Transactions on Information and Communications Technologies vol 10, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

Use of the symbolic concurrent programming Erlang to test the integrated digital networks access

V. Dobrota

Technical University of Cluj-Napoca, Department of Communications, 3400 Cluj-Napoca, Romania

Abstract

The purpose of this paper is to present an efficient tool, as well as a new teaching method, using a symbolic concurrent programming language Erlang (hosted on a workstation running UNIX/DOS/Windows) for a telecommunications application. The aim is to realise a software simulator for the ISDN access, being an ISUP state machine implementation, but also the paper is focused on underlining the stages that a teaching course would have to address in order to generate, step by step, a rapid prototype. It can be useful to understand the mechanisms of a client/server architecture and a socket/port based inter-processes communication. Obviously the software simulator is a powerful instrument to verify the specific signalling protocols involved as well.

1 Introduction

The experiments with programming telecommunications applications using different languages (such as Lisp, Prolog, Parlog) stated that productivity gains can be achieved by a symbolic language, which provides primitives for concurrency and error recovery, and the execution model does not have back-tracking [9]. Most of the telecommunications processes are asynchronous in operation and they must have a unique correspondent process in the language, which means granularity of concurrency. The Computer Science Laboratory of Ellemtel Utvecklings AB (Sweden) presented Erlang at ISS'90 and one year later a faster implementation of this new software package was released to users. The main features of Erlang, as a symbolic concurrent language, recommend it for building the robust concurrent systems, and also make it suitable for rapid prototyping (compare to the time it takes using conventional imperative languages). Erlang is being used for prototyping cordless applications at Ericsson Radio, for controlling photonic switching at Ericsson Telecom etc, as well as in universities around the world and in European collaborative projects for telecommunications such as RACE *(Research and Development in Advanced Communication Technologies in Europe)*. The first book about concurrent programming in Erlang was published by Prentice Hall in 1993 [1]. In MAGIC, which is a RACE project investigating advanced Broadband-ISDN signalling protocols, it has been necessary to develop a software simulator for the ISUP *(ISDN User Part)* protocol, in order to test the interworking of B-ISDN and the existing Narrowband-ISDN signalling systems. The approach of the project was to prototype the ISUP state machines in Erlang. This paper intends to present the experimental results of this, but not from the technical point of view. The aim is to underline the stages that a teaching course would have to address, for an easy understanding and efficient learning of this new language, and then for having the ability to implement a rapid prototype concerning a specific domain (telecommunications for instance).

2 Overview of Erlang for teaching purposes

The data types and objects provided by Erlang that are mentioned in this paper are the following:

Numbers:	For example: integers, floating point numbers, ASCII characters.
Atoms:	Individual objects of a textual content.
Tuples:	An object that contains a fixed number of other objects.
Lists:	This provides a means of storing a variable number of other objects in a sequential manner.
Modules:	Groups of Erlang functions that are independently compiled.
Processes:	Independently executing Erlang programs (Note Erlang processes are lightweight processes and execute within a single UNIX process).
Ports:	The means of sending messages between processes.
BIF:	Built-In Functions (functions provided by the Erlang language).
References:	Globally unique symbols.

The first step in learning how to use Erlang for a symbolic concurrent programming is to understand its application to sequential programming. This paper supposes that the readers are already familiar with other programming languages because it is very interesting to adopt a permanent comparative method, to discover the advantages and disadvantages in different circumstances. The following examples illustrate the basic Erlang data types, such as: numbers (integers, floats), atoms, tuples, lists, pids, ports, references.

```
* Examples of integers in base 10:  6, -6
* Examples of integers in base 16:  16#B0CF, 16#A010
* Examples of integers in base  2:  2#11110011, 2#10101010
* Examples of  ASCII values: $A, $s
* Examples of floats:  16.536, -9.2311, 20.45E-14
* Examples of atoms:  'This is an atom', this_is_an_atom, abcd
* Examples of tuples: {this,that},{12345,{this,that},'Tuple'}, {}
* Examples of lists: [this,that],[12345,{this,that}],[],"i_am",
                     [105,95,97,109], ""
* Examples of variables: I_am_a_variable, VariableName, J
```

There are no restrictions concerning the length of the atoms or the characters included. To store any size but fixed number of items it is suitable to use a tuple. For variable number of items it is proper to use a list which is dynamically sized. Variables must start with a capital letter and they can only be bound once, i.e once set their value cannot be altered, which is an important feature of Erlang.

Every process is a complete virtual machine having at its creation a unic Pid *(Process identifier)*. Suppose we are in the process A , with Pid_A, and we want to create another process named B. The code is: `Pid_B = spawn(Module, Func, Args)`, where `Module` represents an independent entity (module) which has a name (must be an atom) and includes different functions (their names must be also atoms) that could be exported or not to other modules. `Func` is the name of a function defined within specified `Module` or within current module. `Args` is a list that represents the arguments of function and because they can be any Erlang data structure it is even possible to have a function with no arguments. There is a special category of functions named BIFs *(Built In Functions)* which are part of a module called `erlang` and they normally do actions that are of general interest or are impossible to be realised in other way. Some of them could return information about the system, such as `date()` or `time()`. Other BIFs provide a conversion between different data types [1]:

```
integer_to_list(Integer)     list_to_integer(AsciiIntegerList)
float_to_list(Float)         list_to_float(AsciiIntegerList)
atom_to_list(Atom)           list_to_atom(AsciiIntegerList)
tuple_to_list(Tuple)         list_to_tuple(List)
pid_to_list(Pid)             list_to_pid(AsciiIntegerList)
```

To illustrate the Erlang data types, the following example shows the conversion of the atom 'Erlang' to a list of ASCII characters (in decimal codes). The example was run on a UNIX machine named helios and the commands to be entered are shown in bold:

```
helios% erl
Erlang (JAM) emulator version 3.3.581
Eshell V2.0
1>atom_to_list('Erlang').
[69,114,108,97,110,103]
```

The port is another data type which provide byte stream interfaces to external UNIX processes. Communication is realised using a socket. On the Erlang side data is represented by a list of integers and on the UNIX side the bytes in the signal are proceeded by the length given in 2 bytes, with most significant one first. So a port can be seen as an external Erlang process and it is started using a BIF like in the following example: `Port = open_port({spawn, Process}).`

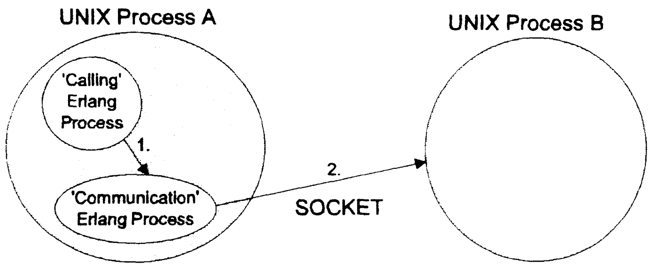


Figure 1: Inter-processes communication using a socket

The exchange of information between processes (including ports) is realised by sending messages, which are in fact tuples containing Erlang data types. The message is sent to the Pid of the destination process as follows: Pid ! {'My message'}. However, in Erlang it is common practice to include the Pid of the calling process in the message, which permits the destination to identify the originating process and to send replies. The Pid of the current process can be found from the BIF *self()*. As an example we will send the message [49,50,51,52] from a process identified by Pid_A to another process identified by Pid_B. The recipient must check the identity of the source.

```
-module(com).
-export([trs/2,rec/2]).
trs(Pid_A,Pid_B) ->
      List = [49,50,51,52],
      Pid_B ! {self(), List}.
rec(Pid_A,Pid_B) ->
    receive
    {Pid_A, Message} ->
    io:format(" Message received by Pid_B ~w~n", [Message]);
    Other ->
    io:format(" Error in receiving message ~n", [])
    end.
```

Finally, the references are data types which provide globally unique symbol guaranteed not to match any other symbol in the system. To create a reference means to apply a BIF called *make_ref()*.

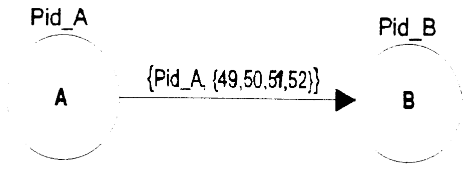


Figure 2: Inter-processes communication using Pid

The above overview illustrates the most important aspects and problems that have to be covered by a teaching course as a first introduction to a new symbolic concurrent language, Erlang. Best results can be obtained following the recommendations from [1] and [2].

3 Use of Erlang to implement an ISUP state machine

The next aim of this paper is to present a method of understanding the mechanism of a client-server architecture and a socket/port based inter-processes communication. It is expected to improve the ability to create a rapid prototype of a software simulator for the ISDN access, being an ISUP state machine implementation. The technical requirements are very important in designing of this tool but they can not be covered by this paper. Because each Erlang process will communicate with other connected Erlang processes, the readers must become familiar with the concept of socket. A socket is a full duplex communication channel between two UNIX processes either over the network to a machine elsewhere or local between processes running on the same machine. There are two parts: the initiator and the connector. The initiator is the UNIX process that opens the socket first and after it issued a series of system calls to set up it waits for a response from another UNIX process, called the connector, which agrees to connect. There is a bidirectional exchange of information between them until the socket is closed. The following building blocks (each of them has one or more Erlang processes attached) have been used: N-ISDN Network Application, Interworking Unit, Communications Link [3]. In order to simplify the implementation it is possible to have two configurations [4]: single ISUP state machine and two ISUP state machines *(Figure 3)*.

3.1 General description of the software simulator

The following requirements were adopted from the beginning: three independent circuits on the N-ISDN links; fully monitored processes; simple to understand user interface;an interactive system for Call Control (both configurations) and Signalling Procedure Control (single ISUP state machine configuration only); an

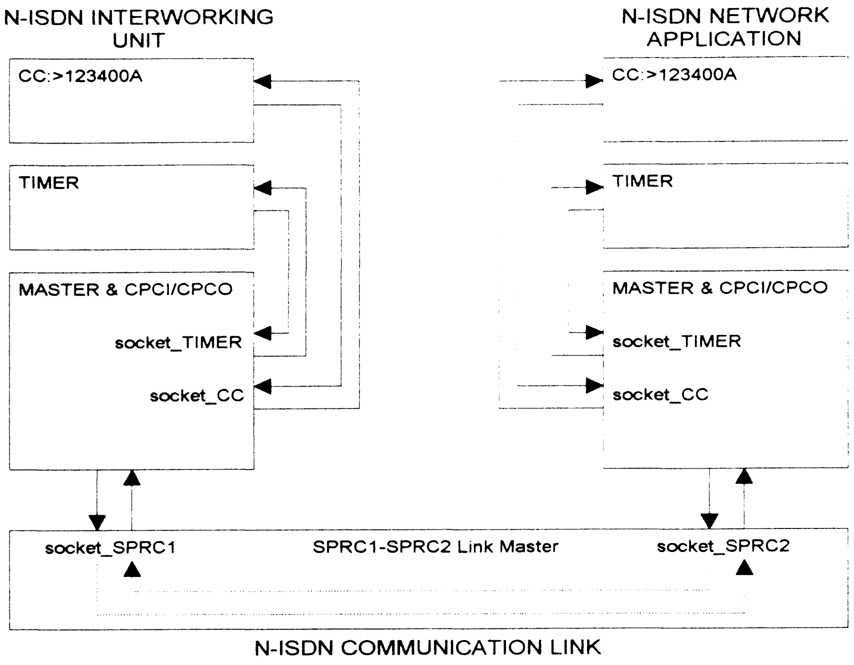
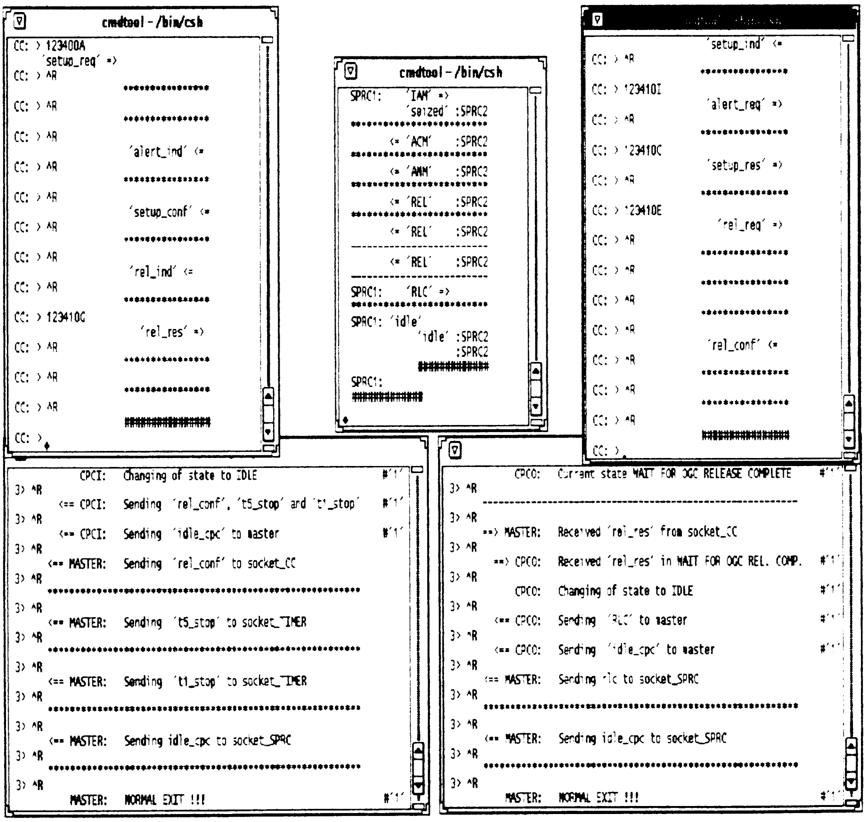


Figure 3: Two ISUP state machine configuration

easily upgradable package by inserting new messages or new facilities. The ISUP simulation makes use of the following main component blocks: **CC** *(Call Control):* This is an entity that simulates the call control processes of the signalling applications. It is realised as an user interface displaying and allowing the user to input signalling primitives. **SPRC** (*Signalling Procedure Control):* This is an entity that simulates the signalling procedure control processes. It is realised as an user interface displaying and allowing the user to input signalling messages, but only for single ISUP state machine configuration. **TIMER** *(Timing Control):* This is an entity that offers four independent timers(T1,T5,T7,T9) for every active circuit. It receives timer_start and timer_stop commands and sends alarms when the tir expired. **MASTER & CPCI/CPCO** *(Master & Call Processing Control Incom...g, Call Processing Control Outgoing):* This is an entity that monitors all the sockets of the ISUP state machine and also implements all the functions of the CPC machine for both incoming and outgoing calls. It is not realised as a user interface but it provides complete details about the exchange of information through the sockets and about every internal transition of the current state machine. **SPRC1-SPRC2 LINK MASTER**: This is an entity that essentially simulates the N-ISDN link between two ISUP state machines. It includes the function of SPRC for both machines and also monitors the exchange of information from one ISUP machine to another.

All these blocks communicate by sockets: ***socket_CC***: Socket between CC and MASTER & CPCI/CPCO, used to transfer primitives; ***socket_SPRC:*** Socket between SPRC and MASTER & CPCI/CPCO, used to transfer messages and primitives; ***socket_TIMER:*** Socket between TIMER and MASTER & CPCI/CPCO, used to transfer timer_start, timer_stop and timer_expired; ***socket_SPRC1:*** Socket between SPRC1-SPRC2 LINK MASTER and MASTER & CPCI/CPCO (IWU), used to transfer messages and primitives; ***socket_SPRC2:*** Socket between SPRC1-SPRC2 LINK MASTER and MASTER & CPCI/CPCO (NA), used to transfer messages and primitives.



```
cmdtool - /bin/csh
CC: > 123400A
     'setup_req' =>
CC: > ^R
                     ****************
CC: > ^R
                     ****************
CC: > ^R
                      'alert_ind' <=
CC: > ^R
                     ****************
CC: > ^R
                      'setup_conf' <=
CC: > ^R
                     ****************
CC: > ^R
                      'rel_ind' <=
CC: > ^R
                     ****************
CC: > 123410G
                        'rel_res' =>
CC: > ^R
                     ****************
CC: > ^R
                     ****************
CC: > ^R
                     ################
CC: >
```

```
cmdtool - /bin/csh
SPRC1:    'IAM' =>
          'seized' :SPRC2
*************************
     <= 'ACM'    :SPRC2
*************************
     <= 'ANM'    :SPRC2
*************************
     <= 'REL'    :SPRC2
*************************
     <= 'REL'    :SPRC2
-------------------------
     <= 'REL'    :SPRC2
-------------------------
SPRC1:    'RLC' =>
*************************
SPRC1: 'idle'
             'idle' :SPRC2
                    :SPRC2
             ############
SPRC1:
############
```

```
                      'setup_ind' <=
CC: > ^R
                     ****************
CC: > 123410I
                      'alert_req' =>
CC: > ^R
                     ****************
CC: > 123410C
                      'setup_res' =>
CC: > ^R
                     ****************
CC: > 123410E
                        'rel_req' =>
CC: > ^R
                     ****************
CC: > ^R
                     ****************
CC: > ^R
                     ****************
CC: > ^R
                      'rel_conf' <=
CC: > ^R
                     ****************
CC: > ^R
                     ****************
CC: > ^R
                     ################
CC: >
```

```
          CPCI:    Changing of state to IDLE                          #'1'
3> ^R
      <== CPCI:    Sending 'rel_conf', 't5_stop' and 't1_stop'        #'1'
3> ^R
      <== CPCI:    Sending 'idle_cpc' to master                       #'1'
3> ^R
    <== MASTER:    Sending 'rel_conf' to socket_CC
3> ^R
    ****************************************************
3> ^R
    <== MASTER:    Sending 't5_stop' to socket_TIMER
3> ^R
    ****************************************************
3> ^R
    <== MASTER:    Sending 't1_stop' to socket_TIMER
3> ^R
    ****************************************************
3> ^R
    <== MASTER:    Sending idle_cpc to socket_SPRC
3> ^R
    ****************************************************
3> ^R
        MASTER:    NORMAL EXIT !!!                                    #'1'
```

```
          CPCO:    Current state WAIT FOR OGC RELEASE COMPLETE        #'1'
3> ^R
    ------------------------------------------------------
3> ^R
    ==> MASTER:    Received 'rel_res' from socket_CC
3> ^R
      ==> CPCO:    Received 'rel_res' in WAIT FOR OGC REL. COMP.      #'1'
3> ^R
          CPCO:    Changing of state to IDLE                          #'1'
3> ^R
      <== CPCO:    Sending 'RLC' to master                            #'1'
3> ^R
      <== CPCO:    Sending 'idle_cpc' to master                       #'1'
3> ^R
    <== MASTER:    Sending rlc to socket_SPRC
3> ^R
    ******************************************************
3> ^R
    <== MASTER:    Sending idle_cpc to socket_SPRC
3> ^R
    ******************************************************
3> ^R
        MASTER:    NORMAL EXIT !!!                                    #'1'
```

Figure 4: Experimental results for two ISUP state machine configuration

3.2 Experimental results

Figure 4 presents the images of windows involved in a two ISUP state machine configuration. As a tool for a new teaching method this software package has a very friendly user interface, by offering a powerful system which is able to simulate the protocol implemented. It could be possible to design another version of the simulator, with the same requirements, but using another type of inter-processes communication (as it was presented in Figure 2). In this case every Erlang process must know the Pids of all connected processes and also it supposes that users have access to a X Window and graphic interface to Erlang. There is available a graphic manager (based on InterViews from Standford University) which is a separate process, that via Erlang buit-in ports is controlled by the Erlang module `iv`. For technical purposes and for more than two state machines configuration, an InterView based solution could be more efficient, but for teaching purposes the implementation presented herein provides a beginning.

4 Conclusions

This paper presented an overview of a new symbolic concurrent programming language Erlang for teaching purposes, as well as an application to implement a software simulator in order to test the integrated digital networks access (interworking of signalling between the Broadband-ISDN and the existing Narrowband-ISDN). The aim was to underline the stages that a teaching course would have to address in order to understand the mechanisms of a client/server architecture by using Erlang. It is also useful in the process of teaching/learning the telecommunications protocols.

Acknowledgements: The author would like to acknowledge the information and the support provided by staff involved in RACE-MAGIC project, in particular thanks go to Luigi Ronchetti and Luca Cipriani from Ericsson Telecommunicazioni (Italy), Richard Boulter and Graham Popple from BT Laboratories (UK). In addition the assistance from Bjarne Dacker from Ellemtel Utvecklings Aktiebolag is also gratefully acknowledged. Very special thanks go to Paul McDonald from BT Laboratories for his competent help.

References

1. Armstrong, J., Virding, R. & Williams, M. *Concurrent Programming in Erlang*, Prentice Hall, 1993
2. ***, *Erlang Standard Libraries, Version 3.2,* Ellemtel Utvecklings AB, 1992
3. ***, *MAGIC R2044/BTL/DS/P/004/b1,* RACE Project R2044, MAGIC - Descriptor Of The Model / Demonstrator, 4th Deliverable, 1993
4. Dobrota, V. *N-ISDN ISUP Simulator*, British Telecommunications plc, 1993.