THE DESIGN AND IMPLEMENTATION OF A DISTRIBUTED PROGRAMMING LANGUAGE

A Thesis

Presented to

The Chinese University of Hong Kong

In Partial Fulfillment of the Requirements

For The Degree of Master of Philosophy

by

Li Wai Kit

May 1985

PREFACE

This thesis covers the design and implementation of a distributed programming language called GDPL. GDPL is a tool for developing distributed software systems for a distributed processing environment. Several previous research efforts and designs for distributed and parallel programming languages have been studied and analyzed. A lot of valuable information about the design philosophies and other issues in concurrent programming languages have been incorporated. The thesis is presented in 6 chapters.

Chapter 1 introduces the attributes and characteristics of a distributed programming language. The research objectives of the thesis are also elaborated in this chapter.

Chapter 2 discusses the current major issues of concurrent programming languages.

Chapter 3 gives an outline of the design philosophy of GDPL and an overview of the language is also presented.

Chapter 4 shows 14 sample programs written in GDPL.

Chapter 5 describes in detail the implementation of the run-time support system and the compiling system.

Chapter 6 contains a summary of the design and implementation of GDPL. Problems encountered and practical experience in using GDPL will be discussed. Finally, a proposal on the future enhancement of GDPL is given.

Preface

Table of Contents

Acknowledgements

I would like to dedicate this thesis to my parents, Mr. S.L. Li and Mrs. T.K.F. Li. Without their encouragement and understanding, this thesis would not have been able to come to fruition.

I would also like to express my special thanks to my supervisor Dr. K.W. Ng, for giving me his valuable guidance and critical advice throughout the years from the design to the implementation of the GDPL language. Without his patience and effort, this project and thesis would not have been able to come to a successful ending.

I must also give my thanks to Mr. Michael Wong, Mr. L.K. Hu, Mr. Simon Lam, Mr. K.L. Lau, and Miss Alison Ng for sacrificing many precious hours in maintaining a smooth operation of the Zilog S8000 system, where the GDPL language has been implemented.

My classmates in the M.Phil. Programme, Miss C.S. Hu, Mr. Bill Kwok, and Mr. W.B. Ngai and many good friends also deserve a vote of thanks for their encouragement throughout the years. Not forgetting Mr. C.S. Law and Mr. W.H. Cheung, for their constructive suggestions.

ABSTRACT

Generalized Distributed Programming Language (GDPL) is a programming language designed to be used in a distributed processing environment and may be used for distributed systems or applications programming. A GDPL program is composed of a number of concurrently executing processes which may reside on a single processor or distributed over a set of processors connected via a network.

GDPL incorporates a number of recent proposals for concurrent programming and represents a synthesis of language features ideally suited for engineering distributed systems. The design of GDPL is based on solving the major issues of concurrent programming languages, such as the nature of concurrency, the model of computation, communication mechanisms, synchronization mechanisms, process creation and termination, access control, exception handling and implementation issues.

The development and implementation of GDPL have been carried out on the Zilog S8000 super micro-computer system with ZEUS (Zilog Enhanced Unix System) as its operating system. Besides the language design, a compiler, a linker and a run-time support system have also been implemented so as to make the language operational. At present, the code generated can only be executed on a single processor with the

multiple processors environment being simulated. Further development effort will be required in order to implement this language on a true multiple processors environment.

CHAPTER 1

Introduction

1.1 Introduction

This chapter introduces the attributes of a distributed programming language and then lays down the design objectives of the GDPL language.

1.2 Attributes of a Distributed Programming Language

A distributed programming language is a concurrent (or parallel) programming language designed for a distributed computing environment. It provides a tool for users to develop distributed computing software running on a distributed computer system. A distributed program contains a number of processes, which are executed overlapping in time either virtually or actually by processor(s). The virtual concurrency can be achieved by interleaving the executions of the processes on a single processor. However, the different parts of a distributed program will execute concurrently in the distributed computer systems configuration [1].

The processes of a distributed program usually must communicate and synchronizate with each other in order to achieve

their common goal. Synchronization is a mechanism to enforce the order of execution of the processes. The methods of current parallel programming languages to express such communication and syncronization fall into two categories [2]:

(i) Using shared memory and explicit syncronization primitives - a typical example is the monitor [35]

(ii) Using the message passing mechanism to achieve communication and synchronization purposes - this is a generalized parameter passing mechanism.

A distributed computing environment (or configuration) consists of a number of processes connected together, either by simple communicatioin links or by a network. The processors usually have distributed memory spaces. Program and data may be distributed over these processors and processes can be executed concurrently. Strategies for distribution of processes and data are embedded in the distributed environment to utilize the resources of the processors in an efficient manner. Hence designing and implementing a distributed programming language may face the isssues of parallel programming language design and the strategies to distribute and execute processes on the distributed computing environment.

1.3 Research Objectives

There are several objectives in the design and implementation of the distributed programming language called

Generalized Distributed Programming Language (GDPL). They are presented below:

(a) To provide considerable expressive power of concurrency for a wider range of parallel programming applications.

(b) To obtain maximum computing speeds using multiprocessors with common or distributed store.

(c) To define a structured, clear access control in order to increase the readability, security and verification of distributed programs.

(d) To implement a compiler and a linker for the language such that executable codes can be generated and distributed onto different processors.

(e) To write a run-time support system to support the execution of the different parts of the distributed program.

(f) To write several sample programs of the language in order to evaluate the language against the above objectives.

(g) To gain the experience of the design and implementation of a distributed programming language.

CHAPTER 2

Major Issues in Parallel Programming Langauges

2.1 Introduction

There is no doubt that the parallelism of execution is an attractive facility to realise the concurrency inherent in application systems and system-oriented systems. The development of distributed computing systems, and networking make parallel programming more realistic and practical. Therefore there is a real demand and neccessity for a notation or programming language to communicate to these sytems. Many such programming languages have evolved over the years. They have their own design philosophies and purposes for various applications and underlying machine models. Twelve parallel programming languages have been studied and several major issues in parallel programming languages have been located and analyzed. Thus, they form the basis for the development of the design and implementation of the distributed programming language GDPL.

The issues involved include the nature of concurrency, the model of computation, communication mechanisms, synchronization mechanisms, process creation and termination, the access control in parallel programs, exception handling and

implementations issues. Actually, some of the issues are closely related or overlapped in some situations. They are treated separately for simplicity, clarity and ease of analysis. The twelve parallel programming languages selected are : Distributed Pascal [1], Distributed Processes (DP) [2], Synchronizing Resources [3] [10], Modula [5] [6] [7], Communicating Sequential Processes (CSP) [8], Gypsy [11], Edison [12], Soma [13], ADA [14] [15] [17] [19], DPL-82 [21], PLITS [22] and Concurrent Pascal [23]. The design and implementation strategies of them will be quoted as examples.

2.2 The Nature of Concurrency

The major objective of parallel programming languages is to express the concurrency inherent in a problem effectively and efficiently. Increasing the concurrency within a program is to gain the potential improvement of the speed of execution.

The concurrency of a program is affected both by the underlying machine architectures and the nature of the application systems. The underlying machine architectures include single processor, multiprocessors with common or distributed store, distributed computer network, etc. The application systems can be divided into loosely coupled and tightly coupled systems. In a loosely coupled system, the computation is spread among several processors which are connected by

some communication paths. The programs are expected to be made up of largely self-contained processes which will share very little information directly. In a tightly coupled system, a larger amount of shared entities are shared by the processes.

The nature of the concurrency ranges from the restricted to the unrestricted form. The restricted one is suitable for tightly coupled application system and is easier to be implemented in a single processor or multiprocessors with common store. The other extreme is the unrestricted one. The maximum degree of parallelism in a program is equal to the number of processes declared in the program. This is at most suitable for loosely coupled application system and implemented in a distributed computing environment. Message passing is the sole means of communication. The processes share nothing, but pass the data between the processors and store in their private stores. Most of the parallel systems lie between the two extremes, a number of mechanisms for expressing and implementing concurrency are discussed in the communication and synchronization mechanisms sections.

2.3 The Model of computation

There are a number of approaches to selecting an underlying mechanism for controlling the computation in a parallel program. The two ends are the data flow mechanism and control flow mechanism. The data/control mechanism lies in the middle of the two ends, which is the MIT Actor [21] model of

computation.

For data-flow mechanism, processes can continue their computations as long as all operands are available. Operands of a process are a list of data to be waiting for. Each process is responsible for sending the data to the other process. The language DPL-82 is a typical example, which depends on the data flow mechanism. The processes of a DPL-82 distributed program are called nodes. The nodes are connected by the communication links which are called arcs. Arcs transmit operands to other nodes, which are tokens and signals. The computation of a node proceeds as soon as all required tokens and signals are available. The availability of operands is detected as an activation condition within the node.

There are a number of languages such as Guarded Procedure (GP) [18], Communicating Sequential Processes (CSP) [8], Concurrent Pascal [23], Soma [13], ADA [14], Distributed Processes (DP) [2] and Modula [5] which apply the control flow mechanism. The control flow mechanism is characterized by the remote procedure call concept or monitor concept. After the procedure call is terminated, the control will return to the invoking process. This is the major difference between the data flow and control flow mechanism.

2.4 Communication Mechanisms

Communication is to convey data from one program component

(e.g. process, module or class) to another. Communication can be divided into: direct and indirect. The direct communication is achieved by the direct access of the shared variables. The indirect communication is achieved either by calling procedures/operations of the other program components or by message passing. Some languages select one or a combination of the three means as the language communication mechanisms. Each one has both advantages and disadvantages over the other. No one of them is ideal for all application systems. Therefore, a language should select the most suitable one for its own envisaged application area.

2.4.1 Direct Shared Variables

Within a program, it is possible to structure the code into a number of concurrent processes which communicate with each other via shared objects. Since the processes run in an unpredictable order, it means that some mutual exclusion mechanisms are required to ensure the correct usage of the shared variables such as semaphores, test and set instruction, P and V operation. The mutual exclusion is either automatically achieved by the machine or explicitly programmed by using conditions.

2.4.2 Procedure or operation calls

The languages under this category are based on the assumption that procedures or operations are much more reliable interactive mechanism than shared data struc-

tures. This category can be subdivided into loosely coupled and tightly coupled mechanism. In loosely coupled mechanism, the data are copied from one process to the other. In tightly coupled mechanism, the calling process is allowed to reference the variables of the called process as long as it is permitted.

(a) Loosely coupled mechanism

For loosely coupled system, after the communication is permitted, the data can be copied from one process to the other. In Communicating Sequential Processes (CSP) [8], the input and output commands are used for communication between concurrent processes. The communication is said to be ready when one process names another as destination for output and the second process names the first process as source for input. In addition, if the type of the variable of the input command is compatible with the value denoted by the expression of the output commands, then the input and output commands are said to correspond. The value to be output is copied from the source to the destination process.

(b) Tightly coupled system

For tightly coupled system, the monitor is the typical construct used to share a number of objects among a collection of processes. Each process can access the shared object indirectly through monitor procedures. However,

guards are required to control the access of shared objects. Concurrent Pascal [1] and Distributed Processes (DP) [2] are based on the monitor concept. The processes are active and the monitors are passive. The monitor provides exclusion of simultaneous access from several processes to shared objects. If a process has called any monitor procedure (critical region or guarded region), another process calling the same or another one of these procedures is delayed until the first process has completed its critical regions. During the execution of the critical region, the calling process is allowed to store its data into the shared objects. If the other process has the chance to enter the monitor afterward, the process can read the data from the shared objects. Hence, the communication objective is achieved.

2.4.3 Message passing

The message passing mechanism is the most suitable for those distributed languages implemented on processors with distributed store. It is subdivided into the asymmetric and symmetric mechanisms. In the asymmetric mechanism, buffers must be required. However for the symmetric mechanism, it may be implemented without intermediate storage for message but the sender of a message must be delayed until the message has been received.

There are a number of advantages to using message

passing:

(1) It simplifies the common exclusion problem.

(2) It can be used as the cleanest way of parameter passing.

(3) It solves the distributed store problem.

(4) It provides a multiple returned values facility.

The two types of message passing mechanisms are described below:

(a) Asymmetric mechanism

Soma [13] unifies the process and the monitor constructs. Somas are communicated directly by exchanging messages via mailboxes. It can be easily implemented on a distributed architecture. In the soma body an arbitrary number of mailboxes can be declared. A mailbox address is associated with each mailbox. The mailbox identification must be made available from one soma to the other to establish the communication link of message passing.

The soma in which a mailbox is declared can obtain a message from this mailbox by executing the receive operation. If the mailbox is empty, the soma is delayed until a message is sent. Any soma having the mailbox identification can send a message to the mailbox by the

send operation, a send operation is never delayed. It is the principle of the asymmetric mechanism.

(b) Symmetric mechanism

DPL-82 [21] depends on the port and message concepts of Rashid's CMU VAX/UNIX Interprocess communication path called arc. The run-time support facilities provide protocols via the IPC for establishing communication path between processes and passing the data on those paths. The processes can proceed as long as all the data required are available.

In Gypsy [11], processes communicate only through message buffers. A message buffer is a finite size queue on which send and receive operations are defined. The queue is manipulated in a FCFS manner. When a send is made on a full buffer, the sending process is suspended until the buffer is not full. On the other hand, a receive on an empty buffer also causes the process to be suspended until it is not empty. A semaphore is associated with every buffer such that mutually exclusive access to the buffer is achieved.

In Synchronizing Resources (SR) [35], the send statements are like message sends. If the operation is invoked by send, the invoking process may proceed as soon as the actual parameter values have been saved or terminated.

2.5 Synchronization Mechanisms

Synchronization mechanisms are used to force the sequence of execution of program statements in a correct manner. It can be implicitly determined by the underlying run-time system or explicitly specified by the programmer. If it is implemented implicitly by the run-time system, a large amount of work for the synchronization is relieved from the programmer and the synchronization mechanism is defined in a more precise manner. It is opposed to those explicitly specified by the programmer. However, more freedom is given to the programmer for controlling the sequence of execution in more sophisticated application systems and it is assumed that the programmer has full understanding of his own system. It is a rare case that the synchronization is achieved solely by the implicit run-time system without the aid of the programmer.

The synchronization mechanism is classified into: indirect by examining the shared variables, and direct by sending and waiting for messages or by invoking the procedures, operations and input/output commands.

2.5.1 Indirect shared variables

In Edison [12], process synchronization is controlled by when statements. The concurrent processes can only execute the when statements one at a time. When statements are used to give concurrent processes exclusive access to common variables when their value of the expression

satisfy certain conditions. If the value of an expression is not true, the process is delayed until the expression is true.

A process executes a when statement in two phases: synchronizing phase and critical phase. In synchronizing phase, the process is delayed until no other process is executing the critical phase of the when statement. After the critical phase (i.e. the synchronized statements) is executed, the process is either returned to the synchronizing phase or the execution of the when statement is terminated.

2.5.2 Invocation of procedures, operations and input/output commands

(a) Procedures

In Guarded Procedure (GP) [18], a guarded procedure can call the guarded procedure of another process but not its own. A process continues to execute its initial statement until either it terminates or waits until the guard of the guarded procedure is ready. Then the process selects arbitrarily one of the ready guarded procedures for execution if more than one guarded procedures are ready. The execution of the guarded procedure in a process is delayed until the last of the same guarded procedure call is completed. This waiting is necessary because there is only one buffer associated

with each guarded procedure of each process to hold the input and the output parameters.

In addition to the interprocess guarded procedure, guarded command is the other way of synchronization mechanism within each process. The guarded command consists of a guard and a command list. A guard is said to be executable if it does not fail. A guard is said to be enabled if it does not fail and also there is a call statement in the guard. The guarded command in GP is more efficient then that proposed by Hoare. In GP, the selection is made between enabled guards instead of executable guards and is delayed until one or more guards become enabled.

In ADA [14], it is possible to set a signal without waiting, and to wait for a signal. However, the major synchronization mechanism is the rendezvous. When the synchronization is achieved, the rendezvous is said to have occurred. The rendezvous consists of executing statement between the do and end keywords following the accept statement. The rendezvous is completed as long as these statements have been executed. The level of synchronization mechanism is higher than semaphores. The conditional accept statement is introduced to specify the conditional execution of entries. The other statement: select statement is introduced to select among a number of alternative accept statements depending on the

order in which entry calls occur.

(b) Operations

In Distributed Processes (DP) [2], a process interleaves the execution between the initial statement and external requests one at a time. A process switches from one operation to another only when an operation terminates or waits for the satisfaction of the condition within a guarded region. A process continues to execute operations except when all of its own operations are delayed for some conditions within guarded regions or when it makes a request of operation of the other process. The process must wait until the operation is completed by the called process.

(c) Input/output commands

In Communicating Sequential Processes (CSP) [8], the input/output commands may appear in guards. The guarded command is selected for execution if the corresponding input and output command is ready to execute the corresponding output and input command. If several input or output guards are ready, only one of them is selected arbitrarily.

2.5.3 Signals

In Soma [13], mailboxes may be used to transfer message containing no information but only for synchronization

purposes.

In Modula [5], the synchronization is explicitly achieved by the use of signals. The signals of Modula correspond to conditions of Hoare and to queues of Brinch Hansen. The processes can be influenced by signals and shared variables only.

Two operations and a test statement can be applied to the signals: wait and send operation, await statement. Wait (s,r) delays the process until it receives the signal s, and the priority r is given to the delayed process. Send (s) sends the signal s to those processes waiting for s with the highest priority. If there are several processes waiting for s with the same priority, the process with the longest waiting time receive the signal s. Await (s) return the value true if there is at least one process waiting for signal s, otherwise it returns false.

If the signal is sent from a process to the other process waiting within the same interface module, then the control is passed from the sending process to the receiving process. The sending process is delayed until the other process has completed its interface procedure. Under this mechanism of send and wait operations, a one-in one-out sequence is achieved and the mutual exclusion problem is solved.

2.6 Process creation and termination

The process creation and termination policy directly affects many design decisions of parallel languages. The design decisions include storage requirement, program verification, modularity, process deadlock, flooding and starvation, formal definition of syntax and semantics, etc. The process creation and termination can be in a restricted manner or a free manner. In a restricted manner, the flexibility of program structuring is reduced. However, it is easier to implement and verify. In a free manner, the programmer is given a lot of freedom to structure his application systems.

Many parallel languages use a process name and a subscript which may contain one or more ranges to stand for a series of processes. Only a slight variation exist in different languages. In ADA, an alternative instantiation statement is used. However, it is more convenient to use subscript instantiation and is especially important when a number of similar processes are defined.

2.6.1 Restricted manner

In Concurrent Pascal [23], As long as a process is created, it exists forever. This greatly simplifies the problem of storage management. The process and monitor can be initialized only once. After initialization, the parameters and local variables of a process and monitor exist forever. They are called permanent variables. Even when the execution of a process terminates its permanent

variables such as monitors continue to exist because they may be used by other processes.

In Modula [5], processes cannot create other processes, process creation is confined in the main program. This means that processes cannot be nested. Modula [6] supports dynamic process generation, however, the generation of new processes is restricted to the main program to reduce the complexities of nested processes and storage requirements can be estimated.

In Communicating Sequential Processes (CSP) [8], there is no recursion of processes and no process-valued variables. Therefore, the number of concurrent processes is known at compilation.

2.6.2 Free manner

In DPL-82 [21], if child nodes are required then the child nodes are declared in the use section. The child startup and initialization of parameters are performed in the initialization section. The connections of nodes are established in the arc interconnection section. The freedom is given to the node in self-termination and child node restarting.

IN ADA, more freedom of process creation is allowed. Processes may be initiated at any point. However, it is not possible to exit the scope of a process until the process is terminated.

2.7 Access Control in Parallel Programs

Besides the communication and the synchronization mechanisms of parallel programming languages, access control is another important issue in order to develop a structured, reliable and secure parallel programming system. Therefore a number of clear and exact access control schemes must be stated explicitly in order that the programmer can use these access rights correctly. Moreover, information is provided for the compiler to check the validity of certain illegal accesses which are ignored by the programmer.

The access control schemes should be able to

(i) define clearly the scope of the data objects and operations that each program component uses in a precise manner

(ii) restrict that only meaningful operations are allowed to operate on the data objects, which is also a requirement of the abstract data type

(iii) schedule the order of access to controlled data objects

The access control schemes can be divided into static access and dynamic access. The access path of the static access scheme is clearly defined at the language definition time. The objective of the static access scheme is to make the access of the data objects well under control. Hence, it reduces the unnecessary or meaningless access of the con-

trolled data objects. The access path of the dynamic access scheme is determined at run-time. The access right of data objects, such as monitors, is passed from one process to the other process.

The static access scheme is also called the scope rules in a program, which is a common way to specify the access rights in a program. In many parallel programming languages, the nested block structure is the most common scheme. More care about the access rights must be taken because the nested block structure is unrestricted in an uncontrolled sense.

There are a number of approaches to explicitly express the static access control scheme such that the compiler can reduce a number of unwanted and meaningless accesses:

(i) A process or module is totally self-contained, which may only access its own variables and operations, no local variables can be examined from the outside, no procedures can be invoked. No nested block structure and recursive procedure call are allowed.

In Distributed Processes (DP) [2], parameter passing between processes can be implemented either by copying within a common store or by input/output between processors that have no common store. No direct access of the data across the process boundary is allowed.

In Gypsy [11], it has no non-local variables, all variables are either local variables or parameters. This enhances the

reliability of program and also simplifies the verification.

In Concurrent Pascal [23], the parameter passing is used at the initialization phase to establish the connections of the program components. All variables accessible to a program component are declared in type definition. This access rule and the init statement make it possible for a programmer to state the access right explicitly and have them checked by a compiler. Nested monitor operations are not allowed.

(ii) Variables must be exported from one process to the other through some explicit statements. In Edison [12], a module is either contained in a procedure or in another module. The module is a block consisting of two kinds of named entities : local and exported entities. The local entities can only be accessed within the module. The exported entities can be accessed by the module and the immediate surrounding block. This means that the scope rules of the variables are restricted to two levels of nested block. The compiler can generate instructions for addressing the references of the local and the exported entities within the local block and immediate surrounding block. In ADA [14], there are two scopes : open scopes and closed scopes. The open scopes are those that are globally declared variables. Closed scopes are those in which the non-local variables must be explicitly imported.

(iii) Process or module may be nested. Explicit statements are required to list the variables that inner process or

module can use. The inner process or module may export variables to the outer process or module as well. In Modula [5], a module is a set of procedures, data types and variables. Modules may be nested and procedures form a block structure. It allows the programmer to control the entities that are imported from and exported to the environment of the module. Other entities are local to the module and cannot be accessed by the other modules. In the heading of the modules, there are two lists of identifiers : define-list and use-list. The define-list identifies all the entities exported to the outside of the modules. The use-list identifies all the entities imported to the module. The entities can be exported further more than two block levels if necessary.

(iv) A new grant feature [16] is introduced in that grant declarations allow the data objects' visibility to be selectively expanded.

The dynamic access schemes are relatively uncommon because they are usually implicitly defined by the communication and synchronization mechanisms. However, there are still a number of approaches to explicitly express the dynamic access schemes to increase the efficiency of implementation and representation.

(i) In ADA [14], it is possible to mark variables that are shared among parallel processes, an unmarked variable that is assigned on one access path and then used on another will

cause a warning.

(ii) the concept of capability is introduced in [16]. All the controlled data objects and their corresponding dynamic access rights are stored in the capability variables as other data objects are stored in variables. Then access rights are passed between processes by passing capability variables as parameters to and from the data objects.

2.8 Exception handling

Among the twelve parallel programming languages, only PLITS [22], Gypsy [11] and ADA [14] provides the facilities for handling exception situations. The purpose of exception handling is to specify what actions should be taken if some software or hardware errors or unusual events occur. Exception handling introduces another way of control from one program component to another. In general, the process of exception handling include declaration, raising and handling of exceptions.

In PLITS [22], exception handling depends on the transaction key. A transaction is a unique key such that the exception handling process can route the correct answer to the originating process.

There are two classes of exception conditions: internal conditions arise within a module and external conditions occur outside the module. The internal conditions include arithmetic overflow, end of file, type mismatch, etc. External

exception conditions include invalid messages, time-out notifications, etc. The external exception condition is raised through messages and handled by an exception process. The internal exception conditions are handled by a set of handlers procedures. When a handler is invoked, it runs under the environment of the block where the exception occurs. The exception may be passed on to another handler if necessary. After the exception handling is completed, the control can return either to the point of invocation or exit to some higher level block.

In ADA [14], the exception conditions include both the hardware, software errors detected during execution, error in built-in operation and user defined exceptions. The raised exceptions are handled by exception handler. There might be more than one handler for the same exception condition. However, the handler is sought first local to the subprogram where exception is raised, then in the environment of call of the subprogram, and the searching is continued in this way until a handler is found for handling this exception condition. Then the exception is executed in the environment of the handler. Therefore, the scope problem should also be considered carefully. After the handler is completed, the environments in which the exception occurs and handled are abandoned. ADA also allows the suppression of exceptions to reduce the run-time checks. However, the behavior of the translation and execution cannot be predicted.

2.9 Implementation issues

In this section, only some key features and issues of implementation of parallel programming languages are stated. The key features include the architecture of the underlying machine, storage management, process scheduling, process switching, timing constraint, context switching between processes and scheduler, formal definition of semantics of the language and proof rules, nondeterministic execution, interfacing between processors, process deadlock, flooding and starvation etc.

Edison [12], Gypsy [11], Modula [5] and Concurrent Pascal [23] are primarily designed oriented towards the single processor. Even when they are implemented on a multiprocessors system, there should be a common store. Guarded procedure (GP) [18], DPL-82 [21], Distributed Pascal [1] and Communicating Sequential Processes (CSP) [8] are designed for distributed computer systems. Although they can also be implemented on a single processor, the power of distributed computations is reduced. Synchronizing Resources (SR) [10], PLITS [22] and ADA [14] are suitable for both single processor and distributed computer systems. Processes should have consistent semantics whether implemented on multiprocessor or interleaved execution on a single processor. However, the drawbacks due to the common store of Concurrent Pascal and Modula can be removed by the Soma construct [13]. Soma makes all program components active and have adequate facilities

allowing the program components communicating directly.

The storage management is highly related to the processes or modules (including their temporary and permanent variables) creation and termination manner and automatic message buffering. Static storage management and dynamic storage management are required for those languages with restricted and free manner of process creation and termination respectively. It is obvious that static storage management can be easier and more efficient to implement. On the other hand, dynamic storage management requires more overhead and storage space to implement, hence it is easier to introduce more system errors. However, the dynamic scheme is a necessity for those languages with more flexible concurrency requirements.

There are three approaches to processes scheduling: implicitly scheduled by the run-time scheduler, explicitly determined by the programmer, or the scheduling decision is embedded in the process structure. The major objective of process scheduling is to minimize the overhead due to the context switching between processes and the scheduler, and to handle the asynchronous event from the external environment such as I/O device interrupts in a correct way.

In general, the computer systems on which concurrent processes run will handle short-term scheduling of simultaneous execution of processes. There are a number of language facilities provided for the programmer to delay or

schedule processes for longer periods of time until some conditions (internal or external) are satisfied. Hidden scheduling decisions should be avoided.

IN ADA [14], the built-in scheduling algorithm is FIFO within the same priority. In Concurrent Pascal [23], a process does not directly call a monitor procedure, the call is supported by a kernel routine which solves the mutual exclusion problem and busy queuing. This kind of kernel facility is implemented by a compiler transparent to the programmer. The queue of Concurrent Pascal can be used by monitor procedures to control medium-term scheduling of processes, a monitor can either delay a calling process in a queue or continue a process waiting in a queue.

For CSP [8], if a group of processes are attempting communication but none of their corresponding input and output commands are executed, it is called a deadlock. When two or more modules in a PLITS [22] program are attempting to receive a message which must be sent by another, deadlock occurs. When a module generates too many messages, flooding occurs. When a module does not receive messages intended for it, starvation occurs. There are three kinds of solutions to the above error conditions: the kernel must provide more facilities to minimize the number of avoidable deadlocks; use system-generated exception; and to prohibit interrupts for the module, therefore no deadlock will occur but the level of parallelism will be reduced.

CHAPTER 3

The Design of GDPL - A Generalized Distributed Programming Language

3.1 Overview of GDPL

GDPL is a distributed programming language designed to cope with the main parallel programming issues and it incorporates a number of good design philosophies. The syntax and semantics of the sequential part of GDPL are the same as the classical sequential programming language Pascal except for the followings:

(a) The sequential control structure is based on Dijkstra's guarded command [19] giving the whole language a uniform control structure.

(b) The 'go to' statement is not allowed, in order to avoid unstructured control flow[23].

(c) Pointers are not provided in order to have a clear and secure program. The cost of the implementation of pointer references in a distributed configuration is also reduced.

(d) Global variables are not allowed, in order to maintain a clear scope rule of program objects.

(e) Files are not provided but the basic I/O statements are

implemented. This reduces a great deal of work required to implement a file management system.

The concurrent programming part of GDPL includes the node, the process, the service and the utility. Interprocess communication and synchronization is based on the shared variable and the message passing mechanism. An overview of the language is given in the following sections. The syntax of GDPL is described using an extended BNF notation (see Appendix B). The detailed language specification is presented in Appendix A.

3.2 Programs

A program consists of global message definitions and one or more nodes which are similar to the resources of SR [4-5]. It has the form

```
[<message definition list>]
<node> {<node>}
```

In GDPL, a program is constructed in such a way that all nodes are executed simultaneously in their virtual processors. The virtual processors may be mapped onto a single processor or multiple physical processors. The physical processors can either be homogeneous or heterogeneous provided that they have the proper communication links and virtual processor software. The programmer can determine which group of processes are tightly or loosely coupled and they are expected to group those tightly coupled processes

together under the same node. Hence, those loosely coupled processes may be distributed onto different virtual processors. If the loosely coupled processes want to communicate with each other, they have to use the message passing mechanism to send their data. Two tightly coupled processes may also communicate with each other using this mechanism.

A message definition defines a number of message constructors, and is similar to a type definition except that a new data type 'address' is also permitted. The data type 'address' is a string of characters to represent the source or destination message addresses. The addresses must include both the name of the process and the node in which the process resides. However, only the process can utilize the facility provided by the message passing mechanism, including the message constructors, 'messin' and 'messout' statements.

3.3 Nodes

Each node consists of a fixed number of processes and services. It has the form

```
node <node name> ;
[ <service> { <service> } ]
begin
  <process> { <process> }
end <node name> ;
```

A process is an active program component. After the process

is initialized, it continues to exist until the last statement of the process is executed or interrupted by some events. The processes of a node are initialized at the same time and they execute simultaneously in their virtual processors. Since all the nodes of a program are initialized at the same time, therefore, all the processes are also initialized at once. After the initialization, all processes within the same node are put into the queue waiting for execution according to their initial values of priority. Then the program is executed by dispatching a process from the queue waiting for execution or from the sleeping processes queue.

3.4 Services

Another program component that can exist within a node is called a service. It has the form

```
service <service name>
      [ '[' <range> ']' ]
[ < constant definition list> ]
[ <type definition list> ]
<common variable definition list>
[ <variable definition list> ]
<utility> { <utility>}
begin
  [ <initialization statement list> ]
end <service name> ;
```

The service is a kind of abstract data type which provides

data and operations (through utilities) in a secure manner and transparent to the users (processes). It means that processes can access the data and operations without considering how the data and operations are implemented. Thus, the implementation can be changed if necessary without affecting the users' view of the utilities. For example, a device control program can be implemented using the service construct. Processes can use the device via utility calls without worrying about the address references and manipulations of the device registers. The machine dependent code is embedded in the utility so that no other program components can reference the device registers.

Putting all the shared objects of each service of the node together, they represent all the shared objects of the node. The shared objects are not only identified by their names, but also by the service in which they are declared. Therefore, two services of a node can use the same name to denote shared objects with no conflict. Each service must contain one or more shared variables, otherwise it is meaningless to use a service without shared objects. Furthermore, the shared objects are visible only to processes in the same node.

An array of services is allowed such that several similar services can be defined simply by a single service definition and an index. A range is added to the service definition in order to define a family of similar services. In

many parallel programming applications a number of similar cooperating services are required to support a single service to the users, therefore the service array is useful. In the family of service array, their declarations and service body are exactly the same except for their identifications. The index of the family of the services and the service name is used to uniquely identify a member of the family. The attribute (me) of the service contains the index of the current service. The initialization statement list forms the body of the service. The body of the service is executed when the node is initialized. After it is executed, the service becomes inactive and waits for activation.

3.5 Utilities

An utility has the following form

```
utility <utility name>
      '(' [ <parameter list> ] ')' ;
[ <constant definition list> ]
[ <type definition list> ]
[ <variable definition list> ]
begin
  <statement list>
end <utility name> ;
```

There are no traditional procedures or functions within the service except the utility. The statements allowed within an utility include all sequential statements, utility calls and

a wait statement. It is assumed that the utilities are composed of a number of critical regions which are usually quite short. Therefore, it is not necessary to use the traditional procedure or function to decrease the repetitive coding. An utility can use the shared objects declared within the service and its local objects which are transparent to the user. Utility calls for other utility of the same or other service in the node is allowed. Recursion of utility is also allowed.

The parameter list of an utility is similar to those in ordinary procedures/functions. This is the unique way of passing data between the calling process and the called utility. Both pass by value and pass by reference are allowed.

3.6 Processes

The process of GDPL is similar to the process used in other parallel programming languages, it is used as the major construct to collect a number of program codings together to be executed simultaneously with other processes in the same or other nodes. Processes in the same node are competing for the resources of the same virtual processor.

Each process defines its own private objects and consists of a number of statements, including sequential statements, procedure/function calls, utility calls and message passing statements. A process has the form

```
process <process name>
     [ '[' <range> ']' ]
     [ '(' <priority> ')' ] ;
[ <constant definition list> ]
[ <type definition list> ]
[ <variable definition list> ]
[ <function declarations> ]
[ <procedure declarations> ]
begin
  <statement list>
end <process name> ;
```

An array of processes is allowed such that several similar processes can be defined simply by a single process definition and an index. A family of processes is valuable for many parallel application systems and systems oriented work. Priorities of each process or family of processes are optional. The default value of the priority is Ø, which is the lowest value, the highest value is 127. Every process can obtain an initial priority in the process definition. They can change the priority using the predefined variable 'priority'. A process has an attribute 'me' which contains the index of the current process. If it is a member of a family of processes, then the index reflects the relative pointer within the family. Otherwise, the value of me is Ø.

When a process issues an utility call, it cannot enter the service immediately. A process is allowed to enter those

services without processes or only with sleeping processes. The control of the process is then passed to the called utility. After the utility call is executed, the control will be returned to the process.

3.7 Communication and Synchronization

Communication and synchronization in GDPL is achieved through shared variables, messages, and some specialized statements.

3.7.1 Shared Variables

When processes communicate through the indirect access of shared objects, the order of execution of the processes within the service is nondeterministic. Therefore the internal synchronization of the utilities are required to enforce a correct sequence of execution. The internal synchronization between utilities within the service is achieved by the 'wait' statement (or conditional critical region statement). The 'wait' statement has the form

```
wait <guarded command>
     { [] <guarded command> }
end
```

A guarded command has the form

```
<guard> -> <statement list>
```

The statement list of a guarded command can be executed

if its guard (Boolean expression) is true. The 'wait' statement is composed of one or more guarded commands. When the current active utility executes a 'wait' statement, all guards of the 'wait' statement are checked. If one or more guards of the 'wait' statement is satisfied (true), then one of the satisfied guarded commands is selected arbitrarily, and the statement list of the selected guarded command is executed. The statement list following each guard should be executed until it is completed. However, if none of the guards of the 'wait' statement is satisfied, then the utility is delayed (i.e. it enters the sleeping state) The process which has called this utility is said to be inactive. If it is time to activate the sleeping process, the guards of the called utility are examined again to determine if there is a satisfied guard. If there is such a guard, the statement list following the guard is executed. The execution of the utility will continue until it is completed or another utility call is issued. When the utility is completed, the process becomes active again.

3.7.2 Message Passing

The unique means of communication between processes in different nodes is through message passing. The message passing mechanism is also used for the purpose of synchronization of two processes. This kind of synchronization is called external synchronization to distinguish

it from the internal synchronization between utilities. Message passing may also be used by processes in the same node. The message passing mechanism is in an asymmetric manner similar to the Soma [10]. There is no need to delay the source process after a message is sent. However, the destination process must wait for the arrival of all required messages. 'Messin' and 'messout' statements achieve synchronization in the following way: if a 'messin' statement is executed, then the execution of the process will be delayed until the corresponding 'messout' statement has been executed.

The 'messin' and 'messout' statements have the following forms

```
messout
  <destination address>
 '(' <message variable> ')'

messin
  ( '(' <message variable> ')' |
  <source address>
 '(' <message variable> ')'
({ '^'<source address>
 '(' <message variable> ')' } |
 { ':'<source address>
 '(' <message variable> ')' }) )
```

The content of the message is stored in the message

variable. In addition to the content of the message, the sender information must also be sent to the destination process. A buffer is required to hold the message sent regardless of whether the corresponding 'messin' statement is executed or not. A message handler in the run-time support system also records the the address of the sending process. If a message is received, the name of the sender is recorded in the attribute 'sender'. The attribute(succ) of the message variable is also set to true. The attribute 'succ' is useful when there are more than one sending processes sending messages to the same receiving process. After the execution of the 'messin' statement is completed, the process can then continue its execution from the statement following the 'messin' statement.

For each message variable, it must be declared with a message constructor which is defined at the beginning of the program. The function of the message constructor is similar to a mail box. A message constructor is treated as a mail box where the same type of message are stored. Each message variable can receive message from its message constructor.

The 'messin' statement has two different forms. The first one is a 'or' 'messin' statement which can be further divided into two subtypes. The first subtype simply contains a message variable without any source

address. It means that the 'messin' statement will be completed only when a message has arrived in its message constructor regardless of the sender identification. The second subtype of the 'or' messin statement may specify one or more sources (sending process) with explicit source addresses The address may either be a constant or a variable, a new data type called 'address' is derived for this purpose. When one of the source processes has sent a message to the message constructor, the 'messin' statement is completed and the process can continue its execution. Otherwise, the process must wait until the above condition is satisfied. The waiting process will be inserted into the waiting for message queue. If more than one source process have sent messages to the receiving process before it executes the 'messin' statement, then the messages would be stored by the message handler in the order of their arrivals corresponding to the different message constructors. When the process executes the 'messin' statement, then the message with the longest waiting time is removed from the message constructor and placed into the message variable of the receiving process. The destination message variables of the 'or' 'messin' statement may be the same or not.

The second subtype of the 'messin' statement is the 'and' 'messin' statement. This statement specifies one or more sources with explicit source addresses, however,

this 'messin' statement is completed only when all the source processes have sent their messages.

3.8 The GDPL Language and Parallel Programming issues.

3.8.1 The Nature of Concurrency.

The design philosophy of GDPL is that it can cater for a large range of computer systems configuration ranging from a single processor to the multiprocessors with common or distributed store. It is also suitable for a wide range of application systems, from strictly tightly-coupled systems to loosely coupled systems. The node is the key structure that can achieve the above objective.

It is assumed that the programmer is the one who knows the most about the relationships between processes and the configuration of the distributed computing environment. The programmer should put those highly related processes into the same node. All the processes within the same node will eventually reside in the same physical processor. They are using the same resources of the physical processor. Since the processes of the same node are using the same common store, the most efficient way to establish communication and synchronization is the shared variable. Hence, the service in a node is constructed to provide shared variables and explicit primitives for synchronization. It is a good computing

environment for those tightly coupled application systems.

However, if multiple processors with distributed store is used, then each reference of a variable represents one copy of data from a memory store of one physical processor to another. This is very inefficient becauses many actual communications between the physical processors are required. GDPL's message passing mechanism is more suitable for implementation on multiple processors with distributed store, and is also suitable for those loosely coupled application systems. They involve distributed computing in their individual physical processor and will use the message passing facility only for a small number of times. Massage passing can be regarded as a generalized parameter passing mechanism in programming languages.

3.8.2 The Model of Computation.

In GDPL, the message passing mechanism can be regarded as a data flow mechanism. As long as a message is available, the process requesting for message can proceed again.

GDPL's utility concept is to achieve the communication and synchronization between processes in the same node in a control flow manner. In GDPL, the word "remote" of the remote procedure call is used for those processes

within the same node but not for those processes in different nodes. After the utility is terminated, the control is returned to the invoking process. Thus, GDPL contains both of the program features of the data flow and control flow mechanism.

3.8.3 Communication Mechanisms

GDPL includes the three commonly used communication mechanisms: direct shared variables, procedure or operation call and message passing [3]. They are present in GDPL in the form of shared objects within services, remote procedure call (utility call), and message passing ('messin' and 'messout' statements). GDPL provides the users with a wider range of tools to fulfill various types of communication requirements.

3.8.4 Synchronization Mechanisms

In GDPL, the service uses wait statements to achieve internal synchronization between utilities. Shared objects must be included in guards of wait statements. If only local variables of utility are included in the guard, the synchronization function of the wait statement is not made use of. The waiting for a true guard may be forever if the guard is not true for the first time. Within the same node, the utility call can be used to achieve synchronization between processes because no more than one process can execute the utility

of a service simultaneously. Message passing is the third mechanism to achieve synchronization. 'Messin' statement cannot proceed until all expected messages have been collected.

3.8.5 Process Creation and Termination

The process creation time in GDPL is statically determined. All the nodes are initialized at the same time when the program is started. At the same time, all the processes of the nodes are also activated. The services of a node are also initialized at the same time. Consequently, the storage allocation for all processes and services can be done only once.

Hence, the job of storage management is greatly simplified. Although the creation of utilities can not be determined, the storage requirement can be determined at compilation time. Storage may be allocated at compilation time or at run time.

Although the processes can terminate at different times, there is no need to worry about the storage allocation problem of the terminated processes. The storage of the terminated process is returned to the free storage pool and the terminated process will not make further requests for storage due to the non-reactivation characteristics of processes in GDPL. The node is terminated when all of its processes have terminated. Before the

node is terminated, it will terminate all services because the services are passive constructs and cannot terminate by themselves. The program is terminated when all the nodes are terminated. The clear creation and termination of the program constructs makes it easier for the program to be verified.

3.8.6 Access Control

In sequential programming languages, access control is an important issue. It should provide a clear access path and access right of each program construct in order to improve the readability, and ease of verification of the program. For parallel programming languages, the concurrency introduces nondeterministic execution paths. It makes the access control a more complicated problem. In GDPL, access control and access right of objects of each program construct is clearly defined both in sequential programming and concurrent programming.

(i) The scope of the data objects of each program construct in GDPL

For sequential programming, no global variables are allowed. Therefore, the parameter list of procedure/function is the unique means to pass data between processes and procedures/functions. A recursive procedure/function is also allowed because it is very useful in many application systems. No nesting or

blocking structure of procedure/function are provided. Although many usages of block structures such as the restricted visibility provided for local identifiers and efficient use of storage, etc. are lost, some of the more severe disadvantages [23] of block structures can be avoided. These disadvantages include poor readability, difficulties in separate compilation, up-level addressing implementation and interference. Furthermore, the implementation of non-nested constructs are easier than those nested constructs.

For concurrent programming, the major objects are shared variables and message variables. Although processes can invoke the utilities of the service, only the service and its utilities can directly access the shared variables. Processes can only obtain the values of shared variables through the parameter list of utilities. A process can access the message variables and its attributes. The attribute (succ) of the message variables is treated as a logical variable. However, the attribute (sender) of the message variable can only be used as an address constant. Message variables can be accessed as ordinary variables with the same data type as message variables. Of course, only the operations of that data type are allowed for manipulating the message variables. Other processes cannot directly access the message variables of a process, they can only obtain a copy of the values of the message variables.

(ii) The operators allowed to operate on the data objects of each program construct in GDPL.

In a service, all sequential statements are allowed to access the shared objects and the local variables of the service. But no wait statement is permitted to manipulate the shared objects in the service. In an utility, both the local variables of itself and shared variables of the service can be accessed by all sequential statements and the wait statement. However, an utility cannot access the shared variables of another service. The message variables can be accessed only within the process, procedure and function. All sequential statements, 'messin' and 'messout' statements are allowed to manipulate the message variables. However, processes cannot use any statements to access the message variables of another process.

(iii) The scheduling of access to data objects

In GDPL more than one process can enter the service, a scheduling scheme is required to control the access of the shared objects. At any time, only one process is active within the service. The other processes can be active only when the process leaves the service or goes to the sleeping state. Hence only one process is allowed to enter the critical region, but the meaning of the critical regions of Concurrent Pascal and GDPL is different. The critical regions of the service include

both the statement list following the guard and other statements within the utility.

GDPL's service is similar to Modula's interface module [15-17] except that there is no define-list and use-list, and no nesting of service definition is allowed. There are three possible states for the service: either all the processes within the service are sleeping, all the processes within the service is sleeping except one is active, or no process is residing within the service. There is no wake up statement in GDPL's service. The wake up of the sleeping processes within a service is handled by a central scheduler. The scheduler will select the next sleeping process only according to their waiting time. A sleeping process of GDPL is not waken up even though it has a satisfied guard. It is because the current active process cannot recognize whether the guard of other processes is satisfied or not. The guards are examined only when the sleeping process is waken up.

The execution of the processes in different node are executed simultaneously. They can access their data objects in their physical processors. No conflict will occur even when messages are passed because the destination process will be temporarily terminated in order to put the message in the message constructor. The execution will recover after the message passing is com-

pleted. The source process will also continue its execution.

The processes within the same node are dispatched by the central scheduler as well. Each process is assigned a fixed time of execution (time slice). When the process is active, it can access its data objects through its statement list or the shared objects of the service indirectly through the utility call. The control of the process will be passed to the central scheduler when the time slice is used up, the process is sleeping or the process is completed. Hence, the execution path in the process is clean and secure.

CHAPTER 4

Sample Programs

4.1 Introduction

14 sample programs of GDPL have been written and executed under the Zilog S8000 super micro-computer system. They are used to test the validity of the design decisions and implementation of the language. Comparisons can be made between GDPL and other parallel programming languages through these sample programs. Comments in the sample programs are placed inside braces {}.

Example 1: Semaphore

A general semaphore can be implemented as a service semaphore that contains two utilities swait and signal. They can be invoked within the node in which the service semaphore resides.

```
node main;
service semaphore;
common s : int;
utility swait();
begin
      wait s>0 -> s := s-1;
      wend;
end swait;
```

```
utility signal();

begin
      s := s+1;

end signal;

begin {the body of the service}
      s := 0;

end semaphore;

begin

{The utility of the semaphore can be called by:

     semaphore.swait() or
     semaphore.signal() }

process dummy;

begin {the statement list is optional}
    skip;

end dummy;

end main;
```

Example 2: Message buffer

The service message_buf provides a ring buffer and two operations: send and receive for its users(processes) such that the users can communicate with sending and receiving character through the service.

```
node main;

service message_buf;

const
   buffersize = 1000; common
   head,tail,avail : int;
  {head points to location in which character is placed}
  {tail points to location in which character
   is received}
  {avail indicates number of available location}
   buffer : array[buffersize] of char;
  {the data structure used can be
   changed transparent to the users}
```

```
utility send(c:char);

begin
   wait avail>0 -> avail := avail - 1;
                   buffer[head] := c;
                   head := head + 1;
                   if head == buffersize ->
                      head := 0;
                   fi;
   wend;

end send;

utility receive(c:char);

begin
   wait avail<buffersize ->
        avail := avail + 1;
        c := buffer[tail];
        tail := tail + 1;
        if tail == buffersize ->
           tail := 0;
        fi;
   wend;

end receive;

begin
   head := 0; {all integer variables are initialized  to
0}
   tail := 0; {hence these two statements are optional}
   avail := buffersize;

end message_buf;

begin

process dummy;

begin
      skip;

end dummy;

end main;
```

Example 3: Resource Scheduler

The service resource_sch can schedule a resource (a fair scheduling scheme is not guaranteed) for its

users(processes) through the two utility calls: request and release.

```
node main;

service resource_sch;

common
    rfree : bool;

utility request();

begin
      wait rfree -> rfree := false;
      wend;

end request;

utility release();

begin
      wait not(rfree) -> rfree := true;
      wend;

end release;

begin
      rfree := true;

end resource_sch;

begin

process dummy;

begin
   skip;

end dummy;

end main;
```

Example 4: Shortest job next scheduler

The service shortest_job_next schedules a resource among n user processes in shortest_job_next order. It consists

of three utilities: request, release and driver. The utility request enters the service_time of the user in its reserved location indexed by the user-supplied identification id. Then the user process waits until it is selected by the scheduler. The utility release returns the resource to the scheduler.

The utility driver is transparent to the user processes. It must be called once by a specific process, then it continues to wait until there are further user processes waiting for the resource. It then selects a user process with the shortest job next algorithm. If the resource is available, then it is allocated to the selected user process. Otherwise, it continues to wait until the resource is available.

```
node main;

service SJN_sch;

const
   nil = minint;
  {nil denotes an undefined process index}
   n = 10; {let n = 10}

common
   user : int;
  {user denotes the current user index}
   no_of_waiting :int;
  {no_of_waiting denotes the number of user
   process waiting within the service}
   queue : array[n] of int;
   queue : array[n] of int;
  {queue is an array to record the
   service time of the user process}

utility request(id,service_time:int);

begin
   no_of_waiting := no_of_waiting + 1;
```

```
   queue[id] := service_time;
   wait user == id -> skip;
   wend;

end request;

utility release();

begin
   user := nil;

end release;

utility driver();

var
   i, min, next : int;

begin
   loop true ->
        wait no_of_waiting > 0 ->
             min := maxint;
             i := 0;
             next := 0;
             loop i < n -> i := i+1;
                  if queue[i] >= min -> skip;
                  [] queue[i] < min ->
                     next := i;
                     min := queue[i];
                  fi;
             lend;
             wait user == nil ->
                  user := next;
                  no_of_waiting := no_of_waiting-1;
             wend;
        wend;
   lend;

end driver;

begin {body of the service}
   user := nil;

end SJN_sch;

begin

process dummy;

begin
   skip;

end dummy;
```

```
end main;
```

Example 5: Readers and Writers

There are two families of processes, called readers and writers, sharing a single resource. The family of processes reader can use the resource simultaneously. However, each member of the family of processes writer must have exclusive access to it. The service resource is used to control the access of the shared resource among the two families. A common variable s defines the current state of the shared resource.

s = 0	1 writer is using the shared resource
s = 1	0 processes is using the shared resource
s = n	n-1 readers are using the resource, where n>=2

```
node main;
service resource;
common s : int;
utility startread();
begin
      wait s>=1 -> s := s+1;
      wend;
end startread;
utility endread();
begin
      if s>1 -> s := s-1;
      fi;
end endread;
utility startwrite();
begin
```

```
      wait s == 1 -> s := 0;
      wend;

end startwrite;

utility endwrite();

begin
      if s == 0 -> s := 1;
      fi;

end endwrite;

begin
      s := 1;

end resource;

{The readers and writers should use the shared  resource
in the following way:

     startread();     the qualifying name  "resource"  is
not
     read();          used because the service resource
     endread();       and the two families are residing
                      within the same node

     startwrite();
     write();
     endwrite;

}

begin

process dummy;

begin
     skip;

end dummy;

end main;
```

Example 6: Dining Philosophers

There are n philosophers alternating between thinking and eating. When a philosopher gets hungry, he joins a table and picks up two forks next to his plate and

starts eating. There are, however, only n forks on the table. So a philosopher can eat only when none of his neighbors are eating. When a philosopher has finished eating he puts down his two forks and leaves the table again.

In order to prevent two philosophers from starving a philosopher between them to death by eating alternately, a fair scheduling scheme is required (in this case, the round robin). An array of status is required to record the status of each philosopher. When the entry is equal to Ø, the philosopher is thinking. When the entry is equal to -1, the philosopher is eating. The philosopher waiting to join the table is inserted into the stack index in a fair scheme (i.e. the neighbor(s) of the index in the stack index is (are) corresponding to the nearest neighbor(s) of the philosopher with this index).

The utility driver is responsible for examining whether there is any philosopher who can join the table. If it is so, then the status of the satisfied philosopher is set to -1. The utility driver continues to wait until there is any philosopher calling the utility join or release. The utility driver must be called by a specific process (forever) only once and it continues to exist forever.

```
node main;
service table;
```

```
const
   no_of_philosopher = 5;
   index_size = 6;
   nil = maxint; common
   status : array[no_of_philosopher] of int;
   index  : array[index_size] of int;
   no_of_waiting,no_of_eating :int;
   max_no_of_eating :int; var
   i : int;
   pointer : int;

utility join(i:int);

begin
   no_of_waiting := no_of_waiting + 1;
   table.insert(i);
   wait status[i] == -1 -> skip;
   wend;

end join;

utility leave(i:int);

begin
   status[i] := 0;
   no_of_eating := no_of_eating - 1;

end leave;

utility insert(i:int);

var
  j,templ:int;

begin
  if index[0] == nil -> index[0] := i;
  else j := 0;
       if ((index[j] < i) and (i < index[j+1]))
       or ((index[j] < i) and (index[j+1] < i)) ->
          templ := index[j+1];
          index[j+1] := i;
          loop templ <> nil ->
               i := templ;
               j := j+1;
               templ := index[j+1];
               index[j+1] := i;
          lend;
       fi;
   fi;

end insert;

utility delete(i:int);
```

```
begin
   loop index[i+1] <> nil ->
        index[i] := index[i+1];
        i := i + 1;
   lend;
   index[i] := nil;

end delete;

utility driver();

var i,j,k,k1,temp1,temp2 : int;

begin
   wait no_of_waiting > 0 ->
        i := 0;
        temp1 := no_of_eating;
        loop (index[i] <> nil) and
             (no_of_eating < max_no_of_eating)  ->  {nil
denotes that the end of the stack is encountered}
             j := index[i];  {j  is  the  index  of  the
selected philosopher}
             k := j+no_of_philosopher-1;
             temp2 := k/no_of_philosopher;
             k := k - temp2*no_of_philosopher;
             k1 := j+1;
             temp2 := k1/no_of_philosopher;
             k1 := k1 - temp2*no_of_philosopher;
             if (status[k] <> -1) and
                (status[k1] <> -1) ->
                status[j] := -1;
                table.delete (i);
                no_of_eating := no_of_eating+1;
                no_of_waiting := no_of_waiting-1;
             else  i := i+1;
             fi;
        lend;
   wend;

   wait (temp1 <> no_of_eating) ->
        skip;
   wend;

end driver;

begin {body of the service}

   max_no_of_eating := (no_of_philosopher-1)/2;
   no_of_waiting := 0;
   no_of_eating := 0;
   pointer := 0;
   i := 0;
   loop i < index_size ->
        index[i] := nil;
```

```
        i := i + 1;
   lend;

end table;

begin

process philosopher[5]; {there are n  philosophers  with
index:
 0,1,2,...,n-1}

procedure think(time:int);

var
   i : int;

begin
     loop i < time*10 ->
          writeln (i);
          i := i + 1;
     lend;
     skip;

end think;

procedure eat(time:int);

var
   j : int;

begin
     loop j < time*5 ->
          writeln (j);
          j := j + 1;
     lend;
     skip;

end eat;

begin
     loop true -> think(me);
           table.join(me);
     {me is the attribute of the family
      of the processes, which contains
      the index of the current process}
           eat(me);
           table.leave(me);
     lend;

end philosopher;

process forever;

begin
```

```
    loop true ->
         sort.driver();
    lend;
end forever;
end main;
```

Example 7: Sorting array

A sorting array with n members can sort n elements or less. A sorting service consists of two utilities: put and get. The items are input through sort service Ø that stores the smallest item input so far and passes the rest to its successor sort service 1. The latter keeps the second smallest item and passes the rest to its successor sort service 2, and so on.

When the m items have been input, they will be stored in an ascending order in sort services Ø,1,...,m, where m <= n. The input item is input in a member of the array of services sort through the utility put, provided that only Ø or 1 item is in the current service. Then the ordered items are obtained through sort service Ø by the utility get, provided that there is exactly one item in the current service.

In each sort service, two locations are required to store the items. This implies that at most two items can reside in one sort service. Each utility driver of the service array sort must be called by a specific process (forever) such that they can continue their work for-

```
over.


node main;

service sort[7]; {an array of sort service with
                  7 members is defined}

common
        length : int;
        two    : array[2] of int;
        succl, next : int;

utility put(c:int);

begin
        wait length < 2 -> two[length] := c;
                           length := length+1;
        wend;

end put;

utility get(var c:int); {c is passed by reference}

begin
        wait length == 1 -> c := two[0];
                            length := 0;
        wend;

end get;

utility driver();

var
        temp : int;

begin
        wait (length == 2) or
             ((length == 0) and (next>0)) ->
             if (length == 2) ->
                if two[0]<=two[1] ->
                   temp := two[1];
                [] two[0]>two[1] ->
                   temp := two[0];
                   two[0] := two[1];
                fi;
                length := length-1;
                sort[succl].put(temp);
                next := next+1;
             [] (length == 0) and (next>0) ->
                sort[succl].get(temp);
                two[0] := temp;
                next := next-1;
```

```
                length := 1;
            fi;
        wend;

end driver;

begin {body of the service}
      succl := me + 1;
      length := 0;
      next := 0;

end sort;


{A user process should use the sorting array to sort an
array of n items in the following way:}

begin

process user;

var
     i,x : int;
     item : array[7] of int; {there are m = 7 items to
be sorted, where m<=n}

begin
       i :=0;
       loop i<7 -> sort[0].put(item[i]);
            i := i + 1;
       lend;
       i := 0;
       loop i<7 -> sort[0].get(x);
            item[i] := x;
            i := i + 1;
       lend;

end user;

process forever[7];

begin
       loop true ->
            sort[me].driver();
       lend;

end forever;

end main;
```

Example 8: Vending Machine

A vending machine is simulated by a service

vend_machine. It accepts one coin at a time and accumulates the value of the inserted coins. Assumes that there are m kinds of items and there are m buttons corresponding to the m kinds of goods.

When a button is pushed, the machine returns an item with change provided that there is at least one item left and the inserted coins cover the cost of the item. Otherwise, all the inserted coins are returned.

```
node main;

service vend_machine;

const n = 10; {let n = 10}

type item = record
               no_of_item : int;
               price      : int;
            end; common
       paid, cash : int;
       stock : array[n] of item;
       {The array is initialized in other place}

utility insert(coin:int);

begin
      paid := paid + coin;
      {accumulates the value of the inserted coins}

end insert;

utility push(var change,goods : int; item_no : int);

begin
     if (stock[item_no].no_of_item > 0) and
        (paid >= stock[item_no].price) ->
        change := paid-stock[item_no].price;
        cash := cash + stock[item_no].price;
        goods := 1;
        stock[item_no].no_of_item :=
        stock[item_no].no_of_item-1;
        paid := 0;
     [] (stock[item_no].no_of_item == 0) or
        (paid < stock[item_no].price) ->
```

```
        change := paid;
        goods := 0;
        paid := 0;
     fi;

end push;

begin
      paid := 0;
      cash := 0;

end vend_machine;

begin

process dummy;

begin
     skip;

end dummy;

end main;
```

Example 9: Coroutine

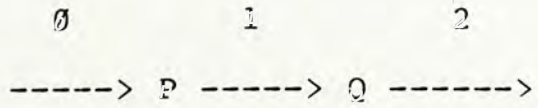
A group of processes can function as coroutines through the simulation of the resume statement. The simulation is accomplished through the message passing mechanism. Assume that process P is one of the processes which function as coroutines and it wants to resume the other process called Q at some place. After the resume message is accepted by Q, then the control of operation is transferred from P to Q. Then process P should wait until the another process resumes it.

```
message resume_slot=int;

node coroutines; {Assume this node contains all the
 processes that function as coroutines}

begin
```

```
process P;

var resume : resume_slot;

begin
      messin (resume);
      skip;
      messout Q(resume);
      messin (resume);
      skip;

end P;

process Q;

var resume : resume_slot;

begin
      messin (resume);
      skip;
      messout R(resume); {resume process R in  the  same
node}
      messin (resume);
      skip;

end Q;

{...}

end coroutines;
```

Example 10: Path expressions

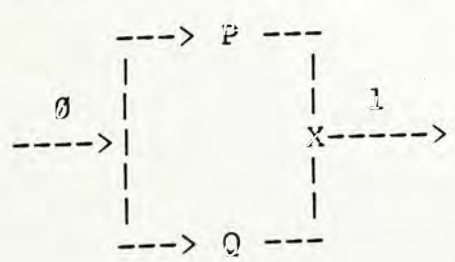
Path expressions define a meaningful sequences of operations. These operations are implemented as utilities and the control sequences of operations is enforced by the wait statement. There are a number of possible paths. The utility P can only be followed by utility Q as shown below:



One possible implementation is through a shared common variable(control), which may contain Ø, 1 or 2 to indicate three different states.

```
node main;

service path1;

common control : int;

utility P();

begin
      {the operations performed by P}
      wait control == Ø -> skip;
      wend;
      control := 1; {control is set to 1 at this place
only}

end P;

utility Q();

begin
      {the operations performed by Q}
      wait control == 1 -> skip;
      wend;
      control := 2; {control is set to 2 at this place
only}

end Q;

begin

end path1;
```

{If utility P is called, it can be executed only when control is equal to Ø. Before P is completed, it sets the control to 1. If utility Q is called, it can be executed only when control is equal to 1. Another path expression is that either utility P or Q can be performed as shown below:



```
It can be implemented similar to the path expression 1.}

service path2;

common control : int;

utility P();

begin
      {the operations performed by P}
      wait control == Ø ->
           skip;
           control := 1; {the paths through P and Q
 are mutually exclusive}
      wend;

end P;

utility Q();

begin
      {the operations performed by Q}
      wait control == Ø ->
           skip;
           control := 1;
      wend;

end Q;


{The other possibility of the path expression is that an
utility  P  can  be  performed Ø or more times. Then the
entry state of the utility P should remain unchanged  as
shown below:

           ---------->
                    |
           Ø |      | Ø
             -- P <-  LP  It  is  implemented  as  shown
below:}

begin

end path2;

service path3;

common control : int;

utility P();

begin
       {the operations performed by P}
       wait control == Ø -> skip;
```

```
        wend;
end P;
begin
end path3;
begin
process dummy;
begin
     skip;
end dummy;
end main;
```

Example 11: Fibonacci

A process fibonacci is responsible for providing a sequence of fibonacci numbers to a user process P. The process fibonacci and P may or may not lie within the same node. They communicate with each other through message passing. It is assumed that there is an overflow handler to provide certain diagnostic actions when an overflow occurs. It is the responsibility of the process fibonacci to send the address (node name and process name) of the process P to the overflow handler.

There are three message names called signal, object, and destination. The message name signal is used for signaling the process fibonacci that the process P requests the next fibonacci number. The object consists of the next fibonacci number and a status. The status is used for indicating the validity of the operation. If status

is equal to 0, then the process fibonacci can generate a valid fibonacci number. Otherwise, an overflow has occured.

The message definition list is listed below:

```
message signal = int; {the type of signal can be of  any
valid
 data instead of int}
        object = record
                   fibno : int;
                   status : int;
                 end;
        destination = address;

node main;

{ The process fibonacci is listed below:}

begin

process fibonacci;

const limit = maxint; {limit is the greatest integer  on
the current computer system}

var
      request : signal;
      fib     : object;
      complainer : destination;
      this, last, previous : int;

begin
      last := 0;
      this := 1;
      loop true ->
           messin (request);
           previous := last;
           last := this;
           if limit - last > previous ->
              this := last + previous;
              fib.fibno := this;
              fib.status := 0;
              messout request'sender (fib);
           [] limit - last <= previous ->
              complainer := request'sender;
              messout overflow_handler (complainer);
           fi;
       lend;
```

```
end fibonacci;

{ The user process P is listed below:}

process P;

var
     request : signal;
     fib     : object;
     next_fibno : int;

begin
       skip;
       messout fibonacci (request);
       skip;
       messin (fib); {the source address is  not  speci-
fied
 because the message may be sent by
 either the process fibonacci or the
 overflow_handler}
       if fib.status == 0 ->
          next_fibno := fib.fibno;
       else skip;
       fi;
       skip;

end P;

end main;
```

Example 12: Stacks

An array of service stacks are used to construct n stack data objects with two operations: push and pop.

```
node main;

service stacks[5]; {5 is the number of the members of
 the service array stacks} const stacksize = 1000;

common
        stack : array [stacksize] of real;
        top   : int;

utility push (object: real);

begin
       wait top < stacksize - 1 ->
            top := top + 1;
            stack[top] := object;
```

```
        wend;
end push;
utility pop (var object : real);
begin
        wait top >= 0 -> object := stack[top];
                         top := top - 1;
        wend;
end pop;
begin
        top := -1; {initialize the stack pointer top}
end stacks;
begin
process dummy;
begin
     skip;
end dummy;
end main;
```

Example 13: Prime number generator

This example is used to illustrate the use of an array of processes. It uses pipeline communication and acknowledgement to implement a parallel version of the sieve of Eratosthenes. Assume that there are two utilities called: getint and printint within a service of the node in which the init process and an array of processes prime reside. Getint is used to get an integer called limit such that prime numbers equal to or less than the square of limit are printed out by the utility printint.

The size of the array of processes prime determine the

number of prime numbers that it can generate. It can generate prime numbers less than m, where m is equal to the square of the (prime_size+1)th prime number.

```
message
        number = int;
        signal = int;

node prime_no_gen;

service input_output;

common dummy : int;

utility getint (var x : int);

begin
        write ("Please input the limit : ");
        readln (x);
        writeln ();

end getint;

utility printint (x : int);

begin
        write ("The prime number is : ");
        writeln (x);

end printint;

begin

end input_output;

begin

process init;

var
     object : number;
        ack : signal;
    n,limit : int;

begin
        input_output.printint (2);
        n := 3;
        input_output.getint (limit);
        loop n <= limit ->
           object := n;
           messout "prime_no_gen.prime"[0] (object);
```

```
          messin "prime_no_gen.prime"[0] (ack);
          {it makes sure that the process
           prime[0] has received the object,
           then it sends the next object}
          n := n + 2;
        lend;

end init;

process prime[19]; {prime_size = 19 defines  the  number
of the
 member of the process array as well as the
 number of prime numbers that it can generate}

var
     object : number;
     ack : signal;
     p, mp, n : int; {p is a prime; mp is a multiple  of
p}

procedure receive_object(var n : int);

var
       object : number;
       ack : signal;

begin
       messin (object);
       n := object;
       messout object'sender(ack);

end receive_object;

begin {main program of prime}

       receive_object (n);
       p := n;
       mp := n;
       input_output.printint (p);
       loop me <  19 - 1 ->
            receive_object (n);
            loop n > mp -> mp := mp + p;
            lend;
            if n == mp -> skip;
            [] n < mp ->
               object := n;
               messout
"prime_no_gen.prime"[me+1](object);
            {send to the next prime number}
               messin "prime_no_gen.prime"[me+1](ack);
            fi;
       []   me == 19 - 1 ->
            receive_object (mp);
            if mp < p*p -> input_output.printint (mp);
```

```
            [] mp >= p*p -> skip;
            fi;
        lend;
end prime;
end prime_no_gen;
```

Example 14: This example is to emulate the data flow environment

This program contains two nodes called Main and Subsidiary. In node Main, there are a service called input_output and two processes called sumsqroot and fourthroot. In the service input_output, there are two utilities called input_stream and output_stream. In node Subsidiary, there are two processes called sqroot[Ø] and sqroot[1]. The processes sqroot[Ø] and [1] both contain a procedure called sqroot_proc.

For the process sumsqroot, it uses the utility input_stream to obtain two input numbers. These two input numbers should be non-negative integers. If one or both of them is negative, then this process would be terminated. Before it is terminated, two messages will be sent to the two processes sqroot[Ø] and sqroot[1] to notify them of its termination. Otherwise, it requests the process sqroot[Ø] and sqroot[1] of the node Subsidiary to evaluate the square root of the two input numbers and return the results to the process sumsqroot through messages. Then the two square roots are added up and the final result is output through the utility

output_stream.

For the process fourthroot, it uses the facility input_stream to obtain an input number. Initially, it checks to see if the process should be terminated or not. If it should be terminated, two messages are sent to the two processes sqroot to notify them of its termination. If the input number is non-negative, then it requests the process sqroot[Ø] to evaluate and return the square root of the input number. Then it requests the process sqroot[1] with the same message to evaluate and return the square root of the square root of the input number. The fourth root is output through the utility output_stream.

In the node Subsidiary, a range is used to define a family of processes called sqroot. This family has two processes called sqroot[Ø] and sqroot[1]. They are both ready for providing services to the process sumsqroot and fourthroot. Only one of the processes sumsqroot and fourthroot is served at any time. When a message is received, it should check whether the termination condition occurs or not. The process sqroot should be terminated if both process sumsqroot and fourthroot are terminated, otherwise the procedure sqroot_proc is called to evaluate the square root of the component x or the component sqx (square root of x) according to whether the component x is equal to maxint or not. Max-

int is a predefined constant equal to the largest integer available on the corresponding physical processor.

After the process sumsqroot and fourthroot have terminated, then the node Main is terminated. After the two processes sqroot[0] and sqroot[1] have terminated, then the node Subsidiary is terminated. If both the node Main and Subsidiary have terminated, then the program is terminated.

```
message
       mess_slot = record
                     x : int;
                     sqx : real;
                   end;

node Main;

service input_output;

common
     shared_object1 : int;
     shared_object2 : char;

var
     continue       : bool;

utility input_stream (var x1,x2 : int);

begin
     read (x1);
     if x2 == maxint ->
        read (x2);
     fi;
     continue := true;

end input_stream;

begin
     continue := false;

end input_output;

begin
```

```
process sumsqroot;

var m1,m2 : mess_slot;
    y : real;

begin
    m2.x := maxint;
    input_output.input_stream
        (m1.x,m2.x);
    loop (m1.x >= 0 and m2.x >=0) ->
         messout "Subsidiary.sqroot"[0]
                 (m1);
         messout "Subsidiary.sqroot"[1]
                 (m2);
         messin  "Subsidiary.sqroot"[0]
                 (m1)
               ^ "Subsidiary.sqroot"[1]
                 (m2);
         y := m1.sqx + m2.sqx;
         input_output.output_stream
             (y);
         m2.x := maxint;
         input_output.input_stream
             (m1.x,m2.x);
    lend;
    if m1.x < 0 ->
       messout "Subsidiary.sqroot"[0]
               (m1);
       messout "Subsidiary.sqroot"[1]
               (m1);
    [] m1.x >= 0 ->
       messout "Subsidiary.sqroot"[0]
               (m2);
       messout "Subsidiary.sqroot"[1]
               (m2);
    fi;

end sumsqroot;

process fourthroot;

var m1 : mess_slot;

begin
      input_output.input_stream
          (m1.x,0);
      loop m1.x >= 0 ->
           messout "Subsidiary.sqroot"[0]
                   (m1);
           messin  "Subsidiary.sqroot"[0]
                   (m1);
           m1.x := maxint;
           messout "Subsidiary.sqroot"[1]
                   (m1);
```

```
            messin  "Subsidiary.sqroot"[1]
                    (m1);
            input_output.output_stream
                    (m1.sqx);
            input_output.input_stream
                    (m1.x,Ø);
        lend;
        messout "Subsidiary.sqroot"[Ø]
                (m1);
        messout "Subsidiary.sqroot"[1]
                (m1);

end fourthroot;

end Main;

node Subsidiary;

begin

process sqroot [2];

var m1,m2 : mess_slot;
    flag1,flag2 : bool;

procedure sqroot_proc (var m1 : mess_slot);

begin
    if m1.x == maxint ->
       m1.sqx := m1.sqx {*}* Ø.5;
    [] m1.x <> maxint ->
       m1.sqx := m1.x {*}* Ø.5;
    fi;

end sqroot_proc;

begin
    flag1 := true;
    flag2 := true;
    messin "Main.sumsqroot"
           (m1)
         : "Main.fourthroot"
           (m2);
    loop (flag1  or flag2) ->
         if m1'succ ->
            if m1.x >= Ø ->
               sqroot_proc (m1);
               messout "Main.sumsqroot"
                       (m1);
            [] m1.x < Ø ->
               flag1 := false;
            fi;
         [] not (m1'succ) ->
            if m2.x >= Ø ->
```

```
                sqroot_proc (m2);
                messout "Main.fourthroot"
                        (m2);
            [] m2.x < Ø ->
                flag2 := false;
            fi;
        fi;
        messin "Main.sumsqroot"
                (m1)
             : "Main.fourthroot"
                (m2);
    lend;

end sqroot;

end Subsidiary;
```

CHAPTER 5

The Implementation of GDPL

5.1 Introduction

GDPL has been implemented on the Zilog S8000, which is a microcomputer system using an enhanced version of the UNIX operating system. The 'C' programming language is used to develop the compiling system of GDPL. The code generated by this compiling system is supported by a run-time support system, which is also written in 'C'. The run-time support system should be loaded and executed on the Zilog S8000 as well. The compiling system is a one-pass recursive descent compiler. It consists of three major components: a parser, a code generator and a linker. The function of the parser is to generate tokens and check the validity of the syntax and semantics of the GDPL source program. The code generator generates 'C' code for the run-time support system in two stages. The first stage is before the linkage time. The codings of all sequential statements are generated. Since the configuration of the distributed computing system is not known before linkage time, only part of the codings of concurrent statements are generated. Then the second stage is after the linkage time. The

remaining codings of the concurrent statements and the control programs of the run-time support system are formed.

The run-time support system consists of six types of control programs: manager control, main control, node control, service control, utility control and process control. Each control program is a complete 'C' program with certain responsibility. Since the service is optional in a GDPL program, a run-time support system may not contain the service and utility control. However, manager control, main control, node control and process control must reside in the run-time support system. The compiling system and the run-time support system are described in more detail in the following sections.

5.2 Run-time Support System

5.2.1 Manager Control

The run-time support system can be viewed as a hierarchical system. There is only one manager control in first level of a run-time support system. The main purpose of manager control is to control the execution of the whole system and work as a common communication station among the main controls. When a main control wants to send a message to another main control, it should store the content of the message into a common message buffer and

then send the address of the destination main control to the manager control. Then the manager control informs the destination main control that a message have been sent. The destination main control will eventually remove the message from the buffer.

A manager control contains the information of the main controls, some facility routines, an initialization routine, a manager/main routine and a manager control body. The information in the manager control include the initial number of main controls, their identifications, the status and the current number of active main controls. The status of each main control shows whether it is terminated or not. If all the main controls are terminated, the manager control should be terminated and the run-time system is completed.

The facilities routines are used to put and get values from manager/main channel and argument list. Channels use the pipe in UNIX which are the communication paths among a number of UNIX's processes. The argument list of the manager control passes the manger/main channel number and identification number of manager control to each main control.

Initially the execution of the run-time support system is started by executing the manager control. Then the manager control executes the initialization routine to create the main controls (use fork in UNIX) and replaces

the newly created UNIX's processes (use execl in UNIX) with the load modules of main controls. Then the main controls are activated. The manager/main routine is activated when a UNIX signal (use kill in UNIX) is passed from a main control. The signals used in any two types of control programs must be different, otherwise the proper control would be lost. The manager/main routine is also used to receive and send the information of the message but not the content of the message. The information include the source address, destination address and the identification number of the message.

The body of the manager control contains the initialization routine and an infinite loop of control statements. After the initialization routine is executed, it enters an infinite loop until all main controls are completed or aborted if some abnormal events occurs in the run-time support system.

5.2.2 Main Control

Main control is at the second level of the run-time support system. There may be one or more main control in the run-time support system with each main control residing in one physical processor. The major function of each main control is to handle the execution switching, message passing and storage allocation within a physical processor. However, all main controls are mapped onto the Zilog S8000 and executed as if a

distributed computing configuration is available.

Each main control contains the information of the main control, some facility routines, a priority list of process controls, manager control identification, main control identification, an initialization routine, process control routines, a manager/main routine, a main/node routine and a main control body. The information of main controls contain the initial number of node controls, their identifications, the status and the current number of active node controls. The status of each node control indicates whether it is terminated or not. If all node controls of the main control are terminated, the main control should be terminated.

Main control controls the execution of its node controls. It also indirectly controls the execution of service controls, utility controls and process controls through node controls. A priority list of process controls is maintained to achieve this purpose. Actually the service controls, utility controls and process controls are 1-1 mapping to the service, utilities and processes of the GDPL source program.

In general, the process with highest priority in the priority list is selected. If there are more than one processes with the same highest priority, the round-robin scheduling scheme is applied. However, there must be a chance for waking up those sleeping processes. The

sleeping processes include those waiting for messages and waiting for a true guard within the utility. There may be a number of strategies to wake up the sleeping processes. In order to simplify the implementation, a simple wake up scheme is selected. At each alternate selection of the next active process control, the sleeping processes pool is examined. If there is a sleeping process, then the first sleeping process in the pool is waken up. The selection of the next sleeping process is also in round-robin, but does not depend on the priorities of the sleeping processes. When all processes come to sleep, selection of the next active process control is taken from the sleeping process pool.

The initialization routine creates the node controls and replaces the newly created UNIX's processes with the load modules of the node controls. The main node channel number and main control identification are passed to node control. Then node controls become active. However, the node controls are active until all initialization of service, utilities and processes control programs are completed. Node controls will then report to the main control and goes to sleep (the sleep system call in UNIX). The node control will be active again only when one of its controls is activated. The manager/main routine in the main control is different from the one in the manager control. It is activated when a UNIX signal is sent from the manager control. The functions of the

manager/main routine include receiving message information, receiving acknowledgement of the destination main control, and receiving the termination command from the manager. The acknowledgement of the destination main control means that the message is successfully received. If the termination command is received from the manager control, the main control should terminate itself.

The main/node routine is activated when a UNIX signal is sent from the node control. The main/node routine should handle receiving acknowledgement of message, receiving the request of sending a message, inserting the current active process into the priority list again according to the completion of the initialization of node controls, completion of the current active process and completion of the node control.

The body of the main control consists of the initialization routine and an infinite loop of control statements. After the initialization routine is completed, the infinite loop is to control the execution of all node controls until all node controls are completed or aborted if some abnormal events occur.

5.2.3 Node Control

Node control is at the third level of the run-time support system. Every node control must be under the control of a main control. Node control is a collection of

control statements corresponding to a node of the GDPL source program. Each node control is used to control the execution of service controls, utility controls and process controls and reports to the main control. It also handles message passing and utility call scheduling.

Each node control contains the information of node control, some facilities routines, a process control, a status table, a service/process list, a main/node routine, a node/service routine, a node/process routine, an initialization routine and a node control body. The information of node control include number of service controls, number of service strings, initial number of processes, current number of active process, service index list, process index list, main control identification number, node control identification, service control identification and process control identification. The number of service controls may not be equal to the number of service strings because an array of services is allowed. The service and process index list are used to record the number of members in the family of services and processes respectively.

The process status table records the status of all process controls of the node control. The status may be sleeping, waiting for execution and being terminated. When process control returns to the node control, it is then time to update the status of the process control

and report to the main control. The service/process list consists of the execution paths within a service control of all process controls, if any. The execution path within a service is the calling sequence of utilities within the service by the process. After a utility is called by a process, it may call itself or another utility of the service. Hence, it is necessary to record the calling sequence of utilities such that the control will be properly returned to the calling one after the called utility is completed.

The main/node routine in node control is not the same as the one in main control. It is activated when a UNIX signal is sent from the main control. The main/node handles the receiving of the message information, reactivating the process control and terminating itself. When the node control receives the message information from the main control, it should pass this information to the destination process. Before reactivating the process control, the status of the process control should be checked. If the process control is already terminated, then the node control should be terminated abnormally. Before termination of the node control, it should terminate all nonterminated process controls and service controls. If the process control is sleeping for a true guard of the utility, the service/process list should be examined to find out the utility control in which the process control is sleeping. Then the service

control where the utility control is under its control is activated.

In that case, the utility call will be eventually activated but not the process control. The reason is that the control of process control is passed to that utility now. If the calling sequence of utility control is completed, the control will be returned to the process control again. The third type of status of the process control is waiting for execution. The node control will activate the process control. If the termination command is received from the main control, the node control will terminate all service controls and process controls and itself.

The node/service routine is invoked when the UNIX's signals are sent from the service control. It handles the exceptions of utility control, utility control calling the temporary completion of utility control and completion of utility control. The exceptions include computation errors, invalid utility call, utility parameters passing problems and invalid UNIX signals passing. When exception occurs, the node control will terminate itself. When a utility control call signal is accepted, it means that the current active utility control issues a utility call. The node control should update the service/process list by adding the new utility control. Then it activates the service control in which the new

utility control resides.

When the utility control is temporarily completed, the utility control should be sleeping for a true guard. Then the process control goes to the sleeping state again. If the utility control is completed, then the node control should check whether the calling sequence of utility control is completed. If the calling sequence has not been completed, the node control will activate the next utility control which has called this completed utility control. Otherwise, the process control will go out of the sleeping state.

The node/process routine is invoked when the UNIX's signal is sent from the process control. It will handle message passing, waiting for messages, utility control calling, temporary completion of process control and completion of process control. The node control can send and receive message information to/from process control and main control respectively. If the process control cannot receive all required messages, it should go to the sleeping state (waiting for messages). The status of the process control will be updated. When the process control issues a utility control call, node control should activate the utility control through its service control. If the temporary completion signal is received, the time slice for the process control must be used up. The status of the process control should be

waiting for execution. When the process control is completed, the node control should check whether all process controls have completed. If all process controls have been completed, the node control should be terminated.

The initialization routine of the node control will create a number of UNIX's processes and replace them by the load modules of service controls and process controls. The node control should initialize all service controls and process controls, and then is forced to sleep and report to the main control. The node control is activated when the process or utility control is activated.

The node control body contains the initialization routine and an infinite loop of control statements. After the initialization is completed, it enters an infinite loop to control the execution of service controls and process controls until all processes are completed or abnormal events occur. The node control should report the status of the process controls to the main control as well as their priority values because the process control may change their priority levels through the keyword

5.2.4 Service Control

The service control is under the control of the node

control program and is at the fourth level of the run-time support system. The service control includes the information of service control, a list of shared variables, a node/service routine, a service/utility routine, an initialization routine and a service control body. The information contains the number of utility controls and service control index, the service control identification number and node control identification number. The service control index shows the membership within the family of service controls. The list of shared variables will be passed to/from utility control.

The node/service routine is invoked when a UNIX's signal is sent from the node control. It can handle the termination of itself, initialization of utility control, activation of utility control and completion of utility control. When the completion of utility control is received, it acknowledges the utility control which has called the completed utility control. The service/utility routine is invoked when a UNIX's signal is sent from the utility control. Its functions include handling utility control call and temporary completion of utility control. Before service control terminates itself, it should terminate all nonterminated utility controls.

The initialization routine of the service control obtains all arguments passed by the node control and

executes the initialization statement list (service body) of the service control. After the statement list is completed, the service control is forced to sleep and reports to the node control. It is activated when a utility control is activated. The service control body contains the initialization routine and an infinite loop to control the execution of utility controls. Since service control is passive in nature, it is activated always by the invocation of its utility control by the process control or utility control.The service control will be terminated when the termination command is sent from the node control or failure occurs.

5.2.5 Utility Control

A utility control is used to encapsulate the codings generated for a utility. In order to be under the control of the service control, some facility routines and control routines are also included. Utility control is at the lowest level of the run-time support system. It contains the number of service strings, a list of shared variables of its service control, a service/utility and a service/process channel, a utility request routine, service/utility routine, an initialization routine, a utility normal termination routine, a utility abnormal termination routine, a utility temporary termination routine and a utility control body.

Every time when the utility control is activated,

temporarily terminated or terminated, the list of shared variables must be passed from/to service control. The service control can maintain an up-to-date list of shared variables. The value of the argument list of the calling process or utility control can be passed through the service/process channel. The utility request routine is used to prepare the utility control call such that the argument list of the utility control and the list of shared variables are passed.

The service/utility routine is different from the one in service control. It can receive the termination command from the service control and completion of utility control. If the utility control receives the termination command, it simply terminates itself. When the completion signal of utility control is received, it means that the utility control call is completed. The values of arguments which are passed by reference are returned and stored. The list of shared variables is also returned. The utility normal termination routine is used to terminate the utility control in normal manner. However, the utility abnormal termination routine is used to terminate the utility in abnormal manner. When the utility control cannot find a true guard in the wait statement, the utility temporary termination routine is used to let the utility control enter the sleeping state.

The utility control body contains the initialization routine and the statement list of the utility control. The statement list is actually the 'C' version of the corresponding statement list of the utility. It may include 'if', 'loop', 'wait', 'read', 'write', 'exit', statement and utility control call. After the initialization routine is completed, the utility control will be forced to sleep and report to the service control. The utility control will be activated again when it is called by a process or utility control. Then the statement list of the utility control will be executed until one of the following events occur: completion of the statement list, issue of another utility call and waiting for true guard with a wait statements.

5.2.6 Process Control

The process control and service control are at the same level of the run-time support system and under the control of the node control program. The process control is used to collect the coding generated for a process. The coding include 'if', 'loop', 'read', 'write', 'exit', 'skip', 'messin', 'messout' and assignment statements, functions, procedures, function calls, procedure calls and utility calls. The information in the process control consists of the time slice allowed for each process control, number of message constructors, number of main controls, number of node controls, number of service

strings, number of utility strings of each service control, number of process strings of each node control and number of node strings of each main control.

The process control contains a list of service index, a list of process index, a list of process starting index, a utility string table, a process string table, a node string table, a 'C' version of the message definition list (if any), a process string initialization routine, a message address routine, a messout routine, a messin routine, a 'or' messin routine, a 'and' messin routine, a utility request routine, an alarm routine, a node/process routine, a process normal termination routine, a process abnormal termination routine, a process temporary termination routine, a process waiting for message routine, an initialization routine and a process control body.

The list of service or process index show the number of member in the family of service and process. Each process control may be assigned an index (for identification) by its node control. The list of index of all process controls of the same node control will be stored in the list of process starting index. The 'C' version of message definition list contains the type definition and data structure of each message constructor. Hence, the message variables declared in the process control will use the type definition and data structure of one of the

message constructors defined in the message definition list. The data structure of each message constructor consists of the source address of the message, the success flag, the content of the message and a pointer to the next message. Therefore, the message definition list of the process control is used to store up all messages with different types of message constructors.

The message address routine can find out the internal representation of the message address. The message address of a GDPL source program may be an 'address constant', 'address variable' or 'sender' of message variable. It is necessary to transform them to a standard representation of message address in the run-time support system. The messout routine sends the information of the message to the node control and the message to a message buffer. After the acknowledgement is received from the node control, the control of the messout routine will be passed to the next statement after the messout statement in the process control.

The messin routine, 'or' messin routine and 'and' messin routine are used to handle all types of messin statements. The messin routine checks whether there is any message in the message constructor corresponding to the message variable. If a message has arrived, the first arrived message is removed from the message constructor and stored in the message variable of the messin state-

ment; otherwise, the process control will be forced to the waiting for message state. The 'or' messin and 'and' messin routines handle the 'or' and 'and' messin statements. In the messin routine, the source address of the message is neglected, however the source address of the message must be matched with the one used in the messin statement for 'or' and 'and' messin routines. The 'or' messin routine checks whether one of the required messages has arrived. If it is true, the routine removes the message from the message constructor and stores it in the message variable; otherwise, the process control is put in the waiting for message state. The 'and' messin routine must check whether all the required messages have arrived. If it is true, then the routine removes all the required messages from their message constructors and stores them in the message variables. For all of the three types of messin routines, the source address and success flag of the message variable must be filled with the source address of the arrived message and true values respectively.

The utility request routine handles the utility call issued by the process control. It will put the values of the argument list of the utility call into the service/process channel and check the validity of the utility call. The utility request routine will then send the utility call signal to the node control. After the utility call is completed, the value of those arguments

which are passed by reference are stored and the control is passed to the process control again. The alarm routine is invoked when the time slice of the process control have been used up. It will temporarily terminate the process control and report to the node control.

The node/process routine is invoked when a UNIX's signal is sent from the node control. The routine can handle the receiving of messages, termination command, completion of utility and activation of the execution of process control. If the message information have been sent from the node control, the node/process routine should remove the message from the message buffer and store it into the specific message constructor. The source address of the message must also be recorded in the message constructor. When the utility call is completed, the control will be passed to the utility request routine. Eventually, the control will be passed to the process control again.

The process normal termination, process abnormal termination, process temporary termination, process waiting for message routines are used to handle normal termination, abnormal termination, temporary termination and waiting for message state. For all cases, except normal termination, the process control will be forced to sleep and report to the node controls. When the process con-

trol is normally terminated, it means that its corresponding UNIX's process will be killed. The node control will know that the process control is normally completed. The initialization routine is responsible for allocation of memory for message variables, setting up of alarm signal and node control signal (use UNIX signal) and establishment of the process table.

The process control body consists of the initialization routine and the statement list of the process control. The statement list includes the 'C' codes generated by the compiling system for the process. When the initialization routine is completed, the process control will be forced to sleep and report to the node control. The process control will be activated when it is dispatched by the main control. The statement list of the process control will be executed until one of the following events occur: completion of the statement list, issue of a utility call, use of the time slice and waiting for message. If the statement list is completed, the process control should terminate itself. When a utility call is issued, the utility request routine is invoked. If the time slice of the process control is used up, an alarm signal may automatically be sent to the process control. The alarm routine will be invoked, the process control will be temporarily terminated. If the message(s) required have not arrived yet, the process control will be put into the waiting for message state for any type

of the 'messin' statements. In order to have a clear and secure run-time support system, the time-up signal is neglected (use UNIX alarm(Ø)) while handling utility call, 'messin' message and 'messout' statement.

5.3 The Compiling System

5.3.1 Parser

The first component of the compiling system is a recursive decent parser which is developed according to the specification of GDPL (see Appendix A). It is composed of a number of facility routines and a collection of recursive routines. The facility routines are described as follows:

(a) Integer to character routine:

This routine converts an integer to a string of characters.

(b) Error handling routine:

This routine records the type of errors and the error line numbers and prints out the error messages when the compilation is completed.

(c) Lexical analyzer:

Lexical analyzer generates the tokens of the source program for the parser. A token consists of a type, a number and a string. The token type includes keywords,

operators, identifiers and constants. The token number is required for the operator token type to distinguish the different kinds of operator. The token string is only useful in identifier and constant token type.

(d) Constant checking routine:

This routine checks whether the identifier is defined in a constant declaration or not. If the idetifier is a constant, its string and its type are returned.

(e) Variable checking routine:

This routine checks whether the variable is declared or not. If it has been declared, the type of the variable will be returned. Otherwise, an error has occurred. If the variable is a message variable, the identification of the message constructor is also returned.

(f) The definition routine:

This routine finds the type definition of the identifier and returns the index of the type definition table when it is found. Each entry of the type definition table contains the detailed information of the type definition of the identifier.

(g) Message address analyzing routine:

This routine analyzes the character string and checks whether it is a valid message address string. The mes-

sage address string should contain an identifier followed by a '.' and another identifier. The first identifier should be a valid node string and the second identifier must be one of the process strings of the node.

(h) Storage allocation routine:

In the compiling system, a number of linked list of special purpose data types are required. The storage for a linked list will be requested during parsing. Therefore, the storage allocation scheme for each element of the linked list is dynamic. The storage allocation routine maintains a pool of elements construct of each special purpose data type. If the element is requested, an element's construct is removed from its pool to fulfil the request. When the pool is empty, the UNIX's system call (calloc) is used to allocate a new element's construct. When a linked list of a data type is released, all the element's construct's are returned to the pool of the data type and the field of each element's construct are cleaned up. The special purpose data types are built for fulfilling the different requirements of the various parts of the parser. They include 'string', 'type', 'proc/fn', and 'utility' data types.

The 'string' data type contains a character string pointer. It is useful for lists of node strings, service strings, utility strings, subscript expression, process

message, etc. When codes are generated, the 'string' data type is the unique data structure to maintain the character strings of codes. The next kind of data type is 'point'. The 'point' data type contains a character string pointer, two integer fields and a character field. The 'pointer' data type is usually used to record the token information, the type of a variable and some conditions. The types of a variable include 'integer', 'character', 'boolean', 'float', 'address', 'array' and 'record'. The message definition list, message variables list, shared variables list, local variables list, type declaration, parameter list and identifier list have the 'point' data type.

The 'string head' data type is designed for construction of a list of 'string' list. Typical examples are variable list and argument list. As the name implies, the 'node information' data type is used to keep the information of a node. The information includes the number of string, process strings, service and processes, service index list, process index list, process pointer list, service file name list and process file name list. The 'error' data type is simply to record the error codes and the line number of the GDPL source program where error occurs. There are nearly one hundred types of compilation error. The constant tables of the parser belong to the 'constant' data type. The symbol and type definition tables of service, utility, process and proc/fn are

built of 'symbol' data type. The 'type' data type is used to establish the type tables of the parser. The 'proc/fn' data type records the information of procedures or functions. The information includes the type of routine, data type of function if necessary, routine name and list of parameter list. The type of routine is either procedure or function.

The collection of recursive routines are actually one-to-one mapping to the nonterminals of the extended BNF of GDPL (see Appendix B). The recursive routines are responsible for checking the syntax and semantics of the source program. The syntax of the source program is simply checked by parsing the tokens generated by the lexical analyzer. If the token of the source program can successfully come across the recursive routines and go to the end of the parser, the syntax of the source program is error free. Otherwise error messages will be printed and the compilation will be terminated when parsing can not proceed any more. The checking of the source program is not an easy task. A number of tables, lists, data structures and flags are used to achieve this purpose. The major checkings for semantics are listed below:

(a) Only a message type definition may contain the 'address' data type.

(b) The starting identifier must match the ending

identifier of the following program constructs: node, service, utility, process, procedure and function.

(c) No array of arrays is allowed.

(d) There should be at least one node within the source program.

(e) There should be at least one process within each node.

(f) There should be at least one utility within each service.

(g) Within a service, there should be a shared variable declaration.

(h) Only the process may contain procedures or functions.

(i) No 'wait', 'messin', 'messout' statement and utility call are allowed in a service.

(j) No 'messin' and 'messout' statement are allowed in a utility.

(k) No 'wait' statement is allowed in the process, procedure and function.

(l) The local variables of a utility must not be declared before in the shared variables of the service and the parameter list of itself.

(m) The local variables of proc/fn must not be declared before in the process and the parameter list of itself.

(n) For the utility and procedure/function, the data types of the parameter list should be declared in the service and process respectively.

(o) The initial priority must be within the range Ø to 127.

(p) The type of L.H.S. and R.H.S of an assignment statement must be compatible.

(q) The attribute 'sender' of message variables cannot be invoked in an arithmetic computation.

(r) The variable of the 'messin' or 'messout' statement should be declared with a data type of the message definition list.

5.3.2 Code Generation

The objective of the code generation is to generate the 'C' version of the GDPL source program (i.e. the run-time support system). The sample outputs of GDPL program constructs will be shown in Appendix C. The code generation is spread among the collection of recursive routines. Since the configuration of the distributed computing environment will be input at the linkage time, the codes depending on the configuration will be generated after the linker is completed. The code

generation will first transform each source statement into 'C' version and then organize them into the manager control, the main control, the node control, the service control, the utility control and the process control according to the specification of the source program. The run-time support system can be divided into two parts for the sake of code generation. They are the 'basis' part and the 'variation' part. The 'basis' part forms a framework of the run-time support system such that the 'variation' part can be added to it to construct the whole system. The 'basis' part of the run-time support system is fixed for any GDPL source program. However, the 'variation' part will vary depending on both the software (GDPL source program) and the hardware (configuration of the distributed computing environment).

For 'exit', 'skip', 'if', 'loop', 'read', 'readln', 'write', 'writeln' and assignment list, the scheme for code generation is simple because there are corresponding statements in the 'C' language. Only a direct translation from GDPL to 'C' is required. Because there is no procedure in 'C', both procedure and function of GDPL are translated into functions in the 'C' language. The difficulties in implementing the procedure/function are the argument passing problems and the distinctions of the parameters and local variables within the procedure and function.

When translating the utility call, the 'wait' statement, the 'messin' statement and the 'messout' statement, some simulation schemes are required to generate their codes. It is because they have no counterparts in the 'C' language. The implementation of the utility call includes the utility argument list passing and the actual invocation of the utility. The argument passing for procedure/function and utility call are completely different. The argument passing mechanism for procedure/function is implemented in the same program construct (process). However, the arguments of utility will be passed from one program construct (e.g. process) to another type of program construct (utility).

The schemes for translating the 'wait' and 'loop' statement are very similar. They should establish the loop structure and the exit conditions. The scheme for translating the 'wait' statement is also required to handle the finding of true guards. When no guards of the 'wait' statement is true, the process should be forced to the sleeping state. Finally, it must handle the checking of guards if the sleeping process is waken up again. The implementation of 'messin' and 'messout' statements are the most difficult job. Initially, they should find out the internal representation of the message address and check whether the address is valid or not. The second step is to find out the number of the message constructor. The number of the message

constructor determines the operations to be used for manipulation of the messages. The third step is to send or receive the message to/from a message buffer. The fourth step is to check for the arrival of messages. The fifth step handles the sleeping state of processes. When the expected messages have not arrived, the process should be forced to sleep. The final step is to check for the arrival of messages again when the sleeping process is waken up. If all the messages have arrived, the messages should be stored in the associated message variables and the attributes of the message variables should also be set.

5.3.3 Linker

The third component of the compiling system is the linker. The functions of the linker are shown below.

(a) Initially, the linker will ask for the programmer to input a configuration of the physical processors. The information of the configuration includes the number of physical processors and the connection of the processors. The information will be validated.

(b) The linker will select the processors from the input configuration to construct the distributed computing environment where the run-time support system will execute. Any two selected processors must at least find a path to the other and the number of the selected proces-

sors is optional.

(c) The nodes of the GDPL source program are mapped to the physical processors. Then the number of main controls is computed. The linker should also determine the physical processor that the manager control will reside.

After the linker has completed the above three functions, the remaining codes are generated for the statements and program construct. The final version of the whole run-time support system is then turned out. Within the run-time support system, a number of exception handlers are embedded. The exception handlers can detect the timing problem of the process control, invalid utility call, invalid message address, activation of a terminated process, sending message to a terminated process, invalid priority value, invalid messages, abnormal termination manager control, main control, node control, service control, utility control and process control.

CHAPTER 6

Conclusion

The design and implementation of GDPL have been successfully completed. Several sample programs of GDPL have been written in order to test the run-time support system and the compiling system. The performance of the two systems are fairly satisfactory. Most of the objectives of the design and implementation of the distributed programming language have been achieved. They are:

(a) The fourteen sample programs have been written in other parallel programming languages such as DPL-82, Concurrent Pascal, Ada, etc. It is not difficult to translate the sample programs into the GDPL versions. It shows that the service concept, the message passing mechanism, the array of services and the array of processes are especially powerful tools to express the concurrency in parallel application systems.

(b) The sample programs are easy to read, modify and verify. They have also a clear, secure and structured program control structure. Although many design purposes and philosophies are integrated, no confusion on

the syntax and semantics of GDPL occurs. This is because the access rights and the order of operations on the data objects are clearly defined. There are no special problems for establishing the communication paths and synchronization primitives within the sample programs. This illustrates that GDPL has clear, simple and structured program control structure, access control of data objects, execution order and process creation/termination time, which are important factors when developing parallel programming systems.

(c) During the design and implementation of the language, a lot of valuable experience in parallel programming language design have been gained, which include solutions for the major issues, the design and implementation considerations and the development of the application systems, run-time support system, and the compiling system of a distributed programming language.

The insufficiencies of the present implementation are also presented below.

(a) The run-time support system can only execute in a single processor. Each node is a virtual processor. In a distributed computing environment, the virtual processors should be mapped to the physical processors. However, all the nodes are mapped into the same processor - Zilog S8000 in this version of implementation, the distributed computing environment can only be simulated.

(b) No separate compilation of the GDPL source program is provided. Seperate compilation is useful for developing large scale application systems. Only the modified part should be recompiled. This could reduce a lot of development time and also give the programmer more flexibility in organizing his application systems. However, the separate compilation of GDPL programs have been taken into consideration in the design. Therefore, it is not difficult to incorporate this feature in the near future.

(c) The 'go to' statement, wake up statement, global variables, block structure and dynamic process creation are not used. Many advantages of these features cannot be obtained. However, a good control structure, clear and secure program and low cost implementation of the language can be maintained although some compromises are necessary.

Some future developments of GDPL are suggested as follows:

(a) The separate compilation of a GDPL program should be implemented.

(b) The run-time support system should be modified to execute in an actual distributed computing configuration. The design strategies have already been available. The new version of the run-time support system is dif-

ferent from the current one in the physical communication links between the physical processors. In the current implementation, the communication link between virtual processors are handled by some temporary files.

(c) More flexible message address scheme should be derived such that the message passing mechanism is enhanced.

(d) A distributed kernel should be implemented in the distributed computing environment in order to support the process switching, storage management, physical communication and exception handling in a more efficient manner.

(e) Optimization of the code generated may improve a lot the performance of the run-time support system.

(f) The current compiling system employs the one-pass compiling technique. If one or two passes are added, the efficiency of the compiling system will be improved.

(g) In an actual application systems development, the input of the distributed computing configuration is not required. Each installation of the GDPL in a distributed computing environment should be assumed to have such configuration. The programmer should write programs to execute in this distributed computing configuration.

Appendix A : The Specification of GDPL

The language specification of Generalized Distributed Programming Language (GDPL) is presented.

The context-free syntax of the language is described using an extended version of Backus-Naur Form (see AppendixB).

(a) Lower case words enclosed in <> denotes syntactic categories, for example

<process>

(b) Optional items are enclosed in sequence brackets, for example

[identifier]

(c) Repeated items are enclosed in braces meaning that the item may appear zero or more times, for example

sequential-statement {sequential-statement}

(d) Lower case words enclosed by the symbols '' are terminals, for example

'a' (represent the first lower case letter)

A.1 Basic Elements

A.1.1 Character Set

(a) <characters> ::= <letters>|<digits>|
<special_characters>

(b) <letters> ::= <lower_case_letters>|
<upper_case_letters>

<lower_case_letters> ::=
'a'|'b'|'c'|'d'|'e'|'f'|'g'|
'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|
'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|
'x'|'y'|'z'

<upper_case_letters> ::=
'A'|'B'|'C'|'D'|'E'|'F'|'G'|
'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|
'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|
'X'|'Y'|'Z'

(c) <digits> ::= '0'|'1'|'2'|'3'|'4'|
'5'|'6'|'7'|'8'|'9'

(d) <special_characters> ::= '+'|'-'|'*'|'/'|
'('|')'|'['|']'|
':'|'''|'^'|'_'|
'{'|'}'|'<'|'>'|
';'|','|'='|'.'|
'"'|'#'

A.1.2 Separator

(a) <separator> ::= <space>|
<statement_separator>|
<object_separator>|
<comment>

(b) <space> ::= ' '

(c) <statement_separator> ::= ';'

(d) <object_separator> ::= ','

(e) <comment> ::= '{'{<valid_characters>}'}'

Any two adjacent word symbols or names must be separated by at least one separator. Comment can be inserted into any place of the program.

(f) <valid_characters> ::= <letter>|<digits>|
'+'|'-'|'*'|'/'|
'('|')'|'['|']'|
':'|''''|'^'|'_'|
'{'| '<'|'>'|
';'|','|'='|'.'|
'"'

A.1.3 Operator

(a) <operators> ::= <arith_operators>|
<relational_operators>|
<logical_operators>|
<other_operators>

(b) <arith_operators> ::= '+'|'-'|'*'|'/'|'**'

(c) <relational_operators> ::= '=='|'<>'|'<'|
'<='|'>'|'>='

(d) <logical_operators> ::= 'not'|'or'|'and'

(e) <other_operators> ::= '[]'|'->'|'^'|':'|':='|'.'

The arithmetic operators, relational operators and logical operators have the same meaning as the classical programming languages such as Pascal. though the syntax is not exactly the same.

'[]' is used in the guarded command list.

'->' is used in the guarded command list as well.

'^' is used in the messin statement and has a meaning "and".

':' is used in the messin statement and has a meaning "or".

':=' is the assignment operator.

'.' is used for qualification.

A.1.4 Constant

(a) <constant> ::= <char_constant>|
<int_constant>|
<bool_constant>|

<real_constant>|

<address_constant>

(b) <char_constant> ::= '''<letters>'''

The letters include all valid characters except '''.

(c) <int_constant> ::= <signed_constant>|

<unsigned_constant>

<signed_constant> ::= ('+'|'-')<unsigned_constant>

<unsigned_constant> ::= <digit> {<digit>}

(d) <bool_constant> ::= 'false'|'true'

(e) <real_constant> ::= <int_constant>(<null>|'E')

[<int_constant>]

All scalar types have its own type constant. All composition types have type constant depending on those members with scalar types.

(f) <address_constant> ::= '"'<address_characters>'"'

(g) <address_characters> ::= <node_name>'.'<process_name>

Both the node and process number must be specified.

(h) <node_name> ::= <name>

(i) <process_name> ::= <name>

A.1.5 Name entity

(a) <alpha_num> ::= (<letters>|<digits>)
{<letters>|<digits>}

(b) <name> ::= <letters> {(<null>|'_')
<alpha_num>}

(c) <null> ::= ''

A.1.6 Reserved word

(a) <reserved_word> ::= <key_word> | <attributes>

(b) <key_word> ::= 'node'|'service'|
'process'|'utility'|
'type'|'of'|'var'|'const'|'common'|
'begin'|'end'|
'if'|'fi'|'else'|'loop'|'lend'|'exit'|
'read'|'readln'|'write'|'writeln'|
'skip'|'wait'|'wend'|
'procedure'|'function'|
'messin'|'messout'|
'int'|'char'|'bool'|'real'|
'address'|
'record'|'message'|'array'|
'true'|'false'|
'return'

(c) <attributes> ::= 'succ'|'sender'|'maxint'|
'minint'|'me'|'priority'

The attribute 'me' is used to indicate the index of

the current service (or process) of the family of services (or processes). If the family of services (or processes) has only one member, then 'me' will contain a value of Ø.

A.2 Declaration and Type

A.2.1 Type declaration

(a) <type_definition_list> ::= 'type'
<type_definition>
{';'<type_definition>}';'

(Note: <null> denotes a null character, i.e. '')

(b) <type_definition> ::= <type_name> '=' <type>

(c) <type> ::= <scalar_type>|
<structured_type>

(d) (scalar_type> ::= 'int'|'char'|'bool'|
'real'|'address'

The constant name may assume the data type 'address' in the following way : <constant_name> = "...", where ... denotes a string of valid characters. The address constants variables of this type can be used by messin or messout statement to specify the source or destination addresses. Actually, the data type 'address' is an array of characters. The attributes 'me' and 'sender' are of the data type

'address'.

(e) <structured_type> ::= <array_type>|
<record_type>

(f) <array_type> ::= 'array''['<subrange>
{','<subrange>}']'
'of' <type>

(g) <record_type> ::= 'record'
<field_name> ':' <type>
{';'<field_name> ':' <type>}
'end'

(h) <type_name> ::= <name>

(i) <field_name> ::= <name>

(j) <subrange> ::= <digits>{<digits>}|
<int_constant_name>

(k) <int_constant_name> ::= <name>

A.2.2 Constant declaration

(a) <constant_definition_list> ::= 'const'
<constant_definition>
{';'<constant_definition>}';'

(b) <constant_definition> ::= <constant_name> '='
(<constant>|'#'<octal_constant>)

The '#' denote the octal byte address. If the '#' is used, then the constant_name is equivalent to a symbolic address with the value equal to the octal constant.

(c) <constant_name> ::= <name>

A.2.3 Message declaration

(a) <message_definition_list> ::= 'message'
<message_definition>
{';'<message_definition>}';'
'end'

(b) <message_definition> ::= <message_name>'='
<message_type>

The message definition is similar to the type definition, in which a number of message constructors is defined. Then the message variable in each process may assume these message constructors.

However, the message definition list is different from the type definition list that only the data type 'address' may be used within the message definition list but not in the type definition list.

(c) <message_name> ::= <name>

(d) <message_type> ::= <scalar_type>|
<array_type>|

```
                                <message_record_type>|
                                <address_type>

(e)  <message_record_type>   ::= 'record'
                                    <field_name>':'
                                    <message_type>
                                {';'<field_name>':'
                                    <message_type>}
                                'end'
```

The message record type is nearly equivalent to the record type except that the member of the message record can contain field with the data type 'address' but the record type cannot.

```
(f)  <address_type>          ::= 'address'
```

A.2.4 Variable declaration

```
(a)  <variable_definition_list> ::= 'var'
                                    <variable_definition>
                                {';'<variable_definition>}';'

(b)  <variable_definition>  ::= <variable_name>':'<type>|
                                <message_variable_name>':'
                                <message_name>

(c)  <variable_name>         ::= <int_variable_name>|
                                 <real_variable_name>|
                                 <char_variable_name>|
                                 <bool_variable_name>|
```

<array_variable_name>|
<record_variable_name>

(d) <int_variable_name> ::= <name>

(e) <real_variable_name> ::= <name>

(f) <char_variable_name> ::= <name>

(g) <bool_variable_name> ::= <name>

(h) <array_variable_name) ::= <name>

(i) <record_variable_name> ::= <name>

(j) <message_variable_name> ::= <name>

A.2.5 Common variable declaration

(a) <common_variable_definition_list> ::= 'common'
<common_variable_definition>
{';'<common_variable_definition>}';'

(b) <common_variable_definition> ::=
<common_variable_name>':'
<type>

(c) <common_variable_name> ::= <variable_name>

Common variables declared can only be used within service and utility.

A.3 Statements

(a) <statements> ::= <sequential_statements>|
<comm_sync_statements>

(b) <sequential_statements> ::= <assignment_statement>|
<if_statement>|
<loop_statement>|
<exit_statement>|
<i/o_statement>|
<skip_statement>

Statements can be divided into sequential statements and communication/synchronization statements.

Statements used in utility and processes are slightly different, therefore the syntax illustrated in this section is only used to show the syntactic structure of all the statement of the language. The statements should be probably indexed when used in the utility and process definition.

A.3.1 Sequential statements

A.3.1.1 Assignment statement

(a) <assignment_statement> ::= <variable> ':=' <expression>

(b) <expression> ::= <arith_expression>|
<logical_expression>|
<char_expression>

(c) <arith_expression> ::= <arith_expression> '+' <term>|

```
                    <arith_expression> '-' <term>|
                    <term>

(d)  <term>         ::= <term> '*' <factor>|
                        <term> '/' <factor>|
                        <factor>

(e)  <factor>       ::= <primary> '**' <factor>|
                        <primary>

(f)  <primary>      ::= '-' <primary>|
                        <element>

(g)  <element>      ::= '('<arith_expression>')'|
                        <constant>|
                        <variable>

(h)  <int_variable>     ::= <int_variable_name>|

(i)  <real_variable>    ::= <real_variable_name>|
```

The '*' and '/' operators are left-associative. The '+' and '-' operators are left-associative.

The precedence is order from high to low :

'-' (unary minus), '**', '*' or '/', '+' or '-'

```
(j)  <logical_expression>    ::= (<logical_expression>
                                  ('or'|'and')
                                  <relational_expression>)|
                                 (<null>|'not')
```

'('<logical_expression>')'|
<bool_variable>|
<bool_constant>

(k) <relational_expression> ::= <arith_expression>
<relational_operators>
<arith_expression>

The logical expression is evaluated from left to right. The precedence of the relational operators are higher than the logical operators. Parentheses are used where appropriate.

(l) <char_expression> ::= <char_variable>|
<char_constant>

(m) <variable> ::= <int_variable>|
<real_variable>|
<char_variable>|
<bool_variable>|
<message_variable>|
<indexed_variable>|
<record_member_variable>|
<message_member_variable>|
<predefined_variable>

(n) <int_variable> ::= <int_variable_name>

(o) <real_variable> ::= <real_variable_name>

(p) <char_variable> ::= <char_variable_name>|

(q) <bool_variable> ::= <bool_variable_name>

(r) <message_variable> ::= <message_variable_name>

(s) <indexed_variable> ::= <array_variable_name>
'['<arith_expression>
{','<arith_expression>}']'

Each element of an array is composed of an array variable name and a subscript which is composed of a sequences of arithmetic expressions.

(t) <record_member_variable> ::= <record_variable_name>
'.'<variable>

Each member of a record variable is denoted by the name of the record variable followed by a "." and the name of the member. The member may also be a record.

(u) <message_member_variable> ::= <message_variable_name>
'.'<variable>

The construct and the usage of the message is the same as the record except that the record cannot be used in the messin and messout statement.

(v) <predefined_variable> ::= 'me'|'maxint'|'minint'|
<message_variable>'"'
('succ'|'sender'|'priority')

A.3.1.2 If statement

(a) <if_statement> ::= 'if' <guarded_command_list>
['else' <statement_list>]
'fi'

An else guard is satisfied only when all other guards of the 'if' statement are not satisfied. However this guard is optional.

(b) <guarded_command_list> ::= <guarded_command>
{'[]' <guarded_command>}

(c) <guarded_command> ::= <guard> '->'
<statement_list>

When a 'if' statement is executed, each guard is examined. If one or more guards are satisfied, then one of the guarded commands is arbitrarily selected and the statement list of the selected guarded command is executed. After the statement list is executed, then 'if' statement is terminated. If no guard is satisfied, then it is a run-time error and the program is aborted.

(d) <guard> ::= <logical_expression>

A.3.1.3 Loop statement

(a) <loop_statement> ::= 'loop' <guarded_command_list>
'lend'

When a 'loop' statement is executed, each guard is examined. If one or more guards are satisfied, then one of the guarded commands is arbitrarily selected and the statement list of the selected guarded command is executed. After the statement list is executed, the above steps are followed again until there is no guard is satisfied.Then the 'loop' statement is terminated.

A.3.1.4 Exit statement

(a) <exit_statement> ::= 'exit'

The 'exit' statement can be used only within a loop. When an 'exit' statement is encountered, then control exits from the loop in which the exit statement resides. One level of exit is allowed.

A.3.1.5 I/O statement

(a) <i/o_statement> ::= <read_statement>|
<write_statement>

(b) <read_statement> ::= 'read' '('<variable_list>')'|
'readln' '('<variable_list>')'|

(c) <write_statement> ::= 'write' '('<variable_list>')'|
'writeln' '('<variable_list>')'

(d) <variable_list> ::= <variable> {','<variable>}

The input pointer will point to the same line after the 'read' statement is executed. While the input pointer will point to the next line after 'readln' statement is executed.

The output pointer will point to the same line after 'write' statement is executed. While the output pointer will point to the next line after 'writeln' statement is executed.

There are two types of parameter passing mechanism. The first one is by value, and the other is by reference. If the parameter is preceded by nothing, it is passed by value. However,if the parameter is preceded by "var", then it is passed by reference. The parameter passing mechanism is applied for both utility, procedure and function.

A.3.1.6 Skip statement

(a) <skip_statement> ::= 'skip'

The skip statement is simply a null statement. It is useful for some checking but without execution.

A.3.2 Communication and Synchronization statements

A.3.2.1 Wait statement

(a) <wait_statement> ::= 'wait' <guarded_command_list>
'wend'

When a 'wait' statement is executed, each guard is checked. If there is one or more guards which are satisfied, then one of them is selected arbitrarily and the statement list following the selected guard is executed. After the statement list is executed, the 'wait' statement is terminated. Otherwise, the process is said to be delayed (or sleeping) and the control is passed to other process.

A.3.2.1 Messin statement

(a) <messin_statement> ::= 'messin'
('('<message_variable>')'|
<source_address>
'('<message_variable>')'
({'^'<source_address>
'('<message_variable>')'}|
{':'<source_address>
'('<message_variable>')'}))

If only message_variable are specified following the keyword 'messin', then it implies that this messin statement will accept all messages sent from any process using the same message name. No combined use of '^' and ':' is allowed.

(b) <source_address> ::= <address_definition>

(c) <address_definition> ::= <message_variable>
'"' 'sender'|

(<address_constant>|
<address_constant_name>|
<message_variable>|
<message_member_variable>)
['['<index>']']

(d) <address_constant_name> ::= <name>

A.3.2.3 Messout statement

(a) <messout_statement> ::= 'messout'
<destination_address>
'('<message_variable>')'

There is only one form of messout statement and only one receiving process is specified.

(b) <destination_address> ::= <address_definition>

A.4 Program

(a) <program> ::= [<message_definition_list>]
<nodes>

A.4.1 Nodes

(a) <nodes> ::= <node> {<node>}

(b) <node> ::= 'node' <node_name> ';'
[<services>]

'begin'

<processes>

'end' <node_name> ';'

(c) <node_name> ::= <name>

A.4.2 Processes

(a) <processes> ::= <process> {<process>}

(b) <process> ::= 'process' <process_name>
['['<range>']']
['('<process_priority>')']';'

[<constant_definition_list>]
[<type_definition_list>]
<variable_definition_list>

[<functions>]
[<procedures>]

'begin'

<statement_list_1>

'end' <process_name>;

In each process, there may be an optional range. The range is used to indicate that a family of processes is to be created

The family of processes contains : <process_name>[0], <process_name>[1] ,..., <process_name>[<range>-1].

The constant and type declarations are optional. However, there must be a variable declaration list in which every variable used within the process is declared.

Only the process can use the functions and procedures. The variables of the process can be referenced only through the parameter passing mechanism.

(c) <range> ::= <unsigned_constant>

(d) <process_priority> ::= <unsigned_constant>

(e) <functions> ::= <function> {<function>}

(f) <function> ::= 'function' <function_name>
'('[<parameter_list>]')'
':' <type> ';'

[<constant_definition_list>]
[<type_definition_list>]
[<variable_definition_list>]

'begin'

<statement_list_1>

'end' <function_name> ';'

The parameters of a function must be declared in the

parameter definition list. In addition to the parameters passed by the process, the function also has a number of local variables. The statements allowed in a process is also allowed in the function.

(g) <procedures> ::= <procedure> {<procedure>}

(h) <procedure> ::= 'procedure' <procedure_name>
'('[<parameter_list>]')' ';'

[<constant_definition_list>]
[<type_definition_list>]
[<variable_definition_list>]

'begin'

<statement_list_1>

'end' <procedure_name> ';'

The parameters of the procedure must also be declared in the parameter definition list. The statements allowed in a process is also allowed in the procedure.

(i) <statement_list_1> ::= <statement_1>
{';'<statement_1>}';'

(j) <statement_1> ::= <sequential_statements>|
<function_call>|
<procedure_call>|
<utility_call>|

<messin_statement>|
<messout_statement>

(k) <function_call> ::= <function_name>
'('[<variable_list>]')'

(l) <parameter_list> ::= (<null>|'var')<variable>
{','<variable>}':'<type>
{';'(<null>|'var')<variable>
{','<variable>}':'<type>}

(m) <procedure_call> ::= <procedure_name>
'('[<variable_list>]')'

(n) <utility_call> ::= [<service_name>]
['['<index>']']'.'
<utility_name>
'('[<variable_list>]')'

(o) <process_name> ::= <name>

(p) <function_name> ::= <name>

(q) <procedure_name> ::= <name>

(r) <service_name> ::= <name>

(s) <utility_name> ::= <name>

(t) <index> ::= <arith_expression>

A.4.2 Services and Utilities

(a) <services> ::= <service> {<service>}

(b) <service> ::= 'service' <service_name>
['['<range>']'] ';'

[<constant_definition_list>]
[<type_definition_list>]
<common_variable_definition_list>
[<variable_definition_list>]

<utilities>

'begin'

[<initialization_statement_list>]

'end' <service_name> ';'

In each service, the constant, type and variable definition list are optional. However, the common variable definition list must be declared.

The variables declared within the variable definition list are local to the service (i.e, they are not shared objects of the node).

The service can use all statements of the language except 'wait', 'messin' and 'messout' statements.

In each service, there may be an optional range. The range is used to indicates that a family of services is to be created The number of a member of the family

is equal to the number specified by the range If a member is referenced, the service name and an index of that member are required.

The family of the services contains : <service_name>[Ø], <service_name>[1],..., <service_name>[<range>-1].

(c) <utilities> ::= <utility> {<utility>}

(d) <utility> ::= 'utility' <utility_name>
'('[<parameter_list>]')'';'

[<constant_definition_list>]
[<type_definition_list>]
[<variable_definition_list>]

'begin'

<statement_list_2>

'end' <utility_name> ';'

The constant declaration, type declaration and variable declaration are optional.

(e) <initialization_statement_list> ::=
<sequential_statements>

(f) <statement_list_2> ::= <statement_2>
{';'<statement_2>}';'

(g) <statement_2> ::= <sequential_statements>|
<utility_call>|
<wait_statement>

(h) <wait_statement> ::= 'wait' <guarded_command_list>
'wend'

Appendix B : Extended BNF of GDPL

<program> ::= <m_d><n_d>

<m_d> ::= 'm'<t_d>

<m_d> ::= e

<t_d> ::= <na_l>'='<t>';'<t_dl>

<t_dl> ::= <t_d>

<t_dl> ::= e

<t> ::= 'rd'<r_d>'en'

<t> ::= 'ad'

<t> ::= 'in'

<t> ::= 'ch'

<t> ::= 'bo'

<t> ::= 're'

<t> ::= 'a'<sub_d>'of'<t>

<t> ::= 'na'

<r_d> ::= 'na'':'<t>';'<r_dl>

<r_dl> ::= <r_d>

<r_dl> ::= e

<na_l> ::= 'na'<na_ll>

<na_ll> ::= ','<na_l>

<na_ll> ::= e

<sub_d> ::= '['<sub1>']'

<sub1> ::= 'ct'<sub2>

<sub1> ::= 'na'<sub2>

<sub2> ::= ','<sub1>

<sub2> ::= e

<n_d> ::= 'n''na'';'<n1><n_d1>

<n_d1> ::= <n_d>

<n_d1> ::= e

<n1> ::= <s_d>'b'<p_d>'en''na'';'

<s_d> ::= 's''na'<s1><s_d>

<s_d> ::= e

<s1> ::= '[''ct'']'';'<s2>

<s1> ::= ';'<s2>

<s2> ::= <tc_d><s3>

<s3> ::= <cv_d><s4>

```
<s4> ::= <v_d><s5>

<s5> ::= <u_d>'b'<s_1>'en''na'';'

<tc_d> ::= 'c'<cd_d1><tc_d>

<tc_d> ::= 't'<t_d><tc_d>

<tc_d> ::= e

<cd_d1> ::= 'na''='<c>';'<cd_d2>

<cd_d2> ::= <cd_d1>

<cd_d2> ::= e

<cv_d> ::= 'co'<vd_d>

<vd_d> ::= <na_1>':'<t>';'<vd_d1>

<vd_d1> ::= <vd_d>

<vd_d1> ::= e

<v_d> ::= 'v'<vd_d>

<v_d> ::= e

<u_d> ::= 'u''na'<u1><u_d1>

<u_d1> ::= <u_d>

<u_d1> ::= e

<u1> ::= '('<p_1>')'';'<u2>
```

```
<u2> ::= <tc_d><u3>

<u3> ::= <v_d><u4>

<u4> ::= 'b'<s_l>'en''na'';'

<p_l> ::= 'v'<na_l>':'<t><p_ll>

<p_l> ::= <na_l>':'<t><p_ll>

<p_l> ::= e

<p_ll> ::= ';'<p_l>

<p_ll> ::= e

<p_d> ::= 'p''na'<p1><p_dl>

<p_dl> ::= <p_d>

<p_dl> ::= e

<p1> ::= '[''ct'']'<p2>

<p1> ::= <p2>

<p2> ::= '(''ct'')'';'<p3>

<p2> ::= ';'<p3>

<p3> ::= <tc_d><p4>

<p4> ::= <v_d><p5>

<p5> ::= <fn_pr>'b'<s_l>'en''na'';'
```

<fn_pr> ::= 'fn''na'<fn1><fn_pr>

<fn_pr> ::= 'pr''na'<u1><fn_pr>

<fn_pr> ::= e

<fn1> ::= '('<p_l>')'<fn_t1>';'<u2>

<fn_t1> ::= ':'<t>

<fn_t1> ::= e

<c> ::= <c1>

<c> ::= '+'<c2>

<c> ::= '-'<c2>

<c> ::= <c2>

<c1> ::= '#''ct'

<c1> ::= 'st'

<c1> ::= 'f'

<c1> ::= 'tr'

<c1> ::= 'max'

<c1> ::= 'min'

<c1> ::= 'me'

<c2> ::= 'ct'

<c2> ::= 'rc'

<v> ::= 'na'<v1>

<v> ::= 'pri'

<v1> ::= '['<sub_e>']'<v1>

<v1> ::= '.'<v>

<v1> ::= '''<v2>

<v1> ::= '('<arg_1>')'

<v1> ::= e

<v2> ::= 'su'

<v2> ::= 'se'

<sub_e> ::= <a_e><sub_e1>

<sub_e1> ::= ','<sub_e>

<sub_e1> ::= e

<l_e> ::= <r_e><l_e1>

<l_e1> ::= <and_or><r_e><l_e1>

<l_e1> ::= e

<r_e> ::= <a_e><r_e1>

<r_e1> ::= 'ro'<a_e><r_e1>

```
<r_el> ::= e

<and_or> ::= 'an'

<and_or> ::= 'or'

<a_e> ::= <term><a_el>

<a_el> ::= '+'<a_e>

<a_el> ::= '-'<a_e>

<a_el> ::= e

<term> ::= <f><terml>

<terml> ::= '*'<term>

<terml> ::= '/'<term>

<terml> ::= e

<f> ::= <pri><fl>

<fl> ::= '**'<f>

<fl> ::= e

<pri> ::= <pril>

<pri> ::= <c2>

<pril> ::= '('<l_e>')'

<pril> ::= 'no''('<l_e>')'
```

<pril> ::= <v>

<pril> ::= <cl>

<pril> ::= '+'<pri>

<pril> ::= '-'<pri>

<s_l> ::= <s_s>';'<s_l>

<s_l> ::= e

<s_s> ::= 'if'<g_c_l><if_s>

<s_s> ::= 'l'<g_c_l>'le'

<s_s> ::= 'wa'<g_c_l>'we'

<s_s> ::= 'r''('<v_l>')'

<s_s> ::= 'rl''('<v_l>')'

<s_s> ::= 'w''('<arg_l>')'

<s_s> ::= 'wl''('<arg_l>')'

<s_s> ::= 'ex'

<s_s> ::= 'sk'

<s_s> ::= 'mi'<mi_s>

<s_s> ::= 'mo'<ad_d>'('<v>')'

<s_s> ::= 'pri'':='<a_e>

```
<s_s> ::= 'na'<s_s1>

<s_s1> ::= '''<v2>':='<a_e>

<s_s1> ::= '('<arg_l>')'

<s_s1> ::= '['<a_e><s_s2>

<s_s1> ::= <s_s3>

<s_s2> ::= ','<sub_e>']'<v1>':='<a_e>

<s_s2> ::= ']'<s_s3>

<s_s3> ::= ':='<a_e>

<s_s3> ::= '.''na'<s_s4>'('<arg_l>')'

<s_s4> ::= <v1>':='<a_e>

<s_s4> ::= '('<arg_l>')'

<g_c_l> ::= <l_e>'->'<s_l><g_c_ll>

<g_c_ll> ::= '[]'<l_e>'->'<s_l><g_c_ll>

<g_c_ll> ::= e

<if_s> ::= 'el'<s_l>'fi'

<if_s> ::= 'fi'

<v_l> ::= <v><v_ll>

<v_ll> ::= ','<v_l>
```

<v_ll> ::= e

<arg_l> ::= <a_e><arg_ll>

<arg_l> ::= e

<arg_ll> ::= ','<arg_l>

<arg_ll> ::= e

<mi_s> ::= '('<v>')'

<mi_s> ::= <ad_d>'('<v>')'<mi_s1>

<mi_s1> ::= '^'<ad_d>'('<v>')'<mi_s2>

<mi_s1> ::= ':'<ad_d>'('<v>')'<mi_s3>

<mi_s1> ::= e

<mi_s2> ::= '^'<ad_d>'('<v>')'<mi_s2>

<mi_s2> ::= e

<mi_s3> ::= ':'<ad_d>'('<v>')'<mi_s3>

<mi_s3> ::= e

<ad_d> ::= 'na'<ad_d1>

<ad_d> ::= 'st'<ad_d2>

<ad_d1> ::= ''''se'

<ad_d1> ::= e

<ad_d2> ::= '['<a_e>']'

<ad_d2> ::= e

Appendix C : Sample outputs of GDPL

The sample outputs of manager control, main control, node control, service control, utility control, process control, utility call, following sections using extended BNF notations.

C.1 Manager control

<information of manager control>

<facility routines of manager>

<manager/main routine of manager>

<initialization routine of manager>

<manager body> := <initialization routine>

<infinite loop of control statement in manager>

C.2 Main control

<information of main control>

<facility routines of main>

<manager/main routine of main>

<main/node routine of main>

<process control routines of main>

<initialization routine of main>

<main body> := <initialization routine of main>

<infinite loop of control statement of main>

C.3 Node control

<information of node control>

<facility routines of node>

<main/node routine of node>

<node/service routine of node>

<node/process routine of node>

<initialization routine of node>

<node body> := <initialization routine of node>

<infinite loop of control statement of node>

C.4 Service control

<list of shared variables in service>

<information of service control>

<facility routines of service>

<node/service routine of service>

<service/utility routine of service>

<initialization routine of service>

<service body> := <initialization routine of service>

[<statement list of the service>]

<infinite loop of control statement of service>

C.5 Utility control

<list of shared variables in utility>

<information of utility control>

<facility routines of utility>

<service/utility routine of utility>

<utility request routine of utility>

<normal termination routine of utility>

<abnormal termination routine of utility>

<initialization routine of utility>

<utility body> := <initialization routine of utility>

[<statement list of the utility>]

<normal termination routine call of utility>

C.6 Process control

<information of process control>

<facility routines of process>

<node /process routine of process>

<utility request routine of process>

<message passing routine of process>

<normal termination routine of process>

<abnormal termination routine of process>

<temporary termination routine of process>

<waiting for message routine of process>

<initialization routine of process>

<process body> := <initialization routine of process>

[<statement list of the process>]

<normal termination routine call of process> C.7 Utility call <disable alarm facility>

<check validity of utility call>

<send the argument list to utility>

<issue the actual utility call>

<receive the returned values from the utility>

<enable alarm facility>

C.8 'Messin' statement <disable alarm facility>

<find the internal representation of message address>

<check the arrival of messages>

[<infinite loop to handle the waiting for message>]

<enable alarm facility>

C.9 'Messout' statement <disable alarm facility>

<find the internal representation of message address>

<stores the content of message to the message buffer>

<send the destination address to the node control>

<enable alarm facility>

References

1. 'An Experiment in Language Design for Distributed Systems', by D. Crookes and J.W.G. Elder, Software-Practice and Experience, Vol. 14(10), P. 957-971, Oct. 1984.

2. 'A Language for Distributed Processing', by Ronald J. Price, (National Computer Conference), P. 957-967, 1979

3. 'Bruwin: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems', by Norman Meyrowitz Margaret Moser, 8th Symposium on O/S Principles, ACM, Dec. 1981

4. 'Japan's Fifth-generation Computer Systems', by Philip Treleaven, Computer, IEEE, p. 79-88, August 1982

5. 'The Programmer's Apprentice: Knowledge Based Program Editing', by Richard C. Waters member, IEEE, Transactions of Software Engineering, Vol. SE-8, No. 1, P. 1-12, January 1982

6. 'The Versatility of PROLOG', by L. Baxter, Department of Computer Science, York University, Downsview, Ontario M3J, P3. ACM SIGPLAN Notices, Vol. 15, No.12, P.15-16, Dec. 1980

7. 'PROLOG: A step toward the Ultimate Computer Language', by Ron Ferguson, 137 University AVE W, Apt 907 Waterloo, Ontario N2L 3E6 Canada, BYTE, Vol. 6, No 11, P. 384-392, Nov. 1981

8. 'Advances in Computers', edited by Marshall C. Yovits, Vol. 20, P. 199-259, 1981

9. 'Can Programming be liberated from the Von Neumann Style? A Functional style and its algebra of programs', by John Backus, IBM Research Lab., P. 613-641

10. 'Distributed Processes: A Concurrent Programming Concept', by Per Brinch Hansen, ACM, Vol. 21, No. 11, P. 934-941, Nov. 1978

11. 'Programming by Refinement, as exemplified by the SETL Representation Sublanguage', by Robert B. K. Dewer, Arthur Grand, Siu-Cheng Liu, and Jacob T. Schwartz, ACM Transactions on Programming Language and Systems, Vol. 1, No. 1, P.84-97, July 1979

12. 'A new approach to proving the correctness of Multiprocess Programs', by Leslie Lamport, ACM Transactions on Programming Language and Systems, Vol. 1, No. 1, P. 84-97, July 1979

13. 'Report of Session on Concurrency', by Jack Dennis, SIGPLAN notices, Vol. 8, No. 9, Sep. 1973

14. 'A data flow language for O/S programming', by Paul R. Kosinski, SIGPLAN Notices, Vol. 8, No. 9, P. 89-93, Sep. 1973

15. 'Application of Extensible Language to Specialized Application Language', by Jean E. Sammet, SIGPLAN Notices,

Vol. 6, No. 12, P. 141-143, Dec. 1971

16. 'Lucid, a Nonprocedural Language with iteration', by E. A. Ashuoft, W. W. Wadge, CACM, Vol. 20, No. 7, P. 519-526, July 1977

17. 'History of Programming Language', edited by Richard L. Werelblat

18. 'High-level programming for Distributed Computing', by Jerome A. Feldman, CACM, Vol. 22. No. 6, P. 353-357, June 1979

19. 'Concurrent Programming Concepts', by Per Brinch Hansen, Computing Surveys, Vol. 5, No. 4, Dec. 1973

20. 'Programming Language Design and Implementation', by Terrence W. Pratt, P. 314-509

21. 'The Programming Landscape', by Henry Legard Michael Marcotty, P. 129-146, P. 247-266, p. 377-405

22. 'Function-level computing, Automating programming, Programming for non-programmers', Computer Software I, II, and III. IEEE Spectrum, p. 22-38, August 1981

23. 'Smalltalk-80 System', BYTE, August 1981

24. 'A sampler of Formal Definitions', by Michael Marcotty and Henry F. Ledgard, Computing Surveys, Vol. 8, No. 2, P. 191-276, June 1976

25. 'Programming Language - the first 25 years', by Peter Wegner, IEEE Transactions on Computers, P. 10-28, Dec. 1976

26. 'The Distributed Programming Language SR-Mechanisms, Design and Implementation', by Gregory R, Andrews, Software-Practice and Experience, Vol. 12, P. 719-753, 1982

27. 'A Comparative Survey of Concurrent Programming Languages', by Paul Scotts. Jr., ACM SIGPLAN Notices, Vol. 17, No. 10, Oct. 1982

28. 'Natural language Communication with Computers', edited by G. /goos and J. Hartmanis

29. 'Formal specification of Programming Languages: A Panoramic Primer', by Frank G. Pagan

30. 'The programming language Concurrent Pascal', by Per Brinch Hansen, IEEE Transactions on Software Engineering, P. 199-207, June 1975

31. 'The architecture of Concurrent Programs', by Per Brinch Hansen, 1977

32. 'Modula: A language for Modular Multiprogramming', by N. Wirth, Software-Practice and Experience, Vol. 7, P. 3-35, 1977

33. 'Communicating Sequential Processes', by C. A. R. Hoare, Communication ACM, 21, P. 666-677, 1978

34. 'Guarded commands, Nondeterminacy and Formal Derivation

of Programs', by Edsger W. Dijkstra, Communication ACM, P. 453-457, August 1975

35. 'Monitors: An Operating System Structuring Concept', by C. A. R. Hoare, Communication ACM, P. 549-557, Oct. 1974

36. 'Synchronizing Resources', by Gregory R. Andrews, ACM Transactions on Programming language and Systems, Vol. 3, No. 4, P. 405-430, Oct. 1981

37. 'The ROSCOE Distributed O/S', by Marvin H. Solomon and Raphael A. Finkel, ACM 1979

38. 'GYPSY: A language for specification and implementation of verifiable programs', by Allen L. Ambler Amdabl Corporation, ACM SIGPLAN, P. 1-10, March 1977

39. 'Access Control in Parallel Programs', by James R. Mcgraw and Gregory R. Andrews, IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, Jan. 1979

40. 'The Design and Implementation of Modula', by N. Wirth, Software-Practice and Experience, Vol. 7, P. 67-84, 1977

41. 'An Assessment of Modula', by J. Holden and I. C. Wand, Software-Practice and Experience, Vol. 10, P. 593-621, 1980

42. 'Edison - a Multiprocessor Language', by Per Brinch Hansen, Software-Practice and Experience, Vol. 11, P. 325-361, 1981

43. 'Abstraction and Verification in Alphard: Defining and

Specifying iteration and Generators', by Mary Shaw and William A. Wulf, Communication ACM, P. 553-564, August 1977

44. 'Extending Concurrent Pascal to all Dynamic Resource Management', by Abraham Silberschatz Richard B. Kieburtz, IEEE Transactions on Software Engineering, Vol. SE-3, No. 3, May 1977

45. 'Capability managers', by Richard B. Kieburtz, IEEE Transactions on Software Engineering, Vol. SE-4, No. 6, Nov. 1978

46. 'The Soma: A programming construct for Distributed Processing', by Joep L. W. Kessels, IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, Sep. 1981

47. 'The design of Ada', --- Requirements for High Order Computer Programming Language - "Steelman"

48. 'DoD's Common programming language effort', by David A. Fisher, Computer, P. 24-33, March 1978

49. 'Programming with Ada: An introduction by Means of Graduated Examples', by Peter Wagner, ACM SIGPLAN Notices, P. 1-46, Dec. 1979

50. 'On the design of a Distributed Operating System using a High Level Distributed Programming Language', by R. K. Arora and N. K. Sharma, North-Holland Publishing Company, Microprocessing and Microprogramming 10, P. 247-254, 1982

51. 'Concurrency in ADA and Multicomputers', by N. H. Gehani, Computer Language, Vol. 7, P. 21-23, 1982

52. 'Language constructs for Real-Time Distributed Systems', by D. M. Berry, C. Ghezzi, D. Mandrioli, Computer Language, Vol. 7, P. 11-20

53. 'DPL-82: A language for Distributed Processing', by Lars Warren Ericson, Carnegie-Mellon University

54. 'A General-Purpose Algorithm for Analyzing Concurrent Programs', by Richard N. Taylor, Communication of ACM, Vol. 26, No. 5, P. 362-376, May 1983

55. 'A Survey Note on Programming Languages for Distributed Computing', by Abraham Siberschatz, Department of Mathematical Sciences, The University of Texas, 21st IEEE Computer Society International Conference, P. 719-722, Sep. 23-25, 1980

56. 'A Language Model for fully Distributed Systems', by Arthur B. Maccabe and Richard J. Leblanc, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, P. 723-728, Sep. 23-25, 1980

57. 'The Starmod Distributed Programming System', by Robert P. Cook, Computer Science Department and Mathematics Research Center, University of Wisconsin-Madison, P. 729-735, Sep. 23-25, 1980

58. 'CSP/80: A Language for Communicating Sequential

Processes', by Mehdit Jazayeri, Carlo Ghezzi, Dan Hoffman, David Middleton, Mark Smotherman, P. 736-740, Sep. 23-25, 1980

59. 'A Communicating Sequential Process Language and Implementation', by T. J. Roper and C. J. Barter, Department of Computer Science, The University of Adelaide, the Australian Computer Journal, P. 17-27, Vol. 15, No. 1, Feb. 1983

60. 'Communications Policy for Composite Processes', by C. J. Barter, Department of Computer Science, The University of Adelaide, the Australian Computer Journal, P. 9-16, Vol. 15, No. 1, Feb. 1983

61. 'A Comparison of the Concurrency Constructs and Module Facilities of Chill and Ada', by C.J. Fidget and R. S. V. Pascoe, the Australian Computer Journal, P. 17-27, Vol. 15, No. 1, Feb. 1983

62. 'LOFE: A Language for Virtual Relational Data Base', by J. K. Debenham and G. M. McGrath, the Australian Computer Journal, P. 2-8, Vol. 15, No. 1, Feb. 1983

63. 'Is Block Structure Necessary?', by David R. Hanson, Department of Computer Science, The University of Arizona, Tucson, Software-Practice and Experience, Vol. 11, No. 8, P. 853-866, Sep. 1980

64. 'Evaluating Synchronization Mechanisms', by Toby Bloom, MIT Laboratory for Computer Science, ACM, Proc. of the 7th

Symposium on O/S Principles, P. 24-32, Dec. 1979

65. "Primitives for Distributed Computing', by Barbara Liskov, MIT Laboratory, ACM, Proc. of the Symposium on O/S Principles, P. 33-42, Dec. 1979

66. 'Concepts and Notations for Concurrent Programming', by Gregory R. Andrews, Department of Computer Science, University of Arizona, Computing Surveys, Vol. 15, No. 1, P. 3-43, March 1983

Publications

1. "GDPL-A Generalized distributed programming language." by Ng, Kam-wing and Li, Wai-kit, In The International Conference on Distributed Computing Systems (U.S.A), p. 69-78, 1984.5.

2. "GDPL-A language for programming distributed systems." by Ng, Kam-wing and Li, Wai-kit, In The First International Conference on Computers and Applications (U.S.A), p. 76-83, 1984.6.