Software Design Language for Concurrent Distributed Systems

by

Yahya Mohamed Zeidan Mustafa

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January, 1992

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1354049

Software design language for concurrent distributed systems

Mustafa, Yahya Mohamed Zeidan, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

SOFTWARE DESIGN LANGUAGE FOR CONCURRENT DISTRIBUTED SYSTEMS

BY

YAHYA MOHAMED ZEIDAN MUSTAFA

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JANUARY 1992

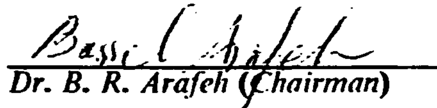
381

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

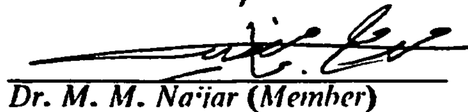
COLLEGE OF GRADUATE STUDIES

This thesis, written by ***YAHYA MOH'D ZEIDAN MUSTAFA*** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of ***MASTER OF SCIENCE*** in ***INFORMATION AND COMPUTER SCIENCE***

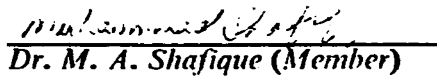
THESIS COMMITTEE



Dr. B. R. Arafeh (Chairman)



Dr. M. M. Najjar (Member)



Dr. M. A. Shafique (Member)

Department Chairman

Dean, College of Graduate Studies

To sweet memory of my father and beloved mother

ACKNOWLEDGEMENT

Acknowledgement is due to *King Fahd University of Petroleum and Minerals* for providing the opportunity to carry out this research work.

I would like to express my deep appreciation and gratitude to *Dr. Bassel Arafeh* who served as the chairman of the thesis committee and to the committee members *Dr. Mamdouh Najjar* and *Dr. Mohamed Shafique* for being so patient to go through the thesis and providing valuable comments and suggestions.

Lastly but not least, let me mention my sincere thanks and appreciation to *Mr. Hassan Al-Yousef* and *Mr. Moayyad Shukri* for their continuous encouragement and morale support and to all my friends and colleagues who have been a constant source of support.

CONTENTS

Chapter *Page*

ACKNOWLEDGEMENT .... **iv**

LIST OF FIGURES .... **x**

LIST OF TABLES .... **xi**

ABSTRACT (English) .... **xii**

ABSTRACT (Arabic) .... **xiii**

1. INTRODUCTION

1.1 Objective of This Study 1

1.2 What is a Software Design Language 1

1.3 Thesis Organization 4

2. LITERATURE SURVEY

2.1 Design methodologies 5

2.1.1 Data-Driven Design 6

2.1.2 Top-Down Functional Decomposition 6

2.1.3 Object-Oriented Design 6

2.1.4 Other Design Methodologies 7

2.2 An Overview of Software Design Languages 9

2.3 Distributed Software Design Methodologies 23

2.4 Distributed Computing Systems 25

2.5 Reasons for the Interest in Distributed Computing Systems 27

2.6 Languages for Programming Distributed Computing Systems 29

2.7 Primitives for the Support of Designing Distributed Software 32

2.7.1 Expressing Parallelism 32

2.7.2 Synchronization 33

2.7.3 Communication 34

2.7.4 Nondeterminism 37

2.7.5 Time 38

2.7.6 Fairness 39

2.7.7 Failures 40

3. LANGUAGE DEFINITION

3.1 Introduction 41

3.2 Syntax Notations 41

3.3 Overall Structure: Modules & Processes 41

3.2.1 Module Structure General Form 42

3.2.2 Process Structure General Form 45

3.4 Lexical Style 46

3.4.1 Lexical Elements 46

3.4.2 Identifiers, Literals, and Comments 46

3.5 Declarations of Classes and Objects 47

3.5.1 Module-type and Process-type 47

3.5.2 Message-type 52

3.5.3 Access-type 54

3.5.4 Array-type and Record-type 54

3.6 Communication Primitives 55

3.6.1 Definition of the Communication Connection 55

3.6.1.1 Entries 55

3.6.1.2 Ports 56

3.6.1.3 Direct Naming 58
3.6.2 Message Reception Statements 59
3.6.2.1 Receive Statement 59
3.6.2.2 Enter Statement 60
3.6.3 Message Transmission Statements 61
3.6.3.1 Send Statement 61
3.6.3.2 Call Statement 62
3.7 Indeterminacy Control 63
3.7.1 Select Statement 63
3.7.2 Selective Loop Statement 64
3.7.3 Delay Statement 64
3.7.4 Exit Statement 65
3.7.5 Terminate Statement 65
3.7.6 Abort Statement 66
3.8 Constructs for Exception Handling 67
3.8.1 Exception and Signals Types 67
3.8.2 Raise Statement 68
3.8.3 Announce Statement 68
3.8.4 Handle Statement 69
3.8.5 Exception Handling Part 70
3.9 Language Constructs for the Support of the Software Design Description and Refinement Process 71
3.9.1 Null Statement 71
3.9.2 Deffered Statement 71
3.9.1 Assignment Statement 72

4. DSDL PROCESSOR

4.1 Introduction 73

4.2 Lexical Analyzer 73

4.3 Syntax Analyzer (Parser) 74

4.4 Error Recovery 78

4.5 Counter Examples 82

5. CONTRASTS AND COMPARISONS

5.1 Introduction 115

5.2 An Overview of Concurrent Message Passing Models 115

5.2.1 Communicating Sequential Processes (CSP) 116

5.2.2 Programming Language In The Sky (PLITS) 116

5.2.3 Concurrent C 117

5.2.4 Synchronizing Resources (SR) 118

5.2.5 Ada 119

5.3 Comparisons 120

5.3.1 Dimensions of Distribution 120

5.3.2 Goals and Structure 120

5.3.2.1 Problem Domain 120

5.3.2.2 Expressing Parallelism 120

5.3.3 Communication 121

5.3.3.1 Synchronization 121

5.3.3.2 Buffering 122

5.3.3.3 Information Flow 122

5.3.3.4 Communication Control 123

5.3.3.5 Communication Connection 124

5.3.4 Other Issues: Time, Fairness, and Failure 124
5.3.4.1 Time 124
5.3.4.2 Fairness 125
5.3.4.1 Failure 125
5.4 Concluding Remarks 126

6. CONCLUSION & FUTURE WORK **128**

BIBLIOGRAPHY & REFERENCES **131**

APPENDIX A : BNF Grammar **137**

THESIS ABSTRACT

NAME OF STUDENT: YAHYA MOH'D ZEIDAN MUSTAFA

TITLE OF STUDY : SOFTWARE DESIGN LANGUAGE FOR CONCURRENT DISTRIBUTED SYSTEMS

MAJOR FIELD : INFORMATION AND COMPUTER SCIENCE

DATE OF DEGREE : JANUARY 1992

A Software Design Language is the formal means of expressing the design decisions and a tool to support the design phase of software development cycle. This work is concerned with the design of such a language, called Distributed Software Design Language (DSDL), for use in designing distributed computing systems. DSDL provides capabilities to support communication primitives in different ways, giving the designer the ability to choose the communication primitives appropriate to his problem and its implementation. The computational model is expressed as modules and processes that communicate through message passing. DSDL provides constructs for handling inherent features in distributed environment, such as communication mechanisms, synchronization, message transmission/reception constructs, exception handling mechanisms, modules/processes creation and termination, and indeterminancy control. A DSDL language processor has been implemented which works as a basis for a distributed software design tool.

MASTER OF SCIENCE DEGREE

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Dhahran, Saudi Arabia

January 1992

خلاصة الرسالة

اسم الطالـــب : يحيى محمد زيدان مصطفى

عنوان الرسالة : لغة التصميم البرمجية للنظم الآنية والموزعة

التخصـــــص: علوم الحاسب الآلي والمعلومات

تاريخ الشهادة : كانون الثاني (يناير) ١٩٩٢ م

تعتبر لغة التصميم البرمجية الوسيله الاعتيادية للتعبير عن قرارات التصميم كما وتعتبر اداة لدعم مرحلة التصميم في دورة تطوير البرامج . يقدم هذا البحث لغة تصميم برمجية لدعم مرحلة تصميم النظم الحاسوبية والموزعة وتزود هذه اللغة المصمم بامكانية اختيار اساسيات الاتصالات المناسبة للنظام المراد تصميمه وبنائه بطرق مختلفة . وتتكون اللغة من وحدات مختلفة تتجاوب مع بعضها من خلال الرسائل . وتشمل لغة التصميم البرمجية المقترحة تراكيب لمعالجة الخواص الاساسية للنظم الموزعة مثل آليات الاتصال والتزامن وارسال واستقبال الرسائل ومعالجة الحالات الخاصة وبدء وانهاء الوحدات وكذلك التحكم غير المحدد . وقد تم خلال هذا البحث بناء معالج يعمل كنواة لأداة تصميم البرمجة الموزعة .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران - السعودية

كانون الثاني (يناير) ١٩٩٢ م

CHAPTER I

INTRODUCTION

1.1 What is a Software Design Language

The heart of software engineering project is the design [1]. Software design is the process which translates the requirements into a detailed design representation. Good software design is the key to produce reliable and understandable software [5]. Historically, software design was very much an "ad hoc" process. Given a set of requirements, usually in natural languages , an informal design is prepared , often in the form of a flowchart. Coding then commenced and the design is modified as the system is implemented. When the implementation phase was complete, the design had usually changed so much from its initial specification that the original design document is an inadequate description of the system. This approach to design was responsible for many project failures.

Lately, it was realized that notations such as flowcharts that are close to implementation language are inadequate means of expressing system design [2]. Several methods have been developed to express system design [3,4,5], such as data flow diagrams, HIPO (hierarchy plus input process output chart), and software design description languages. Design languages are the formal way of expressing design since they have much in common with current programming languages, and they are specifically intended for documenting and

communicating software design.

Software design language (SDL) is a language with special constructs used to develop, analyze, and document a design [2]. When one starts writing down a design in an SDL the important design decisions have already been taken and the major modules of the system have been specified using a design methodology. Also, a design notation has probably been used to depict on paper an overview of the design. An SDL helps to write it down, to analyze it, and get to the implementation of the decisions already taken. The need for SDLs also emerged from the fact that software design is an iterative process with the possibility that details discovered in the later stages of design may lead to modifications in previous portions. Since the SDL is easier to change than actual code, cleaning up a program design in SDL is usually more cost-effective than cleaning up program code [18]. It should be noted that an SDL does not substitute any design methodology. An SDL is not a programming language from the point of view that it does not require very strict constructs, although it evolves during the implementation phase into a programming language [2].

Software design is a process through which requirements are translated into a representation of software. Initially the representation depicts a holistic view of software. Subsequent refinements lead to a design representation that is close to source code in procedural detail. The former representation is accomplished during the architectural design stage, while the latter during the detailed design stage. The architectural design phase is responsible for decomposing requirement specifications to form a system structure. It emphasizes the module-level system

representations which can be evaluated, refined, or modified in the early software development process. The detailed design phase is responsible for transforming the system structure produced by the architectural design phase into a procedural description of a software system. This phase emphasizes the selection and evaluation of algorithms to implement each module. At this phase, all the details and decisions of each module are well defined. The language we introduce is not an implementation langguage; it is a step before the actual code.

1.2 Objective of the Study

Due to the advances in communication technology and the increasing role of computers in our daily life, software designers are being challenged with large and critical requirements for developing distributed computing systems. A distributed system adds new dimensions for designing software that has to be supported with concurrency control, time-dependencies, synchronization, reliability and security, and high performance. Furthermore, distributed computing occurs in many application areas such as operating systems, databases, real-time systems, artificial intelligence, and robotics.

For the target Distributed Software Design Language (DSDL), the dimensions of all required primitives must be supported with constructs. Each construct should be flexible and powerful enough to express design decisions freely. A DSDL should provide capabilities to support a primitive in different alternative ways, giving the designer to choose the approach or mechanism appropriate to his problem and its implementation. For this reason, our approach does not favor an SDL based on one of the concurrent programming

languages such as Ada, Concurrent C, CSP, ...etc.

In this study, we emphasize the role of software design languages as a vehicle to design distributed software. In this regard, we introduce DSDL (Distributed Software Design Language) and the design issues relating to selecting language (linguistic) constructs to support distributed computing. Also, we introduce a processor that checks the syntax of programs that will be written in the language. The objective of DSDL is to provide support for the detailed design phase of distributed software applications development.

1.3 Overview of the Thesis

Chapter 2 will review software design languages, design methodologies, distributed computing systems classification, reasons for the interest in distributed computing systems, languages for programming these systems, and finally, the primitives required for designing a distributed software.

Chapter 3 will present DSDL computational model, the specification of software components as modules and processes, the general formats of modules and processes, the communication mechanisms, synchronization, message transmission/reception constructs, and constructs to support exception handling.

Chapter 4 will explain DSDL processor design, and illustrates some examples written in DSDL.

Chapter 5 will review some message passing models, compare among them including DSDL. Finally, in chapter 6, the conclusion and recommendations which can be carried out as future work are proposed.

CHAPTER II

LITERATURE SURVEY

2.1 Design Methodologies

during the past two decades or so, the computer scientists have realized that the development of software must be upgraded from an art to an engineering science. As a result, methodologies such as structured programming and object-oriented programming have emerged and been developed. Additionally, many practitioners have begun to think in terms of how to design software (as opposed to how to code programs), and several software design methodologies now exit [24] .

Much progress has been made in software design techniques since the mid-1960's. Various systematic approaches have been developed for software design in order to produce reliable and maintainable software systems. These systematic ways to software design are actually a process which translates software requirements into a detailed representation [5] .

There is no single 'best' methodology, and all methodologies work well in some situations and for some classes of application. However, no one methodology is ideal for all situation, and the designer should not concentrate on any methodology and exclude the others. Furthermore, large software systems encompass sub-application areas where many different types of methodologies are applicable. Thus, in the design of large systems, the software

designer may make use of different methodologies for different sub-systems. Next, we will highlight on design methodologies.

2.1.1 Data-driven Design

This approach to design of software systems was first proposed by Jackson in 1975 and Warnier-Orr in 1977 [25]. The basis of this methodology is that the data processed by a system have an inherent structure. This should be reflected in the structure of the programs that are involved in the processing of the data.

2.1.2 Top-down Functional Decomposition

This approach to software design assumes that the system may be represented as a hierarchy of functions. It is essentially based on divide-and-conquer strategy for solving software design problems [5]. The level of detail expressed in any particular function is dependent on the level of that function in the hierarchy. The top-most functions are fairly abstract whilst those at the lowest levels deal with concrete system details. By contrast, a bottom-up approach to design tries to identify useful functions at lower levels in the function hierarchy and defines higher level functions using these components

2.1.3 Object-Oriented Design

This approach to software design is based on the concept of information hiding and abstract data type [5]. It considers the software system to be made

up of a set of objects which interact by passing messages to each other. Each object has its own internal state and the messages passed to an object determine what operations on that state to be initiated [26].

2.1.4 Other Design Methodologies

In addition to the previous methodologies, there are other approaches used in designing the software systems. An overview of these approaches is the following [5]:

* ***Modular Programming Approach***

This approach divides a complex system into several modules. Each module performs a single function. After all modules are coded and tested, then they are integrated and the whole system is tested.

* ***Data Flow Design Approach***

This approach uses information flow as a driving force for software design process. It uses various mapping functions to transform information flow into software structure.

* ***HIPO* : (*hierarchy, plus input, process, output*)**

This approach is developed by IBM, consists of a set of diagrams to represent input, output, and process. It decomposes the system in a hierarchical way without involving logic details.

* *Graphical Representation Approaches*

Flowchart is the oldest and widely used since it is simple and provides visual aid. For each program structure, there is a corresponding graphic chart. Nassi-Shneiderman Diagram was developed to support structured programming. There is special graphical symboles to represent sequence, for, if, case, and goto statements to describe a program. It provides visual aid and can represent recursion easily.

2.2 An Overview of Software Design Languages

The work that has been done in the field of software design language resulted in two groups. The first group, such as SLAN-4 [6], Flex [7], Gamma [8], and ADAPT [9], is not based on any existing high-level programming languages. SLAN-4 is a specification and design tool, Flex and ADAPT are software design toolsets, and Gamma is an integrated design system. The second group is based on existing high-level languages such as APL, COBOL, and Ada [2].

Most of early efforts on SDLs are design languages accompanied with a language processor for producing handy output of the design [2]. These languages have in common the support of some control constructs as well as provisions for the inclusion of pseudocode. The pseudocode acts both as program stubs to be refined at a lower level of design and as design documentation.

Program Design Languages (PDLs) are actually a form of pseudocode for procedural design description. They followed the introduction of many programming techniques such as structured programming, and top-down functional design. PDLs do not provide a powerful mechanisms for the design of user-defined data structures, as the more SDLs do. The PDL developed at Caine, Farber and Gorden Inc. [2], is a mixture of English and of some constructs that relate directly to the constructs of structured programming. A design product in this PDL consists of a set of flow segments that correspond roughly to a procedure in the final implementation, and a set of text segments

that contain textual information such as data formats, assumptions, and constraints. It works equally for both large and small projects . The PDL of Mc-Donnel Douglas Automation Company [2] was used to teach structured programming to programmers. It is followed by a design methodology that encourages different applications such as indentation rules and application-oriented names. An example of a PDL is given in Figure 2.1 [2] , looking at the example, one sees that the statement CREATE A PLACE FOR IT can be expanded into more statements getting into more detail. The PDL processor automatically underlines keywords (IF, DO, etc.), indents statements to correspond to structure nesting levels, provides automatic continuation from line to line, and provides a document of the design.

The main virtue of PDLs is that a holistic view of an entire problem solution can be quickly and easily constructed. This level of design can be easily understood by people other than the designer, and can be easily validated. Thus, criticisms, suggestions, and modifications can be quickly accommodated into the design possibly leading to complete rewrites of major sections. When the design stabilizes at this level more detail can be added in successive passes through the design with decisions at each point affecting smaller and smaller areas each time. The PDLs are independent of the implementation language, their constructs have a relation with a family of high-level languages which might be used in the implementation phase [2] .

Pseudo Language (PL) [2] is the next effort on SDLs and is accompanied by PL processor and an associated library of implemented functions (verbs).

PL is still a kind of "Structured English" , but has more formal characteristics than the previous PDLs since its constructs are Pascal-like. It is the first design language to include constructs such as the cobegin , coend for concurrent programming. The syntax of PL is rather formal. PL program is divided into the introduction and the body

< body > = = > BEGIN < statement list > END

Figure 2.2 shows the PDL example of Figure 2.1 in PL program form. The P processor maintains all of the verbs (functions) which are implemented. For every such verb the calling program form command is transformed properly; for example, the command FIND could be transformed into :

CALL FIND(PLACE, PRODUCT-CODE, STORED-PRODUCTS)

Thus, every useful library can be obtained containing many program forms. The processor also finds syntax errors, produces cross-reference tables, detects misuse in variables, checks parameter passing, and produces indented listing using the parse tree. PL has all of advantages of PDLs, and it allows more checking in its program forms and opens a path for reusable designs. A drawback of its processor is that the programmer may use a verb that is already implemented and contained in the library under different name [2] .

Schematic Pseudocode (SPC) [2] is a more recent SDL aiming at representing the program control flow schematically and introducing a systematic, top-down approach in the documentation of source code. A program in SPC evolves from pseudocode to target source language through a stepwise

refinement process. During all of the phases of this evolution, the control flow constructs (sequential, conditional, and repetitive) are represented using graphic symbols that make the structure of the program immediately perceptible. Documentation also evolves through operational comments; the design becomes more detailed by expanding operational comments. This process continues until operational comments become easily implementable in any procedural target language [2] .

The SPC syntax allows three control constructs: sequential, conditional, and repetitive. An example of a sequential construct is given in Figure 2.3(a). The narrative comment describes the function of a portion of the design. The assignment statement is expressed in symbolic notation, preferably in the target programming language. The operational comment is preceded by a reference number and denotes an operation that will be specified later on. When this operation is designed, the operational comment becomes a narrative comment [2] . An example of conditional construct is given in Figure 2.3(b). This analogous to :

IF B1 THEN S1 ELSE IF B2 THEN S2 ELSE S3.

Finally, an example of a repetitive construct is given in figure 2.3(c).

SCHEMACODE is the software tool which supports SPC. It provides two outputs : a description of the program in SPC and the source program in the generalized source code. SPC uses visual notation of program semantics that makes designs as well as source code programs easily perceptible, but it does not

introduce any other formalism in the design process.

The advent of the second generation SDLs [6,7,8,9] was the result of accumulated experience and advances in software engineering. Most of these SDLs have emerged as toolsets or design systems that intend to automate the design phase. Some produce intermediate code to enhance the portability of the designs, while some others produce executable designs in an implementation language. They all produce checkable designs, provide mechanisms for abstraction, enforce modularization, and information hiding.

SLAN-4 was developed in 1980 [6] due to advancement of research in software specification and the influence of design methodologies. It allows the designer to introduce the information hiding concepts of Parnas [10] and the stepwise refinement [2] of a given design. Its specification part describes what are the intended functions of a module in a formal and clear way, whereas the pseudocode part of the language describes how these functions are to be achieved. Thus, SLAN-4 is both a specification and a design tool [6].

The basic constructs of SLAN-4 are classes and modules. Classes consist of a collection of modules. A class defines a data type, i.e. its structure composed of basic types together with all operations allowed on objects of this type. Thus, a class represents an abstract data type while a module represents an operation acting on the data type. Whereas classes are used to group several operations together, modules describe what will be perceived as a single action [6]. A class description follows the syntactical form [6]:

```
class-name : CLASS
interface-declaration
class-specification
declaration
{ class | module }*
statements
END-CLASS  class-name
```

A module has the following syntactical form [6] :

```
module-name : MODULE
interface-declaration
module-specification
declaration
{ class | module }*
statements
END-MODULE  module-name
```

The pseudocode part of SLAN-4 has been designed to offer a way of presenting algorithms independent of the language in which the final program is to be written. SLAN-4 is a language which can be used as a software specification, design, communication, and documentation tool. The most important aspects of SLAN-4 are the abstract data types, and the axiomatic specification of modules.

With its various control constructs, predefined data types, and the user defined types, SLAN-4 enables designers to introduce rigor and formalism in the design.

Flex was developed in 1979 by the Naval Research Laboratory and the university of Maryland [7]. A program in Flex consists of three kinds of segments : data segment, definition segment, and routines. Data segments contain shared data that can be accessed by routines. Definition segments contain type and operator definitions. There are two kinds of routines : functions that are value-returning and procedures. There are two special types of functions : access functions that return by reference and iteration functions that define the manner in which the elements of a certain data structure are to be traversed during an iteration loop (control abstraction). Routines are not block structured, they can access data residing in data segments in the programming system, but not to internal data in other routines. Routines can be recursive and generic [7].

Flex is a very flexible language. It can take the form its users want. It enforces information hiding, providing the ability of type and control abstraction. User-defined operators is a notable characteristic. The job of the designers is helped by reusable modules that can be kept in a data base and which can be queried for any information. Flex language is not executable; it is only checkable by the processor. So very powerful tools designed in it can be quite inefficient if translated into target language. The Flex system processor has been observed to be relatively slow [7].

Gamma software engineering system is an integrated software design system developed in 1981 [8]. The main characteristic of Gamma is abstract modularization. An abstract module is a module that corresponds to an abstract construction (i.e data structure, algorithms, etc.) of software design. In the Gamma system, there are four types of abstract modules : procedure, group, class, and process. A procedure is similar to procedures in high-level languages, except that it may be eventually implemented as a macro, task, program, or subroutine. A group is a group of modules. A class is a group of modules which defines a data type with its operation. A process encapsulates the idea of a process running independently but exchanging data with other processes [8].

The heart of Gamma system is a database that contains all of the abstract modules for one or more system developments, together with libraries of useful components and designer's work in progress. A problem with Gamma is that it encourages the designer to break down a design into very small procedures that could lead to an acceptably slow program. To solve this problem, Gamma enables the designer to decide which procedures are to be implemented by subroutines and which to be implemented by in-line expansion [8].

Abstract Design And Program Translator (ADAPT) is a set of tools for the design and development of software systems [9]. ADAPT makes a distinction between the design of overall structure of the software system and the design of each specific module. ADAPT designs are transformed to an executable program by a translator. The translator performs strong type checking and verifies that all source module references are consistent with

specifications and declarations within the module. In ADAPT there are three kinds of modules : capsules, procedures, and iteratores [2] .

A capsule is analogous to the class of SLAN-4 and the class in Gamma. It specifies a type abstraction, and consists of an internal data representation and operations that manipulate this representation. A capsule defines which encapsulated modules are exported and its internal data representation. It can take a type as parameter so as to represent a generic data type. A procedure is similar to procedures in high-level languages, but parameters may be user-defined data types, in addition to data types provided by the language itself. An iterator is a control abstraction that is used to produce the elements of a data type so that other procedures do not have to care about the internal structure of data abstraction [9] .

ADAPT provides executable designs via its translator to PL/1. This encourages prototyping and improves design correctness. ADAPT supports the separate compilation of modules and its code tends to have many procedure calls.

All of the reviewed SDLs were designed to be used during the design phase of software life cycle. There are some high-level languages that are proposed to be used as software design languages such as Ada [11] , APL [12] , and COBOL [2] .

Ada is a language that embodies many modern software design methodologies , and a powerful tool to express programming solutions [11] .

Ada provides two forms of abstraction : data abstraction and type abstraction [17]. Data abstraction may have two forms : data encapsulation and structural abstraction. The mechanism for data abstraction in Ada is the package [17]. Type abstraction is provided by the use of generic units. Ada enforces the use of information-hiding principle, supports concurrency via the task construct, supports error handling via the exception mechanism, allows dynamic declarations, and initializations of objects in declaration, and permits identical names for subprograms which are then distinguished on the basis of their parameters [11,13,17]. The features of Ada has led to its use as SDL in three distinct methodologies [13]: bottom-up incremented, top-down semi-incremental, and large-small traditional programming. These methodologies use the " with " and " separate " clauses of Ada, the package construct, and the ability of separate compilation.

Since Ada is mainly an implementation language, the question is whether this language should be used as an SDL in the form it actually is or in a modified form. Different methods have been proposed to use Ada as an SDL in a modified form. For instance, the method, developed by IBM, suggests using a subset of Ada as an SDL, another method, developed by Heart at TRW, proposes the use of a freer form of an Ada SDL that relaxes some of syntactic rules [2].

APL or an extension thereof has been proposed to be used as an SDL. APL is suited for the early stages of design [12]. APL is mathematically amenable, highly symbolic, and expressions are easy to write without temporary

storage variables as well as without bothersome loop indicies on array variables. The advantages of APL are its functional orientation for specifying design functions and its mathematical amenability for proof of correctness techniques. Its most obvious disadvantage is the fact that the sheer power of the language is much more susceptible to abuse.

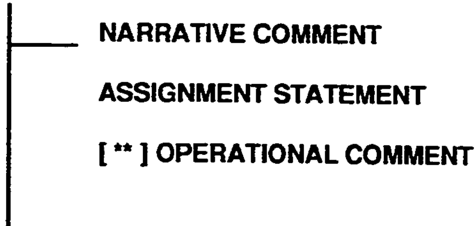
This survey gives a clear picture about software design languages. We recognize that these languages helped to solve many design problems, since, they support strong typing, declaration of data objects of both standard and user-defined types, abstraction, information hiding and modularity. But these languages do not support enough effective features for the design and development of concurrent and distributed computing systems.

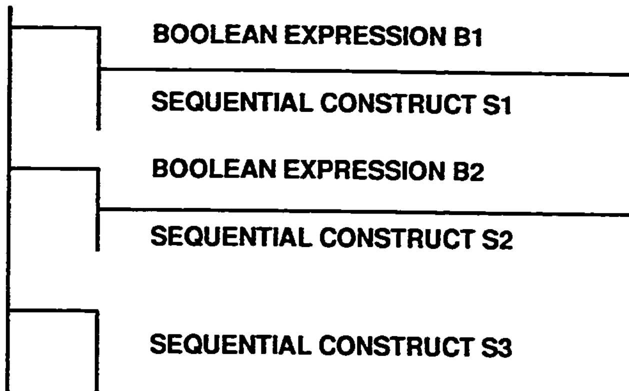
```
DO WHILE THERE ARE NO MORE ITEMS

  DO CASE OF ITEM-CODE

       RAW:   INCREMENT THE RAW MATERIALS COUNTER

    PRODUCT: INCREMENT THE PRODUCTS COUNTER
             GET THE CODE OF PRODUCT
             IF PRODUCT DOES NOT EXIST IN STORE
                CREATE A PLACE FOR IT
                INITIALIZE THE COUNTER OF THIS STORED PRODUCT
             ENDIF
             INCREMENT THE COUNTER OF THIS STORED PRODUCT
               .
               .
               .
               .
  ENDDO
ENDDO
```

FIG.2.1 : EXAMPLE OF PDL PROGRAM DESIGN

```
PRODUCT ITEMS

INPUT PARAMETERS :  ITEM-LIST, RAW-COUNT, PRODUCT-COUNT,
                    STORED-PRODUCTS

OUTPUT PARAMETERS : RAW-COUNT, PRODUCT-COUNT, STORED-PRODUCTS

DICTIONARY :        ITEM-LIST; ITEM-CODE; RAW-COUNT; PLACE;
                    PRODUCT-COUNT; STORED-PRODUCTS;
                    PRODUCT-CODE; STORED-PRODUCT-COUNTER

BEGIN
 WHILE EXIST MORE ITEM-CODE
  DO
   GET ITEM-CODE FROM ITEM-LIST
   CASE ITEM-CODE OF
    1: INCREMENT THE RAW-COUNT
    2: BEGIN
        INCREMENT THE PRODUCT-COUNT;
        GET PRODUCT-CODE FROM ITEM-LIST;
        FIND THE PLACE OF PRODUCT-CODE IN STORED-PRODUCTS;
        IF PLACE = NULL THEN
          BEGIN
           CREATE A PLACE FOR PRODUCT-CODE IN STORED-PRODUCTS;
           INITIALIZE THE SPECIFIC PLACE FOR THE PRODUCT-CODE IN
                       STORED-PRODUCTS

          END
        INCREMANT THE STORED-PRODUCT-COUNTER WHICH IS IN
              THE INDICATED PLACE OF STORED-PRODUCTS
       END
       .
       .
       .
       .
    END
   OD
  END
```

FIG. 2.2: PL PROGRAM DESIGN EXAMPLE





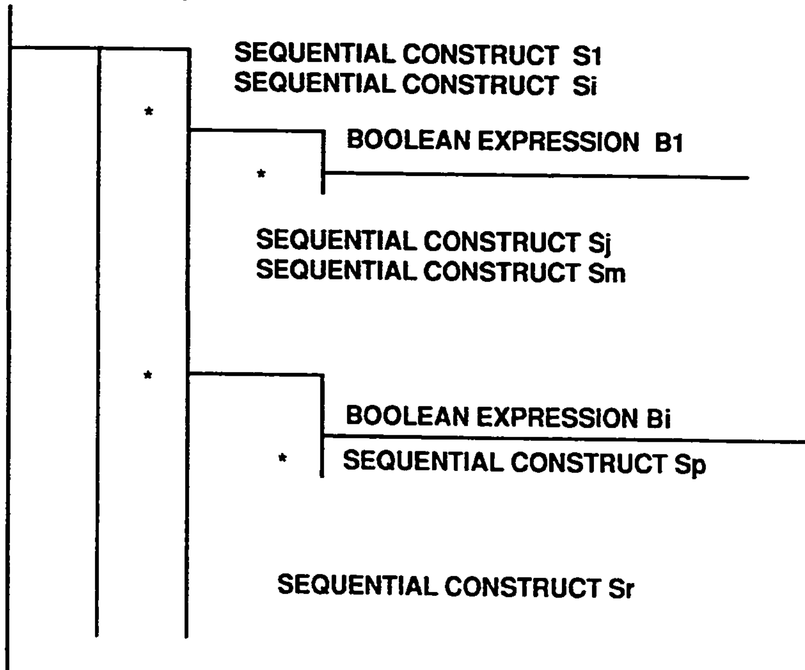


FIG. 2.3: SPC CONTROL CONSTRUCT

2.3 Distributed Software Design Methodologies

Distributed computing systems represent a wide variety of computer systems. The design of software for distributed systems is more complicated due to many design constraints and interactions of software components of the system. In the following paragraphes, we will focus on current advances in distributed software design methodologies.

Yau, Yang, and Shatz [19] have presented an approach for developing the design specification for a distributed software system. In their approach, the data and functional components are considered separately and all interactions among the functional components are allowed only through the access of shared resources. A precise description of all aspects of software design, including the data and functional components, their structural relations, and iterations , is developed. They also proposed a method to estimate the performance of the resultant software.

Lu, Yau, and Hong [5] have developed a formal methodology using attributed grammars for multiprocessing system software development. This methodology includes the design, representation, and validation of the software design using attributed grammars. The objective of this methodology is to provide continuity between the development phases so that design analysis and system validation can be automated. The continuity is provided through the use of a model for representing the control and data flow of a software system. The model is used for automated test-case generation and automated validation of the execution of the software system.

Another design representation of distributed software systems is to use petri nets and their modifications [5,20]. In this approach, a model has been developed for representing and analyzing the design of a distributed software system. The model enables one to represent the structure and the behavior of the distributed software system at a desired level of details of design especially communication among processors. Behavioral properties of the design representation can be verified by translating the modified petri nets into equivalent ordinary petri nets. The model emphasizes the unified representation of control and data flows, partially ordered software components, hierarchical component structure, abstract data types, data objects, local control, and distributed system state.

2.4 Distributed Computing Systems

During the past decade, many kinds of distributed computing systems have been proposed and built. These systems cover a wide spectrum in terms of design goals, size, performance, and applications. A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network [22].

Distributed systems can be characterized by their communications networks. The network determines the speed and reliability of interprocessor communication, and the spatial distribution of the processors. Traditionally, a distributed architecture in which communication is fast and reliable and where processors are physically close to one another is said to be closely coupled; systems with slow and unreliable communication between processors that are physically dispersed are termed loosely coupled.

closely coupled distributed systems use a communication network consisting of fast, reliable point-to-point links, which connect each processor to some subset of the other processors. Examples of such systems are Cosmic Cube, Hypercubes, and transputer networks. Communication time for this type of systems used to be on the order of a millisecond, but are expected to drop to less than a microsecond in the near future.

A more loosely coupled type of distributed systems is a workstation-LAN. The local-area network (LAN) allows direct communication between any two

processors. Communication time is typically on the order of milliseconds. In many LANs, communication is not totally reliable. Occasionally, a message may be damaged, arrive out of order, or not arrive at its destination at all. Software protocols must be used to implement reliable communication. A workstation-WAN can be seen as a very loosely coupled distributed system. Communication in a WAN is slower and less reliable than in a LAN; communication cost may be on the order of seconds.

In summary, there is a spectrum of distributed computing systems, ranging from closely coupled to very loosely coupled systems. Although communication speed and reliability decrease from one end of the spectrum to the other, all systems fit into the same model : autonomous processors connected by some kind of network that communicate by sending message.

2.5 Reasons for the interest in Distributed Computing Systems

Distributed computing systems represent a wide variety of computer systems. These systems are used for many different types of applications. The reasons for programming applications on distributed systems fall into four general categories [22] :

* Decreasing turnaround time for a single computation

Achieving speed up through parallelism is a common reason for runnin an application on a distributed computing systems.

* Increasing reliability and availability

Distributed computing systems are potentially more reliable, because they have the so called partial failure property : since the processors are autonomous, a failure in one processor does not affect the correct functioning of the other processors. Reliability can therefore be increased by replicating the functions of the application on several processors. If some of the processors crash the others can continue the job.

* The use of parts of the system to provide special functionality

Some applications are best structured as a collection of specialized services. A distributed operating system like MULTIX, for example, may provide a file service, a print service, a process service, a terminal service, a time service, a boot service, and a gateway service. It is most natural to implement such an application on distributed hardware. Each service can use one or more

dedicated processors, as this will give good performance and high reliability. The services can send requests to each other across the network. If new functions are to be added or if existing functions need extra computer power, it is easy to add new processors. As all processors can communicate through the network, it is easy to share special resources like printers and tape drives.

* The inherent distribution of the application

There are applications that are inherently distributed. One example is sending electronic mail between people's workstations. The collection of workstations can be regarded as a distributed computing system, so the application (email) has to run on distributed hardware.

2.6 Languages for Programming Distributed Computing Systems

When distributed systems first appeared, they were programmed in traditional sequential languages usually with the addition of a few library procedures for sending and receiving messages. As distributed applications became more common place and more sophisticated, this ad hoc approach became less satisfactory. Researchers began designing new programming languages specifically for implementing distributed applications.

These languages are classified into two categories [22] : logically distributed and logically nondistributed languages. In languages based on logical distribution, parallel computations (e.g. processes) communicate by sending messages to each other. The address space of the different computations does not overlap, so the address space of the whole program is distributed. In a logically nondistributed languages, the parallel units have a logically shared address space. The distinction between two categories is based on the logical model of the language; the presence of logically shared data does not imply that physical shared memory is needed to implement the language [23] .

The languages in the two categories are further partitioned into a number of classes based on their communication mechanisms. Figure 2.4 illustrates the classification of these languages and communication mechanisms they are based on. Figure 2.5 illustrates a sample of these languages and the communication mechanisms they based on [23] .

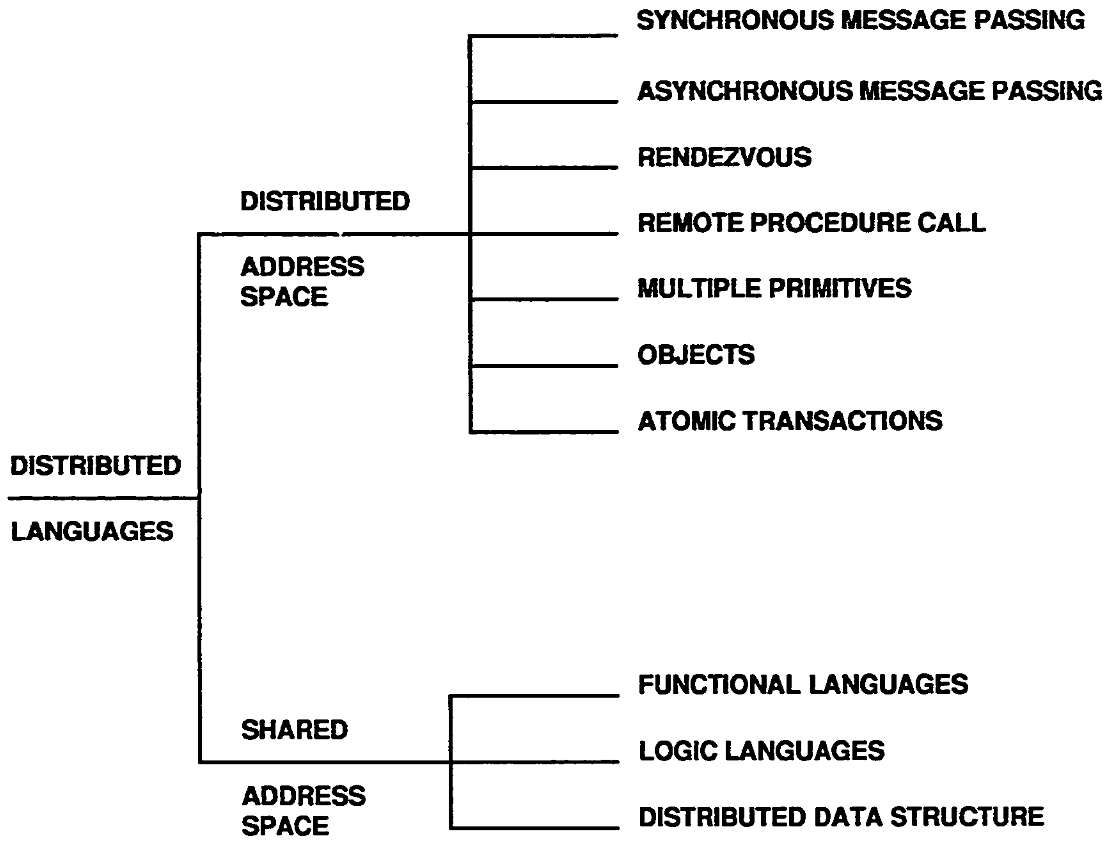


FIG. 2.4 : CLASSIFICATION OF LANGUAGES FOR DISTRIBUTED PROGRAMMING

COMMUNICATION MECHANISM	SAMPLE OF LANGUAGES
* SYNCHRONOUS MESSAGE PASSING	CSP, OCCAM, CCSP, CSM, Joyce, LIMP CSP180, GDPL, Pascal-m, Planet
* ASYNCHRONOUS MESSAGE PASSING	AMPL, CAMY, CONIC, DPL-82, FRANK LADY, PLITS, ZENO, Concurrent C
* RENDEVOUS	Ada, MC, BNR Pascal, Concurrent C
* REMOTE PROCEDURE CALL	Ceder, Concurrent CLU, LYNX, P+ Distributed Processes
* OBJECTS	SR, DisLang, Pascal-FC, StarMod
* ATOMIC TRANSACTIONS	Argus, Avalon, Aeolus, Camelot-Library
* FUNCTIONAL LANGUAGES	FX-87, PML, QLISP, Blaza, Concurrent-Lisp
* LOGIC LANGUAGES	Quty, Vulcan, Delta Prolog, Oc, BRAVE
* DISTRIBUTED DATA STRUCTURES	Linda, Orca, SDL, Tuple

FIG. 2.5

2.7 *Primitives for the Support of Designing Distributed Software*

A commonly held view is that designs for concurrent or distributed systems are best modeled on message passing systems in which each node is a module and each message corresponds to the parameters of a procedure call. In such systems, nodes may be located in different memories for distributed processing, and all nodes that have received messages may process them concurrently. With that view, many of the simplifying concepts used in the design of sequential programs, such as stepwise refinement, data abstraction, and modular design can be exploited in the design of concurrent and distributed systems [28]. Coordinated computing systems provide the opportunity to concurrent solutions. These systems have rich primitives and concepts that are lacking in other systems. In the following paragraphs, we concentrate on the aspects of software design for message passing distributed systems, and the required primitives to specify them.

2.7.1 *Expressing Parallelism*

Distributed computing systems are characterized by executing computing tasks simultaneously. A computing task is commonly referred to as a process, a module, or a task, which is a synonym for a logical processor that executes code sequentially and has its own state and data. The definition of the unit of parallelism in a distributed system is usually made explict by the designer. In our work, the unit of parallelism is referred to as a process, a module or a software component in general. Software components can be looked upon as allocatable objects. Component described to be fixed for the liftime of a software

system are called static components since they have static allocation to processors. While components specified to be created and destroyed during program execution are called dynamic components, since they have dynamic allocation to processors. Dynamic components are created either explicitly, by an executable statement such as a create construct, or implicitly through a component type declaration and then a component is created by declaring a variable of that type.

A software component may terminate normally by the completion of its program execution, or through some other scheme. Therefore, associated with the ability to specify dynamic component creation, there must be a provision for component destruction, including a one that allows a component to terminate another component. Inter-component communications and cooperations among the components of a software must be described thoroughly. Aspects of inter-component communication are covered in the next two subsections.

2.7.2 Synchronization

There are two kinds of message transmission modes that control the ordering of computing events for coordination and cooperation, namely the synchronous and asynchronous message transmissions. With synchronous message transmission, the sender waits (i.e blocked) until the message has been accepted by the receiver. In this case, both parties of a communication connection are said to be synchronized. While with asynchronous message transmission, the sender does not wait for the receiver to be ready to accept its message. The sender is free to continue computing after the message is

transmitted, usually referred to as send-and-forget communication.

Most distributed systems are asynchronous, in the sense that each component operates at its own speed (which may vary in time) and may wait when its own operation requires information not yet available, or to be provided by another component. The speed of the overall system is determined by the speed of the slowest component. For such an asynchronous system, the communication mechanism has to provide a flow control facility for correlating the effective processing speeds of the different components [24] .

Some other systems are synchronous, in the sense there is a fixed relation between the processing speeds at the different components. The synchronization between the components is maintained by a common clock which is usually provided through the communication medium [24] .

The controversy of whether to have a synchronous or asynchronous transmission is a longstanding issue in literature. However, in this work, the issue is bypassed in favor of the design of an SDL with rich and powerful message transmission constructs that are capable of expressing design decisions using synchronous or asynchronous transmission.

2.7.3 Communication

Communication means the exchange of information between components of a distributed system and does not imply synchronization. For loosely coupled systems, message passing must be used to implement both synchronization and information flow. Information flow between two communicating components

(processes) may be unidirectional, where the information flows from the sender to the receiver, or bidirectional, where each component (the sender or the receiver) may transmit information to the other component.

The association between the sender(s) and receiver(s) of all the possible software components is called the connection. Each connection provides a potential communication path from the sender(s) to the receiver(s). The definition and design of this connectivity depends on the specification of the components interfaces, the connection patterns, and the naming convention used.

To emphasize modularity, software components should be considered as independent entities, which interact with other components through their interfaces. A component interface describes the form of the message which can be sent and received and the potential sources and destination of this message. Accordingly, there must be a data type associated with the definition of each message. This message data-type provides the structural information through which messages should be interpreted. Therefore, the interface of a component is given by the types of communicating messages and the names of the connected recipient(s) and sources(s) of the messages.

The possible connection patterns are: one-to-one, one-to-many, many-to-one, and many-to-many. Concurrent programming languages provide mechanisms fot the definition of a connection based on one or more patterns. For instance, CSP provides a one-to-one connection between the sender and the receiver; and Ada provides a many-to-one connection, where a destination entry point can be

called by a number of sources.

Generally, the one-to-one and one-to-many connections simplify synchronization and protection problems, since there is only one source. On the other hand, the many-to-one connection is approprite for the provision of servers (e.g. data bases, or I/O devices). Both one-to-many and many-to-many patterns support provisions for broadcasting. Connectivity may be set up statically, at compile time, or dynamically, at run time. Dynamic connectivity can be created by allowing component names (i.e addresses) to be passed in a message and used by the receiver.

The naming convention refers to the techniques adopted by a distributed system to identify the source(s) and destination(s) of messages involved in communication. It is also the provision by which connections are established between communicating components. In general, naming can be direct or indirect. With direct naming, a connection is specified explicitly when the sender or the receiver names the connected component(s). If each side of a connection communicates by naming the other side, communication is said to be symmetric. When only one side of a connection names the other side, a caller-callee relation is established and the communication is said to be asymmetric. In this case, the sender (client) explicitly names the receiver (server) component, but the receiver does not name the sender. Ada is an example of language in which direct asymmetric connection is used between communicating tasks.

With indirect naming, the receiver or the sender refers to a locally defined name of a port. The sender transmits a message to a local port name with an

export capability, and the receiver accepts messages from a local port name with an import capability. The actual binding of local names of ports to the sender or the receiver is performed separately. The association between the exporting and importing ports can occur at a later stage, during system generation or run time. This approach for naming provides flexibility to the software designer or the programmer of a component, since he does not need to worry about the actual names of the sender(s) or receiver(s).

Mailboxes is an alternative form for an indirect naming mechanism. A mailbox is a typed communication object that is globally defined. A sender process transmits a message to a named mailbox and the receiver process accepts the message from the same mailbox. The main advantage of mailboxes is that they support all communication patterns.

2.7.4 Nondeterminism

Nondeterminism in software designed for distributed systems stems from the basic fact that software components cooperate and coordinate in order to achieve their goal in solving a problem. In general, communication primitives describe the mechanisms for cooperating to be achieved, but they are not sufficient to control it. For example, a process may wait indefinitely on an entry point or a port for the arrival of a message from an unidentified source, or a process may receive messages from multiple sources and needs to select one to be accepted. At the receiving side of a connection, nondeterminism may arise due to two requirements. First, there is the requirement to select among different entries or sources to process messages waiting to be accepted. In this

case, selection may take the form of an arbitrary selection with no specified method, a fixed predefined method, such as the overall order of message arrival, or a user controlled method, such as using a message priority scheme.

Second, there is the requirement to select a message at an entry point, if a number of messages are waiting at that entry. A method must be used to decide which message should be selected for acceptance. The selection is usually controlled by a fixed method, such as the use of a FIFO order of arrived messages, the use of a message priorty scheme, or user-controlled. The receiver process may select a message from the buffer based on the values of some local variables or the values of some fields in the message itself.

2.7.5 Time

The notion of time in distributed programming has two significant uses. In one hand, time is needed to deal with situations and constraints that arise in distributed real-time applications by the occurrences of events (e.g. the opening or closing of valves, moving robot arm, etc). One the other hand, time can be used as a mechanism to control nondeterminism. The use of absolute time, generally, complicates the problem of synchronizing the distributed clocks. However, the use of relative time intervals suffice to enable solving problems raised by blocking, deadlock situation or failure. Therefore, the ability to specify a time constraint in an SDL or a programming language is needed. For instance, a component waits (blocks) until the message arrives or the time period expires (i.e. time-outs). Similarly, time intervals (time-outs) can be used as mechanism to detect the occurrence of communication failures or process

failures, and then, to escape from the blocking situation.

2.7.6 Fairness

One issue that a software designer should worry about is the fairness of the target distributed system. Fairness means to give each component its rightful turn to access a resource, and the amount of delay that a component has to wait before its demand for access is achieved. In this regard, there is the fair allocation of computing resources, and the fair allocation of communication opportunities among the components. Distributed systems are described based on their attitude towards fairness as antifair, weakly fair, or strongly fair. In an antifair system, a component can be indefinitely blocked from accessing a resource. A weakly fair system implies that a component is sure to access the resource although there is no limit on how long it may take before being served. Finally, a strongly fair system provides components access rights to resources in turn. For instance, those with equivelent priorities should be served in the order of their requests.The most common mechanism to enforce strong fairness is with using queues. Associated with each component entry point there is a queue for buffering messages. A component accepts messages on these entries in the queue order. Language constructs used to escape from blocking situations, such as time-outs and guarded send and receive statements, can be used to express, explicitly, what the designer may think a non-fair situation to communicating components of his application.

2.7.7 Failures

The last issue a DSDL must address is the support for coping with total failures and partial failures. Naturally, distributed systems have higher potential for reliability and availability than centralized systems. However, the underlying communication system is subject to delays and failures. Furthermore, the software components are subject themselves to failures due to either hardware or software faults. One objective of distributed systems is to be fault tolerant. A system is said to be fault tolerant if it can still function properly in the case of hardware or software failures. The design of a fault tolerant distributed application is not an easy task to handle. At the design process, the issue of fault tolerance could be ignored altogether, or left to the responsibility of the operating system or the language run-time system. Another alternative is to let the software designer specify exceptions and exception handlers to deal with different kinds of expected failures.

CHAPTER III

LANGUAGE DEFINITION

3.1 Introduction

The desire to use concurrent and distributed design languages stems from the existence of a wide variety of concurrent and distributed applications. The aim of this chapter is to introduce the description of a ***Distributed Software Design Language*** (DSDL) for distributed processing application along with its structure, constructs, features, and communication primitives.

3.2 Syntax Notations

The syntax of our software design language is described using BNF notation, with two extensions :

[X] specifies the optional occurance of item X

{ X } specifies zero or more occurance of item X

X | Y specifies either item X or item Y

non-terminal symbols are written in small letters and enclosed by the symbols < and > , keywords are written in capital letters. The BNF grammar comprises a set of production rules, each rule has a left side and a right side separated by the meta symbol ' ::= '.

3.3 Overall Structure : Modules & Processes

The distributed computing model for the DSDL is a loosely-coupled

message passing system at the module level and tightly-coupled shared memory system at the process level. Where, modules and processes are the two main software components that facilitate expressing and specifying parallelism. A module is stand-alone software component that can be allocated physically or logically to an autonomous host computer in a distributed system. Each module is able to communicate with other modules in the system through passing messages. A module is logically allocated to a host computer if it is a member of a cluster of modules which where allocated to the same host. The communication capabilities of a module are specified by a set of definitions that describe connectivity, flow of information, and synchronization.

A process is a software component that expresses parallelism defined within a module component. Processes belonging to the same module may communicate with each other using shared memory. While, a process must communicate with other modules/processes across the boundary of its module through message passing. Both software components can be as statically created or dynamically created.

3.3.1 Module Structure General Form

A module has the following form:

<module> ::= <module-heading>
    [<definition>] [<classes>] [<objects>] [<components>]
    [<initialization>]
    <module-body>

<module-heading> ::= MODULE <module-name> [(instances)] ";"

Each module component is defined by a module heading that includes the keyword MODULE followed by a module name and possibly a number of instances. The module name acts as a reference or an address to the module structure. If the number of instances is specified in the module heading, it indicates that an array of modules is to be created with the specified size and structure.

The <definition> syntactic category defines communication connection with other modules for exchanging information. The <definition> syntactic category has the following form:

<definition> ::= DEFINE <list-of-communication-primitives > END ";"
<list-of-communication-primitives> ::= <communication-primitive>
{ communication-primitive }
<communication-primitive> ::= <entry-def> | <name-def> | <port-def>

The <classes> syntactic category defines the description of all classes of objects that are sharable within the structure of a module. Of particular importance for a design language is the declaration of user-defined classes (i.e types) for the following super-classes: module-type, process-type, message-type, array-type, record-type, and access-type. The <classes> syntactic category has the following form:

<classes> ::= CLASS <list-of-classes>
<list-of-classes> ::= <class-type> | <list-of-classes> <class-type>
<class-type> ::= <process-type> | <msg-type> | <record-type> |
<array-type> | <module-type> | <access-type>

The <objects> syntactic category declares the names and types of all object specified in the shared memory of a module structure. The objects or (variables) of the DSDL may belong to either one of the built-in types, or to one of user-defined types in the class part. The <objects> syntactic category has the following form:

<objects> ::= OBJECT <list-of-objects>

<list-of-objects> ::= <object-def> | <list-of-objects> <object-def>

<object-def> ::= <identifier-list> ":" < type > ";"

The <components> syntactic category is basically the definition of stati processes/modules encapsulated within the module structure. Only one level of specified static modules/processes is allowed. The <components> syntactic category has the following form:

<components> ::= <component> | <components> <component>

<component> ::= <process> | <module>

The <initialization> syntactic category is a block of statements that is specified to set up actions at the instantiation time. However, the initialization block should not include any communication activities for sending or receiving messages, but it can have any other actions including the specification of dynamically created processes/modules. The <initialization> syntactic category has the following form:

<initialization> ::= INITIALIZE

BEGIN <sequence-of-statements> END ";"

The <module-body> category is the specification of a sequence of statements to be executed as part of module task. The module-body must include a NULL statement, if no actions need to be specified. The body has the following form:

<module-body> ::= BEGIN <sequence-of-statements>
[<exception-handler-category>]
END

An exception handler is an optional category that specifies the actions to be taken in case of raising exceptions locally, within the module body, or remotely, by other components.

3.3.2 Process Structure General Form

A process has the following form:

<process> ::= <process-heading>
[<definition>] [<classes>] [<objects>] [<initialization>]
<process-body>

<process-heading> ::= PROCESS <process-name> [(instances)] ";"

Each process component is defined by a process heading that includes the keyword PROCESS followed by a process name and possibly a number of instances. The process structure has categories similar to those defined for module structure with few exceptions, in particular no static processes/modules are allowed within its structure.

3.4 Lexical Style

3.4.1 Lexical Elements

The design using DSDL is written as a sequence of lines of text containing the following characters

* the alphabet A-Z and a-z
* the digits 0-9
* various other delimiters ( ) [ ] , ; : . + - * / | & ~ < = > _
* the space character

the compound delimiters /* and */ are used to specify the beginning and end of a comment.

3.4.2 Identifiers , Literals , and Comments

An identifier starts with a letter followed by zero or more instances of letters or digits optionally preceded by a single underscore. The identifier has the following form:

< identifier > ::= < letter > { [_] letter | digit }

The language permits the use of integer and real literals, character , and boolean. It allows also using comments any where in the text to increase understandability. The comment form follows compound delimiter /* and terminated by */

/* This is a comment */.

3.5 Declarations of Classes and Objects

In general, the entities or objects of the DSDL belong either to the built-in language types, or to user-defined types. The definition of user-defined types is specified in the CLASS part. Each user-defined type is referred to as a class. The DSDL defines super-classes from which a user defined class may be specified. The super classes are for module-type, process-type, message-type, array-type, record-type, and access-type (pointer-type). The user-defined classes are used as a template to specify the instantiation of any number of objects of the same class either lexically, by declarations, or dynamically, by a create statement.

The object part defines all objects to be created in the shared memory for all processes in the component's scope. The scope of a component includes all processes created, statically or dynamically, by the component or any of its descendents. The DSDL built-in types are for integers (INT), reals (REAL), boolean (BOOLEAN), characters (CHAR), exceptions and signals (EXCEPTION & SIGNAL). In the following sub-sections we shall highlight super-classes.

3.5.1 Module-type and Process-type

The specification of a component class is similar to the specification of the component structure. A module class must be specified by the keyword MODULE_TYPE instead of MODULE. A process class must be specified by the keyword PROCESS_TYPE instead of PROCESS. Module-type has the following form:

<module-type> ::= <module-type-heading>

[<definition>] [<classes>] [<objects>]

[<initialization>]

<body>

<module-type-heading> ::= MODULE_TYPE <module-type-name>

[(<instances>)] [(<list-of-parameters>)] ";"

process-type has the following form:

<process-type> ::= <process-type-heading>

[<definition>] [<classes>] [<objects>]

[<initialization>]

<body>

<process-type-heading> ::= PROCESS_TYPE <process-type-name>

[(<instances>)] [(<list-of-parameters>)] ";"

The component's name refers to the user-defined class name. With the ability to define process-type classes, processes can be specified to be dynamically created by a module or a process. However, modules are not allowed to be specified by a process for creation. Therefore, a module-type class definition within a process structure is forbidden. Instances of the component class are specified by declaration in the object part, or specified by a create statement as part of the run-time specified activities. A list of parameters could be associated with the definition of a module-type or a process-type. Where, each parameter specifies a parameter name and its type. The list of parameters are considered as local objects. They are initialized when a component object of

the specified component class is initialized. The semantic effect of the declaration of a component object is the instantiation of a component of the class specified. However, the specification of a component activation is defined by the INIT statement. If the component requires the passing of a list of arguments, then a list of arguments must be included in the INIT statement too. The general form of INIT statement is as follows:

<init-statement> ::= INIT <component-name> [(<list-of-arguments>)] ";"

The following two examples describe the mechanism for both ways of specifying parallelism:

```
/* Example 1 */

  CLASS
    PROCESS_TYPE Philosopher;
    ....
    /* Declarations and definitions */
    BEGIN
        /* specification of process activities */
    END /* Philosopher */
    Philos : ARRAY [0..4] of Philosopher;
    Ph_indicator : ACCESS Philosopher;
  OBJECT
    Ph1: Philosopher;
    Ph2: Philos;
    Ph_ptr: Ph_indicator;
  INITIALIZE
    BEGIN
          INIT Ph1; init Ph2;
          CREATE(Ph_ptr);
    END
```

```
/* Example 2 */

  CLASS
    MODULE_TYPE Consumer;
    ....
    /* Declarations and definitions */
    BEGIN
        /* specification of module activities */
    END /* Consumer */

    MODULE_TYPE Producer(cons: Consumer);
    ....
    /* Declarations and definitions */
    BEGIN
        /* specification of module activities */
    END /* Consumer */
    Producer_ptr : ACCESS Producer;
    Consumer_ptr : ACCESS Consumer;
  OBJECT
    One-Producer_ptr: Producer_ptr;
    One_Consumer_ptr: Consumer_ptr;
  INITIALIZE
    BEGIN
          CREATE(One_Consumer_ptr);
          CREATE(One_Producer_ptr)(NAME(One_Consumer_ptr));
    END
```

In Example 1, a class for a Philosopher process-type has been defined. The Philosopher class definition is used to instantiate processes dynamically, using two methods. The first method is the specification by declarations. In the object part, Ph1 is declared as of type Philosopher, and Ph2 is declared as an array of five processes of type Philos. In the INITIALIZE part, Ph1 and Ph2 are activated by init statement. The second method is the specification by a create statement. In this case, the creation activities are specified by the create statement and the use of a pointer object, Ph_ptr, of type Ph_indicator. The class Ph_indicator is defined as an access type to an object of the Philosopher class.

In Example 2, the NAME function is introduced. It takes an initialized pointer (i.e., access object) and returns an equivalent name for the dynamic object to which the pointer is pointing to. The semantic effect of the name function is, such that, it facilitates passing names for dynamic objects to other components, since all information passed through parameters or messages are passed by value-result.

3.5.2 Message-type

Message-type has the following form:

```
MESSAGE  <message-name-type> : [ IN | OUT | IN_OUT ]
         <slot-definitions>
         { slot-definitions }
END-MESSAGE ";"
```

Each message is defined by a message-name-type, followed by one of the designations IN, OUT, IN_OUT. An IN designated message is a one that is expected to be received at a certain communication connection. An OUT designated message is the one that is expected to be transmitted on one of the communication connections. Finally, an IN_OUT message is a one specified for transmission and reception. The default message designation is assumed to be OUT.

Each message is a structure of heterogeneous slots; that is, they may or may not be of the same type. Each slot is defined by a slot name and a type. However, the types that are not allowed within a message structure are EXCEPTIONS, access-types. module-types, process-types, message-types, and record-types. The declaration of objects of a particular message class creates instances of the required messages in the shared memory of the components scope. An example might be

```
CLASS
        MESSAGE    msg  : IN
                     c  : CHAR ;
                   i , j  : INT  ;
        END ";"
```

In object part we may instantiate any number of messages of type msg

```
OBJECT

     mymsg  : msg  ;
        .
        .
        .
```

3.5.3 Access-type

Access-type has the following form:

<component-indicator> ":" ACCESS <component-type-name> ";"

The class <component-indicator> is defined as an access type to an objec which is either process-type or module-type class. <Component-indicator> works as a template where we may create a pointer (i.e access object) that points to the address of <component-type-name> by using create statement

3.5.4 Array-type and Record-type

An array is a composite object consisting of elements of the same type. The bounds of the array are specified at the time the array is declared. An array has the following form:

<array-name> ":" ARRAY [lbound .. upbound] OF <type>

A record is a composite object consisting of elements that may be of different types. Record-type is defined as

RECORD <record-name> <component-list> END-RECORD ";"
<component-list> ::= <comp-declaration> { comp-declaration }
<comp-declaration> ::= <identifier> { , identifier } ":" <type> ";"

A record field may be of any type except a signal type, a module-type, a process-type, or a message-type.

3.6 Communication Primitives

In this section, the communication primitives provided by the DSDL are described. In this regard, the language constructs to define communication connections, message reception statements, and message transmission statements are introduced.

3.6.1 Definition of the Communication Connection

The DSDL adopts three different syntactic forms for the specification of communication connectivity, these are: ports, entries, and direct naming. The definition of communication connectivity must be provided in the DEFINE part as a sequence of definitions. Each definition belongs to one communication connection method. The objective of the DEFINE part is to specify the communication connections as inter-module shared objects. The DEFINE part has the following form:

<define-part> ::= DEFINE
        <communication-def>
        { communication-def }
    END ";"

<communication-def> ::= <entry-def> | <port-def> | <name-def>

3.6.1.1 Entries

Entry definition has the following form:

<entry-def> ::= <list-of-entry-names> ":" <entry-type> ENTRY ";"

<entry-type> ::= SYNCH | ASYNCH

An entry name may belong to an entry within the module in which it is defined in, or to a process that have declared the entry name and wishes to export it to other modules for communication. An entry name, belonging to a process that is specified to be created lexically, can be declared for exporting if the full path name of the entry is specified. For an entry belonging to a dynamically specified process by a create statement, the entry definition is not allowed by the semantics of DSDL for exporting to other components. An entry point can be specified as either accepting synchronous or asynchronous communication. With synchronous communication, the sender would block until an acknowledgement is received from the receiver. Communication through an entry is asymmetric, when an entry accepts a message without identifying the sender. To avoid the use of a large list of entry names inside a component, an entry can be specified as an array of entries with upper bound value. Accordingly, entry points are defined based on their index value, and all references to an individual entry from other components must specify its index number for proper communication.

3.6.1.2 Ports

Ports represent, logically, communication channels. They are declared locally by a module to support message passing in loosely and tightly coupled multiprocessors. Also, they represent channels on which broadcasting can be performed. The port definition has the following form:

<port-def> ::= <list-of-port-names> ":" <mode> PORT [< connect >] ";"

<mode> ::= UNI-DIR | BI-DIR

<connect> ::= FROM <list-of-component-names> |

TO <list-of-component-names> | BROADCAST

A port can be specified as a uni-directional or bi-directional channel. A uni-directional port can be used for either transmission or reception, but not both at the same time. Messages transmitted or received on uni-directional ports assume asynchronous communication. A bi-directional port is able to transmit and receive simultaneously. Messages transmitted or received on a bi-directional port assume synchronous communication. Associated with each port, a connection specification may be included using the FROM clause, the TO clause or the BROADCAST clause. The FROM clause defines a uni-directional receiving connection from the component(s) named. When a list of components is included in the FROM clause, a many-to-one connection is specified, and it defines a Selective-Server capability. The TO clause specifies a uni-directional transmitting connection to the components(s) named. When a list of components is included in the TO clause, a one-to-many connection is specified, and it defines a multi-casting capability. The BROADCAST clause specifies a uni-directional or bi-directional transmitting capability to all components. A port definition without a connection specification can assume transmission or reception based on its mode specification. In case of a large number of ports need to be specified inside a component, an array of ports can be defined by associating an upper bound positive number with a port name. Accordingly, individual ports are referenced using their indices for proper communication.

3.6.1.3 Direct Naming

A process or a module can establish communication with other software components through direct naming convention. The sender explicitly names the receiver of the message, while the receiver explicitly names the sender of the received message. Therefore, direct naming convention scheme is referred to establish symmetric communication. The direct naming definition has the following form:

<name-def> ::= <list-of-component-names> ":" <direction> ";"
<direction> ::= IMPORT | EXPORT | EXIM

Each module or process has to declare the names of the lower level components that participate in this type of connection scheme, and define its communication direction. The names of components declared for direct naming communication must be specified statically or lexically in the component which includes the declaration. In this scheme, we define possible directions: IMPORT, EXPORT, and EXIM (EXport & IMport). A component name associated with an IMPORT direction expects to receive messages from other components. A component name associated with an EXPORT direction expects to send messages to other components. Finally, a component name with EXIM capability expects to send and receive messages to other components. Only the names of components declared in the DEFINE part are visible by other modules and processes for communication through direct naming. Components with IMPORTing or EXPORTing capabilities are assumed to be engaged in asynchronous type of communication. Only those with an EXIM capability are

considered to be engaged in synchronous type of communication.

3.6.2 Message Reception Statements

Message reception statements are the Receive and Enter statements. In a reception statement, a message object must be specified to belong to a message-type, which has a designation IN or IN-OUT, and has a structure and ordering of message slots matching the expected message to arrive.

3.6.2.1 Receive Statement

Receive statement has the following syntactic form:

```
<receive-statement> ::= RECEIVE <msg-name>
                        [ FROM <component-name> ]
                        [ ON <port-name> ] [ IFF <condition> ]
                        [ BY <expression> ][ Reply<message-name> ]";"
```

Receive statement expects to receive a message that is collected by a message name. Where a message name has already been declared of a compatable type as that of the message expected. The FROM clause specifies from which component the message is sent. The ON clause specifies the port on which the message is to arrive. Both the FROM and the ON clauses can be specified simultaneously or separately. The IFF clause defines a condition for reception. The reception is completed when the condition has been satisfied. A condition is a boolean expression that is going to be evaluated to true or false. The variables of the condition can be either belonging to the state of the receiving process or the state of some message slots (fields) in the message to be

received. The BY clause defines an arithmetic expression, where its value puts a priority level on a coming message. For instance, if several messages of the same type are received on the same port and/or from the same source, then these messages can be ordered and accepted based on their priority level defined by the BY clause. The REPLY clause provides a specification for returning back a message, when the connection capabilities permits that, such as specifying a receive on a bi-directional port from an EXIM component name. The semantics of the receive statement is assumed to provide an acknowledgement, depending on the type of communication connection specified in the receive statement and defined in the DEFINE block.

3.6.2.2 Enter Statement

Enter statement has the following syntactic form:

```
<enter-statement> ::= ENTER <entry-name> <msg-name>
                      [ FROM <component-name> ]
                      [ IFF <condition> ] [ BY <expression> ]
                      [ DO < sequence-of-statements >  END ] ";"
```

The enter reception statement is designated for accepting messages on a predefined entry point. The entry point of reception is based on the remote procedure call mechanism. When an entry has been declared with the synchronous mode of communication, the rendezvous mechanism of an entry is assumed, similar to that in Ada. In this case, the rendezvous mechanism must return an acknowledgement and may produce a reply message. The designation of the received message must be IN-OUT in order to produce a reply message.

The reply message is the same received message after having some modifications or changes, if any, performed upon its slots. The modifications or changes must be carried out during the rendezvous interval, that is, within the sequence of statements delimited by the DO and END keywords. If the entry name has been declared with an asynchronous mode, no replies can be conducted, and no rendezvous is possible. The semantics of calling an enter statement is such that, when a calling component issues an entry call and the receiving component is not ready to accept the call (i.e., not ready to rendezvous), then the calling component is suspended and its call is put on a queue associated with the entry. Entry calls are accepted from the queue in FIFO order. FROM, IFF, and BY clauses are defined similar to their definitions in the receive statement.

3.6.3. Message Transmission Statements

3.6.3.1 Send Statement

The Send statement has the following syntactic form:

```
<send-statement> ::= SEND <msg-name>
                     [ TO <destination> ]
                     [ ON <port-name> ][ IFF <condition> ] ";"
<destination> ::= <entry-name> | <component-name>
```

The send statement is the construct that specifies an asynchronous transmission. The destination of a message transmission is specified by the TO clause, ON clause, or both of them. The TO clause defines the component name or an entry name which has to receive the transmitted message. The ON clause

names a port on which the message has to be sent. The semantics of the send statement is assumed to provide an asynchronous (unblocked) transmission. The transmission of the message can be specified to be based on the output guard defined in the IFF clause. The definition of the IFF clause is similar to the one described previously.

3.6.3.2 Call Statement

The Call statement has the following form:

<call-statement> ::= CALL <callee> <msg-name>
[IFF <condition>] ";"
<callee> ::= <destination> | ON <port-name>

The call statement is the construct that specifies a synchronous transmission. The semantics of the call statement assumes that the calling component is blocked until the called component accepts the message and an acknowledgement is received. Similarly, the called component might be suspended until the calling component receives the transmitted message. In this case, the caller and the callee (receiver) are said to be synchronized. The receiver of a specified message is determined by the definition of an entry name, a component name, or a port name. Call to an entry name must have been defined to be a synchronous entry point, and call by direct naming a component must have been defined with EXIM capability. A port name must have been defined as a bidirectional port locally or by a remote module. The IFF clause blocks transmission until the guard is satisfied.

3.7 Indeterminacy Control

3.7.1 Select Statement

The select statement is based on the guard command introduced by Dijkstra [2]. The structure of the statement allows indeterminacy control specification, where any one of several alternative execution paths may be selected for a given input. The syntax of the select statement is as follows:

```
<select-statement> ::= SELECT  <guarded-command>
                       { OR guarded-command }
                       [  ELSE <guarded-command>  ]
                       END-SELECT ";"
```

Each alternative path is defined as a guarded command as follows :

```
< guarded-command > ::=  [ WHEN <condition>   => ]
                              <sequence-of-statements>
```

A guarded command is built from an optional guard and a sequence of statements. The guard is a boolean expression. When the guard is evaluated to be true, the following sequence of statements are considered as a candidate for execution.

The semantics of the select statement is as follows: the The system evaluates all the boolean expressions, corresponding to the guarded commands. One guarded command, whose guard is evaluated to be true, is then selected nondeterministically for execution. We assume that the system simulates

parallel guard evaluation. The selection mechanism of the guarded command is assumed to be arbitrary. As such, select statement permits a component to accept entry calls from more than one component in a non-deterministic fashion.

Each alternative of the select statement commences with a possible WHEN clause, and then followed by a sequence of statements. If no guarded command is selected, then the ELSE part has to be executed. If no guarded command is selected and no else part, then the select statement will be skipped.

3.7.2 Selective Loop Statement

The selective loop statement has the following syntactic form:

<loop-statement> ::= LOOP <sequence-of-alternatives> END-LOOP ";"

<sequence-of-alternatives> ::= <guarded-command>
{ OR guarded-command }

The selective loop statement expresses an infinite repetition of a sequence of alternative guarded commands, that are defined as the body of the loop. In each iteration, all guards of the alternative paths are evaluated simultaneously. Accordingly, all guarded commands having true guards are candidates for selection. We assume the system simulates parallel guards evaluation, and the selection mechanism of true guards is arbitrary.

3.7.3 Delay Statement

The delay statement has the following syntactic form:

<delay-statement> ::= DELAY <expression> ";"

The floating point expression specifies the delay time in seconds. The significance of the delay statement is in specifying a duration time for the suspension of a component's activities. Basically, the delay statement is used to specify the control of non-determinism in concurrent programs, to detect communication failure or component failure, or to escape from blocking situations.

3.7.4 Exit Statement

The exit statement is mainly used to control the iterative mechanism of a selective loop. It has the following syntactic form:

<exit-statement> ::= EXIT [WHEN <condition>] ";"

An exit specification would transfer control to the statement following the selective loop. When the WHEN clause is specified, it defines a conditional state for escaping out of the selective loop. Only when the condition is evaluated to be true the exit event would occur.

3.7.5 Terminate Statement

The function of the terminate statement is to specify a termination action of a component within the scope of a module after it completes its task or due to a decision to destruct itself under certain conditions. A terminate statement has the following syntactic form:

<terminate-statement> ::= TERMINATE <local-component>
[WHEN <condition>] ";"

<local-component> refers to the component name that has to terminate its activities. When the WHEN clause is specified, it indicates the condition under which the termination has to take place. The termination action is assumed to complete when all descendents of the component are terminated or aborted. Furthermore, no cyclic chain of communication between processes/modules is assumed.

3.7.6 Abort statement

The abort statement is used to specify an action to terminate (i.e abort) the activities of one or more other components outside the scope of a module. The syntactic form of the abort statement is as follows:

<abort-statement> ::= ABORT <list-of-components>
[WHEN <condition>] ";"

When the WHEN clause is specified, it indicates the condition under which the abort statement has to take place. The semantics of the abort statement is assumed not to allow a component to abort any of its ascendent components. Also, it is assumed that an abort causes the blocking of a component until all specified components in the statement will be terminated. Furthermore, no cyclic chain of communication between processes/modules is assumed.

3.8 Constructs for Exception Handling

3.8.1 Exception and Signals Types

There are two built-in types to describe exceptions, namely, the EXCEPTION and the SIGNAL types. Variables declared as belonging to an EXCEPTION type are raised by a raise statement, and handled locally within the software component they are declared in, or within the shared-memory environment it is defined.

Variables of the SIGNAL type are used to handle exceptions that are raised remotely by an announce statement. A variable of the type SIGNAL is either an IN SIGNAL or OUT SIGNAL . The raise of an OUT SIGNAL variable in a software component is performed using the announce statement. The announce statement is used to specify the raising and transmitting of an OUT SIGNAL variable to another module in which the proper action may be specified to handle the signal. The semantics of the announce action do not assume to cause an interruption of the component in which the OUT SIGNAL is declared in. An IN SIGNAL variable is used to specify the reception of a signal that has been announced remotely. The reception of a signal is specified by the handle statement that defines the raise of an IN SIGNAL variable and the interruption of the receiving component. An exception handler part may be specified to handle exceptions raised locally, by the raise statement, or to handle IN SIGNALS raised remotely, by an announce action.

3.8.2 Raise Statement

Exceptions are raised explicitly by means of the raise statement , which has the following syntactic form:

<raise-statement> ::= RAISE <list-of-exception-names> ";"

The semantic effect of a raise statement is assumed to suspend the execution of the component in which it is specified, and passes control to an appropriate exception handler. A raise statement without an exception name may appear only in an exception handler; it causes raising the corresponding exception again for further actions specified by other components.

3.8.3 Announce Statement

The specification of an announce statement in a component is to describe the raising of OUT SIGNAL variable(s) that may cause the interruption of one or more remote components. The syntactic form of an announce statement is as follows:

<announce-statement> ::= ANNOUNCE <list-of-signals>
[ON <port-name>] [TO <component-name >]
[WITH <msg-name>] ";"

The announce statement acts as a mechanism for raising the signals and transmitting them as control messages . Transmission can be specified to be on a port for broadcasting, using the ON clause; or, it can be specified to be point-to-point by describing the name of the recipient component directly using the TO

clause. The direct naming scheme for transmission can be specified with or without the definition of a port name. An ordinary transmission of a message may be specified, through which the defined signal are carried out as message slots. Also, an ordinary transmission can take place WITH a message name to the required destination. The signal variables must be slots of the message to be transmitted. The semantic effect of the announce statement is such that it does not cause the interruption of the component executing it, and all signals raised in an announce statement are assumed to be reset once the transmission process is completed. Therefore, the same set of signal variables can be announced several times due to the occurance of several events. The announce transmission is basically asynchronous.

3.8.4 Handle Statement

The handle statement is used to specify the reception of signal(s) transmitted by the announce statement due to the occurance of an event at a remote component. The handle statement format is as follows:

```
<handle-statement> ::= HANDLE <list-of-insignals>
                       [ IN <msg-name> ] [ ON <port-name> ]
                       [ FROM <component-name> ] END-HANDLE ";"
```

The semantic effect of the handle statement is as if a raise statement is executed locally. The component in which it is specified is interrupted at the position where the handle statement is completed, given that any one of the IN SIGNALs to be received is raised. The interruption of the receiving component may transfer control to an exception handler if there is one exists.

The handle statement receives signals specified explicitly whether they are associated with a message or not. Reception may be specified to be on a defined port, and/or from a component name directly. The IN SIGNAL variables must belong to slots of the message to be received. The handle statement is not completed before checking the status of all the listed signals. The rules of termination of a component and the propagation of IN SIGNALs are similar to the ones for exception type variables. If none of the IN-SIGNAL in a handle statement are received at the time of the execution of a handle statement, then the handle statement completes execution without interrupting the component, the process or module in which the handle is specified continues execution normally. Basically , the reception of the handle statement is assumed to cause no blocking.

3.8.5 Exception Handling Part

The exception handler part is an optional section at the end of a component body description that consists of one or more statements to handle exception(s) raised locally or remotely as described previously. When an exception is raised, the remainder of the statements in the component's body are abandoned. Instead, execution of an appropriate exception handler can take place, if any is specified. When the specified exception handler completes, or there is no exception handler for raised exception the component terminates. The exception is propagated to the parent component if no exception handler is specified. The exception handler part has the following form:

EXCEPTION

WHEN <exception-choice> { exception-choice } = > <sequence-of-stmts>

.
.
.

WHEN <exception-choice> { exception-choice } = > <sequence-of-stmts>

< exception-choice > ::= < exception-name >

The left hand side of the "= >" is a list of one or more exceptions or signals (IN SIGNALs). The semantic effect of each exception handler is to execute the sequence of statements specified on the right of "= >" , if any one of the specified exceptions has been raised.

3.9 Language Constructs for the Support of the Software Design description and Refinement Process

3.9.1 Null Statement

Null statement is the statement that does nothing. It is used in situations where no action is to be performed, but the syntax require presence of at least one statement. It has the following following form:

<null-statement> ::= NULL ";"

3.9.2 Deffered Statement

Deffered statement is used to indicate the existance of some actions that their description has not been defined at this stage. The exact definition of the actions is deffered to a later stage in the design stage. It has the following form:

<deffered-statement> ::= DEFFERED ";"

3.9.3 Assignment Statement

Assignment statement has the following syntactic form:

<assign-statement> ::= <variable> := <expression>

Executing the assignment statement causes the value of expression to be assigned to the variable.

CHAPTER IV

DSDL PROCESSOR

4.1 Introduction

Once the requirements specification has been completed, the next major phase in the software development cycle is the design phase. The design phase of the software life cycle involves creating a solution that meets the specifications outlined in the analysis and specification phase.

DSDL is a design language that assists designers to create an efficient design. Its processor features enable the designer to end up with a design written in DSDL format and checked syntactically. The syntax deals with the mechanical aspects, namely whether or not a sequence of words is a sentence in the language. What the sentence means is determined by the semantics of the language. In this chapter, we shall describe the DSDL processor features.

4.2 Lexical Analyzer

The lexical analyzer program is written with ***Lex*** that generate the analyzer in C on the UNIX system. Lex turns the user's expressions and actions (called source) into the host general-purpose language; the generated program is named ***yylex***. The yylex program recognizes expressions in a stream called input and performs the specified actions for each expression as it is detected as shown in Figure 4.1.

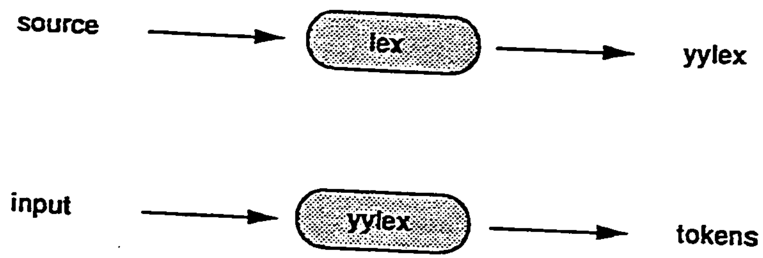


FIG. 4.1 : An overview of Lex

4.3 Syntax Analyzer (Parser)

Yacc , a powerful utility in the UNIX system is used to check the DSDL grammar. Yacc accepts a grammar specified in BNF and indicates any problems which make the grammar unsuitable for language recognition.

Yacc builds a parser while analyzing the grammar. The parser is a push-down automaton (stack machine) consisting of a large stack to hold current states, a transition matrix to derive a new state for each possible combination of current state and next input symbol, a table of user-definable actions which are to be executed at certain points in the recognition, and finally an interpreter to actually permit execution. The result is packaged as a function ***yyparse()*** , which calls repeatedly on a lexical analyzer function ***yylex()*** to read standard input, and which returns zero or one to indicate whether or not a sentence was presented as input file. Figure 4.2 visualize the parser function.

Since Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is appropriate. Lex is used to partition the input stream into tokens, and Yacc assigns structure to the resulting tokens. The flow of control in such a case is shown in Figure 4.3.

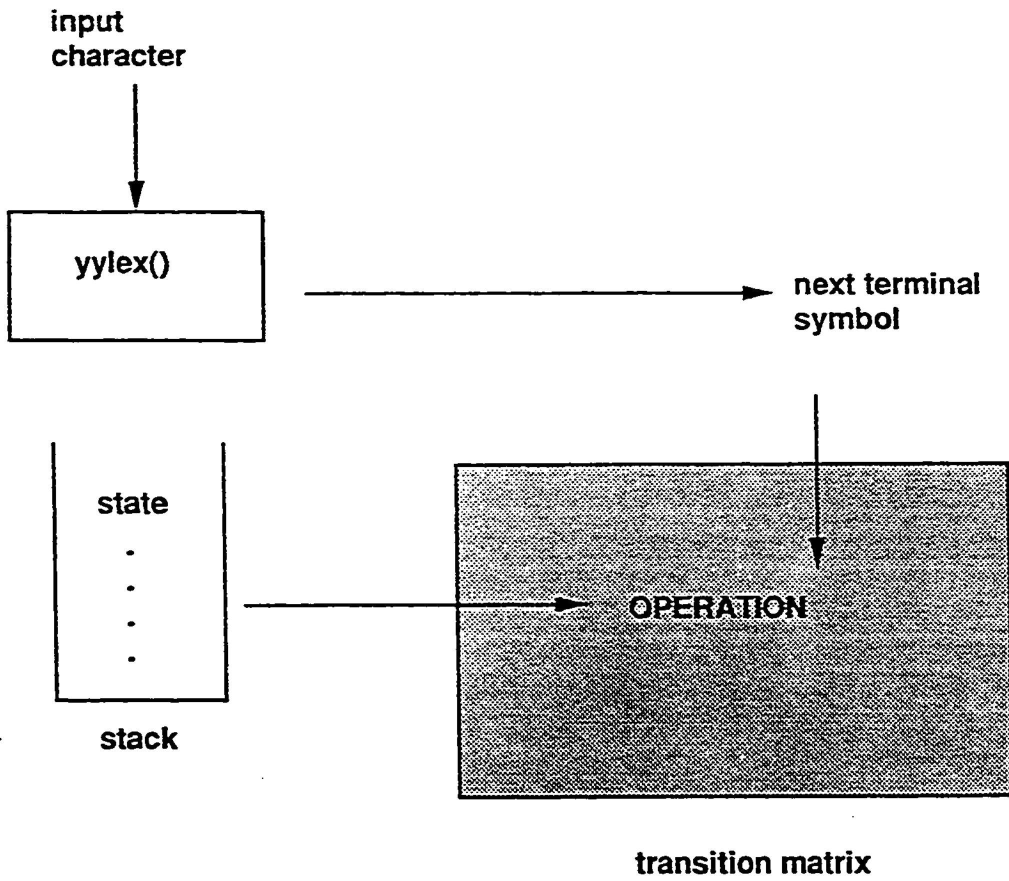


FIG. 4.2 : Parser Function

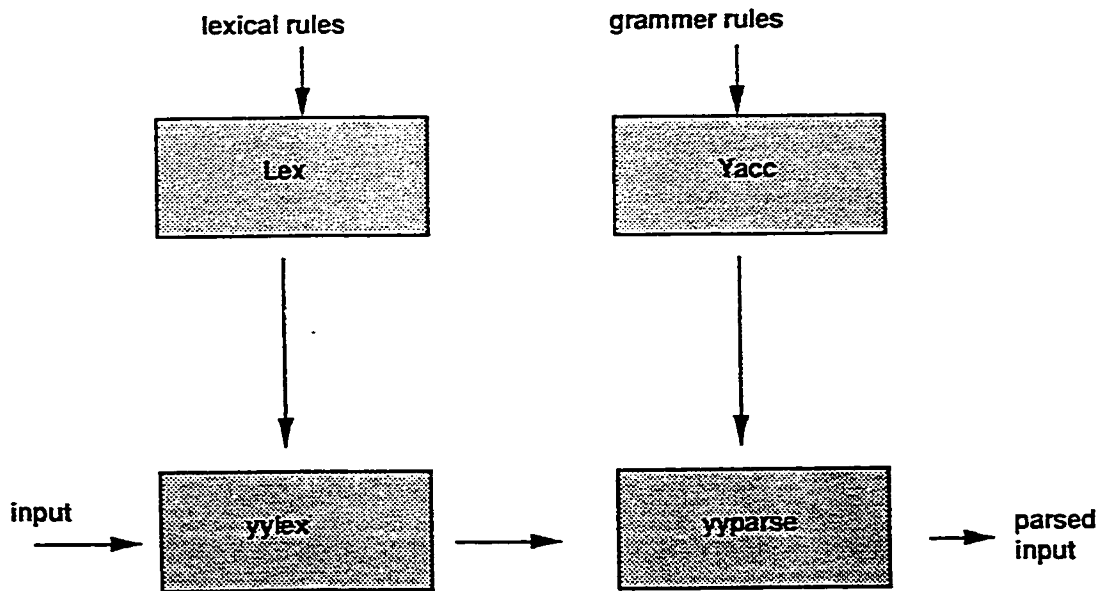


FIG. 4.3: Lex with Yacc

4.4 Error Recovery

It turns out that once an input error is encountered, the parser "fall off the stack" during an error operation so, to make our parser robust against input errors we used a technique to cope with any error; since Yacc provides a special ***error*** terminal symbol to influence the parsing algorithm. The error symbol and the yyerrok action are the Yacc features to use in making a parser robust.

Our parser can recognize and manipulate a sentence. The following example shows what happens if we present the incorrect input " a - - " to the parser, which is based on the rule

```
expression : expression '-' expression
           | Identifier
```

with '-' defined to be left-associative. yyparse() falls of the stack; the second '-' leads to an error operation in the transition matrix. Once the error message has been issued, yyrparse() seems to remove all states from the stack. Since the stack is cleared in the process, yyparse() returns with a function value of one, and the parsing is aborted on encountering the first error in the input.

Our lexical analyzer has the following entry at the end of the pattern

```
return yytext[ ];
```

This pattern is intended to pick up all single character operators. However, this entry will return the integer value of any single character as function value of yylex(). Unexpected input characters are thus passed from the lexical analyzer to the parser as if they were legitimate terminal symbols represented by single characters. To deal with this problem, we treat an input error as a special case

of a terminal symbol "error". "error" can be used in formulations just like a terminal symbol; however, error is not produced by a call to the lexical analyzer. Instead, the parser believes "error" to be the next terminal symbol if the actual next terminal symbol leads to an error operation in the transition matrix for the current state. Once the error symbol has been internally generated in this fashion, and the error message issued, yyparse() will set out to accept error almost like any other symbol.

Consider the following modification to the rule above:

```
expression : expression '-' expression
           | Identifier
           | error
```

A parser based on this grammar will accept erroneous input. If " a - - b " is given as an input, yyparse() will assume that the next terminal symbol is error, and the transition matrix prescribes that the symbol error is to be accepted, and we can move on to next state.

Error recovery, in fact, consisted of inserting the error symbol between the two '-' symbols in the input. The resulting input was acceptable to the extended grammar.

Sometimes the error recovery mechanism must discard an input symbol to keep it going. This is why "error" is treated by Yacc and yyparse() differently from normal terminal symbols. Let us see what happens if " a + - b " is the input to the above rule. Here, '+' is an illegal character passed by yylex() as a valid terminal symbol. At this point, yyparse starts popping its stack looking for

a state on the stack in which the "error" symbol can be accepted. If no such state is on the stack then yyparse will clear the entire stack and terminate with a function value of one. If there is a state , yyparse() discards the '+' (the next terminal symbol) and completes its task.

If we consider " a - - b + c " as a given input. In this case, the parser must shift three terminal symbols beyond the point of error, before another error results in an error message. This way a cluster of error may result in only a single error message. With yyerrok action, the parser can accept enough terminal symbols and report errors close to each others. The following modification to the above rule is done:

```
expression : expression '-' 'expression
           | Identifier   { yyerrok }
           | error
```

The placement of error symbols is guided by the following, conflicting goals:

* ***as close as possible to the start symbol of the grammar.***

This way there is always a point to recover from, since there should always be a state very low on the stack in which error can be accepted. The parser then is never able to clear its stack early, i.e., to not complete by recognizing the end of file from the lexical analyzer.

* ***as close as possible to each terminal symbol.***

This was only a small amount of input would be skipped on each error. This can be improved using yyerrok actions.

* ***without introducing conflicts.***

This may be quite difficult. In fact, accepting shift/reduce conflicts is reasonable as long as they serve to lengthen strings. E.g., one can continue parsing an expression past an error, rather than accepting the same error at the statement level, thus trashing the rest of the expression. Following these goals, the following typical position for error symbols are recommended:

* into each recursive construct, i.e., into each repetition.
* preferably not at the end of a formulation.

This should result in a robust recovery,i.e., in a recovery from which the continuation is meaningful. Adding a trailing error and yyerrok action may lead to cascading error messages, or even to loops if the parser can not discard input.

* non-empty lists require two error variants, one for a problem at the beginning of the list, and one for a problem at the current end of the list.
* possibly empty lists require an error symbol in the empty branch.

4.5 Counter Examples

In this section, the role of DSDL processor will be emphasized as a tool to maintain strong type checking within a module/process, the consistency checking among communicating modules and/or processes, the scope of objects declared within the module/process object part, and the scope of communication connectivities defined in the definition part of the module/process, along with the efficiency and flavor of DSDL as a design language, are illustrated by the following examples:

Example 1 : Producer-Consumer Problem

The problem is to read data, transform it and then print the result. For this example, the data is a stream of characters, and the transformation converts lower-case letters to upper-case letters. The purpose of the example is to show the different forms for expressing the creation of components in DSDL, and to show how modules are connected together using ports, not how to do interesting transformation. Also, explained in this example the DSDL role in semantic analysis and consistency checking.

The DSDL program will be structured as follows: one module called the "Producer", reads the data from the standard input and sends it to another module called the "Consumer". The Consumer module converts the data to upper case and then prints it on the standard output. DSDL presents three versions of this program.

a) Specification of Static Modules

Static creation of modules will be used to represent both the Producer and the Consumer modules. Figure 4.4 illustrates the design expressed in DSDL.

Module Producer produces a message called ***pro-msg*** and sends it on the ***con-port*** port for processing. The module Consumer accepts the arrived message with message ***con-msg*** on the specified port and process it. Communication between Producer and Consumer is assumed an asynchronous one, so no acknowledgment is sent back by module Consumer. Notice that, while the Producer reads the next character, the Consumer module transforms and prints the previous character.

The processor will check if the ***pro-msg*** to be sent to module Consumer has the OUT designation, and the ***con-msg*** that will receive the sent message has IN designation. Also the processor will check the order and types of message slots of both send and receive messages. Furthermore, the processor will check if the port where the message received on is declared in the definition part of the Consumer module.

```
module producer_consumer;

  module  consumer ;

    define
        con_port : uni_dir port ;
    end;

    class
        message  xmsg : in
                 ch : char ;
                 cx : int;
        end_message;

    object
          con_msg : xmsg;

    begin
          loop
               receive  con_msg on con_port;
              /* consume the arrived message and process it */
          end_loop;
    end  /* consumer */

  module producer ;

    class
          message  msg : out
                   c : char ;
                   x : int;
          end_message;

    object
            pro_msg : msg;

    begin
        loop
       /* produce a message and send it to the consumer for processing */
              send  pro_msg on consumer.con_port;
        end_loop;
    end  /* producer */


begin      /* producer-consumer-body */
     null;
end  /* producer_consumer program */
```

FIG. 4.4: Producer-Consumer Program (Static Modules)

b) Specification of Lexical Modules

Lexical elaboration of modules will be used to represent both the Producer and Consumer modules. Figure 4.5 illustrates the design expressed in DSDL.

In this solution, a class for Producer module-type and Consumer module-type have been defined. Both Producer and Consumer class definitions are used to instantiate modules lexically by declarations. Producer module-type has declared a parameter ***cons*** of type Consumer which will receive the messages send by the Producer. In object part, ***one-producer*** is declared as of type Producer, and ***one-consumer*** is declared as of type Consumer. In the initialize part, one-consumer and one-producer are activated by the ***init*** statement. The activated one-consumer will be passed as an argument in the init statement that activates the one-producer. The passing of parameters is needed to be able to send messages to a specific Consumer, otherwise the communication between Producer and a certain Consumer cannot be established. The consumer always checks its port to see if there is messages sent to it. The activated modules communicate in asynchronous fashion.

The processor does the same job discussed in previous solution to the problem. In initialization part, the processor will check if the objects and parameters in the ***init*** statement are declared as types of Producer and Consumer module-types.

```
module producer_consumer;

  class

      module_type  consumer ;

         define
            con_port : uni_dir port ;
         end;

         class
              message  xmsg : in
                     ch : char ;
                     cx : int;
              end_message;

         object
              con_msg : xmsg;

         begin
            loop
                 receive  con_msg on con_port;
                /* consume the arrived message and process it */
            end_loop;
         end  /* consumer */


      module_type producer (cons : consumer );

         class
              message  msg : out
                     c : char ;
                     x : int;
              end_message;

         object
               pro_msg : msg;
         begin
           /* produce a message and send to the consumer for processing */
            loop
                 send  pro_msg on cons.con_port;
            end_loop;
         end  /* producer */
```

FIG. 4.5: Producer-Consumer Program (Lexical Modules)

```
object
        one_producer : producer;
        one_consumer : consumer; /* declare module objects */

initialize
          begin
              init one_consumer;
              init one_producer(one_consumer);
          end;
begin
      null; /* body part */

end  /*  producer_consumer program */
```

FIG. 4.5 (continued)

c) Specification of dynamic Modules

Dynamic creation of modules will be used to represent both the Producer and the Consumer modules. Figure 4.6 illustrates the design expressed in DSDL.

In this solution, a class for Producer module-type and Consumer module-type have been defined. The ***producer-ptr*** and ***consumer-ptr*** are defined as access-types. The creation activities are specified by the ***create*** statement and the use of pointer objects ***one-producer-ptr*** and ***one-consumer-ptr*** of type producer-ptr and consumer-ptr respectively. The ***name*** function is introduced. It takes an initialized pointer and returns an equivalent name for the dynamic object to which the pointer is pointing to. The passing of parameters will be conducted to establish communication between created modules. Communication between created modules is asynchronous.

The processor does the same job discussed in part (a). Furthermore, in initialization part the processor will check if the pointers in the ***create*** statement are declared in object part as types of access types of Producer and Consumer module-types, and check whether any parameters need to be passed at the dynamic creation of the components. Dynamic creation has been illustrated here to occur in the initialize part, however, eventually it may occur also at any point within the body of the Producer-Consumer module.

```
module producer_consumer;

  class
       module_type  consumer ;

           define
                 con_port : uni_dir port ;
           end;

           class
                message  xmsg : in
                        ch : char ;
                        cx : int;
                end_message;

           object
                 con_msg : xmsg;

           begin
                loop
                     receive  con_msg  on  con_port;
               /* consume the arrived message and process it */
                end_loop;
           end /* consumer */


       module_type producer ( cons: consumer );

          class
               message  msg : out
                        c : char ;
                        x : int;
               end_message;

         object
               pro_msg : msg;

         begin
         /* produce a message and send it to the consumer for processing */
              loop
                   send  pro_msg on cons.con_port;
              end_loop;
         end /* producer */

       producer_ptr : access producer;
       consumer_ptr : access consumer;
```

FIG. 4.6: Producer-Consumer Program (Dynamic Modules)

```
object
        one_producer_ptr : producer_ptr;
        one_consumer_ptr : consumer_ptr;

initialize
                begin
                    create(one_consumer_ptr);
                    create(one_producer_ptr) (name(one_consumer_ptr));
                end;
begin
        null; /* body part */
end  /* producer_consumer program */
```

FIG. 4.6 (continued)

Example 2 : House Security System

The use of computers at home is becoming common. One of their uses is to monitor the state of a house, which includes routine tasks such as keeping track of the energy usage, and being on the alert for exceptional situations like fires and break-ins. In these cases, the computer has to be programmed to respond to exceptional situations. In this example, the DSDL capabilities to express exception handling are explained.

In each house, many fire sensors and burglar sensors are installed in different places. In this example, three varieties of modules: ***house-control-system***, ***fire-dept***, and the ***police-dept*** are described. Figure 4.7 illustrates the design expressed in DSDL for the house security system. The Module House-Control-System is used to describe monitoring fires and break-ins inside a house. There are 100 instances of House-Control-System modules, one instance is assumed for each house in the security system. In case of fire, the house-control-system module announces a signal to fire-dept module that will accept the announced signal and does the proper action. In case of break-ins, the house-control-system module announces a signal to police-dept module that accepts the announced signal and does the proper action.

The heading of the module house-control-system describes an array of 100 modules to be created statically, each module corresponds to a particular house control has declared two messages ***fire-msg*** and ***police-msg***. Each message contains the house address and a signal to be announced to a specified remote module. Both fire-sensors and movement-sensors are declared in object part as

arrays of boolean. Also, the total-fire-sensors and total-burglar-sensors are declared as integer objects. In the initialize part, the fire-msg and police-msg messages are initialized to contain the local address of the house. Also, the total-fire-sensors and the total-burglar-sensors are initialized too.

Module fire-dept has declared ***dept-msg*** message to receive the messages that might be sent by the house-control-system modules. The fire-dept module defines a class of a process-type, called ***house-handler*** where each house-control-system module in the control system will have a corresponding instance of the house-handler process-type created. At initialization, all instances of the house-handler are to be created dynamically, with an Active State, then they will be in one of the following states: ***Ready***, ***Active***, or ***Terminated.*** When an instance of a house-handler receives an interrupt signal from the corresponding house-control-system, the process instance is interrupted and enters an exception handler part where its state is flagged to be Terminated. When information about the status of the house has been received to be in normal situation, the process instance state is changed to the Ready state. At the detection of the Ready state for a process instance, that instance for the house-handler will be created dynamically, then its state is flagged to be Active again.

All instances of the house-handler are referenced by the an access type named ***fire-crew.*** The array type ***house-array*** is an array of pointers to house-handler instances. An object of house-array is created for this problem with 100 elements.

The state of the house-handler instances are kept in the shared memory in

the form of an array of characters, each element is corresponding to a house-handler instance. The state of a house-handler instance is referred to by "A" for Active, "R" for Ready, and "T" for Terminated. A class, called ***process-state***, is defined for representing the states of all house-handler instances, an object of that class is declared, referred to by ***house-process-state.***

To handle communication between a control-house-system module and the corresponding process instance of house-handler, a uin-directional port is defined. As a whole, the array of 100 ports, ***fire-port***, is defined to provide the required communication services.

Module police-dept has declared ***police-msg*** message to receive the messages that might be sent by the house-control-system modules. The police-dept module defines a class of a process-type, called ***crime-handler*** where each house-control-system module in the control system will have a corresponding instance of the crime-handler process-type created. At initialization, all instances of the crime-handler are to be created dynamically, with an Active State, then they will be in one of the following states: ***Ready***, ***Active***, or ***Terminated.*** When an instance of a crime-handler receives an interrupt signal from the corresponding house-control-system, the process instance is interrupted and enters an exception handler part where its state is flagged to be Terminated. When information about the status of the house has been received to be in normal situation, the process instance state is changed to the Ready state. At the detection of the Ready state for a process instance, that instance for the crime-handler will be created dynamically, then its state is flagged to be Active again.

All instances of the crime-handler are referenced by the an access type named ***cops-crew***. The array type ***cops-array*** is an array of pointers to crime-handler instances. An object of cops-array is created for this problem with 100 elements.

The state of the house-handler instances are kept in the shared memory in the form of an array of characters, each element is corresponding to a crime-handler instance. The state of a crime-handler instance is referred to by "A" for Active, "R" for Ready, and "T" for Terminated. A class, called ***process-state***, is defined for representing the states of all crime-handler instances, an object of that class is declared, referred to by ***house-process-state***.

To handle communication between a control-house-system module and the corresponding process instance of crime-handler, a uni-directional port is defined. As a whole, the array of 100 ports, ***police-port***, is defined to provide the required communication services.

In case of fire occurance, module house-control announces a signal within a message ***fire-msg*** to fire-dept module on a specified ***fire-port*** port. The announced fire-msg contains in one of its slots a signal to be raised remotely at fire-dept module, and the house address in another slot. The processor will check if the signal to be announced from ***house-control-system*** module to ***fire-dept*** module is an OUT-SIGNAL and the message that contains the signal has an OUT designation. If so, the processor will check if the port that the message announced on is declared in definition part of fire-dept module. Module fire-dept will receive the sent message by the ***handle*** statement. The processor will

check if the received message variable in the handle statement has an IN-OUT/IN designation, and the corresponding slot has an IN-SIGNAL type once the announced signal is received.

In case of movement detection, we announce a signal within a message ***police-msg*** to police-dept module on a specified ***police-port*** port. The announced police-msg contains in one of its slots a signal to be raised remotely at police-dept module, and the house address in another slot. The processor will check if the signal to be announced from ***house-control-system*** module to ***police-dept*** module is an OUT-SIGNAL and the message that contains the signal has an OUT designation. If so, the processor will check if the port that the message announced on is declared in definition part of police-dept module. The ***handle*** statement receives the transmitted signal by the announce statement due to the occurance of an event at house-control module. The processor will check if the received message in the handle statement has an IN-OUT/IN designation, and the corresponding slot has an IN-SIGNAL type.

```
module security_system;

    module House_control_system (100);

        class
            string   : array { 1 to 70 } of char;

            message f_msg : out
                fire_alarm      : out_signal;
                house_location : string;
            end_message;

            message p_msg : out
                burglar_alarm  : out_signal;
                house_location : string;
            end_message;

            heat_sensor : array { 1 to 10 } of boolean;

            entry_sensor: array { 1 to 20 } of boolean;

        object
                fire_msg                : f_msg;
                police_msg              : p_msg;
                fire_sensor             : heat_sensor;
                movement_sensor         : entry_sensor;
                total_fire_sensors      : int;
                total_burglar_sensor :  int;
                i                       : int;
                k                       : int;

        initialize

            begin

                /*  initialize fire_msg & police_msg to contain the
                    local address of the house */

                total_fire_sensors = 10;
                total_burglar_sensors = 20;
            end;
```

FIG. 4.7: Security System Program

```
begin  /*  house-control module body  */

   /* assuming a round robin selection if there is no parallel
       simulation of the loop alternatives available */

      loop
         i = 1;

            loop
               when fire_sensor(i)  implies
                     begin
                         announce  fire_msg.fire_alarm  to  fire_dept
                               on fire_dept.fire_port(k)  with fire_msg ;
                     end;
               i = i+1;
               exit  when i > total_fire_sensors;
            end_loop;

        or

         i = 1;

            loop
               when movement_sensor(i)  implies
                     begin
                       announce  police_msg.burglar_alarm  to  police_dept
                            on police_dept.police_port(k)  with police_msg ;
                     end;
               i = i+1;
               exit  when i > total_burglar_sensors;
            end_loop;

       end_loop;

 end  /*  house_control_system  */
```

FIG. 4.7 (continued)

```
module fire_dept;

        define
             fire_port(100) : uni_dir port;
        end;

        class
             string : array { 1 to 70 } of char;

             message fire_dept_msg : inout
                    fire_answer     : in_signal;
                    house_location : string;
             end_message;

             process_type  house_handler (k: int);

                  object
                        dept_msg : fire_dept_msg;
                               k : int;

                  begin  /* house-handler body */

                    loop
                         handle dept_msg.fire_answer
                             in dept_msg  on fire_port(k) end_handle ;
                    or
                         delay 5;
                    end_loop;

                    exception
                           when dept_msg.fire_answer  implies
                             house_process_state(k)  =  "T";/* terminated *//

                  end  /* house_handler */

             fire_crew    : access house_handler;

             house_array : array { 1 to 100 } of fire_crew;

             process_state : array { 1 to 100 } of char;

        object
            j                          : int;
            num_of_houses        : int;
            house_process_state : processs_state;
            house                    : house_array;
```

FIG. 4.7 (continued)

```
initialize
     begin
   /*  initialization / activation puts all processes in the active state  */

          num_of_houses = 100;
          i = 1;
              loop
                  create( house(i))(i);
                  house-process-state(i) = "A";
                  i = 1 + 1;
                  exit when i > num_of_houses;
             end_loop;

     end;

begin  /* fire-dept module body */

     loop
       j = 1;
          loop
               when house_process_state(j) = "T" /* terminated */ implies

                   /* give orders to send fire men to house address j */

           j = j + 1;
           exit when  j > num_of_houses;
         end_loop;

    or

     j = 1;
          loop
               when house_process_state(j) = R /* ready */  implies

                           /* fire men controlled the fire */
                          begin
                               create(house(k))(k);
                               house_process_state(j) = "A" /* active * ;
                          end;
              j = j + 1;
              exit when  j > num_of_houses;
         end_loop;

    end_loop;

end /* fire_dept module */
```

FIG. 4.7 (continued)

```
module police_dept;

        define
             police_port(100) : uni_dir port;
        end;

        class
             string : array { 1 to 70 } of char;

             message police_dept_msg : inout
                     police_answer  : in_signal;
                     house_location : string;
             end_message;

             process_type  crime_handler (k : int);

                  object
                       police_msg : police_dept_msg;
                       k          : int;

                  begin  /* crime-handler body */

                    loop
                         handle police_msg.police_answer
                             in police_msg  on police_port(k) end_handle;
                    or
                         delay 5;
                    end_loop;

                    exception
                        when police_msg.police_answer  implies
                             house_process_state(k)  =  T; /* terminated */

                  end  /* crime_handler */

             cops_crew   : access crime_handler;

             cops_array  : array { 1 to 100 } of cops_crew;

             process_state : array { 1 to 100 } of char;

        object
            L                   : int;
            num_of_houses       : int;
            house_process_state : processs_state;
            cops                : cops_array;
```

FIG. 4.7 (continued)

```
initialize
     begin

   /*  initialization / activation puts all processes in the      active state  */

          num_of_houses = 100;
          i = 1
               loop
                 create(cops(i))(i);
                 house-process-state(i) = "A"
                 i = 1 + 1;
                 exit when i > num_of_houses;
              end_loop;
     end;

begin /* police-dept module body  */

     loop
        L =  1;
          loop
             when house_process_state(L) = "T" /* terminated */ imlies
            /* give orders to send policemen to house address L */

                 L = L + 1;
                 exit when L > num_of_houses;
         end_loop;

    or

     L = 1;
          loop
               when house_process_state(L) = "R"/* ready */  implies

                       /* when policemen handled the situation */

                       begin
                             create(cops(k))(k);
                             house_process_state(L) = "A" ;  /* activ*/
                       end;
               L = L + 1;
               exit when L > num_of_houses;
         end_loop;

   end_loop;

end /* police_dept module */
```

FIG. 4.7 (continued)

```
begin  /* body of security_system module */
       null;
end  /*  security system program */
```

FIG. 4.7 (continued)

Example 3 : The Mortal Dining Philosophers

Five philosophers spend their lives eating and thinking. They eat at a circular table in a dining room. The table has five chairs around it, and chair number I has been assigned to philosopher number I. Five forks have also been laid out on the table, so that there is one fork between every two chairs. Consequently, there is one fork to the left of each chair and one to its right. Fork number I is to the left of chair number I.

Each philosopher must enter the dining room and sit in the chair assigned to him in order to be able to eat. A philosopher must have two forks to eat; if he cannot get two forks immediately then the philosopher must wait until he gets them before he can eat. The forks are picked up one at a time with the left fork being picked up first. When a philosopher is finished eating he puts the forks down and leaves the room.

In this example, the DSDL capabilities to express synchronous communication, communication connection by entries, and rendezvous mechanism are shown. Figure 4.8 illustrates the design expressed in DSDL.

The dining philosophers program has three varieties of module-types: ***philosopher*** , ***fork*** , and the ***room.*** Philosopher calls the room to enter and exit, and calls the fork to pick forks up and put them down.

In philosopher module-type, a synchronous entry ***get-id*** is defined to establish communication with other modules, ***ph-id*** message is declared to receive the philosopher number, and ***philos-msg*** is declared to be sent to module

fork containing the left and right forks.

In fork module-type, two synchronous entries ***pick-up*** and ***put-down*** are declared. A message ***our-msg*** is declared in order to receive the specified left and right forks sent by the philosopher.

In module-type room, two synchronous entries ***room-exit*** and ***room-entrance*** are declared. A message ***host-msg*** is declared to receive the philosopher number sent by the module philosopher. The module-type room is introduced to avoid deadlock. The module-type room makes sure that there are at most four philosophers in the dining room at any given time. Each Philosopher must request permission to enter the room and must inform it on leaving. Clearly, there is no possibility of a deadlock since there will be at least one philosopher in the room who will be able to eat.

In module dining, the five philosophers and the five forks are instantiated lexically as modules using two arrays of modules in the module dining. On activation, each philosopher first gets an identification number. Using this number, a philosopher can determine the numbers of the forks on either side of him. Communication between modules is synchronous one. The rendezvous will be conducted each time a module calls an entry in another module. The ***call*** statement is used to call an entry in another module, and the other module accepts the call by the ***enter*** statement. The calling module is blocked until the called module accepts the message and acknowledgment is received. Also, the called module is suspended until the calling module receives the transmitted message.

The processor will check if the designation of both sending and receiving messages are compatible to each other. Also, it will check the order and type of slots in both sending and receiving messages.

```
module Dining;

    class

        module_type Fork;

            define
                 pick_up , put_down : synch entry ;
            end;

            class
                message  fork_msg : inout
                           lf  : int;
                           rf  : int;
                end_message;

            object
                our_msg : fork_msg;

            begin  /* fork body */

                 /*  a fork can be picked up by one philosopher at
                     a time. It must be put down before it can be picked
                     up again  */

                loop

                    begin
                        enter pick_up our_msg ;
                        enter put_down our_msg ;
                    end;

                 or

                        exit;

               end_loop;

           end /* fork */
```

FIG. 4.8: Dining Philosophers Program

```
module_type Philosopher ( ourfork : fork; Host : room );

   define
         get_id : synch entry ;
   end;

   class
         message philosopher_msg : inout
                  left  : int;
                  right : int;
         end_message;

         message ph_number : inout
                  id : int;
         end_message;

  object
        left, right     : int;       /* fork numbers */
        philos_msg    : philosopher_msg;
        ph_id          : ph_number;

  begin      /* philosopher body */

        enter  get_id  ph_id;
        left  = ph_id.id;
        right = ph_id.id mod 5 + 1;

    loop
               /* think for a while then enter dining room */
        begin
             call Host.room_entrance ph_id;

                /* pick_up forks */
             call ourfork.pick_up philos_msg;

                /* eat & put_down forks */
             call ourfork.put_down philos_msg;

                /* leave the dining room */
             call Host.room_exit ph_id;
        end;
    end_loop;

 end /* philosopher */
```

FIG. 4.8 (continued)

```
module_type Room;

    /* To avoid deadlock occurance, we introduce a room module,
       making sure that there are at most four philosophers in the
       dining room at any given time. Each philosopher must
       request a permission to enter the room and inform when leaving  */

     define
           room_exit , room_entrance  :  synch entry;
     end;

     class
           message  room_msg : inout
                    Ph : int;
           end_message;

     object
           occupancy  : int;
           host_msg   : room_msg;

     initialize

          begin
              occupancy  =  0;
          end;

     begin /* room body */

        loop

           when  occupancy  <  4 implies

                   enter room_entrance host_msg;
                   occupancy  =  occupancy  +  1;

          or

                   enter room_exit host_msg;
                   occupancy  =  occupancy  -  1;

        end_loop;

    end /* Room */
```

FIG. 4.8 (continued)

```
    /* continuation of class part of module dining */

    message  ph_seq : in
                 id : int;
    end_message;

    Forks : array { 1 to 5 } of Fork;

    Phs   : array { 1 to 5 } of Philosopher;

  object
     k          : int;
     pd_number : ph_seq;
     Philos     : Phs;
     ourfork    : Forks;
     Host       : Room;

  initialize

     begin
         k = 1;
         init ourfork; init Host; init philos(ourfork,Host);
     end;

  begin /* dining module body */

     loop

         begin
            call Philos.get_id ph_number;
            k = k + 1;
            exit when k >  5;
         end;

     end_loop;

end /*  Dining */
```

FIG. 4.8 (continued)

Example 4 : The Eight-Queens Problem

The problem is to place eight queens on a chess board so that they do not attack each other, i.e., every row, column and diagonal of the chess board has at most one queen (in this case, exactly one queen will be on each row and each column, since there are exactly eight rows and eight columns on a chess board).

A placement of the eight queens on the board is called a ***configuration.*** A configuration is partial if all the eight queens have not been placed. A configuration is safe if none of the queens attack each other.

The problem has been studied extensively in the computer science literature [29]. There are about 2**32 ways (64! / (56! x 8!)) combinations in which the eight queens can be placed. The problem has been solved using recursive backtracking. We place the nth queen on some row of the nth column, check to see if it can be captured by any queen already on the board, and ,if it cannot, recursively try to place the remaining queens in the remaining columns. We repeat this process until the eighth queen is successfully placed. If the queen can be captured, or the recursive attempt fails, we move this queen to another row and repeat the process. If we cannot place the queen on any row, we backtrack, reporting failure to the previous column.

We consider two functions: ***place*** function, which takes a chessboard, row, and column and returns a new updated board with a queen at that intersection; ***safe*** function, which is true if its argument board has no mutually attacking queens. Our solution using DSDL is similar, except that instead of trying each queen placement in turn, we try all the possible queen positions in a column

concurrently.

A class for queen module-type has been defined. The target-queen is defined as access-type. The creation activities are specified by the ***create*** statement and the use of a pointer object ***target-queen-ptr***. Eight queen modules are created dynamically. Each module receives the message ***queen-msg*** on message m with two slots: column slot that tells in which column to try to fill, and a board slot that contains a representation of those squares of the board already filled.

The action of module Queen is as follows: It receives a message m containing a board and column . It first checks to see if ***m.board*** is safe.If not, the module terminates. If it is safe, the module determines if it has been asked to fill in the ninth column (off-the-edge-of-the-board). If so, it has an answer, ***m.board***. If this is not a terminal search point, then, for each row, the queen modules copies its board, adds a new queen at (row , m.column), creates a new queen module and sends that module the new board, asking it to solve the next column. After all the created modules of a queen module have reported completion of the task, the current module reports completion and terminates.

```
module  Eight_Queens;

    define
          queen_port : bi_dir  port;
    end;

    class
        chess_board : array { 1 to 8, 1 to 8 } of int;

        message  msg : inout
              q_column : int;
              q_board   : chess_board;
        end_message;

        module_type  Queen;

            define
                  q-port : bi-dir  port;
            end;

            class
                message  xmsg : inout
                      column : int;
                      board   : chess_board;
                end_message;

                target-queen : access Queen;

                All-Queens  : array { 1 to 8 } of target-queen;

            object
                    m : xmsg;
                  row  : int;
              new-queen : target-queen;

            begin     /* Queen's body */

                  receive  m  on Eight_Queens.queen_port;

      /* a logical function called ( SAFE ) checks if the board i
            safe  (i.e the board has no attacking queens ). If
            safe = > TRUE then the following actions will take
            place ; otherwise the created module terminates. */
```

FIG. 4.9: Eighit-Queens Program

```
            select
                    when  m.column  =  9  implies

              /* display the solution which is represented in m.board
                      and terminate  the created module */
            else

               /* create another 8 modules to try all posible quee
                    positions in the current column  */

          begin

                 row  =  1;

             loop

   /*  place function is called to add a new queen at ROW and M.COLUMN  */

                 begin
                    m.column  =  m.column  +  1;
                    create(new_queen(row));
                    row  =  row  + 1;
                    send m on q-port;
                    exit when row  =  9;
                 end;

             end_loop;

          end;

     end_select;

end   /*  Queen */

target_queen : access Queen;

All-Queens : array { to } of target-queen;

object
            target_queen_ptr  : target_queen;
                 i            : int;
             queen_msg        : msg;
```

FIG. 4.9 (continued)

```
initialize

    begin

            /* initialize the board to empty and column slot in the queen-msg to 1 */

            null;

    end;
  begin   /* 8-queens body */

         i = 1;
         loop

            begin
              create(target_queen_ptr(i));
              send queen_msg  on queen_port;
              i = i+1;
              exit when i = 9;
            end;

         end_loop;

  end /* Eight_Queens */
```

FIG. 4.9 (continued)

CHAPTER V

CONTRASTS AND COMPARISONS

5.1 Introduction

Over the years, several parallel programming models have been proposed. These models can be classified into two fundamental categories: the shared memory and the message passing models. In the shared memory models, processes communicate with each other by updating and reading common memory. In the message passing models, processes communicate with each other by sending and receiving messages. Messages can be synchronous (blocking) or asynchronous (non-blocking).

In the previous chapters, we gave a full description for a distributed software design language. In this chapter, we shall review some message passing models of languages that were used to design coordinated computing systems; then, we shall compare among them in one hand, and between them and DSDL in the other hand. Finally, we draw some conclusions.

5.2 An Overview of Concurrent Message Passing Models

In this section, we take a closer look at several languages that were used for designing distributed systems. It is difficult to determine exactly how many such languages exists. We have selected a subset for closer study. Ada, CSP, PLITS, SR, and Concurrent C are the languages that we are going to focus on.

5.2.1 Communicating Sequential Processes (CSP)

CSP is a model-language hybrid for describing concurrent and distributed computation. CSP provides a simple parallel command to create a static set of parallel processes. Interprocess communication is done using synchronous ***send*** and ***receive***. The sending process specifies the name of the destination process and the receiving process specifies the name of the sending process. A process that executes a communication primitive (send or receive) is blocked until its partner has executed the corresponding primitive. A rendezvous is established when one process is ready to execute an input statement and the second process is ready to execute the corresponding output statement. If either process is not ready, then the other process is forced to wait. Information transfer is unidirectional. Guarded commands are used to introduce indeterminacy [22, 14, 38].

5.2.2 Programming Language In The Sky (PLITS)

PLITS is a language based on communication with asynchronous messages. PLITS is based on modules that communicate asynchronously by sending each other messages. PLITS queues and sorts messages for the receiving module. The message sending primitive takes a message and destination and delivers that message to that destination (***send*** M ***to*** V). In sending, a module can specify a transaction key which is differentiating module's messages (e.g. ***send*** M ***to*** V ***about*** K), where ***about*** field of message M is the transaction key "K". ***send*** is an asynchronous operation; the sending modules continue computing after sending. Messages are ultimately routed to the destination module's queue. This requires

that there should be an unbounded queue of unreceived messages kept for each module. Messages from a single sender using a particular transaction key arrive in the order sent and are received in the order sent. PLITS has mechanisms for selective message reception. Modules can choose to accept messages in FIFO order. Furthermore, a module can specify that a particular reception is to be the next message about a particular transaction key (***receive*** M ***about*** K); this statement causes the message with transaction key "K" to be removed from the message queue and assigned to M. Also, a module can specify the next message from a particular source module (***receive*** M ***from*** S), or the next message about a particular key and from a particular source module (***receive*** M ***from*** S ***about*** K). PLITS also has mechanisms for abstracting and protecting message information. Modules can be dynamically created by using ***new*** function and destroyed by executing the command ***self-destruct.*** Information transfer is unidirectional in PLITS [14].

5.2.3 Concurrent C

Concurrent C extends the C language by adding support for distributed programming. It is based on Ada's rendezvous model. A process in Concurrent C has a specification part and a body, just like tasks in Ada. Concurrent C processes interact with each other by means of transactions which are associated with processes. There are two kinds of transactions: synchronous and asynchronous. A process calling a synchronous transaction is blocked until the called process accepts the transaction and returns the result if any. Synchronous transactions allow bidirectional communication between the interacting processes. They can also be used for process synchronization. The calling and

receiving processes synchronize at the transaction call. A process calling an asynchronous transaction is allowed to continue immediately after making the call. The transaction is accepted by the called process whenever it is ready to do so. An asynchronous transaction can be used only for unidirectional information transfer. A timed transaction call allows the calling process to withdraw a synchronous transaction call provided the call has not been accepted within a specified period. A transaction is equivalent to an Ada entry and it differs from Ada entry in that a transaction may return a value. Processes are created explicitly using ***create*** primitive which can pass parameters to the created processes. The new processes can be given a priority. Processes communicate through rendezvous mechanism. Concurrent C asynchronous transactions (equivalent to asynchronous message passing) do not return a value [22, 38].

5.2.4 Synchronizing Resources (SR)

SR is a language for programming distributed operating systems and applications. An SR program consists of one or more resources. A resource can contain several processes and these may share data. Synchronization among these processes is supported by the use of semaphores. A resource may contain an initialization and a termination process. A resource terminates when it is killed by the destroy command. A program terminates when all its processes terminate or block. Processes communicate either through the common storage of a resource or by requests to named entries in other processes. These requests can be either synchronous or asynchronous. SR uses a construct similar to select statement to deal with nondeterminism. SR supports mechanisms for handling failures. Exception handlers can be used to handle failures detected by the run-

time system [14, 22].

5.2.5 Ada

Ada was designed to fit the needs for concurrent and distributed programming. Concurrency is based on sequential processes called tasks. Each task has its type. A task consists of a specification part which describes how other tasks can communicate with it, and a body which contains the executable statements. Ada has both explicit processes creation, and process creation through lexical elaboration. Tasks usually communicate through the rendezvous mechanisms. Tasks can also communicate through shared variables. A task can call an entry of another task by using entry call statement similar to a procedure call statement. The rendezvous mechanism is based on entry declarations, entry calls, and accept statements. Ada uses a ***select*** statement to express nondeterminism. Ada has an exception handling mechanism for dealing with software failures [14, 22, 17].

5.3 Comparisons

5.3.1 Dimensions of Distribution

The comparisons among systems will be held according to three issues: general goals and structure of each system, intrasystem communication, and other issues.

5.3.2 Goals and Structure

5.3.2.1 Problem Domain

Much of the diversity among systems arises from differences in task domains; the different systems are meant to be solutions to different problems. For example, SR is directed at implementing operating systems, Ada concerned with embedded systems, and Concurrent C concerned with loosely-coupled distributed networks or tightly-coupled multiprocessor. CSP is not only a language for systems implementation, but also a model of both hardware and program semantics. PLITS provides pragmatics to ease the task of programming as strong typing, symbolic tokens, and data abstractions. These pragmatics do not change semantics of the systems. However, the domain of DSDL is to support the design of software for concurrent and distributed systems such as real time systems, operating systems, and distributed applications.

5.3.2.2 Expressing Parallelism

Almost all distributed systems support concurrent computations. They use

explicit processes to express concurrency. In an explicit-process system, the programmer causes process entities to be created. Once created, these processes run concurrently. Several systems call their processes, "processes", others use another name for the same concept: tasks in Ada and modules in PLITS.

Some systems require that all processes to be defined at system creation (static processes) such as CSP and SR systems. Others allow creating new processes during execution (dynamic). Two different methods support dynamic process creation: explicit allocation (dynamic) and lexical elaboration. Systems with explicit allocation have a statement to create new process; Ada and PLITS are examples that support dynamic creation. Lexical elaboration creates processes by declaration; Ada, PLITS, and Concurrent C are examples of such systems.

In DSDL, the unit of parallelism is refferd to as a process, a module or a software component in general. Components either created statically and considered to be fixed for the lifetime of software system, dynamically by an executable statement such as create statement, or lexically by declarations.

5.3.3 Communication

5.3.3.1 Synchronization

The differences that distinguish concurrent computation from serial processing center on communication and synchronization among processes. The issue of synchronous versus asynchronous is a notable division between systems. Systems like PLITS are asynchronous, while Ada and CSP are synchronous

ones. SR and Concurrent C provide both synchronous and asynchronous communication. However, DSDL provides a rich and powerful message transmission constructs that are capable of expressing designs in synchronous or asynchronous fashion.

5.3.3.2 *Buffering*

A system's buffer size is the number of messages from a given process that can be pending at one time. Some systems allow unbounded buffer, others allow bounded buffer size. Synchronous systems have bounded buffering. Message-based asynchronous systems as, PLITS, SR and Concurrent C have unbounded buffering. Systems like Ada and SR permit processes to share memory. Shared memory is a form of bounded asynchronous communication. DSDL permits processes to share memory at module level. Furthermore, DSDL assumed to support bounded and unbounded buffering.

5.3.3.3 *Information Flow*

Information flow between two communicating processes may be unidirectional or bidirectional. The various systems propose different organizations for information flow. Unidirectional communication is a sequence of message-based asynchronous systems as PLITS and SR. Only a single synchronous system CSP has unidirectional information flow. Synchronous systems either have simultaneous bidirectional communication or delayed bidirectional in which processes enter a rendezvous. Ada, SR, and Concurrent C are examples of systems who use delayed bidirectional. Concurrent C and SR has unidirectional information flow. DSDL supports both unidirectional and

bidirectional communications.

5.3.3.4 Communication Control

Communication control concerns with the actions that processes take to communicate including the facilities they have for choosing a communication partner. Systems specify roles for the calling and called processes. Some of these systems treat the called process as a passive server (***passive***) that accepts requests without controlling the order of their reception. Other systems like Ada, PLITS, SR, and concurrent C allow the called process some freedom in choosing which requests to serve; such systems treat the called process as an active server (***active***). Some systems introduce input and output guards (***I/O guards***) that provide further concurrency control. Ada, CSP, and Concurrent C provide I/O guards. Furthermore, Ada and Concurrent C permit time-outs by both calling and called tasks/processes. CSP treat communicators as equals (***equal***) Some systems add their own features for communication control; PLITS and SR allow filtering of requests by origin (***send-filt***) SR orders requests by priority (***priority***), and Concurrent C does that using ***by*** and ***suchthat*** clauses. Their guards examine the internal state of the process and/or the message (***msg-grd***).

DSDL treats the called module/process as an active server. The called module/process has a freedom in choosing which request to serve. DSDL provides I/O guards by nondeterminacy mechanism supported by the language ***select*** and ***selective loop*** constructs. The IFF-condition clause supports (msg-grd), where the condition is a boolean expression; its value depends on the state of current process or on the contents of message received. This condition

determines whether to accept the message or not. The IFF clause in send statement allows filtering of requests by origin (send-filt). The BY clause orders requests by priority. DSDL treats communicators as being equals or non-equals; and the TO, FROM, and ON clauses determine that whether communication is synchronous or asynchronous. DSDL permits time-outs by both calling and called processes.

5.3.3.5 Communication Connection

Connection is an issue of naming: to what does a communicating process refer ?. Systems use different syntactic forms to establish communication: ports, names, and entries. PLITS and CSP are an example of systems who use naming to communicate. Ada and SR focus communication on an entry in the called process. In Ada, a called process can have several entries and can accept requests from them in an order determined by program control. Concurrent C uses transactions to establish communication. DSDL supports the use of ports, entries, and direct naming.

5.3.4 Other Issues: Time, Fairness, and Failure

5.3.4.1 Time

In distributed systems, it is meaningless to refer to the absolute time. Absolute time complicates the problem of synchronizing the distributed clocks. However, the use of relative time intervals suffice to enable solving problems raised by blocking, deadlock situation or failure. The time intervals (time-outs) are used in systems like Ada and Concurrent C. DSDL supports the notion of

DELAY construct where it specifies a time interval under which a component waits (blocks) until the time interval expires. When the DELAY construct is used with a selective receive mechanism, a component blocks until the message arrives or the time interval expires.

5.3.4.2 Fairness

Since processes interact through communication, our concerns are with communication fairness. Distributed systems exhibit three attitudes towards fairness as antifair, weak fairness, or strong fairness. CSP is an antifair system. Ada, PLITS, and SR are queue-based and strongly fair systems. DSDL semantics assumes a strongly fair system model.

5.3.4.3 Failure

Distributed systems have high potential for reliability and availability. However, the communication system is subject to delays and failures. Several systems have mechanisms for dealing with failure. Ada has mechanisms for handling failure and distributed termination including time-outs and exception handlers. Concurrent C supports fault tolerance by allowing to replicate critical processes. A program continues to operate with full functionality as long as at least one of the copies of a replicated process is operational and accessible. All details of interaction with replicated processes are handled by the fault tolerance concurrent C run-time system. Fault tolerance also provides facilities for requesting notification upon process termination, detecting processor failure during process interaction, and automatically terminating slave processes. DSDL supports the handling of failures, including distributed termination, time-

outs and exception handlers. Exception handlers are two types: local exception handler where exceptions are raised locally within the scope of a module or a process, and distributed exception handler, where signals are announced to other remote modules as discussed thoroughly in section 3.8. Table 5.1 summarizes the features supported by these languages including DSDL.

5.4 Concluding Remarks

We are seeking for a unified ideal language model, one that describes what to do by specifying an appropriate set of primitives which are sufficient to achieve any type of required behaviors, one that has the ability to express all types of behaviors for design description of distributed computing software.

What mechanisms built from these primitives should an ***ideal*** language model provide ?. What features this ideal language has to have ?.

The design of software design languages has yet to be perfected. Each language has many shortcomings, but each can be considered "successful" in comparison with other languages that have been designed. Comparing an ideal language for distributed computing with DSDL, we can not claim that DSDL is an ideal one, but in any means it is close enough to generality.

System	Ada	Concurrent C	CSP	PLITS	SR	DSDL
Task domain	systems embedded	systems	systems semantic	Pragma-tics	systems (OS)	Design of dist. systems
Unit of Parallelizm	Task	Process	Process	Module	Process	Process Module
Process creation	dynamic lexical	Lexical	static	dynamic lexical	static	dynamic static lexical
synchroniza-tion	synch	synch asynch	synch	asynch	synch asynch	synch asynch
Buffering	bnd	bnd unbnd	bnd	unbnd	bnd unbnd	bnd unbnd
Information flow	Bi	Bi Uni	Uni	Uni	Bi Uni	Bi Uni
Communication control	Active I/O grds time-out	Active I/O grds priority time-out	equal I/O grds	Active send-filt	Active I-grd msg-grd send-filt priority	Active I/O grds msg-grd send-filt priority time-out
Comm. Connection	entry	entry (transaction)	name	name	entry	port entry name
Time	time-out	time-out	-	-	-	time-out
Fairness	strong	strong	anti	strong	strong	strong
Failure	local exception handler & dist. term.	replicated critical processes & termination	-	-	-	local & dist. exception handlers & dist. term.

TABLE 5.1

CHAPTER VI

CONCLUSION AND FUTURE WORK

Getting a program to work is not sufficient, but getting it designed right is the important thing. Given double the budget and schedule, it is preferable not to spend the extra on testing but spend it in design. Design languages are a clear and concise communication medium. Their purpose is to help designers communicate by identifying commonly understood terms and concepts and to capture the design decisions in a machine processable form as distinguished from machine executable form. A software design language serves as a go-between of the early design phase and the implementation phase. So it has to compromise two needs : the need to be easily modifiable and as informal as possible so as to be able to express change; on the other hand, the need to be strict in form so as to be easily transformed to an implementation language.

The early SDLs allowed informality in the expression of programs in order to satisfy the first need and included some simple and basic control constructs to satisfy the second. These SDLs helped to solve many design problems but proved to be inadequate for large and complex software projects. Thus, a new family of SDLs appeared which introduced abstraction as means of connection with the early design phase and both modularity and formality, similar to that included in many high-level languages, for connection with the implementation phase. This formality includes strong typing, declaration of data objects of both standard and user-defined types, and constructs such as classes.

DSDL is adequate tool to support detailed design phase. It is based on a rich base of communication primitives, control constructs, and allow concurrency in a distributed environment. DSDL presents the system design in terms of modules and processes that communicate through message passing.

DSDL provides communication connection through ports, entries, and direct naming. It allows different connection patterns between communicating modules/processes. DSDL provides facilities for both synchronous and asynchronous message passing. DSDL provides constructs for nondeterminism. Exception handlers are provided to handle upnormal situations. DSDL allows the use of comments every where in the design. DSDL has a processor which checks the syntax of its programs.

It should be noted that the design decisions expressed by DSDL can transformed to a concurrent programming language. If the primitives are not provided by the programming language , then a set of library routines can be accessed to simulate the actions to be provided.

It has been found that designers and programmers learn how to use SDLs effectively. Experience has proved that much less valid program errors are found if SDLs are used as design tools [1]. As future work, we recommend the following:
Semantic checking/analysis should be supported by the language processor
Supporting the processor with a language-based editor. This editor will provide a template displaying a construct format that will enable the designer to check the design syntactically and semantically in an interactive way.

Maintaining separate designs in a database; a knowledge base of design description has to be created and will be accessed by a database manager of design descriptions.

Coming up with an automated tool to transform the design expressed by DSDL into a suitable concurrent programming language. This will open the door to automatic programming where the code will be generated automatically from design specifications.

BIBLIOGRAPHY AND REFERENCES

[1] I. Sommerville, ***Software Engineering***, Addison-Wesley, Workingham England, 1987, pp 334.

[2] Panayotis E. Pintelas & Vasilios Kallistros, ***"An Overview of Some of Software Design Languages"***, The Journal Of Systems and Software, Vol. 1, pp 125-138, 1989.

[3] R. S. Pressman, ***Software Engineering: A Practitioner's Approach***, McGraw-Hill Book Company, 1987.

[4] L. J. Peter, ***"Software Representation and Composition Techniques"***, Proceeding of the IEEE, Vol. 68, No. 9, pp 1085-1093, September 1980.

[5] Stephen S. Yau, Jeffery J. Tasi, ***"A Survey of Software Design Techniques"***, IEEE Trans. on Software Engineering, Vol. SE-12, No. 6, pp 713-721, June 1986.

[6] F. Beichter, O. Herzog, H. Petzsch, ***SLAN-4 "A Software Specification and Design Language"***, IEEE Trans. on Software Engineering, Vol. 10, pp 155-162, March 1984.

[7] S. A. Sutton, V. R. Basili, ***"The Flex Software Design System : Designers need languages, Too"***, Computer, Vol. 14, pp 95-102, November 1981.

[8] M. E. Falla, ***"The Gamma Software Engineering System"***, The Computer

Journal, Vol. 24, No. 3, pp 235-242, 1981.

[9] J. L. Archibald, B. M. Leavenworth, L. R. Power, ***"Abstract Design and Program Translator: New Tools for Software Design"***, IBM System Journal, Vol. 22, pp 170-187, 1983.

[10] D. L. Parans, ***"On the criteria to be used in Decomposing System into Modules"***, Communications of ACM, Vol. 15, pp 1055-1058, 1972.

[11] Grady Brooch, ***Software Engineering with ADA***, The Benjamin Cummings Pub. Co. Inc., 1986.

[12] W. T. Jones, S. A. Kirk, ***"APL as a Software Design Specification Language"***, Computer Journal, Vol. 23, No. 3, pp 230-232, 1980.

[13] V. Rajlich, ***"Paradigms for Design and Implementation in Ada"***, Communications of the ACM, Vol. 28, No. 7, pp 718-727, 1985.

[14] Filman & Friedman, ***Coordinated Computing Tools and Techniques for Distributed Software***, McGraw-Hill Book Company, 1984.

[15] D. W. Brown, C. D. Carson, W. A. Montgomery, P. M. Zislis, ***"Software Specification and Prototyping Technologies"***, AT&T Technical Journal, July/August 1988.

[16] J. A. Stankovic, ***"A series Problem for Next-Generation Systems, Software Specification and Prototyping Technologies"***, AT&T Computer, Oct. 1988.

[17] Narian Gehani, ***Ada An Advanced Introduction***, Prentice-Hall Inc. 1983.

[18] P. V. Leer, *"Top-down Development Using a Program Design Language"*, IMB System Journal, Vol. 15, No. 2, pp 155-170, 1976.

[19] S. S. Yau, C. C. Yang, S. m. Shaz, *"An Approach to Distributed Computing System Software Design"*, IEEE Trans. on Software Engineering, Vol. SE-7, pp 427-436, July 1981.

[20] S. S. Yau, M. U. Caglayan, *"Distributed Software System Design Representation Using Modified Petri Nets"*, IEEE Trans. on Software Engineering, Vol. SE-9, pp 733-745, November 1983.

[21] B. S. Chen, R. T. Yeh, *"Formal Specification and Verification of Distributed System"*, IEEE Trans. on Software Engineering, Vol. SE-9, pp 710-722, November 1983.

[22] H. E. Bal, J. G. Steinet, A. S. Tanenbaum, *"Programming Languages for Distributed Computing Systems"*, ACM Computing Surveys, Vol. 21, No. 3, pp 261-322, September 1989.

[23] G. V. Bochman, *Concepts for Distrbuted systems Design*, Springer-Verlag Berlin Heidelberg 1983.

[24] H. Freeman, P. M. Lewis ii, *Software Engineering*, Academic Press, Inc. 1980.

[25] B. T. Mynatt, *Software Engineering with Student Project Guidance*, Prentice-Hall, Inc. 1990.

[26] M. Sloman, J. Kramer, ***Distributed Systems and Computer Networks***, Prentice-Hall, Inc. 1987.

[27] G. R. Andrews, F. B. Schneider, ***"Concepts and Notations for Concurrent Programming"***, Computing Surveys, Vol. 15, No. 1, pp 3-43, March 1983.

[28] B. I. Witt, ***"Communicating Modules : A software Design Model for Concurrent Distributed Systems"***, Computer Vol. 18, No. 1, pp 67-77, January 1985.

[29] J. Tremblay, P. G. Sorenson, ***The Theory and Practice of Compiler Writing***, McGrawe-Hill, Inc. 1985.

[30] W. M. Waite, G. Goos, ***Compiler Construction***, Springer-Verlag New York Inc., 1984.

[31] A. V. Aho, R. Srthi, J. D. Ullman, ***Compilers principles, Techniques, and Tools***, Addison-Wesley Publishing Company, 1988.

[32] A. T. Schreiner, H. G. Friedman, ***Introduction To Compiler Construction With UNIX***, Prentice-Hall, Inc. 1985.

[33] B. W. Kernighan, R. Pike, ***The UNIX Programming Environment***, Prentice-Hall, Inc. 1984.

[34] A. D. Mcgettrick, ***The Definition of Programming Languages***, Publishers Production, Inc. New York 1980.

[35] A. B. Tucker, ***Programming Languages***, McGrawe-Hill, Inc. 1986.

[36] F. G. Pagan, ***Formal Specification of Programming Languages: A Panoramic Prime*** Prentice-Hall, Inc. 1981.

[37] T. W. Pratt, ***Programming Languages: Design and Implementation***, Prentice-Hall, Inc. 1984.

[38] N. Gehani, ***The Concurrent C Programming Language***, Prentice-Hall, Inc. 1989.

[39] J. P. Queille, ***"The CESAR System: An Aided Design and Certification system for Distributed Applications"***, The 2nd International Conference on Distributed Computing systems, pp 149-161, paris, 1981.

[40] C. M. Li, M. T. Liu, ***"DISLANG: A Distributed Programming Language/System"***, The 2nd International Conference on Distributed Computing systems, pp 162-172, paris, 1981.

[41] R. Williamson, E. Horowitz, ***"Concurrent Communication and Synchronization Mechanisms"***, Software- Practice and Experience Vol. 14, No. 2, pp 135-151, February 1984.

[42] G. R. Andrews, ***"Synchronizing Resources"***, ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, pp 405-430, October 1981.

[43] M. L. Scott, ***"Language Support for Loosely Coupled Distributed Programs"***, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, pp 88-103, January 1987.

[44] P. A. Ng, ***Modern Software Engineering: Foundations and Current Perspectives***, Van Nostrand Reinhold, New York, 1990.

[45] L. Kleinrock, ***"Distributed Systems"***, Communications of the ACM, Vol. 28, No. 11, pp 1200-1213, November 1985.

```
<module> ::= <module-heading>
             [<definition>][ <classes> ][ <objects> ][ <components> ]
             [ <initialization> ]
             <body>

<module-heading> ::= MODULE <module-name> [ (instances) ] ";"

<module-name> ::= <identifier>

<instances> ::= "(" digit ")"

<definition> ::= DEFINE <list-of-communication-primitives> END ";"

<list-of-communication-primitives> ::= <communication-primitives> |
             <list-of-communication-primitives> <communication-primitives>

<communication-primitives> ::= <name-def> | <entry-def> | <port -def >

<name-def> ::= <list-of-component-names> ":" <direction> ";"

<list-of-component-names> ::= <component-name> | <list-of-component-names>
                                                 "," <component-name>

<component-name> ::= <module-name> | <process-name>

<direction> ::= EXPORT | IMPORT | EXIM

<entry-def> ::= <list-of-entry-names> ":" <entry-type> ENTRY ";"

<list-of-entry-names> ::= <entry-name> | <list-of-entry-names> "," <entry-name >
```

```
<module> ::= <module-heading>
                [<definition>][ <classes> ][ <objects> ][ <components> ]
                [ <initialization> ]
                 <body>

<module-heading> ::= MODULE <module-name> [ (instances) ] ";"

<module-name> ::= <identifier>

<instances> ::= "(" digit ")"

<definition> ::= DEFINE <list-of-communication-primitives> END "; "

<list-of-communication-primitives> ::= <communication-primitives> |
                 <list-of-communication-primitives> <communication-primitives>

<communication-primitives> ::= <name-def> | <entry-def> | <port -def >

<name-def> ::= <list-of-component-names> ":" <direction> ";"

<list-of-component-names> ::= <component-name> | <list-of-component-names>
                                                  "," <component-name>

<component-name> ::= <module-name> | <process-name>

<direction> ::= EXPORT | IMPORT | EXIM

<entry-def> ::= <list-of-entry-names> ":" <entry-type> ENTRY ";"

<list-of-entry-names> ::= <entry-name> | <list-of-entry-names> "," <entry-name >
```

<entry-name> ::= <identifier>

<entry-type> ::= SYNCH | ASYNCH

<port-def> ::= <list-of-port-names> ":" <mode> PORT [<connect>]

<list-of-port-names> ::= <port-name> | <list-of-port-names> "," <port-name>

<mode> ::= UNI-DIR | BI-DIR

<connect> ::= FROM <list-of-component-names> |
TO <listcomponent-names> | BROADCAST

<port-name> ::= <identifier>

<classes> ::= CLASS <list-of-classes>

<list-of-classes> :: <class-type> | <list-of-classes> <class-type>

<class-type> ::= <process-type> | <module-type> | <msg-type> |
<record-type> | <array-type> | <access-type>

<process-type> ::= <process-type-heading>
[<definition>][<classes>][<objects>][<initialization>]
<body>

<process-type-heading> ::= PROCESS-TYPE < process-type-name >
[(instances)][(list-of-parameters)] ";"

<list-of-parameters> ::= <parameter> | <list-of-parameters> <parameter>

```
<parameter> ::= <identifier-list> ":" <type>

<process-type-name> ::= <identifier>

<module-type> ::= <module-type-heading>
                  [ <definition> ][ <classes> ][ <objects> ][ <initialization> ]
                  <body>

<module-type-heading> ::= MODULE-TYPE <module-type-name>
                          [ (instances) ][ (list-of-parameters) ] ";"

<module-type-name> ::= <identifier>

<msg-type> ::= MESSAGE <msg-name> : [ IN | OUT | INOUT ]
                        <slot-definition>
                        { slot-definition }
               END-MESSAGE ";"

<msg-name> ::= < identifier >

<slot-definition> ::= <identifier> ":" <slot-type> ";"

< slot-type > ::= INT | REAL | CHAR | BOOLEAN | IN-SIGNAL | OUT-SIGNAL |
                  <array-name>

<record-type> ::= RECORD <record-name>
                         <field-definition>
                         { field-definition }
                  END-RECORD ";"
```

<record-name> ::= <identifier>

<field-definition> ::= <identifier> ":" <field-type> ";"

<field-type> ::= INT | REAL | CHAR | BOOLEAN | EXCEPTION |
<array-name> | <record-name> | <component-indicator>

<array-type> ::= <array-name> ":" ARRAY [lbound .. upbound] OF <type> ";"

<array-name> ::= <identifier>

<lbound> ::= <digit>

<upbound> ::= <digit>

<access-type> ::= <component-indicator> ":" ACCESS <component-type-name> ";"

<component-indicator> ::= <identifier>

<objects> ::= OBJECT <list-of-objects>

<list-of-objects> ::= <object-def> | <list-of-objects> <object-def>

<object-def> ::= <identifier-list> ":" <type> ";"

< identifier-list > ::= < identifier > { , identifier }

<type> ::= INT | REAL | CHAR | BOOLEAN | IN-SIGNAL | OUT-SIGNAL |
EXCEPTION | <process-type-name> | <module-type-name> |
<record-name> | <array-name> | <msg-name> |
<component-name>

<initialization> ::= INITIALIZE

BEGIN <sequence-of-statements> END ";"

<components> ::= <component> | <components> <component>

<component> ::= <process> | <module>

<process> ::= <process-heading>

[<definition>][<classes>] [<objects>][<initialization>]

<body>

<process-heading> ::= PROCESS <process-name> [(instances)] ";"

<process-name> ::= <identifier>

<body> ::= BEGIN

<sequence-of-statements> [<exception-handler-part>]

END

<exception-handler-part> ::= EXCEPTION <sequence-of-exception-handlers>

<sequence-of-exception-handlers> ::= <exception-handler> |

<sequence-of-exception-handlers> <exception-handler>

<exception-handler> ::= WHEN <exception-choice> { , exception-choice } = >

<sequence-of-statements>

<exception-choice> ::= <exception-name>

<exception-name> ::= <identifier>

<sequence-of-statements> ::= <statement> { statement }

<statement> ::= <receive-statement> | <send-statement> | <call-statement
<enter-statement> | <announce-statement> | <raise-statement> |
<handle-statement> | <delay-statement> | <abort-statement> |
<terminate-statement> | <select-statement> | <loop-statement> |
<null-statement> | <deffered-statement> | <create-statement> |
<exit-statement> | <assign-statement> | <init-statement>

<receive-statement> ::= RECEIVE <msg-name> [FROM <component-name>]
[ON <port-name>] [IFF <condition>]
[BY <expression>] [REPLY <msg-name>] ";"

<send-statement> ::= SEND <msg-name> [TO <destination>]
[ON <port-name>] [IFF <condition>] ";"

<destination> ::= <entry-name> | <component-name>

<enter-statement> ::= ENTER <entry-name> <msg-name>
[FROM <component-name>] [IFF <condition>]
[BY <expression>]
[BEGN <sequence-of-statements> END] ";"

<call-statement> ::= CALL <callee> <msg-name>
[IFF <condition>] ";"

<callee> ::= <destination> | <ON <port-name>

<announce-statement> ::= ANNOUNCE <list-of-outsignals>
[ON <port-name>] [TO <component-name>]
[WITH <msg-name>] ";"

<handle-statement> ::= HANDLE <list-of-insignals> [IN <msg-name>]
[ON <port-name>] [FROM <component-name>]
END-HANDLE ";"

<raise-statement> ::= RAISE <list-of-exception-names> ";"

<delay-statement> ::= DELAY <expression> ";"

< list-of-exception-names > ::= < exception-name > { , exception_name }

<terminate-statement> ::= TERMINATE <local-component> [WHEN <conditon>] ";"

<local-component> ::= <component-name>

<abort-statement> ::= ABORT <list-of-components> [WHEN <condition >] ";"

<list-of-component> ::= <component-name> { , component-name }

<select-statement> ::= SELECT <guarded-command> { OR guarded-command }
[ELSE <guarded-command>]
END-SELECT ";"

<guarded-command> ::= [WHEN <condition> =>]
<sequence-of-statements>

<loop-statement> ::= LOOP <sequence-of-alternatives> END-LOOP ";"

<sequence-of-alternatives> ::= <guarded-command> | <sequence-of-alternatives>

OR <guarded-command>

<null-statement> ::= NULL ;

<deffered-statement> ::= DEFFERED ";"

<create-statement> ::= CREATE <component-name>

[(function-name(list-of-arguments))] ";"

<function-name> ::= <identifier>

<list-of-arguments> ::= <identifier-list>

<init-statement> ::= INIT <component-name> [(list-of-arguments)] ";"

<exit-statement> ::= EXIT [WHEN <condition>] ";"

<assign-statement> ::= <variable> := <expr>

<variable> ::= <identifier>

<expr> ::= <digit> { digit }

| <identifier>

| <expr> "+" <expr>

| <expr> "-" <expr>

| <expr> "*" <expr>

| <expr> "/" <expr>

| <expr> "<" <expr>

| <expr> ">" <expr>

| <expr> "=" <expr>

| <expr> "&" <expr>

| <expr> "£" <expr>

| <expr> "**" <expr>

| <expr> "< =" <expr>

| <expr> "> =" <expr>

| <expr> "< >" <expr>

| "~" <expr>

| "-" <expr>

| "(" <expr> ")"

<identifier> ::= <letter> { [_] letter | digit }

<letter> ::= A | B | C | ..... | Z | a | b | c | ..... | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9