ERLANG - AN EXPERIMENTAL TELEPHONY PROGRAMMING LANGUAGE

J. L. Armstrong and S.R.Virding

Computer Science Laboratory, Ellemtel Utvecklings AB

ABSTRACT

We present an experimental programming language called **Erlang** *which is suitable for programming telephony applications. We discuss some of the requirements for such a language and introduce the language by a series of simple examples which show how both sequential and concurrent activities can be programmed. We discuss the error recovery mechanism used in Erlang together with the performance characteristics of the current implementation.*

1. INTRODUCTION

Erlang is a programming language designed for prototyping concurrent real-time distributed applications. It grew out of a series of experiments [1,2,3] aimed at creating a language suitable for programming large scale telephony applications. Some of the main characteristics of the language are as follows:

1.1. Robustness

In programming large systems many small programming errors *will* be made - we view this as inevitable. Formal systems and exhaustive test procedures are currently not capable of ensuring fault free software for systems of the size and complexity of modern telecomms applications. Given that errors *will* be made, we are interested in the problem of detecting and handling those errors in such a manner that the system as a whole exhibits satisfactory behaviour *in the presence of errors*.

1.2. Real Time

We will be concerned with real time applications. Response times are critical. In many circumstances a particular operation must be performed within a certain time. Erlang has mechanisms for expressing real time dependencies.

1.3. Distributed

The kinds of application which we are interested in are *essentially distributed*. Such systems are built from a variable number of communicating nodes. Each node operates autonomously and there is no central or master node in the system. Nodes communicate with other nodes using some form of underlying transport medium. There is only a weak coupling between the actual transport system and the distributed Erlang system. Programs written in Erlang can easily be ported from a uni-processor implementation to a multi-processor or multi-node implementation with minimal code changes.

1.4. Fine Grain

Erlang provides fine grain computational processes. By fine grain we mean that the computational effort required to create or destroy a parallel activity in Erlang is small. The *granularity* of computation has an important influence on how we program. In general fine grain computations are preferable to heavy weight processes for telephony applications (since the natural way of programming a telephony application is *as if* it were composed of a large number of small independent agents).

1.5. Functional Notation

Erlang looks like a single assignment functional language. There is a subtle difference between Erlang and a strict functional language. Erlang was designed as a language for controlling real time applications where the control of hardware and strict sequencing of events is very important. In a pure functional language, calling the same function twice with the same arguments is guaranteed to produce the same result. However, in our language, where a function call may result in a hardware action we cannot ensure that the hardware will always produce the same result. Erlang can be viewed as a strict functional language as regards computational functions (i.e. those not dependent upon interaction with the real world for their success) but as a conventional imperative language as regards for sequencing hardware events.

1.6. Concurrent

Erlang applications are built from a large number of parallel processes. We take the view that the degree of concurrency in the application should exactly match the degree of concurrency in the software structure of the program controlling the application. Programming in an unconstrained parallel language (where *every* statement in the language represents a parallel activity) is difficult and error prone. Within any individual process we describe sequential behaviour, but allow any number of such sequential processes to execute concurrently. The only method of synchronising processes or exchanging data between processes is by message passing.

1.7. Modular

Erlang has an in-built module system to make it easier to construct large systems.

2. SIMPLE EXAMPLES

Erlang programs are constructed from a large number of small functions.

Rather than give a formal definition of the Erlang language and its semantics we show some of the language features by example. The syntax of Erlang data structures and variables are similar to those used in Prolog [4].

We start with the factorial function, this is built from two Erlang *equations:*

```
factorial(0) ->
        ^1.
factorial(N) | N > 0 ->
        N1 = N - 1,
        F1 = factorial(N1),
        N * F1.
```

The general equation syntax has the form:

```
Head | Guard ->
        Body.
```

Function evaluation proceeds by a form of pattern directed function evocation. When a function is called, the system

searches for an appropriate equation whose *Head* matches the function call. Any variables occurring in the head are bound to the values supplied in the call. The *Guard* is then evaluated. If the head matches and the condition implied by the guard is true then the system *commits* and the *Body* of the equation is evaluated. Equations are matched sequentially in the order they appear on the page until an appropriate match is found. It is an error for no equation to match.

Body evaluation is sequential. The return value of the function is the result of evaluating the last statment in the function body. A statment of the form *Lhs = Rhs* occurring in the body of a function causes *Rhs* to be evaluated and the result to be bound to *Lhs*.

Lists are written as in Prolog. Thus, for example, a function which reverses the order of elements in a list could be written:

```
rev(X) -> rev(X, []).

rev([], L) ->
        ^L.
rev([H|T], L) ->
             rev(T, [H|L]).
```

So, for example:

```
rev([a,b,c,d])
```

evaluates to *[d,c,b,a]*.

Note that the notation *^X* denotes a return value and is short for *quote(X)*, where the function *quote* merely returns its unevaluated argument.

The previous examples have shown how functions can be defined using sets of recursion equations. This formalism, while simple, has considerable power. We can show some of the expressive power of Erlang by examining a simplified telephony example.

The example is a dialled number analyser. We define a function *lookup(Seq,Struct)* which analyses some number sequence *Seq* to see if it represents a valid subscriber number. *lookup* can be defined as follows:

```
/*
 * lookup(Seq, Struct) returns
 *       subscriber(Who)    if a valid sequence
 *       get_more_digits    if an ambiguous sequence
 *       invalid            if an invalid sequence
 */

lookup(Seq, nil) ->
          ^invalid.
lookup([], Struct) ->
          ^get_more_digits.
lookup([Digit|More_digits], Struct) ->
          Arg = index(Digit),
          X = get_arg(Arg, Stuct),
          lookup1(X, More_digits).

lookup1(nil, _) ->
          ^invalid.
lookup1(subscriber(Who), _) ->
          ^subscriber(Who).
lookup1(Struct, More_digits) ->
          lookup(More_digits, Struct).
```

How does this work? *Struct* is a data structure defined recursively as follows:

1) *nil* is a *struct*
2) *subscriber(Who) is a struct*
3) *{S1,S2,...S10}* is a *struct* iff
   S1..S10 are all *struct*s

The object *{S1,S2,...S10}* is a 10 argument **tuple** where each argument represents an action to be taken if a particular digit is dialled.

If we were to analyse the number sequence [2,1,4,5] we would first look at the 3rd argument in the tuple (The subscriber can dial a digit from 0 to 9, these are represented by arguments 1 to 10 of the tuple), the third argument would itself by a tuple - we would then look at the 2nd argument in the sub-tree etc.

Eventually we should arrive at a terminal node in the *Struct* which is either *nil* representing an invalid number, or *subscriber(Who)* representing a valid user number.

In our example we used two auxiliary functions. *index* which is defined thus:

```
index(N) ->
          N + 1.
```

and the built in function *get_arg(N,Struct)* which fetches the *N*'th argument from *Struct*. So for example: *get_arg(3, foo(a,b,c,d,e))* evaluates to *c*.

The non recursive equations in both *lookup* and *lookup1* represent boundary conditions, i.e. analysing a sequence in an empty tree always gives the result invalid, analysing an empty sequence always requires the user to supply more digits.

The 3rd equations in both *lookup* and *lookup1* specifies the recursive tree walking case where we successively follow the branches of the tree, which branch depending upon the next digit in the sequence.

Manipulation of a 10-way tree shows some of the power of Erlang for a typical telephony programming problem. Tree insertion is equally simple and is left as an exercise for the reader.

3. EXAMPLES WITH COMMUNICATION

Erlang has primitives for creating a parallel process (**spawn**), and for sending and receiving messages (**send** and **receive**). To illustrate this we take the example of a simple process that could be created to count charging pulses in a telephony metering system.

Firstly a process that records debiting pulses.

```
-module(counter).
-export([start/2]).

start(Sys, Proc) ->
          counter(Sys, Proc, 0).

counter(Sys, Proc, Tot) ->
     receive {
          Proc ? bump =>
             Tot1 = Tot + 1,
             counter(Sys, Proc, Tot1);
          Sys ? report =>
             Sys ! Tot,
             counter(Sys, Proc, Tot);
          Sys ? reset =>
             counter(Sys, Proc, 0).
```

The function *start* is evaluated to start a counter process. The counter process receives either a *bump* message from the process *Proc* or one of the messages *report* or *reset* from

the process *Sys*.

Note that in this example we have included module declarations. The module declaration in the example causes the function *start* to be **exported** from the module, whereas the function *counter* is purely local to the module. Exported functions are the only functions which can be accessed from outside the module.

A set of such a counter process could be started as follows:

```
-module(sys_start).
-export([init/2]).

init(Sys,0).
init(Sys,Max) ->
        Id = spawn(line_handler:start(port(Max))),
        spawn(counter:start(Sys, Id)),
        Max1 = Max - 1,
        init(Sys, Max1).
```

The function *sys_start:init(Sys, Max)* when evaluated spawns *Max* line handler processes and connects to each line handler process a process to count the number of charging pulses sent by the line handler process.

Note that inter-module references require fully qualified names - thus we refer to *counter:start* **within** the module *sys_start* but simply *start* within the module *counter*.

A flexible message passing and process spawning mechanism provides a convenient framework in which simple fragments of a telephony program can be expressed. For example we can illustrates part of the code for a conventional POTS like application.

The code describes the situation which occurs when a call is being set up. We assume that the calling (or A) subscriber has dialled a valid number and the called (or B) number was free.

In such circumstances the caller will hear a ring tone, and the called subscribers phone will be ringing.

This situation can be described by a pair of communicating functions *ringing_a/2* and *ringing_b/2*.

```
ringing_a(A, B) ->
    receive 10 {
        A ? on_hook =>
            A ! stop_tone,
            B ! terminate,
            idle(A);
        B ? answered =>
            A ! stop_tone,
            start_switch(A, B),
            conversation_a(A, B);
        timeout =>
            A ! stop_tone,
            B ! terminate,
            wait_clear(A)
    }.
```

```
ringing_b(A, B) ->
    receive {
        B ? off_hook =>
            B ! stop_ring,
            A ! answered,
            conversation_b(A, B);
        A ? terminate =>
            B ! stop_ring,
            idle(B)
    }.
```

The A side of the call evaluates the function *ringing_a*, the B side *ringing_b* . From the A sides point of view three things can happen. The A side can stop the call attempt by going on hook (the message *on_hook* is received from the process *A*), the B partner can answer (the message *answered* is received from the process *B*) or a timeout can occur (ie. after 10 seconds the event *timeout* occurs).

The Erlang syntax used to express this is:

```
receive 10 {
    A ? on_hook =>
        ...
    B ? answered =>
        ...
    timeout =>
        ...
}.
```

where ... represents the code to be executed if the condition is true.

If an *on_hook* message is received the ring tone that the calling subscriber hears is stopped by sending the message *stop_tone* to the process *A* (*A ! stop_tone*), and a message is sent to the called partner to abandon the call attempt (*B ! terminate*). Finally, the function *idle(A)* is evaluated.

If the called partner answers, the ringing tone is removed and the switch started (*start_switch(A,B)*).

The call is now in progress and the functions *conversation_a* and *conversation_b* will control future progress in the call.

One further point to note about the **receive** primitive is that it is selective, that is to say it takes the first message that matches from a queue of messages that are waiting for the attention of the receiving process. It also queues any unmatched messages.

In our previous example the process which was evaluating the function *ringing_a* was expecting one of the messages *on_hook* or, *answered*. Any other messages which arrive at the process while it is waiting will be automatically queued for future processing.

For example: Suppose the following messages have been sent (but not yet received) to a process, eventually they end up in an input queue to the process:

```
p1,on_hook
p2,off_hook
p3,digit(3)
p3,digit(4)
...
```

The notation *P,M* means there is a message *M* from a process *P* in the input message queue of the process

p1,on_hook is the first entry in the queue.

A *receive* statement of the form:

```
receive { p1 ? on_hook => ... }
```

would remove the top item in the queue, whereas,

```
receive { Id ? digit(Digit) => ... }
```

would remove the third entry of the queue binding the variables *Id* to *p3* and *Digit* to *3*. After the reception of this message the state of the input queue would be:

```
p1,on_hook
p2,off_hook
p3,digit(4)
...
```

The selective behaviour of receive is intended so that Erlang programs can easily model CCITT SDL [5] specifications. In SDL message reception unwanted messages can be *saved* for later processing. The mechanism is implicit in Erlang.

The advantage of embedding a fairly sophisticated message reception mechanism in the language frees the programmer from worrying about what to do with messages that should be processed at a later time (a common problem in telephony programming). This mechanism also avoids some of the deadlock problems which occur in languages like Occam, CSP or Parlog where an unexpected message can permanently block a communication channel.

4.   ERROR   DETECTION   AND RECOVERY

Erlang provides a sophisticated error detection mechanism. The purpose of the error detection mechanism is to allow the system to take certain default actions in the event of an unplanned event.

Using the error detection mechanisms programs can be written to build a system which can restore itself to a safe state in the event of certain classes of error.

The error detection mechanism provides a method whereby one process can monitor the behaviour of another process. If the monitored process dies then the monitoring process is informed - it can then take actions to protect the system and restore it to a safe state.

4.1. Basic Mechanisms

The error recovery mechanisms used in Erlang can be described in terms of two concepts, process *links* and *exit* signals. During execution, processes can establish links between themselves and other processes. The set of processes to which a given process is linked we call the **link set** of the process.

If a process terminates abnormally (with a run-time error) then on termination a special *exit* signal is sent to all the processes in its link set. If a process terminates normally then an *exit(normal)* signal is sent to all the processes in its link set

On receipt of an *exit* signal the default action of the receiving process is to terminate and send *exit* signal to all the processes in its link set. An exception to this is when the signal *exit(From,normal)* is received - such a signal is simply ignored.

The default handling of *exit* signals can be overridden to allow a process to take any required action on receipt of an *exit* signal.

4.2. Simple Example

In the following example evaluating the function *init* creates two processes, the child process (which is evaluating the function *driver:start*) is linked to the parent process (which is evaluating the function *monitor*).

```
init ->
    process_flag(trap_exit, yes),
    spawn_link(driver:start, 10),
    monitor.
```

Spawn_link is an atomic command equivalent to *spawn* **immediately** followed by a *link* command. This is to prevent the situation where a process can die before it is linked so that the parent process never sees the *exit* signal. *process_flag(trap_exit, yes)* is a command which when evaluated tells the system to allow this process to treat *exit* signals as if they were normal inter-process messages (i.e. they can be received within *receive* statements).

```
monitor ->
    receive {
        Id ? Mess =>
            write(received(Id, Mess));
        exit(Id,Why) =>
            write(process_exited(Why)),
            spawn_link(driver:start)
    },
    monitor.
```

If a run time error occurs in the child process, an *exit* signal will be sent to the parent process. This will be trapped by the parent process since it has previously issued a *process_flag(trap_exit, yes)* command.

The parent traps the *exit* signal as follows:

```
receive {
        exit(Id,Why) =>
                ...
        }
```

Id will be bound to the process name of the daughter process and *Why* will be bound to a term describing the reason for the *exit* signal.

4.3. A Robust Server

The following example will demonstrate how we can construct a robust device allocation process. Let us suppose that we have a number of resources (represented by processes) which are allocated by a common device allocation process. The top loop of the device allocation process can be written as follows:

```
top(Free, Allocated) ->
        receive {
            Id ? alloc =>
                    top_alloc(Free, Allocated, Id);
            Id ? release(Resource) =>
                    Allocated1 = delete({Resource,Id}, Allocated),
                    top([Resource|Free], Allocated1)
        }.

top_alloc([], Allocated, Id) ->
        Id ! no,
        top(Free, Allocated).
top_alloc([Resource|Free], Allocated, Id) ->
        Id ! yes(Resource),
        top(Free,[{Resource,Id}|Allocated]).
```

```
delete(H, [H|T]) ->
        ^T.
delete(X, [H|T]) ->
        T1 = delete(X, T),
        ^[H|T1].
```

A process that wishes to reserve one of the devices being managed by the device allocation process sends a message *alloc* to the process and waits for the reply *no* (indicating that no resource is available) or *yes(Process)* indicating that *Process* has been allocated to the requesting process.

When the requesting process has finished with the resource it sends the message *release(Resource)* to the resource allocation process.

The resource allocation process maintains two lists, *Free* a list of the free processes, and *Allocated* a list of {*Resource,Id*} pairs which remembers which Resource *Resource* has been allocated to which process *Id*.

Such a server will work correctly *if we can guarantee that the client processes obey the client/server protocol*. In other words, we must guarantee that the client process releases the reserved resources when they are finished with them.

Let us now suppose that a programming error occurs in the client process. In such circumstances the client may terminate abnormally and therefore not send a *release* message to the allocation process. This situation can be handled with the following modification to the allocation process:

```
top_recover(Free, Allocated) ->
        receive {
            Id ? alloc =>
                    top_recover_alloc_1(Free, Allocated, Id);
            Id ? release(Resource) =>
                    unlink(Id),
                    Allocated1 = delete(Resource, Allocated),
                    top_recover([Resource|Free], Allocated1);
            exit(Id, Reason) =>
                    Resource = lookup(Id, Allocated),
                    Allocated1 = delete(Id, Allocated),
                    top_recover([Resource|Free], Allocated1);
        }.

top_recover_alloc_1([], Allocated, Id) ->
        Id ! no,
        top_recover(Free, Allocated).
top_recover_alloc_1([Resource|Free], Allocated, Id) ->
        Id ! yes(Resource),
        link(Id),
        top_recover(Free, [{Resource,Id}|Allocated]).

lookup(Id, [{Resource,Id}|_]) ->
        ^Resource.
lookup(Id, [_|Allocated]) ->
        lookup(Id, Allocated).
```

In this example we have modified the previous code so that when a resource is allocated to a client process a link is created (by evaluating the *link(Id)* function) between the client and the resource manager.

If the client releases the resource in the normal manner (with a *release* message) the link is removed (by evaluating the *unlink(Id)* function) .

If the client terminates without releasing the resource then an *exit* signal is received by the server (note that it must have previously evaluated the *process_flag(trap_exit, yes)* function) - the server establishes which resource was allocated to the terminating process (by examining the allocation list with the function *lookup*) and releases the resource for future use. In addition, any links between the client and server are automatically removed.

4.4. Error Kernels

In the previous section we showed how to construct a robust device allocation server. This server should function correctly in the presence of programming errors in the client processes. This leads us to the observation that not all code in a large system has to be correct in order that the system as a whole should provide acceptable behaviour. By acceptable we mean that at some level the system can protect itself from minor errors.

An extension of this observation leads to the concept of an Error Kernel.

The Error Kernel we define to be the smallest part of a system which has to be correct in order that the system as a whole should provide acceptable behaviour.

One of the major goals of system design should be to isolate the Error Kernel and to program it in as clear a manner as possible so as to increase the probability that it is correct. Formal tools and proof proceedures then only need to be applied to the code in the error kernel and not to the entire system.

Empirical studies (not described here) have shown that the error kernel for a system comprising some tens of thousands of lines of Erlang code can be as small as a few hundred lines.

5. IMPLEMENTATION

Several Erlang interpreters have been written in various prologs and ported to a number of different machines (VAX 11/750, SUN 3/60, PC/AT). These interpretors have been used to control experimental hardware (see [3] for further details) from the Ericsson MD 110 PABX [6] system which is connected to a network of SUN 3/60 workstations. Distribution is provided by using the MD-110 Group switch.

The performance of the current interpretor has proved adequate for experiments in telephony programming. The purpose of these experiments was to evaluate Erlang as a programming language for programming telephony applications. In this evaluation we were interested in clarity of expression rather than performance.

Future implementations will concentrate on improved performance - we are, for example, investigating how Erlang can be compiled to a flat concurrent logic programming language such as STRAND [7] or KL/1.

6. CONCLUSIONS

We have presented a language designed for building prototypes of concurrent real time robust distributed applications. The language is free from side effects and contains mechanisms with which robust systems can be programmed. An explicit notion of concurrency and an in-built error handling mechanism alows many of the problems encountered in interaction between a language and conventional operating system to be avoided.

7. ACKNOWLEDGEMENTS

REFERENCES

[1] Däcker B., Elshiewy N., Hedeland P., Welin C-W. and Williams M. C., *Experiments with Programming Languages and Techniques for Telecommunication Applications* Proc. 6th Int Conf. on Software Engineering for Telecommunication Switching Systems, Eindhoven, The Netherlands, 14-18 April 1986.

[2] Armstrong, J. L., Elshiewy, N. A. and Virding, S. R., *The Phoning Philosopher's Problem or Logic Programming for Telecommunications Applications* Third IEEE Symposium on Logic Programming, September 23-26, 1986, Salt Lake City, Utah

[3] Armstrong, J.L., Williams, M.C. Using Prolog for Rapid Prototyping of Telecommunications Systems. 7th Int Conf. on Software Engineering for Telecommunication Switching Systems, Bournemouth 3 - 7 July 1989.

[4] Clocksin W. F. and Mellish, C. S. *Programming in Prolog.* Springer Verlag 1981.

[5] CCITT Recommendation Z.100 *Specification Description Language. (SDL)*

[6] Reprint of Ericsson Review articles on MD 110 from nos 1 and 2 1982

[7] *STRAND: New Concepts in Parallel Processing.* Ian Foster and Steven Taylor. Prentice Hall, 1989.