IMPLEMENTING CONCURRENCY IN A PROCESS-BASED LANGUAGE

Nurudeen Lameed

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

March 2008

© Nurudeen Lameed, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-40943-5
Our file Notre référence
ISBN: 978-0-494-40943-5

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

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

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Implementing Concurrency in a Process-Based Language

Nurudeen Lameed

Object-oriented programming has been very successful for general purpose programming tasks for almost two decades. It is hard to imagine another paradigm replacing it. But software systems are becoming ever more complex and hard to maintain. Adapting to new hardware will create further problems. The current combination of complex scope rules, inheritance, aspects, genericity, and multithreading cannot provide the flexibility needed for the effective implementation, maintenance, and refactoring of parallel and distributed systems. We believe it is time for a change of approach to software development.

Software must be modified to match today's needs but must not place even greater strain on software developers. The prevailing software development practice makes management and maintenance of software unnecessarily difficult. We describe an approach that we believe will reduce the difficulties of software development and maintenance.

The programming language Erasmus is being developed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory, UK. Erasmus is based on *communicating processes* organized within cells that communicate through message passing.

This thesis describes issues related to the implementation of concurrency in Erasmus. In particular, it explains how the meaning of a program can be separated from its deployment

onto multiprocessor/distributed systems. Through this approach, software investment may be preserved when new features are added or when functionality does not change but the environment does.

Acknowledgments

I thank the Almighty God for His divine guidance and blessing that have made this thesis possible.

My profound gratitude goes to Dr. Peter Grogono for his advice and support. I have gained a lot of knowledge through his wisdom and insights. His advice has also been very invaluable throughout this thesis work.

I also wish to thank all the faculty members and staff of the Computer Science Department particularly the Graduate Advisor, Ms. Halina Monkiewicz for her advice in my first year at Concordia University.

I am grateful to the faculty of Engineering and Computer Science, Concordia University for supporting in part this thesis work.

I thank all my friends and colleagues with whom I shared wonderful and memorable moments.

Finally, I would like to thank my family, my beloved wife, Aderonke and my daughters, Hanifah and Azizah for their support, encouragement and understanding without which this thesis would not have been possible.

Contents

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Background 3

1.1.1 The Object Model 6

1.1.2 The Process Model 7

1.2 Thesis Outline 8

1.2.1 The Erasmus Project 8

1.2.2 The Objectives of the Thesis 8

2 Related Work 10

2.1 Language Style 11

2.2 Concurrent Pascal 11

2.3 Communicating Sequential Processes (CSP) 14

2.3.1 Communication in CSP 15

2.4 Ada 17

2.4.1 Communication in Ada 18

2.5 Occam 24

2.5.1 Occam-π 31

2.6 ABCL/1 33

2.7 Erlang 36

2.7.1 Communication in Erlang 36

2.7.2 Support for distributed applications 39

2.8 Joyce 40

2.9 Cilk 43

2.10 Java Programming Language 45

2.11 Mozart/Oz 49

2.12 SALSA 52

2.12.1 Actor Model 53

2.12.2 Actors in SALSA 54

2.13 Discussion 58

3 Overview of Erasmus Programming Language 60

3.1 Objectives 60

3.2 Language Description 62

3.2.1 Types 64

3.2.2 Statements 65

3.2.3 Communication in Erasmus 68

3.2.4 Channels and Protocols 69

3.2.5 Ports 72

3.2.6 Select statement 76

3.2.7 Recursion, Dynamic Process Creation and Composition 79

4 Communication in Erasmus 84

4.1 Communication 84

4.1.1 Notation and Definitions 85

4.1.2 Channel Structure 85

4.1.3 Basic message transfer 86

4.1.4 `select` Statements 88

4.1.5 Embedded Receives 92

4.2 Mapping of Cells to Processors 95

4.2.1 Design 98

4.2.2 Testing and Results 106

4.2.3 Granularity of Communication 112

5 Communication in Erasmus and other languages **114**

5.1 Language Syntax for Communication 114

5.2 Communication via shared variables 115

5.3 Communication by message passing 117

5.3.1 Channel and Protocols 118

5.3.2 Select statement and Non-determinism 120

5.3.3 Message Typing and Serialization 122

6 Conclusions and Future Work **123**

6.1 Conclusions 123

6.2 Future Work 125

Bibliography **126**

Appendices **134**

.1 Syntax 135

List of Figures

1 A diagram corresponding to the program of Listing 14 70

2 A simple configuration file 70

3 A diagram corresponding to the program of Listing 15 75

4 A diagram corresponding to the program of Listing 19 83

5 A diagram corresponding to the program of Listing 22 97

6 Communication between a client and a server processes 101

List of Tables

1 Summary of concurrent languages 58

2 Functions used in pseudocode 86

3 Policies for the `select` statement. 89

4 Summary of tests in Case one (Frequent Communications) 112

5 Summary of tests in Case two (Infrequent Communications) 112

Chapter 1

Introduction

The desire to build reliable operating systems motivated the introduction of concurrent programming [19]. This has had a profound effect on software development. Many challenging problems have been solved. Today, software controls machines such as NASA's Mars Exploration Rover and many more interesting, sophisticated and crucial applications have been built. Success in software development however is generating increasing demands for ever more complex and challenging computations. Consequently, software is becoming increasingly complex to design. To cope with modern challenges in software development, various tools, languages, techniques and approaches have been proposed, implemented and utilized. The transition from procedural to object-oriented programming helped to facilitate construction of software that otherwise would have been inconceivable. However, increase in software complexity as a result of changes in environment and functionality has exposed some significant limitations of objected-oriented approach: current practice makes software enhancement and refactoring difficult. To address modern software requirements and challenges, Object-oriented languages have been extended with many features. This has further

increased the complexity of object-oriented programming.

Hardware technologies aimed at boosting computational performance remain a major driver of advances in software development. The drive for continued performance gains is causing major processor manufacturers, notably Intel and AMD, to produce microprocessors with multiple *cores* on a chip. The hope is that multi-core architectures will boost performance by having parallel processors execute different instructions. But existing applications are largely sequential. Hence, to achieve true performance gains software must be carefully written to exploit hardware parallelism [46, 52]. Many believe future computations will be largely driven by the multi-core technology.

The proportion of concurrent applications is on the increase, yet concurrent programming has not become the mainstream programming technique. It has however been described as the next major revolution in software development [54]. It is generally believed that concurrent programming is hard. Experts also agree that concurrent programming is hard because mainstream programming languages do not provide suitable abstractions for expressing and controlling concurrency [43, 53, 34].

In responding to the foregoing challenges, Peter Grogono of Concordia University, Canada and Brian Shearing of The Software factory, UK, designed and proposed a new language, *Erasmus.* The language is based on *communicating processes.*

In this thesis, we describe issues related to the implementation of concurrency in Erasmus. In particular, we explain how the meaning of a program can be separated from its deployment onto multiprocessor/distributed systems.

1.1 Background

The development of the multiprogramming system led to the introduction of concurrent programming. Following this, various notations, constructs and languages were proposed. The idea of organizing programs into collections of processes was pioneered at MIT in the CTSS project [48]. Processes are program modules with private data and sequence of statements that operate on those data. Processes are loosely coupled because they only have any effect on one another during rare and brief communications. Processes may have parameters.

Dijkstra [24] proposed the first notation for expressing parallel processes. Parallel execution of statements is constructed by enclosing the sequence of statements within a `parbegin` and `parend`. For example:

```
begin S1; parbegin S2; S3; S4; parend; S5 end
```

statement *S1* is executed followed by parallel execution of $S2, S3, S4$. $S5$ is executed after $S2$, $S3$, $S4$ have finished. Processes communicate via shared variables. It was soon discovered that simultaneous access of shared variables often led to data inconsistencies.

To overcome the problem of data inconsistencies, critical sections were proposed. A critical section is a group of statements that a process can execute with exclusive access to the processor. If a process accesses shared variables only in its critical sections, inconsistencies cannot arise. However, processes may have to wait for a process to complete its critical section; this can cause inefficiency. Critical sections must be coded with care to avoid deadlock.

A special kind of shared variable called *semaphore* [24] was proposed to address the limitations of shared variables as communication channels. Semaphores can *only* have *non-negative integer* values and are typically implemented with queues — processes waiting for a semaphore are put in the semaphore's queue.

Semaphores simplify mutual exclusion problems. Semaphores, used for mutual exclusion are generally called *binary semaphores* because they can have values 0 or 1 only. General semaphores may have higher values; they are called *counting semaphores* because they can be used to indicate the number of available resources. Two primitive actions, `P` and `V` operations are defined for accessing semaphores. The `P` and `V` operations have one parameter and are atomic (indivisible) operations. `P` and `V` are short for Dutch words, they are conventionally called **`wait`** and **`signal`** respectively in English [11]. The `P` operation is described in pseudocode below:

```
P(sem)
   if sem  > 0
     sem = sem - 1;
```

the operation ensures that value of semaphore is non-negative. When a process executes `P` operation and the current value of semaphore is 1, it decrements its value and can thereafter enter its critical section. If on the other hand, a process finds the value of semaphore to be 0; it waits until another process performs a `V` operation. The following pseudocode defines the `V` operation.

```
V(sem)
  sem = sem + 1;
```

it increments value of the semaphore, *sem* by 1. Any process that executes `V` operation, increments the value of *sem* by 1;

Semaphore provides a cleaner way of resolving synchronization problems; however semaphores are error-prone and can easily lead to deadlocks. For instance, if a process performs a `P` operation but subsequently fails to perform a `V` operation, other processes may not be able to progress. For this reason, Brinch Hansen suggested *message buffering* as the basic means for inter-process communication. This eventually led to the invention of `monitors` [12, 38].

A monitor encapsulates shared variables and operations on the variables. It also defines an initial operation that is executed when the variables are created. The procedures that operate on shared variables are called *monitor procedures.* Access to shared variables is possible only through the monitor procedures. Monitors have been utilized in operating systems and programming languages. However, monitors are based on locks and lock-based programming can easily lead to deadlocks [53].

Advances in networking technologies and the increasing needs for programming distributed systems led to the development of message passing techniques, including remote procedure call (RPC). RPC is an extension of local procedure call: process A sends a request message to process B by calling a procedure defined in B; B processes the request and sends a reply to process A; A then continues with the rest of its statements. RPC is known as remote method invocation (RMI) in object-oriented languages.

1.1.1 The Object Model

Objected-Oriented programming has for some time remained the main programming paradigm. The basic concept in OO programming is that an object encapsulates data and provides controlled access to the data via its methods; the behavior of an object is determined by its class and can be extended incrementally through inheritance. In practice however, OO programming has become more complex than this model suggests [30]. For instance,

- Levels of visibility has been extended because the original levels of visibility — private, protected and public have been found to be inadequate. For example Java provides at least thirteen varieties of access: public scope, private scope, protected scope, package scope, static method, instance methods, interfaces, classes, packages, local inner classes, member inner classes, anonymous inner classes, static member inner class. All this makes maintenance of software difficult.
- An Object does not have control over the sequence in which its methods are called. As a consequence, it cannot ensure that an initialization method is called before any other methods. This makes objects difficult to design. The designer must either allow for all possible calling sequences, or must trust clients to call methods in the intended sequence. The compiler cannot enforce correct call sequencing.
- Refactoring OO programs is difficult. Consider a situation where fat clients are to be replaced by thin clients; moving code from one memory space to another is complex with OO approach and may be infeasible in a commercial environment.

- Modeling some applications using OO design can be unnecessarily difficult. For instance, consider a hospital application where the programmer designed classes Consultant and Patient as derived classes of class Person. This indicates that a consultant is a person and, a patient is also a person but it is possible for a consultant to be a patient at the same time [30]. This highlights occasional complexity that may arise in OO Design.

1.1.2 The Process Model

Languages that provide concurrency by extending other sequential languages with multithreading facilities have serious limitations. Safe concurrency is impossible without a sound and complete type system and many sequential languages, including C, lack such facilities. Stroustrup notes that lack of type safety is the main cause of many problems associated with correctness and performance; exact type information can improve performance [51]. Many languages provide concurrency through libraries. Problems with this approach have been recognized: Boehm shows that providing concurrency by implementing threads as libraries could produce unreliable software [33]. Libraries cannot enforce syntax and semantics checking as a programming language would normally do. A new language that is based on established concepts is more appropriate for providing concurrency.

Erasmus is a process-based concurrent language. We highlight some potential benefits of this model.

- Processes are more general. It is easier to map processes onto multiple address spaces than the object-oriented model.

- It is often easier to model applications with processes than objects. Process-based code can be refactored into object-based code but the converse is difficult, if not impossible.
- Processes are loosely coupled and have more autonomy as compared to objects.

1.2 Thesis Outline

This section gives an outline of the thesis. First, it describes the Erasmus project and then describes the goal of this thesis.

1.2.1 The Erasmus Project

The Erasmus project seeks to develop an industrial-strength, process-oriented language that will contribute to the development of concurrent software that can be maintained, refactored, and deployed on different physical systems. Toward this end, a prototype compiler has been developed. The compiler simulates concurrency by time-sharing a single processor.

1.2.2 The Objectives of the Thesis

The goal of the research described in this thesis was to design and develop a modified version of the prototype compiler described above. Code generated by the new compiler uses true processes that communicate with ports and channels. Several processes may run on the same processor or on different processors. The mapping from processes to processors is specified in a metafile, not in the source code.

The thesis is structured into the following chapters: in chapter two we review some related work; chapter three introduces the Erasmus programming language; chapter four addresses communication in Erasmus; chapter five compares communication in Erasmus and other languages and, finally, chapter six concludes the thesis.

Chapter 2

Related Work

Early attempts at finding solutions to problems in operating systems and computer architectures led to an explosion of research in concurrent programming techniques. Toward this end, a number of concurrent languages have been proposed. By 1989, nearly a hundred concurrent and distributed programming languages had been proposed [7]. However, despite this phenomenal growth in concurrency research activities, proposed languages have yet to enjoy widespread acceptance by mainstream software development. Concurrent applications are still written with sequential languages such as C/C++ and Java. Concurrency is therefore far from a solved problem. Many languages were proposed for particular architectures, e.g., Occam (for programming Transputer), and many allow several styles of programming by including features and constructs that favor those styles.

This chapter reviews some concurrent programming languages. First, it distinguishes between languages designed mainly to provide concurrency and those that support multiple programming styles. Next, it gives a brief introduction to each one of the languages. In

particular, it considers languages, including: Concurrent Pascal, Communicating Sequential Processes (CSP), Ada, Occam, ABCL/1, Joyce, Erlang, Cilk, Java, Mozart/Oz, SALSA, and Occam -π. The chapter considers how the languages represent parallel units; how parallel units communicate; use of channels and similar capabilities; recursion and support for distributed systems.

2.1 Language Style

Many languages were designed mainly to address problems related to concurrent executions of multiple program units. Among the languages reviewed here, the following languages fall into this category: Concurrent Pascal, CSP, Occam, Erlang, Joyce, SALSA and Occam-π. Others support more than one style of programming, for instance, Java supports concurrent programming as well as object-oriented programming. Mozart/Oz is a multi-paradigm language supporting object-oriented programming, functional programming, logic and constraint programming. Ada supports concurrent programming and also provides packages for structuring large programs. Object-oriented programming is also supported by Ada 95. ABCL/1 provides concurrency and object-oriented programming.

2.2 Concurrent Pascal

Concurrent Pascal, designed by Per Brinch Hansen in 1975, was designed for structured programming operating of systems [13]. It was the first programming language to use monitors. It is an extension of the programming language, Pascal, which incorporates monitors,

classes and processes. Processes in Concurrent Pascal are independent entities with private data structures and sequential code that operates on those data. It is not possible for one process to access the private data of another process. Processes communicate through shared data encapsulated within monitors. Access to shared variables within a monitor is possible only through the monitor procedures. Monitor procedures execute one at a time, i.e. if two or more processes attempt to execute a monitor procedure, only one of them will be allowed to execute the procedure. The rest of the processes suspend (queued) until the running process completes the execution of the procedure and notifies other processes.

Listing 1 defines a `process` type, *jobprocess* with two parameters of *diskbuffer* type. The process type defines a private variable *block*, whose type is *page*. It also defines a sequence of statements — represented by the ellipsis in Listing 1 — that are run one after the other until `end` is encountered. Listing 2 shows *Diskbuffer* type which is a `monitor` with monitor procedures *send* and *receive*. Monitor procedures are distinguished from the other procedures by the keyword `entry`.

Listing 1: Process type definition

```
type jobprocess =
process (input, output: diskbuffer);
var block: page;
cycle
  readcards(block);
  buffer.send(block);
//...
end
```

Classes in Concurrent Pascal are like monitors in that they define private data and procedures that operate on those data, but they are unlike monitors in that they cannot be

Listing 2: A diskbuffer monitor definition

```
type diskbuffer =
monitor (...);
var disk: virtualdisk; ...

procedure entry send (block: page);
begin
...
end;

procedure entry receive (var block: page);
begin
...
end;

procedure other(...);
begin
...
end;

begin "init statement"
 init disk (...);
 ...
end
```

called simultaneously by other components. Class procedures are accessed by class variables defined by other components. There are a fixed number of processes, monitors and classes. Dynamic process creation is not supported. To prevent deadlock, monitors cannot be called recursively. All this simplifies implementation but limits the usefulness of the language. Many algorithms are easier to express using recursion. A monitor can invoke procedures defined by other monitors. Concurrent Pascal was designed for single-processor computers; therefore it does not include tools for programming distributed systems and applications. Solo [14] operating system was written in Concurrent Pascal.

2.3 Communicating Sequential Processes (CSP)

In 1978, Hoare proposed communicating sequential processes (CSP) as a method of structuring programs [39]. CSP is not a programming language but rather a formal language for structuring, expressing and controlling concurrent programs. This section reviews the main features of CSP.

CSP uses Parallel command to specify concurrent execution of processes. All processes start simultaneously and the parallel command ends when they are all finished. Examples of parallel commands include the following. The command

```
[P::CL || Q::CL]
```

denotes a parallel command for concurrent execution of processes P and Q. An array of processes can be created. The command

```
[X(i:1..n) :: CL]
```

stands for:

```
[X(1):: CL1||X(2)::CL2||X(3)::CL3|| ... ||X(n)::CLn]
```

CL_1, CL_2, CL_3 denote command lists of processes X(1), X(2) ... X(n). A Parallel command terminates successfully when all the processes have terminated successfully. The command

```
[room::ROOM || fork(i,0..4)::FORK||phil(i,0..4)::PHIL]
```

denotes a parallel command with eleven parallel processes namely: room, fork(0), fork(1), fork(2), fork(3), fork(4), phil(0), phil(1), phil(2), phil(3), phil(4). The behavior of room is

specified by the command list ROOM, the behavior of the five processes — fork(i, 0..4) — is specified by FORK. Similarly, PHIL specifies the behavior of phil(i, 0..4).

2.3.1 Communication in CSP

Input and output commands are used for communication between concurrent processes. CSP uses direct naming — a process names another process as the destination for output and the second process names the first process as the source of input. Revised version of CSP [40] uses indirect naming via communication channel with a channel alphabet (a set of messages that can be transferred over the channel). A Channel links two processes. Communication in CSP is synchronous: A process that is ready to execute its input or output statement waits until the other process is ready to execute output or input statement. Processes may not communicate with each other by updating global variables. The following boxes illustrate how processes communicate.

$ProcessA$	$ProcessB$
$B?x$	$A!e$

processes A and B communicate using input and output statements. Process A names B as its source and B names A as its destination. When A executes $B?x$, it *reads* a value from B and assigns it to the variable x. Process B, on the other hand, *writes* an expression e to process A by executing $A!e$. Communication takes place when both A and B execute their respective input and output commands at the same time and the type of variable x matches that of the expression, e. The result is that x now contains the value of the expression.

CSP introduces and controls non-determinism using guarded commands [25]. Input commands may appear in guards. As an example, suppose process A above wants to read from

either process B or process C. This could be expressed with a guarded command. Consider a program that computes the maximum of two values: x and y. This can be expressed in CSP with an alternative command as shown below:

```
[ x >= y -> max := x
 [] y >= x -> max := y
]
```

If $x \geq y$, it assigns x to max; if $y \geq x$ then it assigns y to max. A loop or a repetition can be expressed with a repetitive command. A repetitive command allows an alternative command to be repeated. For instance, suppose A wants to read from three other processes, B, C, D that generate values repeatedly. Actions by process A can be expressed as follows:

```
*[
  B?x -> PROCESSX
  [] y>0; C?y -> PROCESSY
  [] D?x -> PROCESSX
]
```

A repeatedly checks which of the branches (alternatives) of the alternative command is eligible for execution. A branch is eligible for execution if its guard succeeds and communication is feasible. Only one of the branches must be selected. For instance, in the code extract above, if process B is ready to output a value, command list *PROCESSX* is executed. If on the other hand, B is not ready, the next branch is tried. This is repeated indefinitely. If all the branches are eligible, then one of them is selected non-deterministically.

As mentioned ealier, CSP is a formal language; an implementation language might include support for distributed systems. Like Concurrent Pascal, CSP does not support recursion.

2.4 Ada

Ada was a product of an international design competition organized by the US department of Defense. Concurrency in Ada is based on processes called, *tasks*. A `task` includes a specification and a body. The specification declares the components of a task, including all *entries* (named subprograms that may be called by other tasks), parameters and subprograms while the body defines the actions of a task, including those of entries and subprograms. A task can be defined as in the following examples. The code fragment

```
task type SquareSvr is
    entry Square(Num: Integer);
end;
```

declares a task type named *SquareSvr*. Task types are like templates from which independent tasks can be created. A task is an instance of a task type; an anonymous task type can be declared. Anonymous task types hide explicit declarations of types. For example, the statement

```
task Reader;
```

declares an anonymous task type; the same statement also creates a task, *Reader*. The statement is equivalent to the code

```
task type Reader_Type;

Reader: Reader_Type;
```

which declares the task type *Reader_Type* and a task, *Reader*. The statement

```
task type Agent(ID: Integer);
```

declares a parameterized task type, *Agent*. An integer value (argument) is passed during an instantiation of the parameterized type, *Agent*. The code

```
task type BoundedBuffer is

    entry Put(E:Item);

    entry get(E: out Item);

end;
```

declares a specification for the task type, *BoundedBuffer*, with two entries: *get* and *put*. The following code snippet defines the body of *BoundedBuffer*.

```
task body BoundedBuffer is

-- declarations

begin

...

end;
```

2.4.1 Communication in Ada

Ada uses *rendezvous* for direct communications between tasks. The basic idea behind rendezvous is that a task communicates with another task by calling some **entry** of the

other task via a *call* statement. This is similar to calling a method of an object in object oriented programming. A call statement directly names the task and the entry that is being called. The other task responds to the call by executing an `accept` statement defined for the called entry by the called task. The idea is analogous to the client/server model where a server task declares a set of services that it provides by declaring a set of public entries in its specification. Each entry identifies a service, the parameters it requires to process the request and the result (if any) that will be returned by the task. For instance, the following task declaration models a *Teller* that provides withdrawal, deposit and balance enquiry services to clients:

```
task type Teller is
  entry WithDraw(AcctNo:in Account; Amt:in Money; Result:out Boolean);
  entry Deposit(AcctNo:in Account; Amt:in Money; Result:out Boolean);
  entry GetBalance(AcctNo: in Account; Balance: out Money);
end Teller;
Atm : Teller; -- creates a Teller task
```

where *Account*, *Money* are predeclared types; and *AcctNo* and *Amt* are passed (indicated by `in`) to the teller — *Atm* — for withdrawer and deposit services and *Result* is returned (indicated by *out*) to the client. Similarly, *AcctNo* is passed to the teller for balance enquiry and the balance is returned to the client. The client task (calling task) requests a service from the server task by making an `entry` *call*, identifying both the server and the required `entry` (service). The following statement calls `entry` *GetBalance* of the task, *Atm.*

```
-- client task
Atm.GetBalance(1004550, MyBalance); -- calling Atm.GetBalance
...
```

Atm indicates its willingness to provide the requested service by executing an `accept` statement as shown in the code fragment below:

```
-- server task
accept GetBalance(AcctNo : in Account; Balance : out Money) do
  -- look up balance for the Account number and
  -- assign the value to Balance
end GetBalance;
...
```

Communication occurs when both tasks issue their respective requests at the same time. A task that is ready to communicate (issues its request first) waits (becomes suspended or blocked) for its partner. Clients waiting on an entry are queued. When both are ready, communication takes place — the two tasks are said to rendezvous because they meet at the `entry` point. When they meet, `in` and `in out` parameters are copied from the client to the server task. The server then executes the code inside the `accept` statement and on completion (`end` encountered), any `out` parameters are copied to the client. Both server and the client then proceed independently and concurrently.

Select Statement

Ada allows a task to indicate its willingness to accept calls for any of its public entries using the `select` statement. The general form of `select` statement is:

$$
\begin{aligned}
\textit{select_statement} \quad &= \quad \textit{selective_accept} \\
&\mid \quad \textit{conditional_entry_call} \\
&\mid \quad \textit{timed_entry_call} \\
&\mid \quad \textit{asynchronous_select}.
\end{aligned}
$$

returning to the *Teller* example above, typically a *Teller* offers more than one service. Therefore the server can be coded to accept calls for any of its public entries from clients. This can be programmed elegantly with a `select` statement as shown in Listing 3. One of the branches of a select statement is selected for execution if more than one rendezvous is feasible. The choice of which branch is selected depends on the implementation. It is safe to assume that any of the alternatives can be selected. Selection of alternatives can be restricted with Boolean guards:

`when` *boolean_expression* `=>`

if the guard succeeds, e.g. in Listing 4, if *cashAvailable* is true, the withdrawal service is eligible for selection. If it is false, it is not eligible even if there are clients waiting on this entry.

Select statements can also have `delay` (or `delay until`) and `terminate` alternatives. A delay alternative allows a task to time-out if an entry call is not received within a certain

period of time. Terminate (`terminate`) alternative can be used to exit a select statement if no other alternatives are eligible for selection. The language defines `else` alternative that can be used in a `select` statement. If present, an `else` part defines actions to be taken if no other alternative is executable. Timed and conditional entry calls allow a caller to cancel its call if it is not accepted within a specified period. Timed entry is specified with a `delay` statement while a conditional entry call includes an `else` part in the selective call.

Listing 3: Using a `select` statement

```
task body Teller is
... -- local declarations
begin
 loop
   select
     accept WithDraw(...) do
        --- code for withdrawal service
     end WithDraw;
   or
     accept Deposit(...) do
        --- code for Deposit service
     end Deposit;
   or
     accept GetBalance(...) do
        --- code for balance enquiry service
     end GetBalance;
   -- more code for this branch to be
   -- executed after rendezvous
  end loop;
end Teller;
```

Ada 95 [1] defines some extensions to Ada 83. For example, efficient communication through shared variables is possible. Such shared variables are declared as `protected types`. Protected types are like monitors; they provide a structured way of encapsulating data items. They allow access to these data only through protected subprograms or entries. Much like

Listing 4: `when` guard in `select` statement

```
--- Teller task
begin
...
select
  when cashAvailable =>
    accept WithDraw(...) do
     --- code for withdrawal service
    end WithDraw;
    -- sequence of statements
or
 ...
end Teller;
```

a task, a protected unit can be declared as a type or as a single instance. It consists of a specification and a `body`; for instance, the Listing 5 gives a solution [21] to *bounded buffer* problem. An `entry` of *BoundedBuffer* (Listing 5) can be called as follows:

```
My_Buffer.Put(100);
```

The language guarantees that subprograms of a protected object will be executed in a way that ensures that the shared data encapsulated by the protected object is updated under mutual exclusion. Protected functions have read-only impact on the data while protected procedures and entries are allowed to update shared object but run one at a time. Asynchronous communication mechanism (asynchronous select) was also added. Ada does not allow selective calls of entries.

Listing 5: Bounded Buffer using a protected type

```
BufferSize : constant Integer := 10;
type Index is mod BufferSize;
...
protected type BoundedBuffer is
  entry Get(Item : out DataItem)
  entry Put(Item : in DataItem);
private
  --- private declarations
end BoundedBuffer;

My_Buffer : BoundedBuffer;

protected body BoundedBuffer is

  entry Get(Item: out DataItem)
    when NumberInBuffer /= 0 is
  begin
   ...
  end Get;

  entry Put(Item: in DataItem)
    when NumberInBuffer /= BufferSize is
  begin
    ...
  end Put;

end BoundedBuffer;
```

2.5 Occam

Occam [45] is a concurrent language that was originally designed to program transputers[1]. It is based on Communicating Sequential Processes (CSP). A program in Occam is a collection of processes. Processes communicate over synchronous channels. A channel connects two processes and has a protocol associated with it. A protocol specifies the formats and the

[1]A transputer is a specially designed microprocessor. It comprises a high-speed processor, memory and fast inter-processor communication links for connecting to four other transputers. It was built by INMOS Ltd (UK), now STMicroelectronics.

types of the messages that can be transmitted over a channel. There are no data races among Occam processes. Processes in Occam include:

1. an assignment statement e.g. `x := y + 1` — assigns the value of the expression, $y + 1$ to x
2. input e.g. `keyboard ? char` — reads char from the channel named, *keyboard*
3. output e.g. `screen ! char` — writes the value of char to the channel named, *screen*
4. `SKIP` — do nothing
5. `STOP` — causes a program to abort
6. `SEQ` — sequential execution of the processes indented two columns below `SEQ`. The code

```
SEQ
  keyboard ? char
  screen ! char
```

   executes the input statement followed by the output statement.

7. `PAR` — parallel execution of the processes indented two columns below `PAR`. The code

```
PAR
  keyboard ? char
  screen ! char
```

   executes both the input and the output statements in parallel. Processes can also be assigned priority. For instance:

```
PRI PAR
  P
  Q
  R
```

The process that immediately follows `PRI PAR` has the highest priority, P, in the code above.

8. `ALT` — an alternation executes only one of a number of processes each preceded by a guard; evaluates the guard and executes the corresponding process if the guard is true. If none of the guards is true, it blocks until one of the processes becomes eligible. If more than one is true, one of them is selected arbitrarily. The code

```
ALT
  east ? ch
    ostream ! ch
  west ? ch
    ostream ! ch
```

writes the input, *ch* received from either the east or the west channel (whichever is ready) to another channel, *ostream*. If both are ready, it selects one arbitrarily.

9. `WHILE` — conditional loop evaluates a condition and proceeds to execute the following processes if the condition is true; otherwise it does nothing. The code

```
WHILE char <> eof
  east ? char
```

```
    screen ! char
  west ? char
    screen ! char
```

repeatedly writes the input, *char* received from either the east or the west channel until *eof* character is received from any of the channels.

10. IF — conditional statement. The code

```
IF
  x < y
    max = y
  x >= y
    max = x
```

IF combines a number of processes, each preceded with a guard; if the guard is true, the associated process is executed. For instance, if x is less than y, y is assigned to max; if $x < y$ is false then $x \geq y$ is true therefore, x is assigned to max. If none of the guards is true, the IF statement behaves like a SKIP process.

11. PROC — procedure definition defines a name for a process. The code

```
PROC incr (INT i )
  i : = i + 1
:
```

defines a procedure, *incr* with only one parameter, x. A parameter may be qualified with VAL, in this case a value must be passed for the parameter during the invocation

of the procedure. Like CSP, procedures cannot be recursive in Occam!

12. `FUNCTION` — defines a name for a *value* process. Functions in Occam are free of side-effects and may be used in expressions. They can return more than one value of any data type. The code

```
INT FUNCTION getMaxScore ( VAL [] INT values)
  INT max :
  VALOF
    SEQ
      max = values[0]
      SEQ i = 1 FOR SIZE values - 1
        IF
           max < v[i]
             max = v[i]
     RESULT max
:
```

defines a function named, *getMaxScore*. The function can be used in an expression as in the statement below

```
maxScore := getMaxScore(v)
```

Every channel in Occam has an associated protocol that specifies the properties of the messages that can be transferred over the channel. Channel declaration is of this form: `CHAN OF` *protocol*. For instance, the code

```
CHAN OF BYTE keyboard: -- allow transfer of bytes

CHAN OF INT::[]BYTE keyboard: -- counted array
```

creates a channel named *keyboard* with a protocol (BYTE), indicating that the channel can only be used to pass data in bytes. The types in channel declarations (protocol part) could be any Occam primitive type, arrays (including counted array) or record types. Protocols can be named; it is useful to a name a complex protocol:

```
PROTOCOL COMPLEX IS REAL64; REAL64: -- protocol for complex num

CHAN OF COMPLEX cplxChan: --allow transfer of pair of values
```

this is a sequential protocol; an input on *cplxChan* is:

```
cplxChan ? real.part; imaginary.part
```

A channel can be used to communicate messages with different formats. Such channels use *case protocol*:

```
PROTOCOL FILES
  CASE
    request; BYTE
    filename; DOSFNAME
    word;  INT16
    record;  INT32; INT16::[]BYTE
    error;    INT16; BYTE::[]BYTE
    halt
  :
```

request, *filename*, etc. are tags and each identifies a tagged sequential protocol. An example of a channel that uses this protocol is declared below:

```
CHAN OF FILES to.dfs :
```

a tag is first sent to the receiver to inform it of the pattern of the rest of the communications. For instance,

```
to.dfs ! request; get.record
```

sends the tag *request* followed by a BYTE value. CASE input is provided for reading from a CASE channel. The code

```
CHAN OF FILES from.dfs :
SEQ
  to.dfs ! request; get.record
  from.dfs ? CASE
    record; recNo; recLen::buffer
    ... some code
    error; errno; errlen::buffer
    ... error handler
```

accepts an input with a *record* tag or an *error* tag from from.dfs channel.

Occam processes can be delayed using a Timer. The code

```
TIMER clock :
clock ? now
```

inputs the current time and assigns the value to a time variable.

Occam allows processes to be mapped to specific processors. Such processes use local memory of the processor on which they are being executed. They communicate by passing messages over the connecting communication channels. For instance, with the code

```
PLACED PAR
  PROCESSOR 1
    terminal (term.in, term.out)
  PROCESSOR 2
    editor (term.in, term.out, files.in, files.out)
  PROCESSOR 3
    network (files.in, files.out)
```

terminal process is placed on processor with identifier 1, *editor* is placed on processor 2 and *network* runs on processor 3. Since Occam channel is unidirectional, *terminal* communicates with *editor* via *term.in* (input) and *term.out* (output) channels. Similarly, *network*, communicates with *editor* using *files.in* (input) and *files.out* (output) channels. These three processes execute in parallel, each on its individual processor.

2.5.1 Occam-π

Although Occam-π is a recent language, we mention it here because it is derived from Occam. Occam-π [9, 8] is a recent extension to the classical Occam. It addresses some limitations in Occam. In particular, it extends the language by adding features that support mobile data, channels and processes; extended-rendezvous; dynamic process creation;

array-constructors, process priorities and other features. Mobile processes migrate from one environment to another, making it easier to dynamically reconfigure systems. Mobility of processes can also facilitate load-balancing for large and complex systems. A simple activation of a mobile process is given in the code below:

```
SEQ
  c ? mp -- arrival of a mobile process
  mp ( ... ) -- mobile process runs from some point to another
  d ! mp -- mobile process migrates to another environment.
```

Occam-π allows bundling of unidirectional channels into a single structure. This has some significant benefits. It is easier to program reliable communications over a bidirectional channel [40]. Occam-π provides a *channel direction specifier.* The channel direction specifier ?, denotes the input end of a channel and ! denotes the output end of a channel. An Occam process sees only one end of the channel. Channel direction specifiers make it easier to program client-server interactions where the input end denotes the *client* and the output end denotes the *server.* This also enables a more accurate reporting of errors on channel usage. Channel ends are mobile and can be shared. The following code defines a procedure with two parameters *in?* and *out!* connected to the same channel as a client and a server respectively.

```
PROC integrate(CHAN INT in?, out!)
```

Extended rendezvous allows an inputting process to perform some action on the received data before the corresponding outputting process resumes the execution of its code.

2.6 ABCL/1

ABCL/1 is an object-oriented concurrent language whose primary design principle was to provide clear semantics of message passing [56]. An object in ABCL/1 is an autonomous unit with state represented as contents of its local persistent memory. An object can have one of three states namely: dormant, active or waiting. An object is initially in a dormant or passive state; it becomes active when it receives a message that matches one of its specified patterns and other constraints. The behavior of an ABCL/1 object is encapsulated within a `script` — a set of methods of the object. The `script` of an object specifies all the patterns of the messages that the object accepts and the actions to be taken when it accepts a message.

Messages can be sent to or received by an object in one of two modes: *ordinary* and *express*. This is analogous to assigning low and high priorities to messages. Messages of the same mode are queued and attended to in a First Come First Serve (FCFS) manner. Messages received in *express* mode take priority over those received in *ordinary* mode and can interrupt the execution of actions carried by messages received in *ordinary* mode unless such actions are within *atomic* constructs[2]. Suspended actions can however be resumed, if no messages received in *express* mode are outstanding and a *(non-resume)* command has not been executed. Messages are transmitted asynchronously using three different message passing semantics: *past*, *now* and *future*.

Past type message passing (also refers to as send and no wait):— when an object sends a message to another object, it does *not* wait for the message to be received or the result to be returned by the receiving object. This is denoted below:

[2] Code within an atomic construct is executed as an indivisible operation.

Ordinary mode		Express mode
$[T <= M]$		$[T <<= M]$

where T denotes the receiving object and M denotes the message.

Now type message passing (send and wait):— an object sends a message to another object and waits for a reply to be received from the receiving object. It has the following form:

Ordinary mode		Express mode
$[T <== M]$		$[T <<== M]$

Future type message passing (reply to me later):— an object sends a message to another object and expects to receive later, a reply to its message in a special *private* object, called a *future* object, specified in the message. The sending object does not need the result immediately; it therefore continues its computations and checks the future object when it needs the reply. It has the form shown below:

Ordinary mode		Express mode
$[T <= M\$x]$		$[T <<= M\$x]$

x denotes the future variable, T and M are as defined earlier.

An object is created by sending a *now* or *future* type message with some initial information to a special object that creates the object and returns as a result of the *now/future* type message, the new object created. An object can be created dynamically. This is described below:

```
[ object CreateSomething
  (script
    (=> pattern-for-initial-info ![object ...]))]
```

CreateSomething denotes the object that creates a new object; [object ...] represents the definition of the new object to be created by *CreateSomething.* For example, the program [56] in Listing 6 creates an alarm clock object. The keyword, ! is used to send a reply to a request sent in *now* or *future* type message passing.

Listing 6: Creating objects in ABCL/1

```
[ object CreateAlarmClock
   (script
     (=> [:new Person-to-wake]
      ![object
        ( state [time-to-ring := nil] )
        ( script
          (=> [:tick Time]
           (if (= Time time-to-ring)
             then [Person-to-wake <<= [:time-is-up]]))
          (=>[:wake-me-at T]
```

The code

```
( => request-pattern ... !expression ... )
```

is a short form of the code

```
( => request-pattern @destination ... [destination <= expression] ... )
```

where destination denotes the reply-destination. For messages sent as *past* type message passing, a reply-destination can be specified as in the statement

```
[T <=request@reply-destination]
```

A reply-destination provides capability for delegating computations in ABCL/1: messages can be sent to another object with actions implementing some delegated computations.

2.7 Erlang

Erlang [5] is a declarative language that was developed at Ericsson Lab for programming concurrent, real-time and distributed fault-tolerant applications. It provides light-weight[3] concurrency through processes [4]. The language evolved from Prolog extended with parallel logic processes to a completely new language. Processes in Erlang are self-contained independent entities. They have disjoint variables; therefore, it is not possible for a process to modify the state of another process. They communicate through asynchronous message passing over unidirectional channels[4]. Process creation is explicit:

```
Pid = spawn (Module, FunctionName, ArgumentList)
```

The primitive `spawn` creates a new process that evaluates a given function. It takes three (3) arguments as shown above. `Module` represents the name of the program module where the function denoted by `FunctionName` is defined and, `ArgumentList` stands for a list of arguments to `FunctionName`. The return value of `spawn` is a process identifier — *Pid* — which is used for communications with the new process. After its creation, a process executes in parallel with its parent process. When a process completes the evaluation of its function, it terminates automatically.

2.7.1 Communication in Erlang

Erlang processes communicate only by message passing. A process sends a message to another process using the primitive '`!`' (`send`), for example, the statement

[3] Processes require little memory and little computational effort for creating and deleting processes and messages.

[4] A channel is an infinite stream of messages.

```
Pid ! Msg
```

sends a message, "Msg" to the process identified by *Pid.* Messages can be sent to a process by using its registered name. Process registration associates a name with a process identifier. Message type can be any Erlang type. The statement

```
getPid(100)! square(num)
```

evaluates *getPid* to obtain a process identifier. It also evaluates *square* for the message to be sent to the resulting *Pid.* As mentioned earlier, communication in Erlang is asynchronous: sender does not wait for message to arrive at the intended destination. A message is received by the destination process through the execution of the primitive `receive`. The syntax for `receive` is given in the following code fragment.

```
receive
   Message1 [when Guard1] ->
       Actions1;
   Message2 [when Guards2] ->
       Actions2;
   ...
end
```

$Message_1$, $Message_2$, ... $Message_n$; n > 0, shown above, are patterns. $Guard_1$, $Guard_2$, ... $Guard_m$, m <= n, are optional guards. $Actions_1$, $Actions_2$, ... $Actions_n$ represent sequences of actions to be executed for a selected branch. The value of a `receive` expression is the value of the last expression evaluated in the sequence of actions for the selected branch.

Every process has a *mailbox* — a message queue — attached to it. Messages destined for a process are queued in the process's mailbox. When the process executes a `receive` expression, the first message that matches any of the patterns in the alternatives is removed and deleted from the mailbox provided that the corresponding guard (if any) succeeds. A process executing a `receive` waits until a message is matched. Any unmatched messages remain in the mailbox until they are matched by subsequent executions of `receive`.

It is possible to receive messages from a particular process. In this case, the sender must include its own *Pid* or the *Pid* of another known process in the message. For example, the statement

```
Pid ! { self(), 123 }
```

sends the message "123" to the process identified by *PID*. The built-in function `self()` returns the process identifier of the running process. The following code extract shows how messages can be received from a process identified by *Pid*.

```
receive
  {Pid, Message} -> ...
end
```

As shown below, `receive` can also have timeout that expires if no message has been received within a particular period. The corresponding action is then evaluated. The code

```
receive
   Message1 [when Guard1] ->
       Actions1;
   Message2 [when Guards2] ->
       Actions2;
   ...
   after
       TimeOutExpression ->
         ActionsTimeout
end
```

shows how a receive statement might include a timeout as an alternative.

Erlang's `receive` primitive is based on first come, first serve model. It works using pattern matching. An attempt is made to match the first message in the mailbox with a message pattern in the `receive` expression, in the order specified by `receive`. If there is a match and the guard (if any) succeeds, the branch is selected. If the message does not match the pattern in the first branch, the next branch is tried. If the message matches none of the patterns in the `receive`, an attempt is made to match the next message from the queue (mailbox).

2.7.2 Support for distributed applications

An isolated Erlang system becomes an Erlang *Node* (part of a distributed system) by executing a built-in function, `alive`. The function (`alive`) has two parameters: the first is the name to be published as the symbolic name for the new node, the second parameter names

a port for communicating with other nodes. Messages can be sent to remote processes. Links can also be created between local and remote processes, as if the processes were local. When *Pids* are used to send messages, communication is location transparent: there is no difference in syntax and semantics whether the communication is between local processes or the communication is between a local and a remote process. A message can be sent to a remote process by name as in the following statement.

```
{name, Node ! Msg}
```

where *name* is the name of the process, *Node* is the global name for the node where the process is running and *Msg* is the message to be received by the process. Connections to other nodes are set-up by the Erlang run-time system the first time it evaluates expressions involving such remote nodes. A mechanism for an in-service code upgrades is also supported.

2.8 Joyce

Joyce is a concurrent language based on CSP and Pascal that was designed for programming multiprocessor systems [16]. A program written in Joyce, defines autonomous processes (comprising nested procedures), called *agents*. They communicate over synchronous channels and can dynamically create subagents that run concurrently with their creators. A channel connects two or more agents and transmits symbols from one agent to another. An agent terminates after all its subagents, if any, have terminated and, all its procedure statements have been executed. Agents are created through the activations of agent procedures[5]. The statement

[5]Each agent procedure defines a class of agents.

```
semaphore( 1, user)
```

denotes an activation of the agent procedure, *semaphore* (defined in Listing 7).

Every channel has an alphabet that defines the fixed types of messages that can be transferred from one agent to another. Channel alphabet is a set of symbols transmissible over the channel. In contrast with CSP and Occam, Joyce channels are bidirectional: they allow transfers of symbols in both directions. Communication takes place when one agent is ready to output a symbol and another agent is ready to input the same symbol from the channel. When this happens, a message from one agent is copied to a variable of the other agent. A channel can also connect two or more agents; if more than two agents are ready to communicate over a channel, a pair of matching agents will be arbitrarily selected by the channel. An agent can use a `poll` statement to find a matching agent for an interaction. Both sending and receiving agents can be polled. Channels are created dynamically and accessed through agents' local port variables.

$T = [s_1(T_1),\ s_2(T_2),\ ...,s_n(T_n)];$

creates a port type T. Port values are pointers to channels; $\mathrm{s}_i(T_i)$ denotes symbol class and consists of all values of type (T_i) prefixed with the same name (s_i). The alphabet is the union of a finite number of distinct symbol classes, s_1, s_2, ..., s_n. A symbol without a type is a signal, for example. The statement

```
screen = [int(integer), eos]
```

defines a port, *screen*, which can be used to transfer integer type messages or to send a signal, *eos*. Channels are created using port statements as in the following statements.

```
type stream = [int(integer), eos]; {1}

var +a: stream;                    {2}
```

Statement 1 above defines a port named *stream* while statement 2 creates a channel and assigns the channel pointer to a port variable a. The alphabet of the channel is given by the *port* type, *stream* defined in statement 1. Messages are transferred over the channel using input and output commands. Consider two communicating agents, one outputs a message over a channel using its local port variable a and the other agent inputs the same symbol from the same channel through its local port variable b. Let port type:

$T = [s_1(T_1),\ s_2(T_2),\ ...,s_n(T_n)];$

the interaction between the two agents is illustrated below:

B outputs the symbol $s_i(e_i)$ through the channel pointed to by the port variable, b. The expression e_i, must be of message type, T_i. Similarly, A inputs a symbol $s_i(v_i)$ through its local port, a. Port variables a and b must be of the same type. The combined effect of the two statements is that v_i contains the value of the expression, e_i. After the communication, A and B continue at their next statements concurrently. An array of channels can be activated. When more than two agents are connected to a channel, two matching agents are selected by the channel. The choice of which pair of agents is selected is non-deterministic. A `poll` statement has the general form:

```
poll
  C1 & B1 -> SL1 |
  C2 & B2 -> SL2 |
  ...
  Cn & Bn -> SLn
end
```

when an agent examines the communication command C_i and guard B_i, it selects one that is feasible and whose guard B_i is true. It then executes the list of statements, SL_i.

Listing 7: Agent type semaphore

```
PV = [P, V] { P and V signals }

agent semaphore (x: integer; user:PV)
begin
  while true do
   poll
     user?P & x > 0 -> x : x -1 |
     user?V -> x := x + 1
   end;
end;
```

Listing 7 declares a port type, *PV* and an agent procedure, *semaphore.* Different agents can be created through different activations of *semaphore.*

2.9 Cilk

Cilk is a concurrent language for writing multithreaded applications. It is based on ANSI C and was developed by researchers at MIT [44]. A Cilk program is viewed as a directed

acyclic graph where parent procedures depend on the child procedures spawned by them. To execute a Cilk program correctly, these dependencies must be observed. The language provides three keywords `cilk`, `spawn` and `sync` for identifying cilk subprograms, creating and synchronizing cilk threads respectively. The program [44] shown in Listing 8 (written in both C and Cilk) computes the nth Fibonacci number.

Listing 8: C and Cilk programs for computing nth Fibonacci number

```
C program                          Cilk program

int fibonacci ( int n)             cilk int fibonacci ( int n)
{                                  {
  if (n < 2) return n;               if (n < 2) return n;
  else                               else
  {                                  {
    int x, y;                          int x, y;

    x = fibonacci(n-1);                x = spawn fibonacci(n-1);
    y = fibonacci(n-2);                y = spawn fibonacci(n-2);

                                       sync;
    return x + y;                      return x + y;
  }                                  }
}                                  }
```

As shown in Listing 8, `spawn` creates a cilk procedure which runs in parallel with its creator. The function *fibonacci* is identified as a cilk procedure by the keyword `cilk`. It spawns two threads that compute the fibonacci numbers for the preceding terms. Each thread spawns two threads depending on the value of n. This forms a directed acyclic graph. The keyword `sync` ensures synchronization between a parent and a child threads. All threads spawned by a thread must terminate before the thread can terminate. After all the child threads have returned, the procedure continues at its next statement after `sync`. Cilk thread may

communicate using shared global variables. This occurs when parallel procedures access the same variables directly or indirectly through pointers.

2.10 Java Programming Language

Java programming language provides concurrency through threads. Threads execute independent code that operates on private and shared global variables (located in shared memory). They communicate via shared variables and access to these variables is coordinated by synchronizations implemented using monitors[6]. A Java thread is an instance of the built-in class `Thread`. A thread begins execution of its code after its *start* method has been invoked. Every object in Java has an associated lock. A thread can use **`synchronized`** statement to lock an object. First, it computes a reference to the object and then attempts to acquire the lock associated with the object. If it succeeds, it executes the code within the statement; otherwise, it suspends on the monitor associated with the object. An unlock action is performed by the thread after the code within the synchronized statement has been executed successfully or aborted successfully. A synchronized method automatically locks the object through which it has been invoked. It executes the body of the method and unlocks the object after the execution of the last statement. If the method is a **`static`** method, it locks the monitor associated with **`Class`** object that represents the class of the object whose static method has been invoked [27]. A thread can also lock an object multiple times; subsequent unlock actions undo the effect of the previous lock actions.

[6]The claim that Java uses monitor has been criticized. Java monitor is considered insecure because access to shared variables encapsulated by the monitor can be compromised by an unsynchronized method of a class. In addition, a shared variable of a class has public visibility within the package in which the class is defined, making it accessible by any program within the package. This can have significant impact on the reliability of multithreaded applications written in Java. See [18] for details.

The behaviors of Java threads are non-deterministic. In other words, except by adequate synchronizations, the results of executing threads in parallel cannot be predicted. Different results can be obtained at different runs. Each thread has no knowledge of other threads. While a thread is reading a shared variable, another thread may be updating the same variable. This typically creates time-dependent inconsistencies. This situation can however be controlled by programmers through explicit synchronization of access to shared objects. `Thread` can be extended as shown in Listing 9. Actions of a thread is defined by implementing an `abstract` method, *run* of `Thread`.

Listing 9: Extending Java built-in class Thread

```
class GetThread extends Thread {
  ...
  public void run() {
          ... // thread code here
  }
}
class PutThread extends Thread {
  ...
}
```

Listing 9 defines two classes, *GetThread* and *PutThread* by extending `Thread`. An interaction by instances of these classes via a shared variable is illustrated in Listing 10. *BoundedBuffer* gives a solution to *bounded buffer* problem. It defines two `synchronized` methods: *put* and *get*. An object of *BoundedBuffer* is conceptually a monitor. *TestBuffer* (Listing 10) creates an object of *BoundedBuffer* which is shared by two threads: p and g. Thread p removes an item from the buffer while p stores a new value into the buffer. Both threads execute concurrently after their *start* methods have been invoked. A thread that executes `wait()`, suspends until another thread executes `notify()` or `notifyAll()`. When

a thread executes `notify()`, one of the threads waiting on the monitor queue is removed from the queue and scheduled. An execution of `notifyAll()` awakens all threads waiting on the monitor queue and one of them becomes runnable.

Listing 10: Bounded buffer in Java; a driver program is shown in Listing 11

```
public class BoundedBuffer {
  private int} full, top, base, buf[];
  private final int MAX = 10;

  public BoundedBuffer(){
    buf = new int [MAX];
    full = top = base = 0;
  }
  public synchronized  void put(int val) throws Exception {
    while (full == MAX ) wait();
    buf[top++] = val;
    top %= MAX; // cycle around MAX
    ++full;
    notifyAll();
  }
  public synchronized  int get() throws Exception {
    int item;
    while (full == 0) wait();
    item = buf[base++];
    base %= MAX;
    --full;
    notifyAll();
    return item;
  }
}
```

Java provides supports for programming distributed applications through Remote Method Invocation (RMI). Remote method invocation is an extension of local method invocation: an object running in a process invokes a method of another object (via the object's reference) in another process. In a client-server architecture, servers typically consist of *remote* objects. Clients send requests by invoking methods of the remote objects running in the

Listing 11: Driver program for Bounded buffer(Listing 10)

```
public class TestBuffer {
   public static void main}(String args[]) {
      BoundedBuffer b = new BoundedBuffer();
      PutThread p = new PutThread(1,b);
      GetThread g = new GetThread(b);
      p.start(); // invokes function run of PutThread
      g.start(); // invokes function run of GetThread
      ...
   }
}
```

server process after obtaining *references* (*remote references*) to them. Remote references are obtained through the naming service. Methods that are intended to be called by objects running in other processes are specified in an *interface* (*remote interface*) which is implemented by the remote object. Listing 12 shows an interface that specifies all the methods that are intended to be called by objects residing in remote processes. The interface extends a built-in `interface` — `Remote`. *BrokerImpl* (Listing 12) extends a java built-in class *UnicastRemoteObject*. It also implements *BrokerInf* `interface` and defines bodies of all the methods specified by the `interface`.

Java also supports mobile code in the form of applets that are downloaded and run on web browsers. An applet runs in a constrained environment for security purpose. Java supports network class loading; this facilitates movements of code from servers to clients as in the case of applets.

Listing 12: Remote interface definition and implementation

```
public interface BrokerInf extends Remote{
  int getStock() throws RemoteException;
  boolean sellStock(Account acc) throws RemoteException;
  ...
}

public class BrokerImpl extends UnicastRemoteObject
                                implements BrokerInf{
  int synchronized getStock() throws RemoteException {
    ...
  }
  boolean synchronized sellStock(Account acc)
                    throws RemoteException{
    ...
  }
  ...
}
```

2.11 Mozart/Oz

Mozart/Oz is a multi-paradigm concurrent language that combines features of object-oriented programming, functional programming, logic and constraint programming. Concurrency is based on sequential dataflow threads[7] that communicates through shared references in the shared store[8] Oz variables are single assignment variables (logic variables). A logic variable is a single assignment variable[9] that can be equated with another variable. The statement

```
thread S end
```

[7]A dataflow thread is a thread that executes its next statement only if all the values that are needed by the statement are available. If the required values are not available, the thread blocks until they become available.

[8]A shared store is not a physical memory but an abstract store for entities; it allows *only* legal operations defined for entities stored.

[9]Once a logic variable is assigned a value, the value cannot be changed but can be read.

spawns a new thread that executes sequential statements, S in parallel with the current thread. Threads are scheduled in a round-robin fashion using time-slice. Much like Java threads, they are given a priority and it is not possible for a high priority thread to starve a low priority thread. Listing 13 shows a concurrent map function in Mozart. The function

Listing 13: A current map function

```
fun {Map Xs F}
   case Xs
   of nil then nil
   [] X|Xr then thread {F X} end | {Map Xr F}
   end
end
```

transforms the list Xs by applying function, F to all the elements of the list. It creates a thread to evaluate each F X for every X in the list Xs. Consider the following code fragement

```
declare

F X

{Browse thread {Map X F} end }
```

the thread blocks until X and F are bound. Given the code fragement below

```
X = 1|2|3|nil

fun {F X} X * X * X
```

X and F are bound to their respective definitions, therefore the thread can resume its computations; it creates 3 threads: one for each element of list X.

Oz provides *future* variables for programming demand-driven computations; futures are read-only capability created for logic variables. A future can be created for variable x as follows:

```
Y = !! x
```

any thread that attempts to use Y will block until Y becomes bound to a value. Procedure `ByNeed{+P ?F}` can be used to execute a procedure P when a thread attempts to access the value of F. P takes one argument; the expression |{PX}— denotes an invocation of the procedure P with argument X.

Oz provides asynchronous communication by message passing over channel represented by "stateful" type, `Port`. A port is an ADT which means it provides interfaces for all the possible operations. A communication channel can be shared by several senders. Every port has an associated stream. For instance, the statement

```
{Port.new S ? P }
```

creates a Port P and connects it to S (the stream). The statement

```
{Port.send P ? M }
```

sends the message M to the port P by appending it to the end of the stream associated with port P, and the statement

```
{Port.Isport P ? B}
```

checks whether P is a port.

Oz threads use locks to coordinates access to shared resources by threads. A thread gains access to a critical region by acquiring a lock and releases the acquired lock on leaving the critical region. The statement

```
{NewLock L}
```

creates a new Lock L, and the expression

```
{IsLock E}
```

returns true if and only if E is a lock. Guarded critical regions have the following syntax:

```
Lock E then S end
```

where E denotes an expression that evaluates to a lock or an error if E is not a lock. This statement blocks until S is executed.

Mozart system adds network transparency to Oz by separating the functionality of a program from its distribution [47]. Distribution is specified separately as a software component. Code modification is not necessary if a program is moved from one environment to another. Mozart also allows multiple Oz sites to be connected together to form a single logical Oz computation.

2.12 SALSA

SALSA (Simple Actor Language System and Architecture) is an Actor-Oriented concurrent programming language designed specifically to facilitate programming of dynamically reconfigurable open distributed applications [55]. It is based on Java. In this section, a

brief background on the actor model is given before a description of SALSA programming language.

2.12.1 Actor Model

The actor model of concurrent computation was proposed by Carl Hewitt and other researchers at MIT [37, 35, 36]. It was further developed by Irene Greif [28], William Clinger [22] and Gul A. Agha [2]. Hewitt and Bishop [36] describe certain "laws" that must be satisfied by computations involving communicating parallel processes. In actor model, an *actor* represents the basic construct of computation. Actors interact by passing messages. An actor sends a message to another actor. If a reply is to be sent by the actor that receives the message, the sender includes another actor, called `continuation` in the message. Any reply from the receiving actor is sent to the continuation. Acquintances of an actor x is the list of all actors that x directly "knows about". The list of acquintances of an actor at any time is finite. In actor parlance, everything is an actor. For instance, the statement

```
[[+ <~~ [request: [5 9], reply-to: c] ]]
```

sends a message containing the tuple `[5 9]` and a continuation, c to an actor, `+`. When the receiving actor (+) receives the message, it computes the addition of 5 and 9 and sends the result to c (the continuation) as given by the statement below:

```
[[ c <~~ [reply: 14] ]]
```

Messages received by an actor are ordered according to their arrival times (an actor's local times when messages are received). When an actor receives a message, it can concurrently:

1. create a finite set of new actors;
2. send a finite set of messages to other actors, including itself
3. designate a new behavior that will govern the response to the next message it receives.

Actors communicate through asynchronous message passing. Each actor has a mail address that identifies its mail queue and therefore messages can be sent to an actor only if its mail address is known. All messages to an actor are received in its mail queue.

2.12.2 Actors in SALSA

Actors in SALSA interact by passing asynchronous messages. There is no shared memory in SALSA. Any actor that wants to communicate with another actor must first obtain the other actor's reference. Once the reference has been obtained, the actor can send message to its communicating partner. An actor is defined by encapsulating its behavior, state and message handlers:

```
module examples;
behavior HelloWorld {
  void act ( String [] args ){
     standardOuput <- println("Hello World");
  }
}
```

defines an actor named HelloWorld whose message handler, *act* sends a message (println("Hello World")) to the standard actor — *standardOutput* that displays the message to the standard output device. SALSA uses <— for sending messages as shown in the following statement:

```
actorReference <- message
```

A message "message" is sent to the actor identified by *actorReference.* An actor can be created by another actor using the keyword `new`, for example, the statement

```
HelloWorld actRef = new HelloWorld()
```

returns a reference to actor, HelloWorld; other actors can communicate with the new actor using the reference i.e. *actRef.* Reference to an actor can also be obtained by calling the built-in function, *getRefenceByName()*, or from messages sent by other actors.

SALSA provides three abstractions for coordinating concurrent activities:

1. Token passing continuations:— is used to specify the ordering of messages. It uses the reserved keyword "@" to join messages. For instance:

   ```
   standardOutput <- print("Hello ") @
   standardOutput <- print("World")
   ```

   prints Hello World whereas:

   ```
   standardOutput <- print("Hello ");
   standardOutput <- print("World");
   ```

   could print either "*Hello World*" or "*World Hello*". Message handler can return a value which can be accessed through the primitive `token`. Token passing continuations can be compared to monadic programming in Haskell [42] where result from previous computations can be passed as input to the next.

2. Join continuations:— allows parallel processing barrier to be specified. Message passing statements with a join block are executed in non-deterministic fashion, but every message within a join block must be executed. The result is joined to the result of the statement following the @ sign, For instance, the code

```
join {
  standardOutput <- print("Hello ")
  standardOutput <- print("World");
} @ standardOutput <- println(" SALSA");
```

prints either “Hello World SALSA” or “WorldHello SALSA”. An array of values returned by statements within a join are collected into **token**.

3. First-class continuations :— allow an actor to delegate a computation to third party independently of the current continuation for a given message's token. Delegation of computation is specified by the keyword **currentContinuation**. To illustrate the use of first-class Continuations, consider the following program [55] for computing the first n fibonacci numbers:

```
module fibonacci;

behavior Fibonacci {
   int n;
   Fibonacci ( int n) {
       this.n = n;
   }
```

```
    int compute(){
       if (n < 2){
           return n;
       }else {
               Fibonacci fib1 = new Fibonacci(n-1);
               Fibonacci fib2 = new Fibonacci(n-2);
               join { fib <-compute(), fib2<-compute() )
                     @ add @ currentContinuation;
       }
    }

    int add ( int numbers[]){
       return numbers[0] + numbers[1];
    }

    void act (String args[]) {
        n = Integer.parseInt(args[0]);
        compute()@ standardOutput <- println;
    }
}
```

It is possible for an actor to migrate to another location. First, a universal actor must be bound to a unique universal name and a universal locator by the naming service. Universal Naming is central to actor migration and remote communication.

Table 1: Summary of concurrent languages

Language	Parallel Unit	Communication Mechanism	Use of Channel	Support for Distributed Sys.	Recursion
Concurrent Pascal	Process	Monitor	-	-	-
CSP	Process	Sync MP	+	-	-
Ada	Task	Rendez/Async MP Protected types	-	+	+
Occam	Process	Sync MP	+	+	-
Occam-π	Process	Sync MP	+	+	+
ABCL/1	Object	Sync(now) MP Async MP	-	+	+
Erlang	Process	Async MP	+	+	+
Joyce	Process	Sync MP	+	+	+
Cilk	Cilk Procedure	Shared Variables and Pointers	-	-	+
Java	Object	Shared Variables Sync MP (RMI)	-	+	+
Mozart/Oz	Thread	Shared References Async MP	-	+	+
SALSA	Actor	Async MP[10]	-	+	+
Erasmus	Cell	Shared Variables[11] Sync MP	+	+	+

2.13 Discussion

Concurrent languages are often designed for certain problem domains. This needs to be considered when comparing programming languages. Many languages are strong for programming certain applications, yet unsuitable for some other problems: Java programming language is excellent for programming internet applications and some embedded systems. In fact, the growth of internet applications has largely been driven by Java technologies. Yet, Java may not be suitable for programming certain defense applications where extremely

high reliability and robustness are required. Ada was designed for this purpose.

We have seen from the foregoing sections that different languages have different ways of representing parallel units. Most use process or a variant of this under a different name. In Object-oriented languages object represents the unit of parallelism: e.g. Java threads are instances of *Thread* class. Tasks in Ada are conceptually, processes, so are Joyce agents. Independent processes that communicate by passing messages are more suitable for building highly reliable systems. It is no wonder that an Erasmus cell is a collection of independent closures (processes) that communicate via individual ports connected to channels.

Erasmus shares with Erlang the philosophy that module system allows structuring of large concurrent systems. Good structuring language constructs improve manageability of large programs. By organizing processes into cells, large programs can easily be constructed. By modifying program code, Erlang applications developed for a single processor machine can be made to run on networks of processors. This is inherently simple to achieve in Erasmus.

An Erasmus cell represents program structuring tool and can be extended incrementally. Reconfiguration of program is less difficult because cells can be moved from one location to another. The clear separation of program logic from its deployment facilitates reconfiguration of programs. Erasmus code compiled for a single processor runs on a network of processors without modifications to the underlying program source code. This capability is one of the strengths of the language. Erasmus uses synchronous message passing mechanism. There is no buffer; buffer overflow problem does not exist. We discuss communication and network transparency in Erasmus in chapter four and compare the communication aspect of Erasmus with the languages discussed here in chapter five.

---

[10]SALSA is based on Java, hence Java facilities are also available.
[11]Shared variables may be used by processes within a cell only.

Chapter 3

Overview of Erasmus Programming Language

The main goal of the Erasmus programming language [32] is to facilitate the development of large-scale, maintainable software. Development and maintenance of large and complex software can be a very daunting task. Erasmus overcomes many of the challenges that are associated with large-scale software development. Some of the specific objectives are highlighted below.

3.1 Objectives

- *Scale-free programming*

  Erasmus takes the view that software construction should be *fractal*: the same notation should be employed at all levels of scale. Erasmus facilitates scale-free software

development by allowing cells and processes to be recursively nested. This is in contrast to languages with a hierarchy of abstractions such as method-class-package, each with slightly different syntax.

- *Type safety*

  Modern languages should be type-safe. This is a necessary condition for building reliable systems. Erasmus is a strongly typed language (all type errors are detected) and type checking is *static*: all checking is performed at the compilation time.

- *Encapsulation*

  Erasmus provides encapsulation in various ways: program code is organized into isolated processes that communicate through message passing. A process cannot modify private variables of another process. Processes are organized into independent cells that communicate *only* by exchanging messages. A cell can have variables that are shared by processes within the cell. Since there is only one thread of control in a cell, race conditions do not arise.

- *Capabilities*

  Capabilities or entities in Erasmus are program components. Cells are given capabilities or entities that are needed to carry out their tasks and, nothing more.

- *Large-scale Refactoring*

  Refactoring is the process of changing the internal structure of a software system without changing its external behavior [26]. Refactoring is often desirable but sometimes infeasible especially in commercial setting where downtime can be extremely costly. Erasmus is designed to facilitate refactoring of large programs. Software components

can be moved easily from one environment to another. For example, a "fat client" can be converted to a "thin client" by moving components from the client to the server.

- *High-level abstractions*

  Erasmus provides high-level constructs and abstractions that simplify programming for most programmers.

3.2 Language Description

Concurrency in Erasmus is based on communicating processes. A *closure* is an autonomous process with its own state and instructions. Closures may have parameters and communicate using certain interfaces (ports) over synchronous channels satisfying some well-defined protocols. A port serves as an interface for a closure to communicate with another closure. Closures *within* a cell may also communicate via shared variables.

The basic building block of an Erasmus program are *cells*, *closures* and *protocols*. A cell is a collection of one or more closures. Protocols define constraints on messages that can be transferred with any port that is associated with the protocol. Cells and processes can be created dynamically. Erasmus supports modular programming. For instance, cells may also be nested. A program is a sequence of definitions followed by the instantiation of a cell.

Program = { *ProtocolDefinition* | *ClosureDefinition* | *CellDefinition* }$_{;}$

Instantiation.

A definition introduces or elaborates a protocol, a closure, or a cell type:

ProtocolDefinition = ProtocolName '=' *Protocol* .

ClosureDefinition = ClosureName '=' *Closure* .

CellDefinition = CellName ('=' | '+=') *Cell* .

where

Protocol = ProtocolName

| '[' *ProtocolExpression* ']' .

Closure = ClosureName

| '{' [{ *Declaration* }$_{;}$] '|' *Sequence* '}' .

Cell = CellName

| '(' [{ *Declaration* }$_{;}$ '|'] { *Declaration* | *Instantiation* }$_{;}$ ')' .

The complete syntax of Erasmus is given in appendix .1.

A complete Erasmus program consists of at least a process definition, a cell definition and a cell instantiation. The code

```
myProc = { | sys.out := "Hello, world!"};
```

names and defines a process with no parameters. A process's sequence of statements is specified after a vertical bar |. The Process *myProc* has only one statement that writes the string, '`Hello, World`' to the standard output device using the standard port, `sys.out`. Process parameters may be specified before the bar.

The code

```
myCell = ( myProc() );
```

names and defines a cell comprising of one process: `myProc`.

The code

```
myCell();
```

instantiates a cell; in this example, an instance of `myCell` is activated.

Process and cell definitions such as `myProc` and `myCell` can be viewed as defining types since their definitions have no effects. Only an instantiation of `myCell` causes any actions by the program.

All communication between cells and processes is performed by ports and shared variables. For simplicity, Erasmus provides standard ports: `sys.in` and `sys.out` that can be used anywhere in an Erasmus program. More detail about ports is given in Section 3.2.3.

3.2.1 Types

The possible relationships between two types T_1 and T_2 are:

$$
\begin{array}{rcl}
T_1 = T_2 & : & T_1 \text{ and } T_2 \text{ are } \textit{equal types} \\
T_1 <: T_2 & : & T_1 \text{ is a } \textit{subtype} \text{ of } T_2 \\
\text{(no symbol)} & : & T_1 \text{ and } T_2 \text{ are unrelated}
\end{array}
$$

The language provides a set of basic types including: `Char`, `Bool`, `Integer`, `Decimal`, `Float`, `Text`. Port types are described under Section 3.2.3. A detailed description of the Erasmus type system can be found in [41].

3.2.2 Statements

Erasmus provides a set of statements for declarations and actions. A *sequence* is a series of statements that are executed consecutively. Erasmus statements have the general form:

Statement = `skip`
| `exit`
| `until` *Expression*
| `while` *Expression*
| *Assertion*
| *Declaration*
| *Instantiation*
| *Assignment*
| *Conditional*
| *Loop*
| *Select* .

some of these statement types are described below:

- `skip statement`:— `skip` causes no action.

- `exit statement`:— `exit` transfers control to the statement that immediately follows the `loop` or `loopselect` that execute the statement. The `exit` statement is only allowed within a `loop` or `loopselect` statement.

- `until` and `while` statements are discussed under `loop`.

- Assertion:— an assertion consists of the keyword `assert` and two arguments. The statement

  ```
  assert(b, msg);
  ```

  The first argument *b* is an expression that must evaluate to `Bool`. The second argument *msg* is displayed if the value of the first argument is `false`. Otherwise, the assertion has no effect. For example, the code

  ```
  n:Integer := 100;
  assert(n > 0, "n must be a non-negative number"); -- has no effect
  ```

  the assert statement has no effect since `n > 0` is true. Whereas, the code

  ```
  n:Integer := -1;
  assert(n > 0, "n must be a non-negative number");
  ```

  prints the message '`n must be a non-negative number`' since `n > 0` is false.

- *CellInstantiation*:— Cell instantiation activates a cell and its processes. For instance, the statement

  ```
  cell();
  ```

activates a cell and causes all its processes to execute in parallel.

- *Conditional* statement:— defines alternatives with `if` statement. Only one of the alternatives is executed. The general form is:

$$\begin{aligned} \textit{Conditional} \quad &= \quad \texttt{if}\ \textit{Rvalue}\ \texttt{then}\ \textit{Sequence} \\ &\quad \{\ \texttt{elif}\ \textit{Rvalue}\ \texttt{then}\ \textit{Sequence}\ \} \\ &\quad [\ \texttt{else}\ \textit{Sequence}\ ]\ \texttt{end}\ . \end{aligned}$$

e.g.:

```
if grade = 'A' then
   sys.out := "Excellent"
elif grade = 'B'  then
   sys.out := "Good"
else
   sys.out :=  "Poor"
end.
```

- *Loop* statement:— `loop` construct is provided for repeated execution of some sequence. The sequence is executed repeatedly until an `exit` statement is executed. The `loop` statement can be used with `until` and `while` statements. In such cases:

`until` $C \equiv$ `if` C `then exit end`

`while` $C \equiv$ `if not` C `then exit end`

Listing 14 shows three equivalent processes that print integers from 0 to 10. *Counter1*

uses `loop` statement while *counter2* and *counter3* use `while` and `until` respectively. The cell, *myCell* defines a cell comprising of three processes, *Counter1*, *Counter2* and *Counter3*. The three processes run concurrently when *myCell* is instantiated as shown in the listing.

- select statement:— The select statement is discussed under communication (Section 3.2.3).

Erasmus Programs have a natural representation as diagrams. Figure 1 shows the diagram corresponding to the program shown in Listing 14. Rectangles with thin edges denote processes while rectangles with thick edges and rounded corners denote cells. Ports are small circles labeled with `'+'`, if a service is provided or `'-'` if a service is needed. There are no ports linking the processes shown in Listing 14.

3.2.3 Communication in Erasmus

Erasmus processes communicate by passing messages over synchronous channels. In order to communicate, two processes must be connected to a *channel* via ports. Erasmus programs say nothing about where processes and cells are executed. A major motivation for Erasmus is to separate deployment of program from its logic. This makes it easier to construct distributed applications. In other words, the source code describes the behavior of the program and a separate XML file describes how cells are mapped to processors. Figure 2 shows a simple configuration file that maps *squarecell* onto processor named *switzerland.encs.concordia.ca*. Similarly, *squareclient* is mapped to *latvia.encs.concordia.ca*. We discuss mapping and its implementation in chapter four.

Listing 14: A program demonstrating the use of `loop`, `while` and `until`

```
counter = { |
   i:Integer := 0;
   loop
     sys.out := "counter " + text i + "\n";
     if i = 10 then
        exit
     end;
     i := i + 1
   end;
};

counter2 = { |
   i:Integer := 0;
   loop while i <= 10
     sys.out := "counter2 " + text i + "\n";
     i := i + 1
   end;
};

counter3 = { |
   i:Integer := 0;
   loop until i > 10
     sys.out := "counter3 " + text i + "\n";
     i := i + 1
   end
};

myCell = ( counter(); counter2(); counter3() );

myCell();
```

3.2.4 Channels and Protocols

Every port has an associated protocol that defines the allowable messages that can be sent through the port. A channel links a *client* port in one process to a *server* port in another process. The protocol associated with the server port must *satisfy* the protocol associated with the client port. For example, the protocol of a teller server might provide *deposit*,

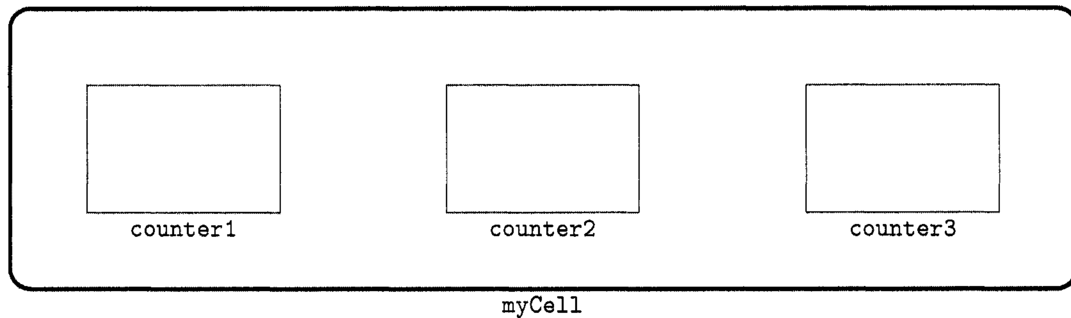

Figure 1: A diagram corresponding to the program of Listing 14

```
<mapping>
  <processor> switzerland.encs.concodia.ca
    <port> 5555 </port>
    <cell> squarecell </cell>
  </processor>
  <processor> latvia.encs.concodia.ca
    <port> 5556 </port>
    <cell> clientcell </cell>
  </processor>
</mapping>
```

Figure 2: A simple configuration file

withdrawal and *balance enquiry* services, but it would allow connection to a client that required only *deposit* and *withdrawal* services. A *query* is a message that travels from the client to the server while messages from the server to the client are called *replies*. Replies are indicated in the protocol syntax by the prefix ^. Every message has a name and an optional type. A message without a type is a *signal*.

ProtocolExpression = ['^'] *Declaration*

| [*Multiplicity*] *ProtocolExpression*

| { *ProtocolExpression* }$_{;}$

| { *ProtocolExpression* }$_{|}$

| '(' *ProtocolExpression* ')' .

A declaration in a protocol declares a message and always introduces a new name. Ports can also be declared within a protocol. Such ports may be sent from one process to another with this protocol. A protocol expression may be preceded by a multiplicity that indicates how often it may be sent:

Multiplicity = '?' | '*' | '+' .

(no operator): the default, means exactly once

'?': means 'optional' (zero times or once)

'*': means 'many' (zero or more times)

'+': means 'at least once' (one or more times)

Protocol expressions are similar to regular expressions. The following are examples of protocol expressions:

```
pairProt = [ first:Integer; second:Integer; ^result];
```

a protocol for transferring a pair of integer values over a channel followed by a result from the server. As mentioned earlier, replies have the prefix ^.

```
pairProt = [ *(first:Integer; second:Integer); finish;  ^result];
```

for repeated transfer of pair of integers followed by a signal *finish* which indicates the end of the stream. A reply, *result* is transferred from the server to the client. The prefix * denotes a repetition, i.e. in this example, zero or more transfers of the pair *first* and *second.*

```
sumProt = [ +(num: Integer) ; finish; ^total:Integer];
```

for one or more transfer of an integer followed by a signal *finish* which indicates the end of the stream. A reply, *total* is transferred from the server to the client. The prefix + also denotes a repetition but the transfer must occur at least once. In this example, the integer, *num* must be sent at least once.

```
initProt = [ ?( start; name:Text; ^reply:Bool) ] ;
```

for zero or one transfer of a signal followed by a text. A reply, *reply* is transferred from the server to the client.

3.2.5 Ports

A port connects two or more processes to a channel. Port declarations associate ports with some well-defined protocols. Ports must be signed. When a port represents an interface that *provides* a service, the port is declared with `'+:'`. A port that **needs** a service is indicated with `'-:'`. Port declarations with `'::'` create channels. For example:

```
p+: initProt; -- declares a port parameter p in a server process S
```

```
q-: initProt; -- declares a port parameter q in a client process C

r:: initProt; -- declares and instantiates a channel r that
              -- can be passed as an argument to p and q,
              -- thereby linking processes S and C
```

A collection or an array of ports may be created, the code

```
servers: Integer indexes +initProt;
clients: Integer indexes -initProt;
```

declares a collection of *servers* and *clients*. An individual port of a collection can be referenced using an index; e.g: *servers*[1] refers to a specific server port. Similarly, *clients*[1] refers to a specific client port.

A message transfer occurs when a qualified name which refers to a field of a protocol is used as lvalue or rvalue. A signal may be sent when such a qualified name is referenced. For example, the statement

```
q.start;
```

q represents a port and `start` is a field of the protocol associated q. The statement sends a `start` *signal* to a server and, the statement

```
q.name := "Registry";
```

references the field `name` of the protocol associated with the port `q`. The statement sends a text `'Registry'` to a server via the port q. The statement

```
service: Text := p.name;
```

receives a text `'Registry'` from a client into a text variable *service.*

Listing 15 shows an Erasmus cell that comprises of two processes: a server and a client. The program in the listing first defines a protocol named, *initProt.* This models an interaction pattern where a client may send a request to a server by first sending a signal, *start* and then a text value indicating the *name* of the service to be started by the server. The server responds by sending a Boolean value that indicates whether or not the service was successfully started. This interaction can occur at most once as indicated by the prefix `?` in the protocol definition.

The process, *startService* declares a port named p before a vertical `|`. This makes p accessible by other components defined in the program. Similarly, the process, *remoteStarter* declares a port named p before a vertical `|`.

The cell definition declares a channel named *ch.* The channel uses the protocol (*initProt*) defined in the program. The client end of the channel is passed to *remoteStarter* and the server end to *startService.*

The client sends a query to the server to start a given service. The server responded by communicating the status of the service to the client. The program illustrates basic sending and receiving of messages in Erasmus. The diagram shown in Figure 3 corresponds to this program.

Receive expressions can be rvalues within expressions. However, receive expressions may not be used in the conditions following `if` or `elif`.[1]

[1] This is a temporary restriction of the prototype implementation.

Listing 15: Sending and Receiving messages

```
initProt = [?(start; name:Text; ^reply:Bool)];

startService = { p+: initProt |
   p.start;
   n:Text := p.name;
   sys.out := "about to start " + n + "...\n";
   -- ... some code here
   p.reply := true;
   sys.out := "server: " + n + " started\n";
};

remoteStarter = { p -: initProt|
   name:Text := "Registry";
   p.start;
   p.name := name;
   started:Bool := p.reply;
   if started = true then
      sys.out := name + " started";
   end
};

cell = (ch::initProt; remoteStarter(ch);
        startService(ch) );

cell();
```

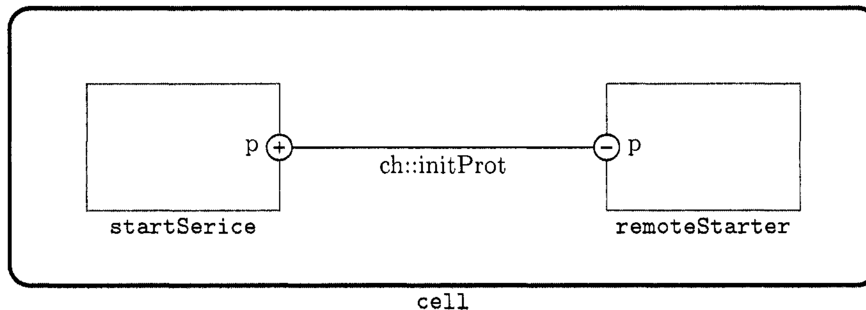

Figure 3: A diagram corresponding to the program of Listing 15

3.2.6 Select statement

There are situations in which a process is willing to communicate with more than one process. The `select` statement allows such process to choose between several possible communications. The process can enable or disable alternative communications (branches) using Boolean *guards*.

Select = (`select` | `loopselect`) *Policy* { *Guard Sequence* } `end` .

Policy = [`fair` | `ordered` | `random`] .

Guard = '`|`' [*Rvalue*] '`|`' .

The `loopselect` is a shorthand for a common situation in which `select` is nested with a `loop`. The following are equivalent:

```
                              loop
loopselect                      select
   ...                             ...
end                             end
                              end
```

A select statement must have at least one branch.

The *Rvalue* of a guard must be a Boolean expression. The empty guard '`||`' is equivalent to the guard '`|true|`'.

The sequence in a branch of a `select` statement cannot be empty and the first statement must be a communication statement. This is called the *principal communication* of the branch. A branch of a select statement is *ready* if its guard evaluates to true and the

principal communication is feasible. The principal communication may be a send or a receive statement.

The *Policy* determines the order in which the ready branch of the *select* statement is tested. The default is `fair`, meaning that the branches are tested in a way that avoids starvation of any of the branches. If the selection is `ordered`, the branches are tested in the order in which they appear in the code. If the selection is `random`, the branches are tested in an arbitrary order.

At most one of the two communicating partners may use a `select` statement. Only one of the branches of a select statement will be executed. The policy makes a difference only when more than one branch of the `select` statement is ready.

Listing 16 shows a process that provides *deposit*, *withdrawal* and *balance enquiry* teller services using a `select` statement. The listing also defines a protocol named *tellerProt* which models a pattern of interactions between a server and a client. The protocol defines three alternative signals: *deposit*, *withdraw* and *enquiry*. The signals represent deposit, withdrawal and balance enquiry services respectively. To deposit or withdraw an amount, the client sends a signal indicating the service required to the server. Then it sends a pair representing an account number and an amount to deposit or withdraw. For a balance enquiry service, a client only sends an account number as shown in the protocol. When the server has completed a service, it sends a reply to the client; in this case, it sends the new balance after a deposit or withdrawal or the current balance, if the service was a *balance enquiry*.

Listing 16 also defines a protocol named `cntrlProt` that a process uses to start or stop the teller server. The protocol defines two alternative signals separated by `|`. The signals are

Listing 16: A teller server that uses select statement

```
tellerProt =
  [*(( ((deposit|withdraw);
         acctNo:Integer; amt:Float) |
         (enquiry; eAcctNo: Integer)
      );
     ^balance: Float)
  ];

cntrlProt = [ *( start | stop); ^reply: Bool];

svr = { p +:tellerProt; q +:cntrlProt|
     q.start;
     -- ...
     cashAvailable: Bool := true;
     loop
       select
         ||p.deposit;
           acctNo:Integer := p.acctNo;
           amt: Float := p.amt;
           bal:Float := 0;
           --- fetch account record into bal
           bal += amt; -- update balance
           p.balance := bal; -- send the new bal
         |cashAvailable| p.withdraw;
           acctNo:Integer := acctNo;
           amt: Float := p.amt;
           bal:Float := 0;
           --- fetch account record into bal
           bal -= amt; -- update balance
           p.balance := bal; -- send the new bal
         ||p.enquiry;
           acctNo:Integer := p.eAcctNo;
           bal:Float := 0;
           --- fetch account record into bal
           p.balance := bal; -- send the balance
         ||q.stop;
           sys.out := "Shutting down ...";
           q.reply := true;
           exit;
        end
     end
};
```

Listing 17: Clients to test the teller server

```
client = { p -:tellerProt|
   p.deposit;
   p.acctNo := 555577;
   p.amt := 2005.55;
   sys.out := "\nBalance=" + text p.balance +"\n";
};

cntrlCl = { p -:cntrlProt|
   p.start;
   n:Integer := 0;
   -- some delay here
   p.stop;
   status :Bool := p.reply;
   if status = true then
      sys.out := "Server terminated.\n";
   end
};
```

used in *cntrlCl* to start and stop the teller server as shown in Listing 16 and Listing 17.

The process, *svr* first receives a signal — *start* — from its port *q* and then proceeds to execute the following `select` statement. When it receives a signal from either port *p* or *q*, it executes the sequence that corresponds to the selected branch. The loop terminates when the process receives a *stop* signal from its port *q*. Listing 17 shows possible clients for the teller server process shown in Listing 16.

3.2.7 Recursion, Dynamic Process Creation and Composition

Erasmus processes may be created recursively. Listing 18 shows a program for generating prime number using sieve of Eratosthenes. The process *filter* has an input port and an output port. It receives a sequence of integers on its input port and sends a subsequence to its output port. It prints and stores the first integer it receives which is a prime number.

Listing 18: Generating prime numbers

```
prot = [ arg: Integer ] ;

filter = {p +: prot|
   prime: Integer := p.arg;
   sys.out := text prime + ' ';
   q -: prot;
   filter(q);
   loop
     n: Integer := p.arg;
     if n % prime <> 0 then
        q.arg := n
     end
    end
};

prog = { p -: prot |
   sys.out := "Primes to ?";
   max: Integer := int sys.in;
   cand: Integer := 2;
   loop
     p.arg := cand;
     cand += 1;
     until cand > max
   end
};

cell = ( p::prot; filter(p); prog(p));

cell();
```

Subsequently, it checks each number that it receives; if the number is a multiple of p, it does nothing; otherwise it sends the number to its output port. The process, *prog* prompts the user for a number N and then send the sequence 2, 3, 4 ..., N to its output port.

Listing 19 shows a client-server application. The server provides a factorial service. The

client makes multiple requests to compute factorials of numbers from 1 to 10. The protocol `factProt` is a simple request-reply protocol: the client makes a request and the server responds with a reply to the request. Much like the program in Listing 18, the factorial program illustrates how an Erasmus closure can *delegate* computation to another closure. Upon receipt of a number from its port p, process `factService` creates an instance of `factorial`. It then passes the number received onto the new process, `factorial` through a new port, q. Each new instance of `factorial` creates an instance of itself if it receives a number that is greater than one. The child and the parent processes run concurrently. The processes `factService` and `client` are encapsulated by the cells `factServiceCell` and `clientCell` respectively. When `mainCell` is instantiated, `clientCell` and `factServiceCell` begin to execute concurrently.

The diagram shown in Figure 4 corresponds to the program shown in Listing 19.

The cells, *factServiceCell* and *clientCell* are nested within *mainCell.* Both cells (*factServiceCell* and *clientCell*) have a process each and are connected via the channel labeled *ch:factProt* in the diagram. Messages are transferred using the ports via the channel.

Summary of the features provided by Erasmus

Erasmus provides safe concurrency through processes and cells. Ports, Channels and Protocols are at the heart of communication in Erasmus. Ports provide interfaces for cells and processes to communicate with other cells or processes. A channel represents the medium for transferring message from one entity to another. Protocols ensure that interactions conform to some well-defined rules, thus helping to detect errors. The `select` statement is an elegant construct for introducing and controlling non-determinism in Erasmus programs.

Listing 19: A factorial service application

```
factProt = [ *( query: Integer; ^reply: Integer)];

factorial = { p +:factProt |

   n: Integer := p.query;
   if n <= 1 then
      p.reply := 1;
   else
      q -:factProt;
      factorial(q);
      q.query := n - 1;
      p.reply := n * q.reply;
   end
};

factService = { p +:factProt|
  loop
     in: Integer := p.query;
     q -:factProt;
     factorial(q);
     q.query := in;
     p.reply := q.reply
  end
};

factServiceCell = ( port +: factProt |
                    factService(port) );

client = { p -: factProt |
 num : Integer := 1
 loop
   p.query := num;
   sys.out :=  "factorial of " + text (num) ;
   sys.out :=  " = " + text( p.reply) + "\n\n";
   num += 1;
   until num > 10
  end
};


clientCell = ( port -: factProt | client(port) );

mainCell = ( ch ::factProt; clientCell(ch);
             factServiceCell(ch)  );

mainCell();
```

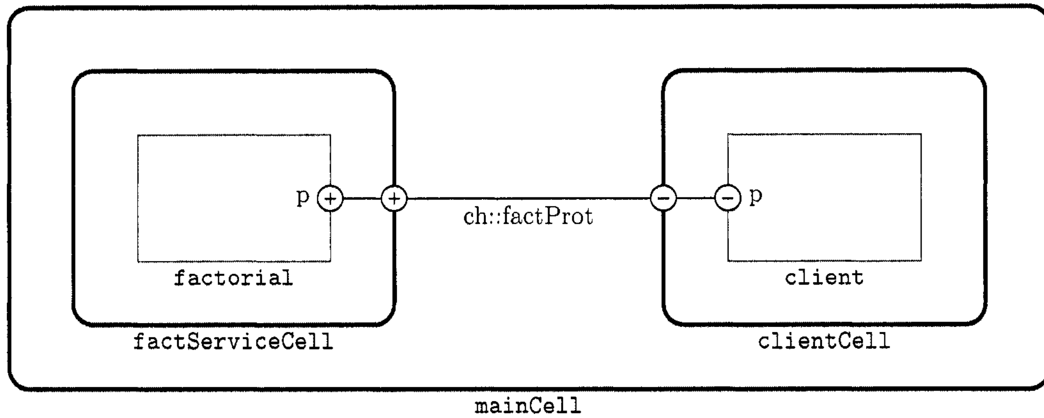

Figure 4: A diagram corresponding to the program of Listing 19

Processes in Erasmus may be created dynamically as shown in the last two examples. Cells may also be nested. This capability helps to build scalable applications. Cells may be extended by adding more cells or processes.

Chapter 4

Communication in Erasmus

In the previous chapter, we presented an overview of the Erasmus programming language. Recall that Erasmus processes in *different* cells communicate only by passing messages over synchronous channels. A cell has a single thread of control and as a consequence, race conditions do not exist when processes within the cell communicate via shared variables. In this chapter, we take a detailed look at communication in Erasmus and, in particular, we discuss the implementation of the mapping of cells to processors.

4.1 Communication

First we discuss the implementation of communication in Erasmus as proposed in [31]. We then discuss the implementation of the mapping of cells to processors.

4.1.1 Notation and Definitions

Small Greek letters, such as ρ and σ denote processes. The letter ρ suggests receiving while σ suggests sending.

A process is *runnable* if it is running or can be run. There exists a *ready queue* for runnable processes. A uniprocessor system has a single, global ready queue. A multiprocessor system has one ready queue for each processor. In a shared-memory multiprocessor system, the number of ready queues depends on how the memory is partitioned. However, a global queue might be a bottleneck, especially if the number of processors is sufficiently large. A queue for each processor, which is more likely, might complicate load-balancing among processors; an hybrid of the two may also be feasible.

A *blocked* process is removed from the ready queue and placed in another queue. It should be assumed that critical sections cannot be interrupted.

4.1.2 Channel Structure

A cell or a closure that creates a channel *owns* the channel. Processes — possibly residing in a different address space — that are connected to a channel have references to the channel. Each channel has a globally unique identifier. As we saw in the previous chapter, a channel is associated with a protocol and protocols have fields. The letter `p` denotes a typical protocol and the letter `f` denotes a typical field. Table 2 gives the meanings of the functions used, in pseudocode. We write `p.f` to denote field `f` of protocol `p`.

Each field of a channel's protocol has two queues: a *reader* queue and a *writer* queue. Each entry in a queue is a pair, (ρ, r) consisting of a process ρ and a reference r. We write

`p.f.writers` and `p.f.readers` to denote the queues of `p.f`. The idea of having a queue for each field of a protocol is based on Brinch Hansen's implementation of Joyce [17].

If process ρ is in queue `p.f.readers`, then ρ is blocked until it can receive data using `p.f`. Similarly, if process σ is in queue `p.f.writers`, then it is blocked until it can write data using `p.f`.

Table 2: Functions used in pseudocode

`empty` q	`true` if the queue q is empty, `false` otherwise.
`first` q	The first element of queue q.
`add`(ρ, r) `to` q	Remove ρ from the ready queue, thereby making ρ non-runnable. Add the tuple consisting of process ρ and reference r to the queue q
`resume` ρ	Put the process ρ in the ready queue, thereby making ρ runnable.

4.1.3 Basic message transfer

Recall that, to receive a message, a process executes an assignment statement involving a port expression as an rvalue. Therefore if ρ wants to read, it executes a statement of the form:

```
v := p.f
```

where v is a variable or an lvalue, in general. This is implemented as follows: if no *writer* is waiting i.e. `p.f.writers` is empty, the pair (ρ, v) is added to `p.f.readers` and process ρ is removed from the ready queue. If on the other hand, some processes σs are waiting on `p.f.writers`, i.e. `p.f.writers` is non-empty, the first entry is removed from `p.f.writers` and the value of the expression is copied into reference v. Process σ is then entered into

the ready queue. The following pseudocode implements this:

```
if empty p.f.writers then
   add (ρ, v) to p.f.readers
else
   (σ, e) := first p.f.writers
   v := e
   resume σ      --  become runnable
```

Similarly, if process σ wants to write, it execute the statement of the form:

```
p.f := e
```

where e is an expression or rvalue. The following pseudocode implements the statement:

```
if empty p.f.readers then
   add (ρ, v) to p.f.writers
else
   (σ, e) := first p.f.readers
   v := e
   resume ρ      --  become runnable
```

A receive operation always waits after storing an lvalue in the channel record. The write operation waits only if necessary for an lvalue to be present, and then uses it to store the appropriate rvalue. Consequently, this implementation assumes that:

A process may store an address belonging to its memory space in a channel; another process may use that address for a *write* operation.

4.1.4 `select` Statements

A select statement consists of several alternatives or branches. Each branch consists of a guard; the principal communication statement and a sequence of statements. The sequence may contain sends and receives which are treated like the "basic message transfer" described in Section 4.1.3.

Since there is no feasible way of implementing communication between two processes that both have `select` statements, we impose the following restriction:

At most one process connected to a channel may access the channel with a `select` statement.

A branch of a `select` statement is ready if its guard is true and the channel for its read (write) operation has a process waiting in its writers (readers) queue. The post-condition of a select statement is that exactly one branch of the select statement has been executed. The post-condition is satisfied by ensuring that:

1. if no branches are ready, the process blocks until a branch becomes ready and then executes that branch;

2. if more than one branch is ready, one branch is chosen for execution.

The choice of which branch is chosen in the second case is determined by the *policy*. A select statement can have one of three possible policies: **`ordered`**, **`fair`** and **`random`**. The actions corresponding to each policy are defined in Table 3.

Table 3: Policies for the `select` statement.

`ordered`	Execute the first ready branch, using the ordering defined by the source text of the program.
`fair`	Choose a branch `fairly` and execute it. This is the default policy.
`random`	Choose a branch at random and execute it.

Where *fairly* is defined as:

> **A branch b of a select statement with N branches is said to be chosen fairly if, when b becomes ready, it is executed after no more than $N-1$ executions of the `select` statement.**

An execution of a select statement can be viewed as performing any of the following actions:

1. if no branches are ready, set the program counter back to the beginning of the `select` statement and yield to the scheduler.

2. If one branch is ready, execute it.

3. If more than one branch is ready, apply the appropriate rule from Table 3.

Case (1) enables the scheduler to schedule another process from the ready queue and to re-schedule the process executing the `select` statement after all of the processes running on this processor have had a chance to run. If the ready queue is implemented as a circular

queue, the desired effect is achieved simply by advancing the queue index to the next entry. Case (2) is the correct action of a `select` statement for each policy and case (3) is correct by definition.

The following section considers a detailed description of the implementation of the `select` statement consistent with the three actions just described.

- S stands for the `select` statement.
- B or b denotes a branch of S.
- There is an unsigned integer counter, c, associated with S and initialized to zero.
- Each branch B has a field $B.c$ that stores a copy of c.
- L denotes a list of branches
- $|L|$ denotes the number of entries in list L.

We show the implementation in pseudocode in Listing 20. The algorithm constructs a list of ready branches and then chooses an action based on the length of the list and the *policy* of `select` statement.

In Listing 20 if the policy is **`fair`** and $|L| > 1$, we claim: (1) the branch with the smallest counter has been waiting longest; and (2) the fairness requirement defined above is satisfied.

From the last three line of Listing 20, we can infer that, after a branch has been executed, it has a higher counter than any other branch, it therefore follows that the branch with the smallest counter has been waiting longest.

Listing 20: Implementation of the `select` statement

```
L := the empty list
for each B of S:
    if B is ready:
        add(B, L)
if |L| = 0:
    for each B of S:
        B.c : = 0
    c := 0
    reset PC and relinquish control
else
    B: Branch
    if |L| = 1:
       B := first (L)
    else
       case policy of:
           ordered:
               B := first(L)
           fair:
               B := the branch in L with minimum B.c
           random:
               B := a branch randomly chosen from L
    execute B
    c += 1
    B.c := c
```

The maximum number of times that the select statement can execute before choosing branch B is the number M of branches with counters less than $B.c$. Clearly, $M \leq N - 1$ (where N is the total number of branches), and the second part of the claim follows.

If $B.c = 0$, it is clearly fair to execute B. All of the counters are set to zero if $|L| = 0$; after this has happened, any branch can be executed fairly and no ready branch will have to wait for more than $N - 1$ iterations. Thus fairness is maintained.

There is the potential for the branch counters in pseudocode in Listing 20 to overflow if the `select` statement is executed many times. The pseudocode shown in Listing 21 remedies

this problem.

Listing 21: Implementation of the `select` statement

```
n := c
loop
   if Bn is ready then
       c := case policy of:
           ordered: 0
           fair: (c + 1) mod |B|
           random: rnd in [0, |B|]
       execute B
       exit
   else
       n := (n+1) mod |B|
       if n = c then reset PC and relinquish control
   end
end
```

4.1.5 Embedded Receives

Erasmus allows receive expressions to be used as rvalues. For example, the following is a valid statement in Erasmus.

```
p.f := q.g + r.h + 1.5
```

`q.g` and `r.h` are receive expressions. Statements like this must be implemented in such a way that their meaning does not depend on the order in which values becomes available. A naive implementation of the statement above is:

```
t1 := q.g

t2 := r.h

p.f := t1 + t2 + 1.5
```

but this implementation can easily lead to deadlock. However, the statement might correctly be implemented with a select statement as shown below:

```
select
    ||p.f := q.g + r.h + 1.5
    ||p.f := r.h + q.g + 1.5
end
```

The only problem with this approach is scalability. The solution requires $n!$ branches where n is the number of receive expressions. However, it is unlikely that typical programs would require large values of n. Erasmus programmers should be aware that expressions with many receiving terms will generate code that is both bloated and inefficient.

In general, if

```
E(r1, r2, r3, ... , rn)
```

is an expression and $\mathbf{r}_i$, $i \leq n$ and $i > 1$ is a receive expression. The compiler should translate `v := E` in a manner that ensures that all of the $\mathbf{r}_i$ are processed in the order in which they become ready, and then `E` is evaluated using the values read. The following code outlines a possible implementation.

```
b1, b2, ..., bn : Boolean := true;

loop while b1 or b2 or b3 ... or bn;

    select

        |b1| t1 := r1; b1 := false;

        |b2| t2 := r2; b2 := false;

        ...

        |bn| tn := rn; bn := false;

    end

end;

v := E(t1, t2, ... , tn)
```

4.2 Mapping of Cells to Processors

We have discussed in the previous section the underlying communication concepts in Erasmus programming language. A process that is ready to communicate waits for its communicating partner. This mechanism reminds one of a telephone call: the caller waits for the person at the other end of the line to pick up the receiver before communication takes place. When both processes are ready, communication takes place. In this section, we look at the implementation of inter-process communication in distributed architectures; in particular, how Erasmus facilitates refactoring of programs through separation of concerns is explained. A similar approach has been used in Mozart/Oz [47].

Software is an investment and refactoring applications when environments change can be very expensive and may be infeasible in commercial environments. To overcome this challenge, Erasmus separates program semantics from its deployment: programmer defines cells and messages that cells exchange; separately, the programmer specifies how cells are assigned to processors — a task commonly referred to as *mapping*.

There are other benefits for explicit mapping of cells onto processors. Cells that communicate too frequently may be mapped onto processors that are close to the scheduler to reduce communication time. Being familiar with the nature of the problem that is solved by the program, programmer may map cells onto processors considering the specific nature of the problem and the characteristics of the individual processors [7].

An Erasmus program may be compiled to run on a uniprocessor system. The same program may be re-compiled to run on a multicomputer — a network of computers with independent memory. Similarly, the same program code may be moved to a multiprocessor system

Listing 22: A simple client-server program

```
sqProt = [ *( query: Float; ^reply: Text)];

square = { p +:sqProt |
 loop
   q: Float := p.query;
   p.reply := text (q * q);
 end
};

squareCell = ( prot +: sqProt | square(prot) );

client = { p -: sqProt |
 num : Float := 10;
 p.query := num;
 sys.out :=  text num + "^2 = " + p.reply + "\n";
};

clientCell = ( prot -: sqProt | client(prot) );

mainCell = ( chan::sqProt; clientCell(chan); squareCell(chan)  );

mainCell();
```

with shared memory. All this is possible without changing the source program. However, compilation of an Erasmus program for a multiprocessor or distributed system requires a separate configuration file that specifies specific detail about mapping of cells onto the participating processors.

Consider the program shown in Listing 22, if the program is compiled without any configuration file, it generates code for uniprocessor systems. However, the two cells: *squareCell* and *clientCell* may be mapped onto different processors by submitting a configuration file to the compiler. Listing 23 shows a sample configuration file that maps *squareCell* onto processor *alpha.encs.concordia.ca* and *clientCell* to *latvia.encs.concordia.ca.*

Listing 23: A simple configuration file

```
<mapping>
  <processor> alpha.encs.concordia.ca
    <port> 5555 </port>
    <cell> squareCell </cell>
  </processor>
  <processor> latvia.encs.concordia.ca
    <port> 5556 </port>
    <cell> clientCell1 </cell>
  </processor>
  <processor> portugal.encs.concordia.ca
    <port> 5556 </port>
    <cell> clientCell2 </cell>
  </processor>
</mapping>
```

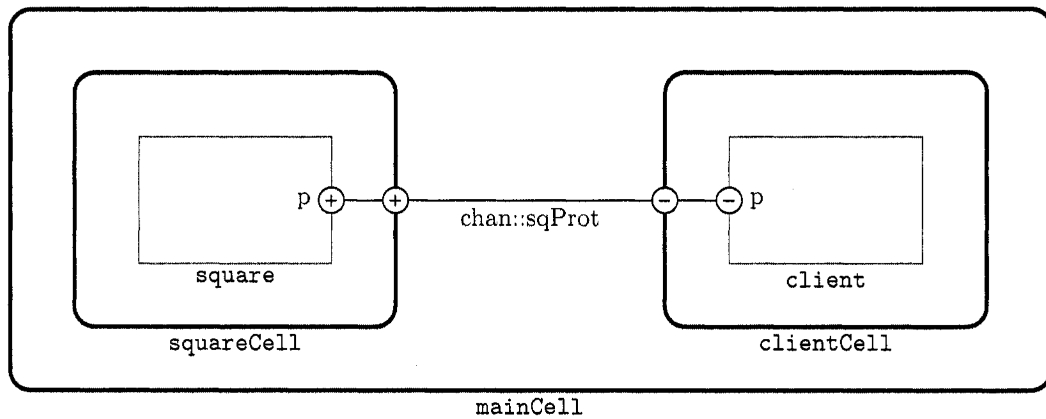

Figure 5: A diagram corresponding to the program of Listing 22

The cells are connected to the channel, *chan.* The processes *square* and *client* have ports that are connected to *chan.* The port *p* of *square* is a server port (i.e. it provides a service) while *clientProc* has a client port (i.e. it needs a service). Figure 5 corresponds to the program shown in Listing 22.

4.2.1 Design

Configuration file

A typical configuration file for mapping Erasmus cells onto processors is a valid XML file with specific tags defined. We are interested in:

1. name/IP address of a processor;
2. an operating system port for a special process, named *broker*. Every processor has a running broker;
3. a list of cells mapped onto the processor.

A mapping specification is enclosed within `<mapping>` and `</mapping>` tags. The tag `<mapping>` is followed by records containing properties of processors participating in the system. Each processor has a record enclosed within `<processor>` and `</processor>` tags and the record comprises of the name or the IP address of the processor, a port number that the processor uses to communicate with cells in other processors, and a list of cells mapped to this processor. The port number is enclosed within `<port>` and `</port>` tags and the name of each cell mapped to the processor is enclosed within `<cell>` and `</cell>` as shown in Listing 23.

XML is a suitable choice for Erasmus because most programmers are familiar with it and there are widely-available tools for processing it. In fact, the XML standard has been adopted by the web community to represent data sent in messages exchanged by clients and servers in web services [23]. The verbosity problem is minimal because mapping file requires only four pair of tags as shown in Listing 23.

Compilation of an Erasmus program whose cells have been mapped to some processors requires a configuration file. The compiler reads the XML file (if any is provided), extracts the data therein and organizes the contents into a table. The compiler generates a unique identification for each cell. Later, it retrieves mapping information for each cell. This includes the processor name, port of the communication *broker* (discussed later). This is eventually appended to a file — hosts.txt — in an order determined by cell id; each line of the file contains a record about a cell. If there is no entry in the configuration file, `'localhost'` and port number `0` is written for that cell. This means that the cell has not been mapped to any particular processor; it may therefore be reasonable to execute the cell on the processor running the broker.

Inter-process Communication

Various approaches for implementing communication and synchronization between isolated processes in concurrent languages have been proposed and implemented. Gregory Andrews [3] proposed a centralized message passing implementation that uses a "*clearing house*". A clearinghouse process matches pairs of communication requests. A template is a message that describes a communication request. When a process wants to communicate, it sends a template or a set of templates, if several communications are possible. When the clearinghouse receives a template, it checks if it has received before a matching template otherwise, it stores the template. If it has a matching template, it sends some synchronization messages to the processes that sent the matching templates. These processes then use the information received from the clearinghouse to communicate. Subsequently, both processes continue at their next statements after exchanging data. A disadvantage of a centralized

implementation is the inherent potential for the clearinghouse to become a bottleneck in the system. This is likely to be exacerbated where large number of processes is involved. Other implementations can be found in [50, 10, 49, 20, 6].

The major challenge is to build a reliable communication and synchronization between processes that exchange messages using the minimal number of messages. The implementation described in this chapter uses a distributed clearinghouse for inter-process communication. Since we require reliable communications and TCP/IP guarantees that messages are delivered in the order in which they have been sent, our implementation is based on TCP/IP.

The Broker process

Each processor runs a special process named *broker.* The broker is responsible for handling all communications by the processes. The port address on which the broker attached to a processor runs is specified as part of the properties of the processor in the configuration file. During start-up, a broker loads a table with the data from the file, *hosts.txt* created earlier during compilation by the compiler. As described earlier, a record in *hosts.txt* comprises of a host name and a port number. Brokers are *daemons* that match communications, i.e., they run in the background without human intervention and are always runnable.

A cell that wishes to communicate with another cell sends the request to its local broker. A broker also maintains a table of valid connections to cells and brokers running on other hosts. Cells must first establish connections to their local brokers before executing any instructions. If a connection has been established, the cell activates its processes. Meanwhile, the broker listens for connections for communication requests from communicating agents (cells or brokers). When a broker receives such request, it stores it until it finds a matching request

from another cell. The message transferred in a particular request depends on whether the request is a read (receive) or write (send) operation.

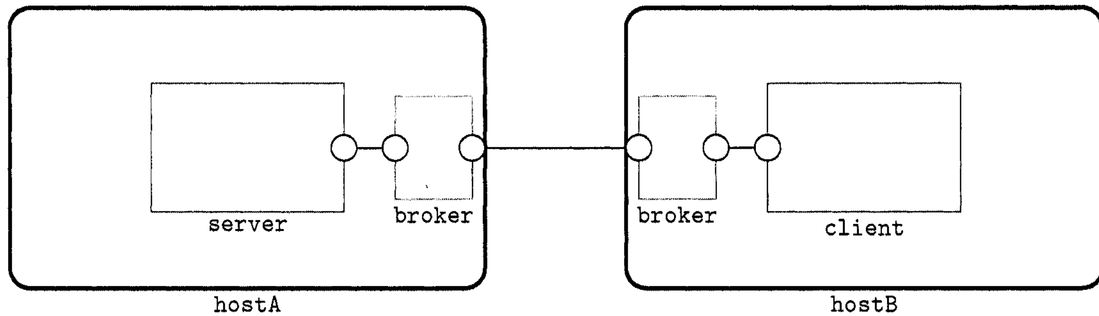

Figure 6: Communication between a client and a server processes

Listing 6 shows a connection between a client process and a server process. The server process labeled *server* in the diagram runs on the host labeled *hostA*. Similarly, the client process, *client* runs on *hostB*. Running on each host is a communication module, labeled *broker*. Every link shown in the diagram is bidirectional. The broker is responsible for matching communication between communicating processes.

The distributed nature of the broker reduces the likelihood of a broker becoming a bottleneck in the system. Messages received by a broker are addressed to cells running in the host that runs the broker. In practice, it is unlikely that a host will execute many cells in different address spaces. A broker does not match communications for pairs in which none of the cells is executed by the host that runs the broker.

Every cell that communicates with another cell is given a port to do so. A port is connected to a shared channel. A channel connects two or more processes. Ports are shown in the diagram as small circles. The following describes an implementation of a port.

```
struct Port
{
  PortState state;
  int field;        // protocol field
  int ccid;         // client cell id
  int scid;         // server cell id
  Process *server;  // pointer to the server process
  Process *client;  // pointer to the client process
  bool *pBool;      // buffer for bool type
  int *pInt;        // buffer for integer type
  double *pDouble;  // buffer for double type
  string *pString;  //  buffer for string type
};
```

The data member *field* above denotes the protocol field referenced in a qualified name that refers to a port variable. The pointers, *client* and *server* store the addresses of the processes connected via the client and server ends of a channel respectively. The pointers are not valid across address spaces. The members *pBool*, *pInt*, *pDouble*, *pString* are buffers and hold data of type `Bool`, `Integer`, `Double`, `Text` respectively.

The data members *ccid* and *scid* of Port shown above hold the ids of the client and the server cells respectively. Communication takes place when a cell is referenced (i.e. a qualified name refers to a field of the associated protocol). When a process in a cell sends a message to another cell, it constructs a message containing some useful headers and sends the message to the broker running on the host. The broker inspects the message and retrieves some of

the headers to determine the location of the communicating partner. The content of the composed message is as follows:

- a port direction which indicates whether the sending port is a server or a client. Possible values are 0 and 1;
- the cell id of the server connected to the channel referenced by the port;
- the cell id of the source of the message — could be a client or a server;
- policy type; values include: 0 (`ordered`), 1 (`fair`) and 2 (`random`) ;
- input/output direction; possible values are 0 (read) and 1 (write);
- the type of the data carried, if the request is a write operation;
- the cell id of the destination cell (client or the server) connected to the channel referenced by the port;
- a tag for the beginning of the data;
- the data to be sent, if it is a write operation; empty if it is a read operation;
- a tag for the end of the data; message may carry more than one request.

When a process sends a write request to a broker, it sends along with the message the data to be written to the other process. If the broker receives a matching request, the data is copied into the variable of the other process and both the sender and the receiver can proceed independently and concurrently. Considering the program shown in Listing 22 and the XML file shown Listing 23, when the *square* process executes

```
q:Float := p.query;
```

assuming that *squareCell* and *clientCell* have been assigned unique ids, 1 and 2 respectively, the compiler builds a message (used later by the runtime system to send and receive data) of this form:

```
0 1 1 0 0 1 2 $$
```

this is explained as follows:

1. the first item (0) means that the port, p used in *squareCell* is a *server* port, i.e. it provides some service
2. the second item (1), is the identifier of the server cell connected to the channel.
3. the next item (1) identifies the source of the message which coincidentally is the same as the server;
4. Item four (0) represents the policy which in this case corresponds to the default policy
5. the fifth item (0) indicates that the process is attempting a read operation;
6. the sixth item (1) indicates that the data type of the data to be read;
7. item seven (2) corresponds to the id of the client that is currently connected to the channel;
8. item eighth and nine ($$) are tags that demarcate the beginning and the end of the data in a message;

This message "`0 1 1 0 0 1 2 $$`" is then sent to the broker. When the broker receives the message, it retrieves the first two items in the message and performs one of the following two actions:

1. if the port is a client port, it checks whether the server that is connected to the channel is running on this host or on a remote host. If the server is on a remote host, it forwards the request to the broker running at the host of the server and waits for a reply from the broker. The reply is subsequently forwarded to the client cell.
2. if the port is a server port, the broker queues the message until (hopefully) a matching request is received from the client.

Therefore in this case, the broker running on *alpha.encs.concordia.ca* forwards the message to the broker running on *latvia.encs.concordia.ca*. This message is matched by the receiving broker when it receives a matching request i.e. a message of the form `1 1 2 0 1 1 1 $d$` from *clientCell*. The symbol d denotes the data which can be of any size. This message is generated when *clientCell* executes the statement

```
p.query := num;
```

Marshalling is the process of converting a collection of data items into a form that is suitable for transmission in a message over a network. The data items are flattened into a stream of bytes. *Unmarshalling* does the opposite by converting a message received at the destination into an equivalent collection of data items. Marshalling and unmarshalling blur the distinction between different data representations across architectures. For instance, there are two ordering for integer: the *little-endian* order represents data by considering

the least significant bytes first before other bytes in a piece of data and, the *big-endian* order which considers the most significant bytes first when representing data. In addition, floating-point numbers also have different representations between architectures.

Typically, the communication broker marshals the data carried in a message before sending the message to a broker on another host. It unmarshals the data when it receives a message from another broker. This process, i.e. marshalling/unmarshalling may be avoided if both the source and the destination hosts are known to have the same architecture. Apart from the Erasmus basic types discussed in Section 3.2.1, Erasmus cells, processes and ports are first-class entities: they can be transmitted from one process to another and may be sent over a network.

Matching occurs when a broker receives two messages that are complementary; one from a client and another from a server. The two input/output requests must be opposite. One of the communicating partners must be a read operation while the other must be a write operation. The type of the data in the messages must also match. During matching, data received is copied and sent to the process executing a read command. The message may contain a set of requests if the process is executing a `select` statement.

4.2.2 Testing and Results

I modified the original compiler (described in Section 1.2) by adding features that implement communications and mapping of cells onto processors. The compiler has been used to compile the program shown in Listing 22 to work on a stand-alone computer. The same program was also distributed on two computers: the server cell running in one process on a computer while the client runs in another process in a different host. The broker

processes on both machines matched communications between these cells. The program works as a client-server system as specified by protocol *sqProt*. The client process sends repeated requests to compute squares of numbers ranging from 1 to 10. Each time, the server responded by sending the corresponding result to the client. Although the program in Listing 22 solves a trivial problem, it nevertheless shows how Erasmus programming language facilitates distribution of programs to different architectures. In practice, this approach may help preserve software investment where software environment changes.

The next section describes some tests conducted to evaluate the impact of communications on the overall performance of programs written in Erasmus.

The two computers used for these tests have the following specifications:

1. *latvia.encs.concordia.ca*: Intel(R) Pentium(R) 4 CPU 3.00 GHz, 2.99 GHz, 1.00 GB of RAM; Operating System: Windows XP Professional, version 2002; Service Pack 2.
2. *lithuania.encs.concordia.ca*: as in 1 above.

The tests are grouped into two cases namely:

1. *Case one:* examines the impact of frequent inter-process communications on performance of programs.
2. *Case two:* examines the impact of infrequent inter-process communications.

Case One

Considering the program in Listing 24, it takes `145s` on *latvia.encs.concordia.ca* for *clientcell* to display the squares of the numbers from 1 to 1500. It should be noted that there is no

Listing 24: A client-server program showing no communications between a client and a server process

```
sp = [ *( ask: Integer; ^answer: Integer)];
square = { p +:sp |
 loop
   in: Integer := p.ask;
   x: Integer := in * in;
   --- do more work
   j: Integer := 0;
   loop while j < 1000000
     j += 1;
   end;
   p.answer := x;
 end
};
client = { p -: sp |
 x: Integer := 1;
 loop
   xs: Integer := x * x;
   stdout :=  text x + "^2 = " + text xs + "\n";
   --- do more work
   j: Integer := 0;
   loop while j < 1000000
     j += 1;
   end;
   x := x + 1;
   if x > 1500 then exit end
 end
};
squarecell = ( p +: sp | square(p) );
clientcell = ( p -: sp | client(p) );
cell = ( p ::sp; clientcell(p); squarecell(p)  );
cell();
```

communication whatsoever between *clientcell* and *squarecell* as shown in the listing.

- **Scenario one:** it takes *clientcell* in Listing 25 289s to display the squares. In this case, `clientcell` communicates with *squarecell* by sending numbers whose squares are computed by *squarecell* through the port *p*.

Listing 25: A client that communicate frequently with square process of Listing 24

```
client = { p -: sp |
 x: Integer := 1;
 loop
    p.ask := x;
    xs: Integer := p.answer;
    stdout :=  text x + "^2 = " + text xs + "\n";
    --- do more work
    j: Integer := 0;
    loop while j < 1000000
     j += 1;
    end;
    x := x + 1;
    if x > 1500 then exit end
 end
};
```

Listing 26: Configuration file for test case in scenario two

```
<Mapping>
  <Processor> latvia.encs.concordia.ca
     <Port> 5555 </Port>
     <Cell> squarecell </Cell>
     <Cell> clientcell </Cell>
  </Processor>
</Mapping>
```

The two processes in Listing 24 and 25 run in one address space. The difference in the elapsed-times for *clientcell* to display all the computed results can be attributed to the communication overhead in executing the program shown in Listing 25.

- **Scenario two:** if the program shown in Listing 25 is compiled using the configuration file shown in Listing 26, it takes 236s on *latvia.encs.concordia.ca* for *clientcell* running in a separate address space to display the squares of the numbers from 1 to 1500. It

is interesting to note that *clientcell* runs faster than in the previous scenario. This suggests that the combined effect of the inter-process communication and the runtime process scheduling overheads is higher in scenario one than the communication delay in scenario two.

- **Scenario three :** if the same program is compiled with the configuration file shown in Listing 27, it takes *clientcell* running on *latvia.encs.concordia.ca* 241s to completely display the squares of all the numbers from 1-1500 as specified in the program of Listing 25. This is slightly higher than that in scenario two. This means that in this case, *clientcell* runs slower than in scenario two. Frequent network communications between the client process in one host and the server process in another host cause delays, which are likely to offset any performance gains in faster execution of *clientcell* due to the host's processor having fewer processes to execute.

Listing 27: Configuration file for test case in scenario three

```
<Mapping>
  <Processor> latvia.encs.concordia.ca
     <Port> 5555 </Port>
     <Cell> clientcell </Cell>
  </Processor>
  <Processor> lithuania.encs.concordia.ca
     <Port> 5555 </Port>
     <Cell> squarecell </Cell>
  </Processor>
</Mapping>
```

In scenarios two and three, communication brokers add to the communication delays. One broker is involved in scenario two but two broker processes (one on each host) are involved in scenario three.

Case Two: Infrequent inter-process communications

Similar to the tests conducted under in case one, the test described here is similar to the last three scenarios in case one. However, all the tests are based on the program shown in Listing 28.

Listing 28: A client process showing infrequent communications with the server process of Listing 24

```
client = { p -: sp |
 x: Integer := 1;
 loop
    xs: Integer;
    if x % 500 = 0 then
       p.ask := x;
       xs := p.answer;
    else
       xs := x * x;
    end;
    stdout :=  text x + "^2 = " + text xs + "\n";
    --- do more work
    j: Integer := 0;
    loop while j < 1000000
     j += 1;
    end;
    x := x + 1;
    if x > 1500 then exit end
 end
};
```

If the processes run in one address space, *clientcell* takes `145s` to display all the squares. However, if the configuration file shown in Listing 27 is used, *clientcell* takes `116s` to finish.

It takes `116s` if the file in Listing 28 is used.

Tables 4 and 5 give a summary of the tests in *case one* and *case two* respectively.

It can thus be concluded that a distributed environment may be more suitable for a system

Table 4: Summary of tests in Case one (Frequent Communications)

Scenario	Execution time
Scenario one	289s
Scenario two	236s
Scenario three	241s

Table 5: Summary of tests in Case two (Infrequent Communications)

Scenario	Execution time
Scenario one	145s
Scenario two	116s
Scenario three	116s

where processes rarely communicate. Frequent communications between the participating nodes in a distributed system may degrade the overall performance of the entire system.

4.2.3 Granularity of Communication

Communication granularity concerns the frequency and the size of messages sent from one process to another process within a system. In a system that uses fine-grained communication; small messages are sent by processes. This might be reasonable in environments where the processors are physically close to one another (closely coupled environments e.g. multi-core architectures). In contrast to fine-grained communication, in a system that uses coarse-grained communication; few large messages are sent from one process to another process. This is most useful in a loosely coupled (processors are physically dispersed) distributed environment where processes perform large computations and send very few but large messages.

Erasmus is intended for communication at any level of granularity. However, programmers are supposed to match granularity to hardware. While it does not make sense to send an

integer over a network to find its square, it might make sense to send it to another processor on the same chip!

Chapter 5

Communication in Erasmus and other languages

In the previous chapter we discussed how Erasmus cells/processes communicate. We also considered how Erasmus cell may be mapped onto processors. In chapter two, we reviewed some concurrent programming languages and concluded the chapter with a discussion around the languages reviewed. In this chapter, we compare communication in Erasmus with the languages discussed in chapter two.

5.1 Language Syntax for Communication

In the previous chapter, we discussed how Erasmus achieves its network transparency by separating the semantics of a program from its deployment. A direct benefit of this approach is that the syntax for expressing local and remote communications is essentially the same. This permits an execution of the *same program* using a different topology/model. This is

similar to the approach used in Mozart/oz [47]. Many languages that provide support for building distributed systems do not provide network transparency. Programmers have to change the underlining source programs when they are ported to different environments. In Erlang, except where process ids have been used throughout the programs, changes may be necessary to the code when they are moved from one environment to another. In Java and many other languages, the syntax for expressing remote communication is different from that required for local communication. For instance, every remote object in Java RMI must implement an interface which must be a sub interface of `Remote`. This makes refactoring of applications written in Java unnecessarily difficult.

5.2 Communication via shared variables

Communicating through shared memory is not new. It has remained the de-facto standard for inter-process communication in multithreaded applications. The danger of this approach which stems from data inconsistency is also well recognized [53]. However, shared memory is an efficient means of communication by communicating processes. Therefore a language that provides shared-memory communication must prevent the problems inherent with this approach. This explains in part why CSP, its derivative languages (e.g. Occam, Occam-π and Joyce) and Actor-Based Language such as SALSA, Erlang and ABCL/1 do not allow inter-process communication via shared memory.

Java and Cilk allow inter-thread communication via shared variables. When more than one thread has access to a shared memory, the result can be largely unpredictable. A thread does not know when another thread is accessing the same location. Programmers must

carefully control the potential non-determinism in this approach.

Typically, shared variables are protected by monitors or its variants such as protected types in Ada^{TM}[1]. Mozart/OZ data stores are abstract stores that allow *legal* operations on stored entities. In practice however, correctness of multithreaded programs in which threads (or processes) communicate through shared variable may be difficult or impossible to guarantee. Despite the obvious performance gain of inter-process communications via shared variables as compared with message passing where data have to be copied from one process to another, message passing approach is a better alternative. Lee contends that by using threads as a computation model, achieving reliability and predictability might be impossible for many applications [43]. Reliability should not be traded for efficiency in language design at least, not in a language designed for programming mission-critical applications, e.g. missile control systems — where the danger posed by unreliable programs can be catastrophic! Brinch Hansen writes:

> ... *efficiency*, *portability*, and *generality* should never be sought at the expense of simplicity, reliability, and adaptability, for only the latter qualities make it possible to understand what programs do, depend on them, and extend their capabilities [15].

The designers of Erasmus emphasize the importance of a 'failure model' for highly reliable software.

Erasmus cells communicate *only* by message passing. Processes within the same cell may communicate *safely* via shared variable: a running process continues until it suspends when it needs to communicate with another process. There is only one thread of control within

a cell hence; only one process can access a shared location at a time. This effectively eliminates data races. Processes in one cell communicate with processes in another cell only by sending messages to them using their local ports that conform to some common protocols.

5.3 Communication by message passing

We have seen that in some of the languages reviewed in chapter two, parallel units (processes or threads) communicate over synchronous channels while in others communication is via asynchronous channel or similar mechanisms. While synchronous communication can be compared with the telephone system, asynchronous mechanism can be compared with the postal service system or recently, telephone voice mail system. In the postal service system, a mail is sent to a given address by the sender and the mail is collected from the same address, at a convenient time, by the owner of the address. While in synchronous message transfer, the sender and the receiver need to be synchronized before an exchange of data takes place, in an asynchronous transfer of message, there is no synchronization between the sender and the receiver. However, asynchronous transfer of messages requires a buffer or *mailbox* between the sender and the receiver of messages.

The functional languages reviewed in chapter two: Erlang, Mozart/Oz and ABCL/1 provide asynchronous communication by message passing. This is not altogether surprising. Objects in functional languages are usually created when needed and often objects can be as large as a computer's available memory which of course, is not infinite. This means that unbounded buffer fits seamlessly into this paradigm.

Indeed, *past* and *future* semantics of ABCL/1 have potentials to increase concurrency of applications. Threads or processes can proceed independently and retrieve responses when needed. However, problems with asynchronous communication are well known. Message buffer can easily be filled with messages thereby raising a buffer overflow error. In addition, direct communication is less complicated and more straightforward than communication via a third-party such as mailbox in Erlang. Hoare argues that unbuffered communication is a better approach where fast interactions are more important than heavy processor utilization [40]. It is also straightforward to implement asynchronous communication (if required, as in Ada 95) with synchronous mechanism by simply introducing a buffer between communicating units.

5.3.1 Channel and Protocols

Erasmus builds on older languages such as Joyce and CSP. For instance, Erasmus port protocol is similar in some respect to Joyce channel alphabet. However, Erasmus protocol is more elegant and versatile. Protocol expressions are like regular expressions. This gives programmers numerous ways of modeling systems with communicating components much more than remote procedure call (RPC)/ remote method invocation (RMI) that is simply an extension of local procedure call. Communication statements within the program code can be checked and analyzed statically for conformity with the associated protocols. This means that Erasmus protocols may facilitate programming of robust applications. For example, if all communications satisfy the associated protocols then the reliability of the system with respect to the given protocols can be guaranteed.

In contrast to other languages, interaction patterns between a client and a server can be

expressed explicitly by an Erasmus port protocol. For instance, consider an interaction where a client must repeatedly send either integer or float values to the server until a stop signal is sent. This pattern of interaction can easily be expressed with an Erasmus protocol as shown in the following protocol definition.

```
-- Erasmus (3)
protocol = [*( intVal: Integer | floatVal: Float), stop]
```

CSP or Joyce's channel alphabet may not be able to explicitly express this kind of interaction completely: it gives alternatives but not sequencing of messages. A Joyce equivalent of the above is:

```
-- Joyce (4)
protocol = [intVal(integer), floatVal(float), stop]
```

Erasmus allows interaction pattern to be made explicit as in (3). This not only improves the expressiveness of the language but also improves program quality by improving its understandability and maintainability. A likely consequence of this is that the long run cost of software maintenance may be reduced. Errors can also be detected during compilation.

CSP inspired several languages, including Occam, Joyce and Occam-π. These languages use communication channels with protocols. Channel simplifies synchronization of interactions. Channel protocol serves to prevent channel misuse and this can be checked at compilation time. CSP and Occam have unidirectional channels. This is an obvious limitation. Unidirectional channels are inadequate for implementing reliable communication where messages need to be retransmitted, if not received by the destination process. The receiving process needs to pass messages in the opposite direction to acknowledge receipts of messages.

Erasmus Ports are of two types: those that provide a service and those that need a service. This is a request-reply mode of communication. The client sends a request and the server responds with a reply.

5.3.2 Select statement and Non-determinism

We saw in chapter two how many of the languages reviewed allow programmers to introduce and control non-determinism in a program. In a seminal paper, Dijkstra [24] formalized non-determinism by introducing `'guarded commands'` (see Section 2.3.1) as a building block for alternative and repetitive program constructs that allow non-deterministic program components. Guarded commands were adopted by Hoare [39] as a means of introducing and controlling non-determinism in CSP.

Many concurrent programming languages including CSP (alternative command), Joyce (`poll` statement), Occam (ALT command) and Occam-π (ALT command), Ada (`select` statement) use guarded commands or some variations for introducing and controlling non-determinism in concurrent programs. Erlang has a `receive` statement; SALSA provides `join continuation`; Mozart/Oz uses Ports (`future` variables) as channels for asynchronous communication.

Erasmus uses `select` statements for sending and receiving messages non-deterministically. This makes it possible to program a variety of safe interactions. Lee argues that non-determinism should be explicit in programs and introduced only when needed [43]. Unlike languages such as Java which allows inter-process communication that is difficult or impossible of being safely and deterministically controlled (communication via shared variables) and like Joyce, CSP, Occam, non-determinism is explicit and localized in Erasmus.

As mentioned in chapter two, output statements cannot be used as guards in CSP. Erasmus does not have this limitation. A principal communication statement may be a send or a receive statement. Receive expressions may also be embedded in an expression as described in Section 4.1.5. Such an expression behaves like a `select` statement, each expression corresponds to a branch of the `select` statement. This capability is an elegant alternative to using an explicit `select` statement and a way of expressing succinctly, solutions to common problems. For example, different values computed and sent from different locations might be combined in one compound expression to compute a value at a different location. Consider the statement

```
grandSales =
            london.totalSales + montreal.totalSales + beijing.totalSales;
```

assuming that *london*, *montreal* and *beijing* are ports connected to processes in London, Montreal and Beijing respectively. The value of *grandSales* is the sum of the values received via the three ports.

For controlling and scheduling processes waiting for communications, CSP leaves fairness to implementation, suggesting FIFO model for selecting matching output commands. As we saw in chapter three, Erasmus provides policies: `order`, `fair`, `random` for `select` statement. The policy is used to determine the order in which a branch is selected when more than one communication is feasible. Joyce implements only the policy that Erasmus called `ordered`. We believe that a production-quality language should be able to ensure fairness.

5.3.3 Message Typing and Serialization

The increasing needs for distributed systems highlights the reasons for languages to provide support for distributed programming. Languages such as Java supports automatic *serialization* of data transmitted from one process to another. Serialization concerns the flattening of the data in a data structure into a serial form that is suitable for storing on disk or transmitted in a message. Java uses *reflection* (the ability to enquire about the properties of a class, such as the names and types of its members [23]) to serialize Java objects so that they can be sent over a network and reconstructed at the destination process.

Apart from the primitive types, Erasmus cells, processes and ports are first-class entities: they can be transmitted from one process to another and can be sent across a network.

Summary: Erasmus and other Languages

Though Erasmus builds on well-established past work, it differs from other languages in many ways. This chapter highlighted the main differences between Erasmus and the languages reviewed in chapter two. The same syntax is used for expressing local and remote communications. The benefit and the disadvantage of shared variables as means of communications were discussed. The chapter also reviewed why synchronous communication is suitable for Erasmus. The chapter compared non-determinism in Erasmus and other languages. The `select` statement represents a powerful abstraction for expressing and controlling non-determinism in Erasmus programs.

Chapter 6

Conclusions and Future Work

In this final chapter, we present conclusions and discuss avenues for future research.

6.1 Conclusions

So far we have considered a number of issues related to the implementation of concurrency in Erasmus. Erasmus provides *modular concurrency* [29] through nested cells. This facilitates development of software that scales over time and needs. In chapter one, we reviewed the object and the process models and showed why the process model is superior to the object model in representing concurrent units.

In chapter two, we reviewed some related work to highlight how Erasmus builds on the success of the past while at the same time avoided the mistakes of the time. Despite the proliferation of concurrent languages, problems associated with concurrent programming are still not uncommon. Erasmus is capable of reducing or eliminating many of these problems.

In chapter three, we considered an overview of the Erasmus programming language. In particular, we showed the importance of protocols and how Erasmus protocols can be used to model a variety of systems. The simple syntax of the language encourages fast learning by both new and experienced programmers. We also described how Erasmus provides non-determinism through the `select` statement.

In chapter four, we described how Erasmus cells and processes communicate. Processes in different cells communicate only by exchanging messages over synchronous channels but processes within a cell may communicate through shared variables. Ports and protocols together form the primary means through which cells communicate. The chapter also considered how cells may be mapped onto different architectures and showed how this capability enhances refactoring of software. We believe through this approach, software maintenance can be made less expensive and less difficult.

In the previous chapter, we compared Erasmus with other languages and showed how Erasmus builds on well-established past work, both theoretical and practical.

To implement communication and mapping of processes on to different architectures, the prototype compiler described in chapter one was modified. The new compiler together with the run-time system is capable of compiling an Erasmus program with processes that run on the same processor. It is also capable of compiling the same source code to run on different architectures; thus achieving the stated objectives of the research. The main contributions of this thesis are as follows:

- modification of an existing compiler;
- introduction of one broker process for each processor;

- design of XML format for metaprograms;
- tests with two Erasmus cells running in the same memory space, in different memory spaces on the same processor, and on different processors.

Time measurements show that true concurrency improves performances. The modified compiler has some limitations. It is capable of mapping cells onto processors in isolated computers and distributed systems. It does not explore multi-core architectures. Further limitations are highlighted in the next section.

Internet use has seen phenomenal growth within the last decade. This was largely driven by advances in telecommunication, government policies (bridging the digital divide), the growth of internet applications and technologies, affordability and other factors. To be considered useful, a modern concurrent language must provide support for building large, scalable real-time and distributed applications that leverage internet platform. Erasmus seeks to make programming these non-trivial applications a triviality.

6.2 Future Work

Erasmus is an ambitious project. There is still a lot of work to be done. In this section, we highlight some important aspects of Erasmus that are yet to be implemented.

- Erasmus requires further rigorous testing of how its capabilities are supported; in particular how cells on more than two hosts may safely interact without deadlock. It is reasonable to build some non-trivial applications that demonstrate the capabilities of the Erasmus language.

- We mentioned that cells, processes and ports may be sent from one process to another and over a network. It would be worthwhile to implement this capability and provides test cases that show how and where this might be useful. To achieve this objective, Erasmus must provide automatic serialization of the structural types such as maps, processes, ports and cells. The current prototype compiler implements only the basic types — `Char`, `Integer`, `Float`, `Bool` and `Text`.

- Erasmus will eventually provides mobile code/cells that may be executed in different hosts over their life times.

- Erasmus must ensure that mobile code/cells run in constrained environments; i.e. they must be prevented from accessing certain resources in the host in which they run.

Bibliography

[1] Ada. Ada 95 Reference Manual. Revised International Standard ISO/IEC 8652:1995, 1995. www.adahome.com/rm95.

[2] Gul A. Agha. *A Model of Concurrent Computation in Distributed Systems.* The MIT Press, 1986.

[3] R. Gregory Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[4] Joe Armstrong. A History of Erlang. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*, pages 6.1–6.26, New York, NY, USA, 2007. ACM Press.

[5] Joe Armstrong, Robert Virding, Clases Wikstrom, and Mikes Williams. *Concurrent Programming in ERLANG.* Prentice Hall, second edition, June 2004.

[6] R. Bagrodia. Synchronization of Asynchronous Processes in CSP. *ACM Transaction on Programming Languages and Systems*, 11(4):585–597, October 1989.

[7] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[8] Fred R.M. Barnes and Peter H. Welch. Communicating mobile processes. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures*, pages 201–218. IOS Press, 2004.

[9] F.R.M. Barnes and P.H. Welch. Prioritized Dynamic Communicating and Mobile Processes. *IEE Proceedings - Software*, 150(2):121–136, April 2003.

[10] Arthur J. Bernstein. Output Guards and Non-determinisn in CSP. *ACM Transaction on Programming Languages and Systems*, 2(2):234–238, April 1980.

[11] Per Brinch Hansen. An Outline Of A Course On Operating System Principles. In C. A. R. Hoare and R. H. Perrot, editors, *Operating Systems Techniques, Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland*, pages 29–36, New York, NY, USA, 1971. Academic Press.

[12] Per Brinch Hansen. *Operating System Principles.* Prentice Hall, Englewood, NJ, jul 1973.

[13] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[14] Per Brinch Hansen. The Solo Operating System: A Concurrent Pascal Program. *Software Practice & Experience*, 6(2):141–149, June 1976.

[15] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Clis, NJ, 1977.

[16] Per Brinch Hansen. Joyce - A Programming Language for Distributed Systems. *Software Practice & Experience*, 17(1):29–50, January 1987.

[17] Per Brinch Hansen. A Multiprocessor Implementation of Joyce. *Software: Practice & Experience*, 19(6):579–592, June 1989.

[18] Per Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[19] Per Brinch Hansen. The Invention of Concurrent Programming. In Per Brinch Hansen, editor, *Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 3–61. Springer-Verlang, New York, 2001.

[20] G. N. Buckley and Abraham Silberschatz. An Effective Implememtation for The Generalized Input-Output Construct of CSP. *ACM Transaction on Programming Languages and Systems*, 5(2):223–235, April 1983.

[21] Alan Burns and Andy Wellings. *Concurrency in Ada.* Cambridge University Press, second edition, 1998.

[22] William Douglas Clinger. Foundations of Actor Semantics. Technical Report TR 633, Department of Artificial Intelligence, MIT, May 1981.

[23] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design.* Addison-Wesley, second edition, 2005.

[24] Edsger W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[25] Edsger W. Dijkstra. Guarded Command, Nondeterminancy and Formal Derivation of Programs. *Association for Computing Machinery (ACM)*, 18(8):453–457, August 1975.

[26] Martin Fowler, Beck Kent, John Brant, William Opdyke, and Don Roberts. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, June 1999.

[27] James Gosling, Bill joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, third edition, May 2005.

[28] Irene Greif. Semantics of Communicating Parallel Processes. Technical Report TR-154, Department of Artificial Intelligence, MIT, September 1975.

[29] Peter Grogono. Modular Concurrency. Keynote Speech for Canadian University Software Engineering Conference (CUSEC) 2006, January 2006.

[30] Peter Grogono, Nurudeen Lameed, and Brian Shearing. Modularity + Concurrency = Manageability. Technical Report TR E–04, Department of Computer Science and Software Engineering, Concordia University, September 2007.

[31] Peter Grogono and Brian Shearing. A Note on Communication. Technical Report TR E–05, Department of Computer Science and Software Engineering, Concordia University, August 2007.

[32] Peter Grogono and Brian Shearing. MEC Reference Manual. Technical Report TR E–06, Department of Computer Science and Software Engineering, Concordia University, January 2008.

[33] Boehm Hans-Juergen. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268. ACM Press, 2005.

[34] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. *ACM SIGPLAN Notices*, 38(11):388–402, October 2003.

[35] Carl Hewitt. Viewing Control Structures as Pattern of Passing Messages. Technical Report AIM-410, Department of Artificial Intelligence, MIT, December 1976.

[36] Carl Hewitt and Henry Baker. Actors and Continuous Functionals. Technical Report MIT/LCS/TR-194, Department of Artificial Intelligence, MIT, December 1977.

[37] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. *MIT*, 13(4):238–242, April 1973.

[38] Charles Anthony Richard Hoare. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[39] Charles Anthony Richard Hoare. Communication Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[40] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice Hall International, third edition, June 2004.

[41] Nima Jafroodi. A Type System for the Erasmus Language. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, January 2008.

[42] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* September 2002.

[43] Edward A. Lee. The Problem With Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[44] Charles E. Leiserson et al. *CilK 5.4.6 Reference Manual.* MIT http://supertech.lcs.mit.edu/cilk, June 1998.

[45] SGS-THOMSON Microelectronics Limited. *Occam 2.1 REFERENCE MANUAL.* SGS-THOMSON Microelectronics Limited, 1995.

[46] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *ACM Queue*, 3(7):26–34, September 2005.

[47] Peter Van Roy. General Overview of Mozart/Oz. Slides for a talk given at the Second International Mozart/Oz Conference (MOZ 2004). Available online at http://www.cetic.be/moz2004/talks/GeneralOverview.pdf, 2004.

[48] Jerome H. Salzer. M. I. T. Project MAC. Technical Report MAC-TR-16, Department of Artificial Intelligence, MIT, March 1965.

[49] Fred B. Schneider. Synchronisation in Distributed Programs. *ACM Transaction on Programming Languages and Systems*, 4(2):125–148, April 1982.

[50] Abraham. Silberschatz. Communication and Synchronization in Distributed Programs. *IEEE Transaction on Software Engineering*, 5(6):542–546, November 1979.

[51] Bjarne Stroustrup. The Design of C++0x. *C/C++ Users Journal*, May 2005.

[52] Herb Sutter. The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005. Available online at http://www.gotw.ca/publications/concurrency-ddj.htm.

[53] Herb Sutter. The Trouble With Locks. *Dr. Dobb's Journal*, March 2005.

[54] Herb Sutter and James Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, September 2005.

[55] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIG PLAN Notices*, 36(12):20–34, December 2001.

[56] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *OOPSLA '86 Proceedings on Object Oriented Programming Systems, Languages and Applications*, pages 258–268, September 1986.

Appendices

.1 Syntax

Keywords

Types: `Bool` `Decimal` `Float` `Integer` `Text`

Functions: `float` `int` `rand` `text`

Other: `alias` `and` `assert` `copy` `div` `elif` `else` `end` `exit` `fair`
`false` `if` `loop` `loopselect` `mod` `not` `or` `ordered` `random`
`region` `select` `share` `skip` `then` `true` `until` `while`

Symbols

Assignment operators: `:=` `+=` `-=` `*=` `/=` `%=`

Binary operators: `+` `-` `*` `/` `%` `.`

Comparison operators: `=` `!=` `<>` `~=` `<` `<=` `>` `>=`

Protocol operators: `?` `*` `+` `|` `;` `^`

Declaration operators: `:` `+:` `-:` `::` `^`

Separators: `,` `;` `|`

Brackets: `(` `)` `[` `]` `{` `}`

Programs

Program = { *ProtocolDefinition* | *ClosureDefinition* | *CellDefinition* }$_;$
Instantiation.

Definitions

ProtocolDefinition = ProtocolName '=' *Protocol* .

ClosureDefinition = ClosureName '=' *Closure* .

CellDefinition = CellName ('=' | '+=') *Cell* .

Descriptions

Protocol = ProtocolName

| '[' *ProtocolExpression* ']' .

Closure = ClosureName

| '{' [{ *Declaration* }$_{;}$] '|' *Sequence* '}' .

Cell = CellName

| '(' [{ *Declaration* }$_{;}$ '|'] { *Declaration* | *Instantiation* }$_{;}$ ')' .

Protocols

ProtocolExpression = ['^'] *Declaration*

| [*Multiplicity*] *ProtocolExpression*

| { *ProtocolExpression* }$_{;}$

| { *ProtocolExpression* }$_{|}$

| '(' *ProtocolExpression* ')' .

Multiplicity = '?' | '*' | '+' .

Declarations

Declaration = *VariableDeclaration* | *PortDeclaration* .

VariableDeclaration = { VariableName }, [':' [*Mode*] *Type*] [':=' *Rvalue*] .

PortDeclaration = { PortName }, *Direction* *Protocol* [':=' *Rvalue*] .

Mode = `copy` | `share` | `alias` .

Direction = '+:' | '-:' | '::' .

Type = *BasicTypeName* | *Cell* .

BasicTypeName = `Bool` | `Integer` | `Decimal` | `Float` | `Text` .

Statements

Sequence = { *Statement* }; .

Statement = `skip`

| `exit`

| `until` *Expression*

| `while` *Expression*

| *Assertion*

| *Declaration*

| *Instantiation*

| *Assignment*

| *Conditional*

| *Loop*

| *Select* .

Assertion = `assert` '(' *Rvalue* ',' *Rvalue* ')' .

Instantiation = (*Cell* | *Closure*) '(' { PortName | VarName | *Rvalue* }, ')' .

Assignment = { *QualifiedName* }, *AssOp* *Rvalue* .

AssOp = ':=' | '+=' | '-=' | '*=' | '/=' | '%=' .

Region = `region` *Sequence* `end` .

Conditional = `if` *Rvalue* `then` *Sequence*

{ `elif` *Rvalue* `then` *Sequence* }

[`else` *Sequence*] `end` .

Loop = `loop` *Sequence* `end` .

Select = (`select` | `loopselect`) *Policy* { *Guard* *Sequence* } `end` .

Policy = [`fair` | `ordered` | `random`] .

Guard = '`|`' [*Rvalue*] '`|`' .

Values

Rvalue = UnOp *Rvalue*

| *Rvalue* BinOp *Rvalue*

| *FunctionName* *Factor*

| *Literal*

| { *Factor* }, .

Factor = *QualifiedName* | *Literal* | '`(`' *Rvalue* '`)`' .

FunctionName = `int` | `float` | `text` | `rand` .

Lvalue = { *QualifiedName* }, .

QualifiedName = [*QualifiedName* ‘.’] (VariableName | PortName) .

Operators

UnOp = | ‘+’ | ‘-’ | `not` .

BinOp = ‘*’ | ‘/’ | `div` | ‘%’ | `mod` | ‘+’ | ‘-’ |

‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘=’ | ‘!=’ |

`and` | `or` .

Constants

Literal = Bool | Integer | Decimal | Float | Text | *Closure* | *Compound* .

Bool = `false` | `true` .

Integer = { Digit } .

Decimal = { Digit } ‘.’ { Digit } .

Float = { Digit } ‘.’ { Digit } [(‘e’ | ‘E’) [‘+’ | ‘-’] { Digit }] .

Text = { Character } .