tasks:

**Subsystem start-up:** It starts-up the children *SCP* objects on a given set of processors identified in the *SCPtable*.

**Subsystem shut-down:** Under special request from the parent *CS* object, it asks the children *SCP* objects to destroy themselves.

**State compilation:** It receives the states of the children *SCP* objects through the *port* and *sitcPort* objects and invokes a method of the *gsdb* object to compile them into a consistent global state of the controlled subsystem.

**State recording:** Records the consistent global states compiled by the *gsdb* object and sends it to the parent *CS* object to be used in the compilation of a more wide part of the state of the system. A compiled state of the monitored subsystem is recorded as a single state with global time value *GT*. The  $i^{th}$  element of the vector *GT* is equal to the maximum  $i^{th}$  element of the global time values of all the compiled states. *GT* is used from the parent *CS* objects for revealing consistency using the *LTCA* or *GTCA*. This is feasible as proved in theorem 7.

**Inter-site communication:** Forwards messages through the *port* object, allows communication between active objects controlled by different *LCS* objects.

### 9.3 Central Site Structure

*CS* objects and *LCS* objects are very similar. The basic difference is that the class *CS* state has an object of class *CStable* as a member that keeps information for the children *CS* or *LCS* objects. The class *CS* is defined as:

```
class CS: GCS
{
    CStable      cs;
    csInfo      info;
public:
                CS(scpId, portInfo);
                ~CS();
}
```

The only difference between *CS* objects and *LCS* objects is that they control active objects of different classes. *CS* objects control objects of class *LCS* or of class *CS*. *LCS* objects control objects of class *SCP*. A *CS* object receives states of *SCP* objects compiled from the dominated *CS* or *LCS* objects. Then it compiles these states into global states of the controlled subsystem. The compiled state is the state of the entire system if this *CS* object is the root of the system. If the *CS* object is not the root, it records the compiled states at the parent *CS* object.

### 9.4 Global State Database

A *gsdb* object is an implementation of the *GTCA* algorithm introduced in chapter 5. A *gsdb* object controls a database of states of *SCP* objects. When a new state is entered in the database it executes the *GTCA* algorithm and the database is updated accordingly.

The class *gsdb* is defined as:

```

class gsdb
{
    ...
public:
    gsdb(int numOfSCP);
    ~gsdb();
    void    enterState(int numOfSCP, SCPstate *state);
    SCPstate*  ~getCGS();
}

```

The state of an *gsdb* could be implemented using an available data-base system, e.g. a relational or an object oriented database system. The behaviour specification consists of the following methods:

**gsdb(int numOfSCP):** Accept as parameter the number of the descendant *SCP* objects and creates a database that may accommodate up to a given number of states for each descendant *SCP* object.

**~gsdb():** It deallocates the resources allocated by the constructor.

**enterState(int numOfSCP, SCPstate \*state):** This method accepts as parameters an array *SCP* states and the cardinality *numOfSCP* of this array. These states are the consistent global state of a part of the system. They are entered in the database according to the algorithm *GTCA* in an attempt to compile a more recent consistent global state of the controlled subsystem.

**~getCGS():** This method returns the most recently compiled consistent global state of the controlled subsystem.

In the above description, *SCPstate* is a class with a number of *processTable* objects, *sitcPort* objects, and one *globalClock* object as component objects. Compiled states of the controlled subsystem form a new *SCPstate* object that contains all the *processTable* objects and *sitcPort* objects of the component *SCPstate* objects. However, The *globalClock* object of the new *SCPstate* is formed from the *globalClock* objects of the component states (theorem 7).

# Chapter 10

## Concluding Remarks and Future Directions

### 10.1 Use of Pomsets for DCS Modelling

The pomset model is a flexible tool for specifying concurrent systems. In this model both concurrency and nondeterminism are considered primitive notions. Unlike the event structure and branching tree models, and like the linear set model, the behaviour of a system is expressed as the set of its possible deterministic behaviours. However, unlike the linear set model, concurrency is expressed explicitly.

Systems which do not consider where nondeterminism occurs, is better specified in a model that does not consider nondeterminism [35]. It is better to use a simpler model than a more complicated one when the extra tools provided are not needed. Since the distributed system specified in this thesis is not concerned with deadlock avoidance, the pomset model is the most appropriate. Although the pomset model is not powerful enough to deal with deadlock avoidance, it can be used to specify algorithms that detect deadlock.

The use of a formal model like pomsets makes precise expression possible. After introducing the formal modelling of a system (e.g. definition of processes, states, composition, etc...), theorems and statements that express the properties of this system are proved in a simple and clear way.

The formal mathematical notation is not well accepted by programmers/implementors but should not be regarded as a disadvantage since the design that possesses

a clear specification has obvious advantages.

## 10.2 DCS Specification in Pomsets

A method for designing distributed computing systems using pomsets has been given. Processes are expressed as sets of pomsets. The *construction rule*, introduced in chapter 2, is a structured way to construct processes by inheriting properties of other processes.

The *composition* of processes to composite processes and the *decomposition* of them to components has been defined by synchronizing their communication events. This synchronization is a result of the added causality on the events of the process that represents the parallel execution of a set of processes.

The *states* of a process have been defined in the pomset context. Consistent global states of a distributed system have been defined through the *state composition* to composite states introduced in chapter 2. *State decomposition* to component states is defined as well.

The operators introduced for process construction, process composition/decomposition and state composition/decomposition are general and can be used for the design of distributed systems and algorithms.

## 10.3 Logical and Global Time Properties

The properties of *logical* and *global* time have been studied in depth. They are powerful tools for synchronizing distributed computations without affecting their specified concurrency.

Theorem 3 shows that logical time can be used for *recording* consistent global states of a DCS. However compilation of consistent global states cannot always be guaranteed because logical time does not uniquely identify the causal relationship between two events of a distributed computation.

Global time can *uniquely identify* the events of a distributed computation and, as proved from theorem 5, can *reveal the causal relationship* between two events. Thus,

given the states of the component processes, global time can detect all global states that can be formed.

Global time is more powerful than logical time as proved in theorem 6 and the global time properties. All the properties of logical time are properties of global time as well.

## 10.4 State Recording and State Compilation Algorithms

Cooperative recording algorithms have a complexity that is proportional to the number of processes or the number of channels of the system. Moreover, many of these algorithms also assume only one recording of the system in its lifetime. These algorithms make this assumption to avoid the development of a compilation algorithm that composes the recorded states into global states.

Probabilistic algorithms cannot guarantee the availability of a consistent global state. Thus, they are not appropriate for a class of applications such as fault tolerance and distributed debugging.

The periodical recording algorithm *LSR* introduced in chapter 5, has worst case complexity  $O(1)$  each time it is executed. It is executed periodically and records the component states of the system to a central site. The central site may execute the *LTCA* or *GTCA* compilation algorithm to compile these component states to global states of the system. Both *LTCA* and *GTCA* guarantee the availability of a consistent global state of the system.

As shown in chapter 5, *GTCA* has superior performance to that of *LTCA* with respect to how recent the available state is. However, the performance of both algorithms deteriorate as the number of component processes of the system increases.

## 10.5 The GSDK Distributed Programming Environment

*GSDK* provides parallel programming primitives in a distributed environment for:

- Process creation.
- Process deletion.
- Process communication.
- Global State Information.

*GSDK* introduces the request for global state information as a programming primitive. This is useful for applications such as fault tolerance, stability detection etc... at the user level. It also provides the *GSDKL* language that facilitates the use of these primitives.

*GSDK* is an object-oriented system. It is designed as a collection of independent entities that are meaningful and useful on their own. Thus, it is easily expandable and modifiable.

## 10.6 Future Directions

The pomset specification model of the DCS introduced in this thesis is general and can be used for the specification of distributed algorithms. However, in our specification model we purposely omit the channel states because our algorithms assume a reliable FIFO communication subsystem that guarantees that a message is in the state of the sender site as long as it is in transit. Thus channel states can be revealed from the site states.

There are, however, a variety of distributed algorithms that consider channel states. These algorithms cannot be specified with our model in its current form. However, the model may be expanded to provide channel state specification.

Although the logical and global clock algorithms are very convenient for process synchronization, they have a disadvantage. The values of the logical and global clocks increase to infinity as the number of totally ordered events of the system increases to infinity. Some reactive systems like airline reservation systems, banking systems, etc... are designed to be executed forever. Thus, algorithms for resetting the logical and global clocks should be designed to extend the use of these tools.



The *GSDK* design includes details related to the mapping of the system on a Unix like environment. More sophisticated environments like *MACH* may be easily modified to provide the services of *GSDK*.

# Bibliography

- [1] R. V. Baron et al. *Mach Kernel Interface Manual*, Department of Computer Science, Carnegie-Mellon University, January 1990.
- [2] D. L. Black. *Scheduling Support for Concurrency and Parallelism in the Mach Operating System*, IEEE Computer, Vol. 23, No. 5, May 90, pp. 35-43.
- [3] J. Boslough. *The Enigma of Time*, National Geographic, vol. 177, no. 3, March 1990, pp. 109-132.
- [4] G. Boudol, I. Castellani. *Concurrency and Atomicity*, Theoretical Computer Science 59 (1988) pp. 25-84.
- [5] K. M. Chandy and L. Lamport. *Distributed Snapshots: determination of global states of distributed systems*, ACM Trans. on Computer Syst., Vol. 3, No. 1, Feb. 85, pp. 63-75.
- [6] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*, Addison Wesley, 1988.
- [7] K. M. Chandy. *The Essence of Distributed Snapshots*, California Institute of Technology, March 1989.
- [8] D. R. Cheriton, W. Zwaenepoel. *The Distributed V Kernel and its Performance for Diskless Workstations*, In Proceedings of the 9th Symposium on Operating System Principles, October 83, p.p.129-140.

- [9] F. Douglass. *Process Migration in the Sprite Operating System*, Tech. Rep. UCB/CSD 87/343, Comp. Sc. Division. University of California, Berkeley, Feb. 1987.
- [10] M. J. Fischer, N. D. Griffith and N. A. Lynch. *Global states of a distributed system*, IEEE Tran. on Software Engineering, May 82, pp. 198-202.
- [11] A. Gafni, *Rollback mechanisms for optimistic distributed simulation systems*. Proc. of SCS Conf. on Distributed Simulation, 1988, pp. 61-67.
- [12] N. H. Gehani, W. D. Roome, *Concurrent C Project*. Computer Technology Research Laboratory Technical Reports, AT&T Bell Laboratories Murray Hill, New Jersey 07974.
- [13] J.L. Gischer, *The Equational Theory of Pomsets*. Theoretical Computer Science 61 (1988) 199-224.
- [14] C.A.R. Hoare. *Communicating Sequential Processes*, Prentice Hall, Englewood Cliffs, N.J., 1985.
- [15] C. A. R. Hoare, *Monitors: An Operating System Structuring Concept*. Comm. ACM, Vol. 17, No. 10, Oct. 1974, p.p. 549-557.
- [16] R. Koo and S. Toneg, *Checkpointing and rollback recovery for distributed systems*. IEEE Transactions on Software Engineering, SE-113 (1), 1987, pp.23-31.
- [17] D. R. Jefferson. *Virtual time*, ACM TOPLAS, 7 (3), 1985, pp. 404-425.
- [18] D. R. Jefferson et al. *Distributed Simulation and The Time-warp Operating System*, Operating Systems Review, 2 (5), 1987, pp. 77-93.
- [19] E. Jul et al. *Fine-Grained Mobility in the Emerald System*, ACM Transactions on Computer Systems, Vol 6, No. 1, Feb 88, p.p. 109-133.
- [20] T. H. Lai and T. H. Yang. *On distributed snapshots*, Information Processing Letters, 25 (3), 1987, pp. 153-158.

- [21] L. Lamport. *Time, clocks and the ordering of events in a distributed system*, CACM,21 (7), 1978, pp. 558-565.
- [22] T.J. LeBlanc and J.M. Mellor-Crummey, *Debugging Parallel Programs with instant-replay*. IEEE Transactions on Computers, C-36 (4), 1987, pp. 471-482.
- [23] H. F. Li, K. Venkatesh and T. Radhakrishnan, *Global states of a distributed system*, manuscript. Department of Computer Science, Concordia University, 1987.
- [24] H.F. Li, B.M. Dang, C.B. Lea, T. Radhakrishnan, *Debugging in a Distributed Environment*, manuscript. Department of Computer Science, Concordia University, 1989.
- [25] H.F. Li, D. Livas, *Spontaneous Global State Detection Using Global Time*, Proc. 1989 International Symposium on Computer Architecture and Signal Processing. pp. 444-449. Oct. 1989.
- [26] H.F. Li, D. Livas, *Periodical Global State Detection*, manuscript. Department of Computer Science, Concordia University, February 1990.
- [27] D. B. Lomet, *A Partial Deadlock Avoidance Algorithm for Data Base Systems*, Proc. ACM-SIGMOD Conf. on Management of Data (1977), pp. 122-127.
- [28] D. B. Lomet, *Coping with Deadlock in Distributed Systems*, Data base Architecture, North Holland (1979), pp. 95-105.
- [29] B. Meyer. *Object Oriented Software Construction*, Prentice Hall, Englewild Cliffs, N.J., 1988.
- [30] R. Milner. *Communication and Concurrency*, Prentice Hall International Ltd., 1989.

- [31] C. Morgan. *Global and logical time in distributed algorithms*, Information Processing Letters, 20 (1985) pp. 189-194.
- [32] *Network Programming*, Sun Microsystems, 5/9/88.
- [33] G. J. Popek, B. J. Walker, editors. *The LOCUS Distributed System Architecture*, Computer Systems Series, The MIT Press, 1985.
- [34] M. L. Powell, B. P. Miller. *Process Migration in DEMOS/MP*, In Proceedings of the 9th Symposium on Operating System Principles, October 83, pp. 110-119.
- [35] A. Pnueli. *Specification and Development of Reactive Systems*, Information Processing 86, pp. 845-858.
- [36] V. R. Pratt. *Modelling concurrency with partial orders*, Int. Journal of Parallel Prog., Vol. 15, No. 1, 1986, pp. 33-71.
- [37] D.K. Probst and H.F. Li, *Abstract Specification, Composition and Proof of Correctness of Delay-Insensitive Circuits and Systems*. Department of Computer Science, Concordia University, CS-VLSI-88-2. April 1988.
- [38] M. Raynal, *A Distributed algorithm to prevent mutual drift between  $n$  logical clocks*, Information Processing letters (Netherlands), vol. 24, no. 3, pp. 199-202. February 87.
- [39] M. Raynal, *Distributed Algorithms and Protocols*. John Wiley and Sons Ltd. 1988.
- [40] H. Shang, *Consistent Global State: Algorithms and an Application in Distributed Garbage Collection*. Masters Thesis, Concordia University, August 1988.
- [41] S.H. Son and A.K. Agrawala, *A non-intrusive checkpointing scheme in distributed database systems*. Proc. of FTCS 1985, pp. 99-104.

- [42] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.
- [43] R. Strom and S. Yemini, *Optimistic recovery in distributed systems*. ACM Transactions on Computer Systems, vol. 3, no. 3, 1985, pp. 204-226.
- [44] A. S. Tanenbaum, *The Amoeba Distributed Operating System*. Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam. Collection of papers on Amoeba system.
- [45] K. Venkatesh, T. Radhakrishnan and H. F. Li, *Optimal Checkpointing and Local Recording for Domino-free Rollback and Recovery*. Information Processing Letters (Netherlands), vol. 25, no. 5, pp. 295-303, 1987.
- [46] K. Venkatesh, T. Radhakrishnan and H. F. Li, *Discrete Event Simulation in a Distributed System*. COMPSAC Proceedings, 1986, pp. 123-127.
- [47] A. I. Wasserman et al. *The Object-Oriented Structured Design Notation for Software Design Representation*. IEEE Computer, Vol. 23, No. 3, March 90, pp. 50-63.
- [48] G. Winskel. *Event Structures*. Proc. Advances in Petri Nets 1986. Lecture Notes in Computer Science 255 (Springer, Berlin, 1987) 325-392.
- [49] L. Wittie and R. Curtis. *Time Management for Debugging Distributed Programs*. Proc. of DCS Conference, 1985, pp. 549-550.