Erlang - A survey of the language and its industrial applications

Joe Armstrong
Computer Science Laboratory
Ericsson Telecommunications Systems Laboratory
Box 1505
S - 125 25 Älvsjö
Sweden
joe@erix.ericsson.se

September 18, 1996

Abstract

This paper describes the Erlang programming language and some of the applications which have been programmed in Erlang.

Erlang has many language features more commonly associated with an operating system than a language. Concurrent processes, scheduling, garbage collection, etc., are all performed by the Erlang run-time system and not by the underlying operating system.

The paper starts with a description of some of the requirements that we wanted to satisfy when designing the language. This is followed by a brief introduction to the language. Finally, we discuss some of the applications which have been programmed in Erlang. We have chosen to concentrate on three applications which are commercial products.

1 Introduction

Erlang [ARM95a] is a parallel functional programming language designed for programming real-time control systems (examples of such systems are telephone exchanges, Automated Teller Machines etc.). We wanted to make a language which addressed many of the problems which are handled in an operating system while maintaining the advantages of a declarative programming language. Our problem domain requires that we address the following problems:

- **Real-time** - Real-time response times in the order of milliseconds are required. Erlang is designed for programming "soft" real-time systems where we do not have to meet all timing deadlines.
- **Very large programs** - Control systems can have millions of lines of code, and are programmed by large teams of programmers.
- **Non-stop systems** - Control systems cannot be stopped for software maintenance. The Erlang abstract machine allows program code to be changed in a running system. Old code can be phased out and replaced by new code. During the transition, both old code and new code can be run at the same time. This enables faults to be corrected and software to be upgraded in systems without disturbing their operation.
- **Portability** - Erlang compiles to abstract machine code which can be run on any of a large number of different operating systems. This approach makes the system source and object code compatible.

  The cost of emulation is not a limiting factor in our systems, experiments [HAU94] show that Erlang can often be executed as efficiently as unoptimised C despite one level of machine emulation being involved. The advantages of having portable object code and of having to support only one version of the compiler outweighs the advantages of native code compilation.
- **Concurrency** - Our applications are best modeled by a very large number of concurrent processes. At any instant in time most of these processes are idle. The number of processes and their memory requirements vary with time and are extremely difficult to predict in advance. Erlang has lightweight processes whose memory requirements vary dynamically. No requirements for concurrency are placed on the host operating system.
- **Inter Process Communication** - The Erlang abstract machine supports communication between light-weight processes. Communication is performed by asynchronous message passing.
- **Distribution** - Erlang is designed to be run in a distributed multi-node environment. Every com-

putation in Erlang is performed within a process. Processes have no shared memory and communicate by asynchronous message passing. An Erlang system running on one processor can create a parallel process running on another system (which need not even be the same processor or operating system) and thereafter communicate with this process.

- **Garbage collection** - Erlang is used to program real-time systems. Long garbage collection delays in such systems are unacceptable. Erlang implementations are written using bounded-time garbage techniques, some of these techniques are described in [ARM95b, VIR95].
- **Incremental code loading** - Users can control in detail how code is loaded. In embedded systems, all code is usually loaded at boot time. In development systems, code is loaded when it is needed. If testing uncovers bugs, only the faulty code need be replaced.
- **Robustness** - The Erlang abstract machine has three independent error detection primitives which can be used to structure fault-tolerant systems. One of these mechanisms allows processes to monitor the activities of other processes, even if these processes are executing on other processors. We can group processes together in distributed systems and use these as building blocks in designing distributed transaction oriented systems.
- **Timing** - Erlang has mechanisms for allowing processes to timeout while waiting for events and to read a real-time clock.
- **External Interfaces** - Erlang has a "port" mechanism which allows processes to communicate with the outside world in a manner which is semantically equivalent to internal message passing between Erlang processes. This mechanism is used for communication with the host operating system and for interaction with other processes (perhaps written in other languages), which run on the host operating system. If required for reasons of efficiency, a special version of the "port" concept allows other software to be directly linked into the abstract machine. Examples of the use of the port mechanism are interacting with the host file system, interfacing to a graphical interface and a low level socket interface.

The execution mechanisms listed above are provided by the Erlang abstract machine. Programs which use these mechanisms are completely portable between implementations of Erlang running on different operating systems and processors.

Erlang started [ARM92a] as a Prolog meta-interpretor which added a notion of concurrency to Prolog. Having started as a set of extensions to Prolog, Erlang slowly drifted from the logic to functional school of programming. Erlang has many of the features found in a modern functional language (higher order functions, list comprehensions etc.). It differs from most modern functional languages by having a dynamic type system (inherited from Prolog) and an eager evaluator.

2 Sequential Erlang programs

Erlang programs are composed of modules in which functions are defined. Functions are written as sets of recursion equations. The familiar factorial function is written:

```
-module(math).

-export([fac/0]).

fac(N) when N > 0 ->  N * fac(N-1);
fac(0)            ->  1.
```

This can be evaluated in the Erlang shell as follows:

```
> math:fac(25).
15511210043330985984000000
```

The ">" symbol is the Erlang shell prompt, the line following the prompt is the value returned by the Erlang evaluator.

The annotation `-module(Name)` denotes the module name and `-export(Funs)` is a list of the functions which this module exports. Functions which are not declared in the export list cannot be called from *outside* the module.

In what follows we will often omit the module declarations where they are implied by the text.

A more complex example might be a function to search for a value in a binary tree:

```
lookup(Key, {Key, Val, _, _}) ->
    {ok, Val};
lookup(Key,{Key1,Val,S,B}) when Key<Key1 ->
    lookup(Key, S);
lookup(Key, {Key1,Val,S,B}) ->
    lookup(Key, B);
lookup(Key, nil) ->
    not_found.
```

Nodes in the tree are are either *tuples* of the form `{Key,Value,S,B}` or the atom `nil`. `S` are trees with keys smaller than `Key` and `B` are trees with keys bigger or equal to `Key`.

Lists are written as in Prolog - as examples the familiar `append` and `member` functions are written:

```
append([H|T], L) -> [H|append(T,L)];
append([], L)    -> L.

member(H, [H|_]) -> true;
member(H, [_|T]) -> member(H, T);
member(_, [])    -> false.
```

The Erlang primitive data types are:

- **atoms** - `true, foo, bar 'Hello Joe'`
- **Integers** - `123146142641264531715 6, 42`
- **Floats** - `3.1415926535`
- **Pids** - process identifiers
- **Refs** - guaranteed unique identifiers
- **Funs** - functional objects

Complex data objects are represented as:

- **tuples** - for storing a fixed number of objects, thus: `{a, 1234}`
- **lists** - for storing a variable number of objects, thus: `[a, 23, hi]`

Functional objects are introduced with the syntax:

```
fun(Arg1,..,ArgN) -> ... end.
```

So, for example, the following sequence of expressions:

```
K = 2,
F = fun(X) -> X * K end,
```

creates a functional object `X * K` which is bound to the variable `F`. In this expression `X` is a free variable and `K` is bound to the integer 2.

We can evaluate `F` with the syntax `F(Args)`.

Functions (funs) are first class objects - they can be passed as arguments to other functions or can be returned by functions.

To illustrate this suppose we define the higher order function `adder(C)` as follows:

```
adder(C) -> fun(X) -> X + C end.
```

Evaluating, for example, `adder(10)` returns a function which adds 10 to its argument:

```
> Add10 = adder(10).
#Fun
> Add10(5).
15
```

Suppose, also, that we have the function `map` defined as follows:

```
map(F, [H|T]) -> [F(H)|map(T)];
map(F, []) -> []
```

We can use this as follows:

```
> map(Add10, [1,2,3,4,5]).
[11,12,13,14,15]
```

Here we used the `Add10` function as an argument to `map`.

List comprehensions are introduced by the syntax `[Term||P1,P2,..,Pn]` where each of the `Pi`'s is either a generator of the form `Pattern <- Expression` or a predicate.

Using list comprehensions we can write the familiar quick-sort routine as follows:

```
sort([X|Xs]) ->
    sort([Y||Y <- Xs, Y < X]) ++
    [X] ++
    sort([Y||Y <- Xs, Y >= X]);
sort([]) -> [].
```

Where `++` is the infix append operator.

3 Concurrent Erlang Programs

Erlang is a *concurrent* programming language – parallel activities can be programmed in Erlang itself and do not make use of any concurrency mechanisms in the host operating system.

Erlang has a process model of concurrency. New process are created by evaluating the Erlang primitive `spawn`; send (written with the infix operator `!`) sends a message to a process and `receive` can be used to receive a message.

The primitive `spawn(M, F, [A1, A2,..,An])` creates a new parallel process. This process evaluates the function `M:F(A1,A2,..,An)`. When the function evaluation has completed the process dies.

We can write an `echo` process thus:

```
-module(echo).
-export([start/0, loop/0]).

start() ->
    spawn(echo, loop, []).

    loop() ->
        receive
            {From, Message} ->
                From ! Message,
                loop()
        end.
```

`spawn(echo, loop, [])` causes the function represented by `echo:loop()` to be evaluated *in parallel* with the calling function. Thus evaluating:

```
...
Pid = echo:start(),
Pid ! {self(), hello}
...
```

causes a parallel process to be started and the message `{self(), hello}` to be sent to the process – `self()` is a built-in function which returns the process identifier of the current process.

Note that `receive` is a *pattern matching operation*; the syntax:

```
receive
    Message1 ->
        ... ;
    Message2 ->
        ...
end
```

means try to receive a message described by one of the patterns `Message1,Message2,...` The process evaluating this primitive is suspended until a message which matches one of the patterns is received. If a match occurs the code after the '`->`' is evaluated.

Any unbound variables occurring in the message reception patterns become bound if a message is received.

Message sending is non-blocking with "send and pray" semantics, that is, the sending process does not wait when sending a message and there is no guarantee of delivery of the message (you send the message and pray that it gets there!). Note that if a *sequence* of messages is sent between two different processes and if the messages arrive, then the order of delivery is the same as the order in which the messages were sent. Any explicit synchronisation between processes must be explicitly programmed by the user.

Note that in order to send a message to a process we need to know the name of the process. Recall that the syntax `Pid ! Msg` sends the message `Msg` to the process `Pid` (this is short for process identifier). Initially, the Pid of a process is known only to the process which *creates* the new process. A consequence of this is that this Pid then has to be communicated to all other processes in the system which may wish to send messages to the process.

The Erlang primitive `register(Atom, Pid)` provides a global process name registry. After evaluating `register`, the value of `Atom` may be used as an alias for `Pid` when sending a message.

We can demonstrate the concurrency primitives by implementing a simple client-server model of a bank transaction system. Firstly we write a server process which represents the "bank".

```
-module(bank_server).

-export([start/0, server/1]).

start() ->
   register(bank_server,
       spawn(bank_server,server,[[]])).

server(Data) ->
  receive
    {From, {ask, Who}} ->
      reply(From, lookup(Who, Data)),
      server(Data);
    {From, {deposit, Who, Amount}} ->
      reply(From, ok),
      server(deposit(Who, Amount, Data));
    {From, {withdraw, Who, Amount}} ->
      case lookup(Who, Data) of
        undefined ->
          reply(From, no),
          server(Data);
        Balance when Balance > Amount ->
          reply(From, ok),
          server(deposit(Who,-Amount,Data));
        _ ->
          reply(From, no),
          server(Data)
      end
   end.

reply(To, X) -> To ! {bank_server, X}.

lookup(Who, [{Who, Value}|_]) ->
    Value;
lookup(Who, [_|T]) ->
    lookup(Who, T);
lookup(_, _) ->
    undefined.

deposit(Who,X,[{Who,Balance}|T])->
    [{Who, Balance+X}|T];
deposit(Who,X,[H|T]) ->
    [H|deposit(Who,X,T)];
deposit(Who,X, []) ->
    [{Who,X}].
```

Evaluating `bank_server:start()` creates a global process named `bank_server`. The local variable `Data` contains a data base of `{Name, Balance}` tuples which store the name and balance of the customer's account.

The bank server is accessed by sending it `{From, Request}` messages. `From` is the Pid of the requesting process, this is included so that the bank process can send a reply to the requesting process (otherwise there would be no way of knowing who had sent the message).

The module `bank_client` is used to access the server:

```
-module(bank_client).
-export([ask/1, deposit/2, withdraw/2]).

ask(Who) ->
    rpc({ask, Who}).
deposit(Who, Amount)  ->
    rpc({deposit, Who, Amount}).
withdraw(Who, Amount) ->
    rpc({withdraw, Who, Amount}).

rpc(Msg) ->
    bank_server ! {self(), Msg},
    receive
        {bank_server, Reply} ->
            Reply
    end.
```

Note that remote procedure calls are not primitives but are programmed in Erlang itself.

4 Distributed Erlang Programs

Erlang applications are often designed to run on distributed networks of processors. We use the term *node* to mean a complete self-contained Erlang system. In a distributed network one or more Erlang nodes may run on a single computer.

In designing the language we were careful to avoid any primitives which would not map naturally onto a distributed model of computation. The only way of exchanging data between process is by message passing. All the primitives involving processes work transparently on a network of Erlang nodes.

Inter-processor communication makes use of a sophisticated caching technique [WIK94] which makes data transfer highly efficient. Indeed a cross-processor remote procedure call in Erlang is often more efficient than in a language such as C.

Recall that the primitive `spawn` creates a new parallel process. The new process is always created on the current node. To create a parallel process on a different node the primitive `Pid = spawn(N, Mod, Func, ArgList)` is used. This creates a new parallel process on the Erlang node `N`. This allows an Erlang process running on one Erlang node to create a parallel process on another node in the system. All operations on the new process behave in exactly the same way as if the process had been created on the local node.

Recall that that `register` primitive created a global alias for a process identifier, the scope of this alias is the node on which the process was registered, so this name is not visible outside the node.

To send a message to a registered process on a different node the syntax `{Atom, Node} ! Msg` is used. This sends `Msg` to the process named `Atom` on `Node`.

To turn the bank client-server model from a single processor solution to a fully distributed program we make a single line change to the code in the `bank_client` module, changing the line:

```
    bank_server ! {self(), Msg},
```

to:

```
{bank_server,'Host@Domain'} ! {self(), Msg}
```

where `Host@Domain` is the name of the node where the bank server is running.

Note that no data transfer languages are needed to describe the structure of the messages which are sent between the different systems. The set of messages to be sent does not need to be known in advance but is encoded at run-time. Problems of confusion between different byte orderings on different machines and different representations of data structures in memory are eliminated.

5 Error Handling in Erlang

Erlang has three different mechanism for trapping run-time errors, these can be used to:

- Monitor the evaluation of an expression.
- Monitor the behaviour of a process.
- Raise an exception when an undefined function is called.

Suppose we evaluate `N/M` where `M` is equal to zero, this will cause a run-time error and the process in which this evaluation is performed will die.

We can detect this error in one of two ways. The first method makes use of the primitives `catch` and `throw` which are used for monitoring the evaluation of an expression.

The expression `X = catch N/M` evaluates to the *tuple* `{'EXIT', What}` where `What` is an error descriptor, in this case the term:

```
{badarith,{erl_eval,eval_op,['/',1,0]}}}
```

`catch` can be though of as a meta-function which changes the normal evaluation model by converting errors into terms which can be manipulated by the program.

If no error occurs then the result of evaluating `catch Expr` is the same as that of evaluating `Expr`.

The primitive `throw(Expr)` evaluates `Expr` which then becomes the return value of any enclosing `catch`. Using `catch` and `throw` programs can be written to monitor and correct local errors within a process.

The second error handling mechanism determines what happens when a process dies. Suppose some code evaluating in a process generates an error and that this error is *not* handled using the catch mechanism described above.

If the process Pid1 evaluates the expression link(Pid2) then a *link* is established between these two processes.

When any process dies it broadcasts a message containing information about why it died to all processes in the system to which it is linked. Thus if Pid1 dies, a message containing the reason for death is sent to the process Pid2. link is symmetric, so if Pid2 dies an error message will be sent to Pid1.

As an example, the function monitor(M,F,Args) creates two parallel processes. One of these processes evaluates the expression M:F(Args) the other monitors the process performing the evaluation. If the process performing the evaluation dies, a new process is created by the monitoring process.

```
-module(monitor).

-export([monitor/3, start_monitor/3]).

monitor(M, F, A) ->
    spawn(monitor, start_monitor, [M, F, A]).

start_monitor(Mod, Func, Args) ->
    process_flag(trap_exit, true),
    restart(Mod, Func, Args).

restart(Mod, Func, Args) ->
    Pid = spawn_link(Mod, Func, Args),
    receive
       {Pid, 'EXIT', Why} ->
       %% the monitored process died
       %% start a new process ...
       restart(Mod, Func, Args)
    end.
```

The primitive process_flag(trap_exit, true) allows the evaluating process to trap exit messages. spawn_link creates a new parallel process and links to it in an atomic action.

The third error handling mechanism is used to trap errors occurring when undefined functions are called. If an attempt is made to evaluate M:F(Args) and no code for the module M has been loaded into the system then the function undefined_function(M,F,Args) in the module error_handler is called.

error_handler is a normal Erlang function. A typical definition might be:

```
-module(error_handler).
-export([undefined_function/3]).

undefined_function(Mod, Func, Args) ->
    case code:is_loaded(Mod) of
      {file,File} ->
        % the module is loaded
        exit({undef_fun,{Mod,Func,Args}});
      false ->
        case code:load_file(Mod) of
          {module, _} ->
            apply(Mod, Func, Args);
          {error, _} ->
            exit({undef_mod, Mod})
        end
    end.
```

The point to note here is not this particular code, but rather the fact that the user can program exactly what happens when a program tries to execute code that has not been loaded into the system.

Different types of system, for example, embedded systems, or, development systems need different types of code loading strategy. The above mechanism allows code loading strategies to be programmed by the user in Erlang itself.

6 The Industrial Use of Erlang

Erlang was developed at the Ericsson Computer Science Laboratory and started to spread outside the lab. for internal use in Ericsson in 1987. The first projects were simple prototypes written using a very slow version of Erlang which was implemented in Prolog.

Work with the interpretor led to the development of a much faster Erlang machine [ARM92b] which was loosely based on the WAM with extensions to handle process creation and message passing.

The availability of a faster Erlang implementation encouraged the spread of the language within Ericsson and a number of experimental projects were started. At the time of writing some of these have developed into full-scale Ericsson products.

In 1994 the first International Erlang Conference was held in Stockholm. This conference, which publishes no proceedings has been held every year since 1994. The 1995 conference attracted 160 delegates from 10 different countries. The Erlang conference is the principle forum within Ericsson for reporting work done with Erlang.

By 1995 three projects had matured into stable commercial products, these were:

- **NETSim** - Network Element Test Simulator.
- **Mobility Server** - The Mobility Server is a fully featured PBX.
- **Distributed Resource Controller** - the distributed resource controller (DRC) is a scalable, robust resource controller written in distributed Erlang and running on Windows-NT.

6.1 NetSim

The first commercial product developed using Erlang was a network simulator which simulates the maintenance and operations behaviour of an Ericsson AXE exchange. NetSim was released in 1994. At the time of writing this product is in use at about twenty sites in several countries.

6.2 Mobility Server

In February 1995, Ericsson Business Networks launched the Consono ®Personal Mobility product range.[1] The principle component in the Consono system is a device called the *Mobility Server*, most of the code in the Mobility Server is written in Erlang.

The Ericsson press release [ERI96] at CeBIT-96 summarised some of the features of the Mobility Server:

> Ericsson's solution is a new system concept called a Mobility Server, which acts as an intelligent call control system when linked to a company PBX. The Mobility Server is primarily designed as an add-on to the Ericsson Consono MD110 PBX, although it can also be used with PBX systems from other suppliers.
>
> One of the principal attractions of the Mobility Server is the introduction of Personal Number services for all mobile users. Each employee is able to publish a single number, to which all phone calls and faxes can be sent. The person can then control when and where and from whom they want to receive phone and fax calls.
>
> The Mobility Server automatically routes the calls appropriately, to an extension within the company PBX, or to an external fixed or mobile phone in the public network. The routing can be selective, so that only certain calls are directed to the person's extension, with all other calls going to a secretary or voice-mail system.
>
> The mobility server can be utilized by organisations with as few as 60 extensions but is expected to be particularly attractive to organisations with 400 or more extensions, and highly mobile workforces for whom the quality and flexibility of communications is important. Examples include sales and service organisations, and businesses with employees who telework, spending some or all of their time working from home.
>
> The services offered can be used with wired or cordless PBX extensions, and externally with cellular and PSTN phones, wide-area or local paging systems, and fax/fax mailboxes.

Interestingly, nowhere in the Ericsson press releases was there any mention that the system was programmed in Erlang!

The Mobility Server is the largest system programmed in Erlang yet written and contains over 230,000 lines of production quality Erlang code. At the time of writing we believe this to be the largest functional program ever written, it also demolishes the argument that "functional programming languages cannot be used for large-scale projects".

The Mobility Server is a good example of a non-numerical, non-scientific system problem which can be programmed in Erlang.

6.2.1 Technical Details

The Mobility Server (MS) is an interesting example of a large function program. Here are some details of the program:

- The MS code was organised into 486 Erlang modules, containing 230,000 lines of commented Erlang code integrated with 20,000 lines of C.
- The Erlang source code was 6.6 Mbytes of text, when compiled and loaded into the system it occupied 1.2 Mbytes of code space.
- The system runs on a 40MHz Force 5TE 40Mhz with 32 Mbytes of memory and the VxWorks operating system.
- The MS was written by a team of 35 Programmers. These programmers had no previous experience. They were helped by four consultants from "Erlang Systems".[2]
- The project started in a small way in 1987 as an experiment in prototyping using a very slow Erlang interpretor written in Prolog.

6.3 Distributed Resource Controller

The DRC is written in distributed Erlang and runs on a network of Windows-NT machines. The DRC has a robust scalable architecture and runs on low-cost hardware. The resource which are controlled are a collection of various audio devices, human operators together with various database systems.

The allocation of resources is achieved using a dedicated scripting language - the meta-interpreter for the scripting language is written in Erlang.

[1] Consono is a trademark owned by Telefonaktiebolaget L M Ericsson for marketing and sales of business communications systems by Ericsson.

[2] Erlang Systems is a company owned by Ericsson which was set up to market Erlang and to provide consulting and training in Erlang.

The first functioning system is currently in use on two sites in Denmark where it is used by 638 operators and controls 400 audio ports.

This project was delivered to the end-user and had completed its acceptance tests in less than a year after the requirements had been agreed upon.

Programming took 9,000 man hours and testing 2,000 man hours with seven programmers and two testers. The resulting program has 40 Erlang modules, two external C programs and 12,800 lines of Erlang code.

7 Comments

7.1 Time to market

Although the actual programming phase of the software development process is only a very small part of the total design work the choice of programming language and tools has a profound effect on the total life cycle costs. Use of Erlang in a number of different projects has dramatically shortened time to market and greatly improved the quality of delivered software.

7.2 Performance

Erlang is *sufficiently fast* for delivering real telecommunication products.

Interestingly Erlang rarely does well in small benchmark comparisons with C. C programs are usually faster than the equivalent Erlang programs.

However, when talking about large *systems* the opposite seems to be true – usually the Erlang programs are *faster* than the C. The reasons for this are not well understood.

7.3 Real-time garbage collection

It seems to be a widely accepted myth that garbage collected languages are unsuitable for programming real-time control systems.

In our experience the opposite appears to be true. In implementing large control systems in conventional languages problems of memory leakage and fragmentation are extremely difficult to eliminate. The cost of ad hoc solutions seems to outweigh that of using a purpose-built garbage collector.

One thing that is clear is that the run-time dynamic memory requirements of a real-time system programmed in Erlang are surprisingly small.

8 Future directions

It is clear that Erlang is suitable for programming a wide range of soft real-time control applications. What is less clear is if Erlang is suitable for programming applications with very demanding real-time requirements.

Research efforts in the Ericsson Computer Science Laboratory are concerned with a number of interesting problems:

- Efficient implementation of Erlang. Here we are investigating several techniques - these include native code compilation of Erlang and porting Erlang to run on multi-processor architectures.
- Efficient implementation of protocol software in Erlang. We are investigating how to implement TCP/IP directly in Erlang.
- Typed Erlang. A prototype type checker for Erlang has been written by Simon Marlow and Phil Wadler of the University of Glasgow [MAR96].

A free version of Erlang [ERL96] can be downloaded from the WWW.

Acknowledgments

I would like to thank Mike Williams and Peter Högfeldt for their help in preparing this paper.

References

[ARM92a] Armstrong, J.L., Virding, S.R. and Williams, M.C. *Use of Prolog for developing a new programming language* The Practical Application of Prolog - 1 – 3 April 1992. Institute of Electrical Engineers, London.

[ARM92b]Armstrong, J.L., Däcker, B.O., Virding, S. R., and M. C. Williams, M. C. *Implementing a functional language for highly parallel real-time applications* Software Engineering for Telecommunication Switching Systems. March 30 – April 1, 1992 Florence.

[ARM95a] *Concurrent Programming in Erlang*, Joe Armstrong, Mike Williams, Robert Virding and Claes Wikström. Prentice Hall, 1995.

[ARM95b] Armstrong, J.L., Virding, S.R., *One pass real-time generational mark-sweep garbage collection.* International Workshop on Memory Management (IWMM'95) September 27-29, 1995 Kinross, Scotland

[ERI96] `http://www.ericsson.se/Eripress/CeBIT-96/19.html`

[ERL96] `http://www.ericsson.se/erlang/bulletin/pres_free.html`

[HAU94] Hausman, B. *Turbo Erlang: Approaching the speed of C.* In Implementations of Logic Programming Systems. Evan Tick, editor. Kluwer Academic Publishers, 1994.

[MAR96] `http://www.dcs.gla.ac.uk/~simonm/erltc_toc.html`

[VIR95] Virding, S. R. *Garbage Collector for the concurrent real-time language Erlang.* International Workshop on Memory Management (IWMM'95) September 27-29, 1995 Kinross, Scotland

[WIK94] Wikström, C. *Distributed Programming in Erlang.* First International Symposium on Parallel Symbolic Computation (PASCO'94) September 26-28, 1994 Linz.