# The development of Erlang

Joe Armstrong
Computer Science Laboratory
Ericsson Telecom AB
joe@cslab.ericsson.se

## Abstract

This paper describes the development of the programming language Erlang during the period 1985-1997.

Erlang is a concurrent programming language designed for programming large-scale distributed soft real-time control applications.

The design of Erlang was heavily influenced by ideas from the logic and functional programming communities. Other sources of inspiration came from languages such as Chill and Ada which are used in industry for programming control systems.

## 1 Introduction

This paper describes the development of the Erlang programming language. Erlang is a language which draws heavily from various traditions in the logic, functional and real-time control programming communities.

Our goal was to make a language which could be used for building large soft real-time control systems. By large I mean systems with possibly millions of lines of code. By a soft real-time system I mean a system which does not fail catastrophically if a real-time deadline is missed.

Erlang was developed by the author and his colleagues at the Ericsson Computer Science Laboratory. Ericsson is currently the world's largest supplier of telecoms systems and the world's leading manufacturer of mobile phones.

Ericsson's flagship product is a telephone exchange called the AXE10 which is programmed in a proprietary language called PLEX. A typical AXE10 system has over 5 million lines of PLEX. The AXE10 is specified to have a "down time" of less than 3 minutes per year - such requirements are common in the telecoms industry.

The programming languages which are used for building such systems have to deal with a range of problems not usually encountered in traditional batch or interactive computer systems. For example, one requirement is that the system should be in "continuous" operation - this means we must do software upgrades without stopping the system. It also implies the use of fault-tolerant processors and a software architecture which protects the integrity of the system from various classes of programmer error.

Erlang grew out of a series of experiments which we performed to see if we could find better ways of programming telecoms systems. Our criterion for success was that we could program telecoms systems with less effort and fewer errors than could be done using conventional technology.

This paper starts with a brief discussion of the various milestones which the language went through. This is followed by sections describing the language, finally future directions of research are mentioned.

## 2 Milestones

- **Early experiments**

  Being a telecoms laboratory we were in the fortunate position of always having hardware to play with. The lab had a Ericsson MD110 (which is a small telephone exchange) which had been modified so it could be controlled by a conventional Unix machine. We programmed POTS[1] in as many programming languages as we could lay our hands on.

  The main conclusion [5] was that declarative language programs for POTS were a lot shorter and easier to understand than imperative language programs. Unfortunately the declarative languages lacked features for concurrency control and had poor error handling facilities.

  We concluded that we would like something like Prolog with added facilities for concurrency and improved error handling. No such language existed at the time.

- **1986 - 1989 The Prolog interpreter**

  Having decided that we liked Prolog for programming POTS we started a series of experiments call LOTS (Lots of POTS) to program not only POTS but also an extremely large number of different telephony features in Prolog. Before long I had programmed a large subset of PABX features in Prolog. To do so I wrote a Prolog meta-interpretor which added the notion of a process to Prolog and which added facilities for advanced error detection and recovery. This work is described in [1].

- **The Bollmora group**

  Using the results of the work with the Prolog interpretor we were able to attract the curiosity and interest of a group of engineers[2] who were responsible for a future software architecture of the Ericsson MD110.

[1] POTS stands for Plain Old Telephony Service, that is, subscriber A calls subscriber B and they chat for a while.

[2] Who worked in Bollmora, Stockholm.

This group thought that they could use our results as a vehicle for investigating their own software architectures. This began a period of collaboration which led to the development of Erlang as a fully fledged language and to a new range of Ericsson products.

During the period 1986-1988 members of the computer science lab worked closely with the Bollmora group, meeting once or twice a week. During this time the Prolog interpretor (and the language which it defined) changed rapidly. The language grew and evolved and somewhere along the line acquired a name, Erlang, named in honor of the Danish mathematician Agner Krarup Erlang (1878–1929) whose name is associated with the telecoms industry.

- **The jam machine**

  By about 1988 is was clear that Erlang as it was now called was a good language for prototyping telephone exchanges [10]. It was a strange mixture, with declarative features (inherited from Prolog), multi-tasking and concurrency (inherited from EriPascal and Ada) and an original combination of error handling mechanisms.

  Erlang was, however, far too slow to be used for product development. The Bollmora group did some measurements and decided that for product development they needed a system that was 40 times faster than the Prolog interpretor.

  This requirement lead to the development of a number of different abstract machines and compilation techniques used for implementing Erlang. A cross compiler from Erlang to Strand [7] was developed. After a number of experiments the JAM [4] machine was invented. This was based on the Warren Abstract Machine with added primitives for concurrency and exception handling.

  By this time the Erlang effort had grown to three people. Mike Williams wrote a byte code emulator for JAM code, I wrote the compiler and Robert Virding wrote the support libraries.

  While we only ever distributed one version of the system to external users, internally there were several competing Erlang "engines" where we experimented with different implementation techniques.

  The resulting JAM implementation was delivered to the Bollmora group in 1989. Fortunately it was 70 times faster than the original Prolog interpretor - unfortunately they had now revised their original estimates and wanted a machine that was 280 times faster than the original.

- **Adding distribution**

  During the period 1989-1994 the Erlang "group" expanded (there has never been a formal group). Claes Vikström joined the group in 1990 and added distribution to the language. The original language had been designed with "hooks" for adding distribution - but this was never actually done until Claes implemented it.

  The Bollmora group had now decided to go ahead and build a new PABX in Erlang. Erlang wasn't fast enough but it was generally thought that the performance problems could be solved. In 1992 they started development of a product called the "Mobility Server". In 1995 this was launched and became part of the "Consono" product range. At the time of writing the Mobility Server is marketed in 12 countries. Among other things it is used to control the DECT mobile phone in the European Parliament in Strasbourg.

- **Fight for acceptance**

  During the period 1992-1996 it was by no means clear that Erlang was suitable for programming large-scale industrial products. While we were very successful in a number of small projects, Erlang programming and programmers still represented a very small percentage of the programming effort at Ericsson.

  The Mobility server project continued and we attracted several new small projects. During this time we were involved in massive "guerilla marketing" activities. I don't think there was a single person in Ericsson who ever came out of Bjarne's[3] office without a copy of the Erlang manual under their arm!

- **The beam**

  Performance has always been a major problem. In 1992 Bogumil Hausman started work on the BEAM.[4] The BEAM compiles Erlang to C which can then be compiled with a conventional C compiler. The BEAM can also compile to threaded code which can be freely intermixed with compiled code.

  Compiled code is faster but takes up more space than threaded code. In a large embedded system with millions of lines of code, the volume of object code can be a major problem.

  Infrequently used and non-performance critical parts of the system can be compiled to threaded code while performance critical parts of the system can be compiled to native code. There are performance tools which we can use to analyse the system and find out which parts of the system need to be native code compiled.

  The BEAM is described in [6]. In many applications the BEAM is comparable with C in performance terms. The BEAM replaced the JAM as the principle system for new product development in 1997.

- **Erlang Systems**

  Up to 1995 most users were "enthusiasts" and Erlang spread internally within Ericsson by "word of mouth". We didn't find new users, they found us.

  As the language spread we needed to produce training material and hold courses to train new users. Initially, all courses were held at the Computer Science Lab by members of the Erlang "group". While holding such courses was great fun, demand soon far outstripped supply.

  In April 1993 a new company "Erlang Systems" was formed to handle sales, marketing, education and consulting of Erlang. We had been in the fortunate position of having supervised a number of Master degree

[3] Boss of the computer science lab.

[4] Bogdans's Erlang Abstract machine.

students in computer science at the University of Uppsala. Having completed a Masters degree many of these students were employed by Erlang Systems.

Mike Williams moved from the Computer Science Laboratory to manage Erlang Systems and had soon built up a strong team of consultants and impressive training facilities. In 1996 over 500 Ericsson programmers attended courses at Erlang Systems.

Erlang consulting is one of the most important factors that have led to the spread of Erlang within Ericsson. In the first Erlang projects there was very close contact between the developers and the users. We knew every user and could provide individual help and tuition - this doesn't scale.

Now, when we start a new project, we always try to get at least two Erlang Systems consultants working on the project. They provide a vital link between the users and developers of Erlang. This is a significant factor in the success of new projects.

It is also interesting to note that the skill level required for an entry-level industrial programmer has increased in recent years. A good degree in computer science is now a pre-condition for working with state of the art software technology. The days of the amateur hacker are fast disappearing.

- **The tools**

  The current Erlang system comes with an extensive toolkit. As well as the obvious software tools (yacc, lex look-a-likes etc.) there is a wide range of tools which are useful for building telecoms applications.

  These include things like cross-compilers for interfacing Erlang to foreign language applications, an ASN.1 interface compiler, an SNMP tool-kit, a HTTP-server etc.

  Many of the new users of Erlang are not initially attracted by the language but rather by the set of tools which comes with the language. For example, it is extremely easy to build an SNMP application using the Erlang SNMP toolkit. The SNMP toolkit, for example, contains a MIB compiler, and set of default methods which allows a non-expert user to build an SNMP agent in a matter of hours. Doing this in a conventional language is much more difficult.

  Many of the first versions of the tools were written by students as part of their Master thesis work.

- **Mnesia**

  Many real-time applications need access to data over long periods of time. For example, in a telephone exchange subscriber data must be stored for many years and must be accessible within a few milliseconds. Charging data (telephone bills) must be kept and never lost.

  Mnesia [9] is a real-time distributed database designed for programming telecoms applications in Erlang. Amnesia is written entirely in Erlang.

- **OTP**

  On 1 January 1996 a new Ericsson division was created to support applications written in Erlang. Our users want a lot more than just a programming language. For example, some users want not only the language but also the operating systems and the hardware platform to be delivered in one package.

  The OTP (Open Telecom Platform) division can provide Ericsson users with anything from a simple Erlang system which runs on a PC to an embedded system complete with hardware. The division has the goal of providing prospective users with a turn-key system which they can turn on and start programming from day one of a project. Standard OTP software comes with extensive libraries which solve common application problems.

  Another goal of the OTP division was to transfer responsibility for the maintenance and support of the Erlang system from the Computer Science Laboratory to a mainstream Ericsson division.

## 3 Erlang in 14 Examples

The previous sections described the development of Erlang. The following sections describe Erlang through a number of small examples.

### 3.1 Sequential Erlang

#### Example 1 - Factorial

All functions are defined in modules, for example, factorial can be written:

```
-module(math).
-export([fac/1]).

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

The annotation `-export([fac/1])` means the function `fac` with one argument is from the module. Only exported functions can be called from outside the module.

Once a module has been loaded into the system the query evaluator can be used for function evaluation:

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

#### Example 2 - 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.
```

Here the *tuple* `{Key,Val,S,B}` represents a node of a binary tree. Tuples store fixed numbers of arguments.

#### Example 3 - Append

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

The notation [H|T] when occurring in a function head denotes a pattern matching operation on a list. H is the first element of the list, T is the remainder of the list. When occurring on the right hand side of a production it denotes a list constructor.

### Example 4 - Sort

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

The notation [Expr || ...] introduces a list comprehension and ++ is the infix append operator.

### Example 5 - Adder

Lambda expressions are introduced with the syntax:

```
fun(...) -> end
```

As an example we can write:

```
> Adder = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Adder(10).
#Fun
> G(5).
15
```

## 3.2 Concurrent Erlang

### Example 6 - An area Server

```
start() ->
    spawn(fun() -> loop(0) end).

loop(Tot) ->
    receive
        {Pid, {square, X}} ->
            Pid ! X*X,
            loop(Tot + X*X);
        {Pid, {rectangle,[X,Y]}} ->
            Pid ! X*Y,
            loop(Tot + X*Y);
        {Pid, areas} ->
            Pid ! Tot,
            loop(Tot)
    end.
```

This creates an "area server" - you can ask the server what the area of a square or rectangle is, or, you can ask it to return the total of all areas that it has been requested to compute.

In the above, spawn(Fun) creates a parallel process which evaluates the lambda expression Fun. Spawn returns a process identifier (Pid) which can be used to communicate with the process.

Here Pid ! M sends the message M to the process Pid. receive ... end is a pattern matching operation which receives a message. Send is an asynchronous non-blocking operation.

### Example 7 - An area client

Client code which uses the above server can be written:

```
Pid ! {self(), {square, 10}},
receive
    Area ->
        ...
end
```

### Example 8 - Global Server

In the above examples, the process identifier of the server had to made known to the client. To provide a global service we can associate a process identifier with a global name as follows:

```
...
Pid = spawn(Fun),
register(area_server, Pid),
...
area_server ! ...
```

This associates the global name area_server with Pid. Thereafter any process evaluating in the node where this name was registered can send a message to the process with the syntax area_server ! Msg

## 3.3 Distribution

### Example 9 - Spawning on a remote node

Spawn can be used to create a process of a different Erlang node.

```
...
Pid = spawn(Fun@Node)
...
alive(Node)
...
not_alive(Node)
```

The primitives alive and not_alive signify a node's willingness to participate in distributed computations. If they have not evaluated alive they remain "anonymous" and cannot take part in distributed computations.

## 3.4 Error detection

Erlang is designed for programming "robust" systems, so there are a number of primitives for trapping errors. Error recovery is not automatic. The programmer must design a fault-tolerant architecture which can be implemented using the error detection mechanisms.

### Example 10 - Catch

```
> X = 1/0.
** exited: {badarith, divide_by_zero} **
> X = (catch 1/0).
{'EXIT',{badarith, divide_by_zero}}
> b().
X = {'EXIT',{badarith, divide_by_zero}}
```

catch(Expr) converts any error occurring within Expr into a term describing the error.

### Example 11 - Catch and throw

```
case catch f(X) of
  {exception1, Why} ->
    Actions;
  NormalReturn ->
    Actions;
end,

f(X) ->
   ...
   Normal_return_value;
f(X) ->
   ...
   throw({exception1, ...}).
```

Non-local returns can be performed with `throw(Expr)`. `Expr` is evaluated and becomes the value of the enclosing `catch`.

### Example 12 - Links and trapping exits

Processes can be *linked* together. If a process dies an error message is sent to all processes to which it is linked.

```
process_flag(trap_exits, true),
P = spawn_link(Node, Fun),
receive
  {'EXIT', P, Why} ->
      Actions;
  ...
end
```

`spawn_link(Fun)` creates a parallel process which evaluates `Fun` and creates a link between the process which performs the spawn (the parent) and the newly created process (the child).

If an exception is raised in the child process then an error message is sent to all processes which the child process is linked to. In our example an error in the child process will cause an error message to be sent to the parent process. The parent process can receive the error message and take appropriate action.

### Example 13 - Undefined processes

If we call `Mod:Func(Arg1,..Argn)` and this function cannot be located in the system, then `Mod:Func` behaves as if it had been defined as follows:

```
Mod:Func(Arg1, ..., Argn) ->
  error_handler:undefined_function(Mod, Func,
                   [Arg1, .. Argn])
```

The function `undefined_function` can be defined by the user or the system default can be used. It is usually defined something like:

```
undefined_function(Mod, Func, Args) ->
 case code:ensure_loaded(Mod) of
  {module, Mod} ->
   case is_exported(Mod, Func, length(Args)) of
    true ->
     apply(Mod, Func, Args);
    false ->
     exit({undef,{Mod,Func,Args}})
   end;
  {module, Other} ->
   exit({undef,{Mod,Func,Args}});
  {interpret, Mod} ->
   int:eval(Mod, Func, Args);
  Other ->
   exit({undef,{Mod,Func,Args}})
 end.
```

## 3.5 Hot code replacement

### Example 14 - Code replacement

Erlang is designed for "non-stop" systems. We need to be able to replace code and data without stopping the system. This example shows how we can change code in a server, without stopping the server.

```
loop(Data, F) ->
    receive
        {request, Pid, Q} ->
            {Reply, Data1} = F(Q, Data),
            Pid ! Reply,
            loop(Data1, F);
        {change_code, F1} ->
            loop(Data, F1)
end
```

The above code represents a server. `Data` represent the local state of the server. `F` is a lambda expression which determines the behaviour of the server. To change the code in the server we send it the message `{change_code, F1}` The garbage collector will remove the old function.

# 4 Higher Order Processes

Most large Erlang systems are programmed by re-using code from a number of programming patterns which we call *behaviours*. This is a processes whose behaviour will be determined by a lambda expression at run-time.

As an example of a behaviour we show how a simple client-server model can be programmed. Firstly the code for the generic part of the client-server:

### Generic Client Server

```
-module(cs).

-export([start/3, rpc/2, loop/2]).

start(Name, Data, Fun) ->
  register(Name,
    spawn(fun() ->
            loop(Data, Fun)
          end)).

rpc(Name, Q) ->
  Tag = ref(),
  Name ! {query, self(), Tag, Q},
  receive
    {Tag, Reply} -> Reply
  end.

loop(Data, Fun) ->
```

```
  receive
    {query, Pid, Tag, Q} ->
      {Reply, Data1} = Fun(Q, Data),
      Pid ! {Tag, Reply},
      loop(Data1, Fun)
  end.
```

The above code provides skeleton code for a client-server. We can parameterise it by instantiating the variable Fun with a lambda expression which gives the server its desired behaviour.

For example, a Key-Value dictionary server can be made as follows:

### A key-Value Server

```
-module(kv).

start() -> cs:start(keydb, [], fun handler/2).

add(Key, Val) -> cs:rpc(keydb, {add,Key,Val}).
lookup(Key)   -> cs:rpc(keydb, {lookup,Key}).

handler({add, Key, Val}, Data) ->
    {ok, add(Key,Val,Data)};
handler({lookup, Key}, Data)   ->
    {find(Key, Data), Data}.

add(Key,Val,[{Key,_}|T]) ->
    [{Key,Val}|T];
add(Key,Val,[H|T]) ->
    [H|add(Key,Val,T)];
add(Key,Val,[]) ->
    [{Key,Val}].

find(Key,[{Key,Val}|_]) ->
    {found, Val};
find(Key,[H|T]) ->
    find(Key, T);
find(Key,[]) ->
    error.
```

In sequential programming a small number of higher order functions (map, fold, zip etc.) provide a library of functions which the programmer can re-use in a number of different contexts. Note there is a *limited* set of such functions. Too many or too abstract functions would not help.

In concurrent programming we have found that most concurrent programs can be built from a small number of higher order processes (such as the client-server) example. In a given application between 5 and 10 such higher-order processes seems to capture 95 % of the concurrency mechanisms.

Interestingly the generic part of the higher-order process is often not well-typed, but the functions which parameterise the process are.

## 5 Enter The Type System

Erlang has a dynamic type system which was inherited from Prolog. All types are checked at run-time. Nonsense expressions (for example `1 + true`) are not type-checked at compile time but result in run-time exceptions.

In 1995 Phil Wadler decided that Erlang needed a type system, and promptly informed us that he was making one. After he had started work on the project he realised that he needed some money and some help. He got his money, and we got a type system.

At the time of writing a Haskell prototype of the type checker [8] is being evaluated and the type checker itself is being re-written in Erlang.

We are evaluating the type system by type checking all the libraries in the standard Erlang distribution. This work is not complete, but we can make certain observations about the type system.

The first point to note is that the type checker is totally free-standing from the Erlang system itself. No changes have been made to Erlang to accommodate the type system.

A consequence of this is that the user is free to compile and run programs which the type checker says are not well-typed. This is in contrast to say Haskell, or ML where a type-incorrect program cannot be compiled or run.

Many programs behave correctly despite the fact they are not well-typed. This is especially true of "systems software" - for example, the programs in the kernel of the Erlang run-time which are responsible for IO and distribution perform highly complex generic operations on arbitrary data structures.

In the Erlang type system a user can declare a type. For example:

```
-type fac(int()) -> int().

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

If a type is supplied it is checked by the type checker. Types can be annotated unchecked. This means that the type checker accepts the supplied type without analysing the function to see if it has the type the programmer intended.

In practise this turns out to be very useful for forcing bizarre code through the type checker. An example is the code for the Erlang "pretty printer" The pretty printer contains code like:

```
-unchecked([pp/1]).
-type pp(1) -> deepstring().

pp(X) then tuple(X) ->
    pp_list(tuple_to_list(T));
pp(X) when list(X) ->
       ...
pp(X) when float(X) ->
       ...
```

pp is a function which turns any arbitrary data structure into a deep list of characters. We tell the system that pp is of type `1->deepstring()` where `1` is the universal type.

This is a correct statement about the program but one which cannot be inferred by the type checker.

In type checking the standard libraries we have found the following:

- The type system has uncovered no errors. The kernel libraries were written by Erlang "experts" - it seems that good programmers don't make type errors. It will be interesting to see if this remains true when we start type checking code written by less experienced programmers.

- Certain libraries (ordsets, dict etc.) passed the type checker at the first attempt. The derived types corresponded in most cases to our intuition of what the derived types should be. Adding type declarations (mostly for documentation purpose) was a trivial operation.

  Interestingly we could often remove a number of comments and replace them by more precise type declarations.

  Thus in dict.erl we replaced comments like:

```
%% fetch(Key, Dict) -> Value
%% find(Key, Dict) -> {ok,Value} | error
%% store(Key, Value, Dict) -> Dict'
```

  With type declaration:

```
-deftype dict(Key, Val) = [{Key, Val}].

-type fetch(Key, dict(Key, Val)) ->
  Val.
-type find(Key, dict(Key, Val)) ->
  ok{Val} | error.
-type store(Key, Val, dict(Key, Val)) ->
  dict(Key, Val).
```

  Which says the same thing only more precisely.

- Many modules did not type check at the first attempt. We had to re-write some functions or add `unchecked` annotation to force the code through the type checker. In complex modules about 10% of the functions needed small changes. In the vast majority of cases a small change to the function corrected the type error. In a minority of cases we had to resort to adding `unchecked` annotations. Even in the kernel system modules we were able to provide readable types for the majority of functions and not resort to the `unchecked` annotation.

  `unchecked` only had to be used when the Erlang functions contained primitives such as `apply` etc. which could not be analysed by the type checker.

- Debugging (i.e. finding the source) of the type error was in some cases extremely difficult. This proved to be very frustrating. Well-tested programs which we believed to be correct were rejected by the type checker - finding exactly what was wrong was sometimes very difficult.

## 6 The future

Where is the research leading us?

- **Very large systems**

  How can we build very large systems? The problems associated with building very large systems seem to have more to do with software architectures than choice of programming language. We have found that the use of program patterns (or higher order processes) greatly helps users structure large software systems. It has often been speculated that the advantages of a strong type system will be seen in very large software systems. We intend to test this hypothesis.

- **Evolving systems**

  As systems evolve and as we learn more we discover better ways of programming things. Early design decisions turn out to be wrong. We often wish to change some of the major system interfaces which turned out to be wrong.

  Project managers worry about "backwards compatibility" and are very reluctant to accept changes to the standard system software. Any change to the basic system invalidates their test procedures and can delay introduction of a new product. While in the short-term is is desirable to keep the system as stable as possible, in the long-term we must allow systems to evolve and phase out old code and design decisions.

  We are interested in techniques (for example, partial evaluation) that can simplify (or in the best case automate) the transition from an older to a newer version of the system.

- **Performance**

  Certain applications cannot be efficiently programmed in Erlang. We are considering adding imperative features to the language to solve these problems.

## 7 Reflections

Erlang has spread successfully from the laboratory to a number of commercial products. We can speculate as to the reasons why this has occurred. Some of the more important factors seem to be:

- **real problems**

  We work on real problems. We tend to make progress when we cannot solve a particular problem with the existing technology. Progress has often come when a user came with a problem which could not be solved in Erlang.

- **working within the organisation**

  We work *within* the Ericsson organisation. It is far easier to "sell" an idea internally than to come to the organisation from outside.

- **organisational support**

  There is a gap between the best that a laboratory with limited resources can produce and what is minimally acceptable for a commercial product. Ericsson has provided financial support and created new jobs as necessary to help fill this gap.

- **we can provide good support**

  Good documentation, courses, e-mail, hot-line telephone support etc. are essential in passing from the "enthusiast" to the "main-stream" phase of development.

- **lots of tools**

  Project managers are not interested in programming languages. They are not interested in formal anything and don't give a hoot about types or calculi.

  They are however, interested in short "time to market" and in writing bug-free software. The provision of large numbers of software tools can greatly reduce software development times and improve the quality of the software.

These tools are specific to our problem domain. Thus we have tools for making SNMP MIBs, for manipulating ASN.1 data types, for building fault-tolerant duplicated data-bases with hot-standby etc.

- **Foreign language interfaces**

  Typical systems are written in several different languages. Erlang is not good at everything. Large parts of a system might use purchased software packages written in C. Efficient integration with C is essential.

## References


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

[2] J. L. Armstrong, M. C. Williams, C. Wikström and S. R. Virding. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall (1995)

[3] J. L. Armstrong. Erlang - A survey of the language and its industrial applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*. 16 – 18 October 1996. Hino, Tokyo Japan.

[4] J. L. Armstrong, B. Däcker, S. R. Virding, and M. C. Williams, *Implementing a functional language for highly parallel real time applications*. 8th Int Conf. on Software Engineering for Telecommunication Switching Systems, Florence 30 March – 1 April 1992.

[5] B. Däcker, N. Elshiewy, P. Hedeland, C. W. Welin C. W. and M. C. Williams. Experiments with Programming Languages and Techniques for Telecommunication Applications. *Sixth International Conference on Software Engineering for Telecommunication Switching Systems*. Eindhoven, 1986.

[6] B. Hausman. Turbo Erlang: Approaching the speed of C. In *Implementations of Logic Programming Systems*, pp. 119-135, ed. Evan Tick and Giancarlo Succi, Kluwer Academic Publishers (1994).

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

[8] S. Marlow, and P. Wadler. A practical subtyping system for Erlang. In *ACM International Conference on Functional Programming*, 1997.

[9] C. Wikström and H. Nilsson. Mnesia - An Industrial DBMS with Transactions, Distribution and a Logical Query Language. *International Symposium on Cooperative Database Systems for Advanced Applications*. Kyoto Japan 1996

[10] K. Ödling. New technology for prototyping new services. In *Ericsson Review* No. 2 1993.