A History of Erlang

Joe Armstrong
Ericsson AB
joe.armstrong@ericsson.com

Abstract

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. Erlang has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

This paper describes the history of Erlang. Material for the paper comes from a number of different sources. These include personal recollections, discussions with colleagues, old newspaper articles and scanned copies of Erlang manuals, photos and computer listings and articles posted to Usenet mailing lists.

1. A History of Erlang

1.1 Introduction

Erlang was designed for writing concurrent programs that "run forever." Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language and not the operating system. Erlang has mechanisms to allow programs to change code "on the fly" so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

The initial development of Erlang took place in 1986 at the Ericsson Computer Science Laboratory (the Lab). Erlang was designed with a specific objective in mind: "*to provide a better way of programming telephony applications.*" At the time telephony applications were atypical of the kind of problems that conventional programming languages were designed to solve. Telephony applications are by their nature highly concurrent: a single switch must handle tens or hundreds of thousands of simultaneous transactions. Such transactions are intrinsically distributed and the software is expected to be highly fault-tolerant. When the software that controls telephones fails, newspapers write about it, something which does not happen when a typical desktop application fails. Telephony software must also be changed "on the fly," that is, without loss of service occurring in the application as code upgrade operations occur. Telephony software must also operate in the "soft real-time" domain, with stringent timing requirements for some operations, but with a more relaxed view of timing for other classes of operation.

When Erlang started in 1986, requirements for virtually zero down-time and for in-service upgrade were limited to rather small and obscure problem domains. The rise in popularity of the Internet and the need for non-interrupted availability of services has extended the class of problems that Erlang can solve. For example, building a non-stop web server, with dynamic code upgrade, handling millions of requests per day is very similar to building the software to control a telephone exchange. So similar, that Erlang and its environment provide a very attractive set of tools and libraries for building non-stop interactive distributed services.

From the start, Erlang was designed as a practical tool for getting the job done—this job being to program basic telephony services on a small telephone exchange in the Lab. Programming this exchange drove the development of the language. Often new features were introduced specifically to solve a particular problem that I encountered when programming the exchange. Language features that were not used were removed. This was such a rapid process that many of the additions and removals from the language were never recorded. Appendix A gives some idea of the rate at which changes were made to the language. Today, things are much more difficult to change; even the smallest of changes must be discussed for months and millions of lines of code re-tested after each change is made to the system.

This history is pieced together from a number of different sources. Much of the broad details of the history are well documented in the thesis *Concurrent Functional Programming for Telecommunications: A Case Study for Technology Introduction* [12], written by the head of the Computer Science Lab, Bjarne Däcker. This thesis describes the developments at the lab from 1984 to 2000, and I have taken several lengthy quotes from the thesis. In 1994, Bjarne also wrote a more light-hearted paper [11] to celebrate the tenth anniversary of the lab. Both these papers have much useful information on dates, times and places that otherwise would have been forgotten.

Many of the original documents describing Erlang were lost years ago, but fortunately some have survived and parts of them are reproduced here. Many things we take for granted today were not self-evident to us twenty years ago, so I have tried to expose the flow of ideas in the order they occurred and from the perspective of the time at which they occurred. For comparison, I will also give a modern interpretation or explanation of the language feature or concept involved.

This history spans a twenty-year period during which a large number of people have contributed to the Erlang system. I have done my best to record accurately who did what and when. This is not an easy task since often the people concerned didn't write any comments in their programs or otherwise record what they were doing, so I hope I haven't missed anybody out.

DOI 10.1145/1238844.1238850
http://doi.acm.org/10.1145/1238844.1238850

2. Part I : Pre-1985. Conception

2.1 Setting the scene

Erlang started in the Computer Science Laboratory at Ericsson Telecom AB in 1986. In 1988, the Lab moved to a different company called Ellemtel in a south-west suburb of Stockholm. Ellemtel was jointly owned by LM Ericsson and Televerket, the Swedish PTT, and was the primary centre for the development of new switching systems.

Ellemtel was also the place where Ericsson and Televerket had jointly developed the AXE telephone exchange. The AXE, which was first produced in 1974 and programmed in PLEX, was a second-generation SPC[1] exchange which by the mid-1980s generated a large proportion of Ericsson's profits.

Developing a new telephone exchange is a complex process which involves the efforts of hundreds, even thousands of engineers. Ellemtel was a huge melting pot of ideas that was supposed to come up with new brilliant products that would repeat the success of the AXE and generate profits for Ericsson and Televerket in the years to come.

Initially, when the computer science lab moved to Ellemtel in 1988 it was to become part of a exciting new project to make a new switch called AXE-N. For various reasons, this plan never really worked out, so the Lab worked on a number of projects that were not directly related to AXE-N. Throughout our time at Ellemtel, there was a feeling of competition between the various projects as they vied with each other on different technical solutions to what was essentially the same problem. The competition between the two groups almost certainly stimulated the development of Erlang, but it also led to a number of technical "wars"—and the consequences of these wars was probably to slow the acceptance of Erlang in other parts of the company.

The earliest motivation for Erlang was "*to make something like PLEX, to run on ordinary hardware, only better.*"

Erlang was heavily influenced by PLEX and the AXE design. The AXE in turn was influenced by the earlier AKE exchange. PLEX is a programming language developed by Göran Hemdahl at Ericsson that was used to program AXE exchanges. The PLEX language was intimately related to the AXE hardware, and cannot be sensibly used for applications that do not run on AXE exchanges. PLEX had a number of language features that corrected shortcomings in the earlier AKE exchange.

In particular, some of the properties of the AXE/PLEX system were viewed as mandatory. Firstly, it should be possible to change code "*on the fly*;" in other words, code change operations should be possible without stopping the system. The AKE was plagued with "pointer" problems. The AKE system manipulated large numbers of telephone calls in parallel. The memory requirements for each call were variable and memory was allocated using linked lists and pointer manipulation. This led to many errors. The design of the AXE and PLEX used a mixture of hardware protection and data copying that eliminated the use of pointers and corrected many of the errors in the AKE. This in its turn was the inspiration of the process and garbage-collected memory strategy used in Erlang.

At this stage it might be helpful to describe some of the characteristics of the concurrency problems encountered in programming a modern switching system. A switching system is made from a number of individual switches. Individual switches typically handle tens to hundreds of thousands of simultaneous calls. The switching system must be capable of handling millions of calls and must tolerate the failure of individual switches, providing uninterrupted services to the user. Usually the hardware and software is divided into two planes called the control and media planes. The control plane is responsible for controlling the calls, the media plane is responsible for data transmission. When we talk in loose terms about "telephony software" we mean the control-plane software.

Typically, the software for call control is modeled using finite state machines that undergo state transitions in response to protocol messages. From the software point of view, the system behaves as a very large collection of parallel processes. At any point in time, most of the processes are waiting for an event caused by the reception of a message or the triggering of a timer. When an event occurs, the process does a small amount of computation, changes state, possibly sends messages to other processes and then waits for the next event. The amount of computation involved is very small.

Any switching system must therefore handle hundreds of thousands of very lightweight processes where each process performs very little computation. In addition, software errors in one process should not be able to crash the system or damage any other processes in the system. One problem to be solved in any system having very large numbers of processes is how to protect the processes from memory corruption problems. In a language with pointers, processes are protected from each other using memory-management hardware and the granularity of the page tables sets a lower limit to the memory size of a process. Erlang has no pointers and uses a garbage collectible memory, which means that it is impossible for any process to corrupt the memory of another process. It also means that the memory requirements for an individual process can be very small and that all memory for all processes can be in the same address space without needing memory protection hardware.

It is important to note that in the AXE/PLEX system and in Erlang, explicit processes are part of the programming language itself and not part of the underlying operating system. There is a sense in which both Erlang and PLEX do not need most of the services of the underlying operating system since the language itself provides both memory management and protection between parallel processes. Other operating system services, like resource allocation and device drivers needed to access the hardware, can easily be written in C and dynamically linked into the Erlang runtime system.

At the time Erlang was first implemented, the view that processes were part of the language rather than the operating system was not widely held (even today, it is a minority view). The only languages having this view of the world that we were aware of at that time were Ada (with tasks), EriPascal (an Ericsson dialect of Pascal with concurrent processes), Chill (with processes), PLEX (with its own special form of processes implemented in hardware) and Euclid. When I first started working at Ericsson, I was introduced to the Ericsson software culture by Mike Williams, who also worked in the Lab. Mike had worked with concurrent systems for many years, mostly in PL163, and it was he who hammered into my brain the notion that three properties of a programming language were central to the efficient operation of a concurrent language or operating system. These were: 1) the time to create a process. 2) the time to perform a context switch between two different processes and 3) the time to copy a message between two processes. The performance of any highly-concurrent system is dominated by these three times.

The final key idea inherited from the AXE/PLEX culture was that the failure of a process or of hardware should only influence the immediate transaction involved and that all other operations in the machine should progress as if no failures had occurred. An immediate consequence of this on the Erlang design was to forbid dangling pointers between different processes. Message passing had to be implemented by copying message buffers between the memory spaces of the different processes involved and not by passing point-

[1] Stored program control—an ancient telecoms term meaning "computer controlled."

1	Handling a very large number of concurrent activities
2	Actions to be performed at a certain point of time or within a certain time
3	Systems distributed over several computers
4	Interaction with hardware
5	Very large software systems
6	Complex functionality such as feature interaction
7	Continuous operation over several years
8	Software maintenance (reconfiguration, etc.) without stopping the system
9	Stringent quality and reliability requirements
10	Fault tolerance both to hardware failures and software errors

Table 1. Requirements of a programming language for telecommunication switching systems (from [12]).

ers to a common memory pool. At an early stage we rejected any ideas of sharing resources between processes because of the difficulties of error handling. In many circumstances, error recovery is impossible if part of the data needed to perform the error recovery is located on a remote machine and if that remote machine has crashed. To avoid this situation and to simplify the process, we decided that all processes must always have enough local information to carry on running if something fails in another part of the system. Programming with mutexes and shared resources was just too difficult to get right in a distributed system when errors occurred.

In cases where consistency is required in distributed systems, we do not encourage the programmer to use the low-level Erlang language primitives but rather the library modules written in Erlang. The Erlang libraries provide components for building distributed systems. Such components include mnesia, which is a distributed real-time transaction database, and various libraries for things like leadership election and transaction memories. Our approach was always not to hard-wire mechanisms into the language, but rather to provide language primitives with which one could construct the desired mechanisms. An example is the remote procedure call. There is no remote procedure call primitive in Erlang, but a remote procedure call can be easily made from the Erlang send and receive primitives.

We rapidly adopted a philosophy of message passing by copying and no sharing of data resources. Again, this was counter to the mood of the time, where threads and shared resources protected by semaphores were the dominant way of programming concurrent systems. Robert Virding and I always argued strongly against this method of programming. I clearly remember attending several conferences on distributed programming techniques where Robert and I would take turns at asking the question "What happens if one of the nodes crashes?" The usual answer was that "the system won't work" or "our model assumes that there are no failures." Since we were interested in building highly reliable distributed systems that should never stop, these were not very good answers.

In order to make systems reliable, we have to accept the extra cost of copying data between processes and always making sure that the processes have enough data to continue by themselves if other processes crash.

2.2 Requirements, requirements, requirements ...

The AXE/PLEX heritage provided a set of requirements that any new programming language for programming telecommunications applications must have, as shown in Table 1.

These requirements were pretty typical. Existing systems solved these problems in a number of ways, sometimes in the programming language, sometimes in the operating systems and sometimes in application libraries. The goal of the Lab was to find better ways of programming telecoms systems subject to such requirements.

Our method of solving these problems was to program POTS[2] over and over again in a large number of different programming languages and compare the results. This was done in a project called SPOTS, (SPOTS stood for SPC for POTS). Later the project changed name to DOTS (Distributed SPOTS) and then to LOTS, "Lots of DOTS." The results of the SPOTS project were published in [10].

2.3 SPOTS, DOTS, LOTS

In 1985, when I joined the Lab, SPOTS had finished and DOTS was starting. I asked my boss Bjarne Däcker what I should do. He just said "*solve Ericsson's software problem.*" This seemed to me at the time a quite reasonable request—though I now realise that it was far more difficult than I had imagined. My first job was to join the ongoing DOTS experiment.

Our lab was fortunate in possessing a telephone exchange (an Ericsson MD110 PABX) that Per Hedeland had modified so that it could be controlled from a VAX11/750. We were also lucky in being the first group of people in the company to get our hands on a UNIX operating system, which we ran on the VAX. What we were supposed to do was "to find better ways of programming telephony" (a laudable aim for the members of the computer science lab of a telecommunications company). This we interpreted rather liberally as "program basic telephony in every language that will run on our Unix system and compare the results." This gave us ample opportunities to a) learn new programming languages, b) play with Unix and c) make the phones ring.

In our experiments, we programmed POTS in a number of different languages, the only requirement being that the language had to run on 4.2 BSD UNIX on the Vax 11/750. The languages tried in the SPOTS project were Ada, Concurrent Euclid, PFL, LPL0, Frames and CLU.

Much of what took place in the POTS project set the scene for what would become Erlang, so it is interesting to recall some of the conclusions from the SPOTS paper. This paper did not come down heavily in favour of any particular style of programming, though it did have certain preferences:

- "small languages" were thought desirable:

  "*Large languages present many problems (in implementation, training etc) and if a small language can describe the application succinctly it will be preferable.*"

- Functional programming was liked, but with the comment:

  "The absence of variables which are updated means that the exchange database has to be passed around as function arguments which is a bit awkward."

- Logic programming was best in terms of elegance:

  "Logic programming and the rule-based system gave the most unusual new approach to the problem with elegant solutions to some aspects of the problem."

- Concurrency was viewed as essential for this type of problem, but:

  "Adding concurrency to declarative languages, rule-based systems and the object based system is an open field for research."

At this time, our experience with declarative languages was limited to PFL and LPL0, both developed in Sweden. PFL [17] came from the Programming Methodology Group at Chalmers Technical University in Gothenburg and was a version of ML extended with

[2] Plain Ordinary Telephone Service.

primitives borrowed from CCS. LPL0 [28] came from the Swedish Institute of Computer Science and was a logic language based on Haridi's natural deduction [15].

Looking back at the SPOTS paper, it is interesting to note what we *weren't* interested in—there is no mention in the paper of dealing with failure or providing fault-tolerance, there is no mention of changing the system as it is running or of how to make systems that scale dynamically. My own contribution to LOTS was to program POTS. This I did first in Smalltalk and then in Prolog. This was fairly sensible at the time, since I liberally interpreted Bjarne's directive to *"solve all of Ericsson's software problems"* as *"program POTS in Smalltalk."*

3. Part II: 1985 – 1988. The birth of Erlang

3.1 Early experiments

My first attempts to make the phones ring was programmed in Smalltalk. I made a model with phone objects and an exchange object. If I sent a ring message to a phone it was supposed to ring. If the phone A went off-hook it was supposed to send an (offHook, A) message to the exchange. If the user of phone A dialled some digit D, then a (digit, A, D) message would be sent to the exchange.

Alongside the Smalltalk implementation, I developed a simple graphic notation that could be used to describe basic telephony. The notation describing telephony was then hand-translated into Smalltalk. By now the lab had acquired a SUN workstation with Smalltalk on it. But the Smalltalk was very slow—so slow that I used to take a coffee break while it was garbage collecting. To speed things up, in 1986 we ordered a Tektronix Smalltalk machine, but it had a long delivery time. While waiting for it to be delivered, I continued fiddling with my telephony notation. One day I happened to show Roger Skagervall my algebra—his response was "but that's a Prolog program." I didn't know what he meant, but he sat me down in front of his Prolog system and rapidly turned my little system of equations into a running Prolog program. I was amazed. This was, although I didn't know it at the time, the first step towards Erlang.

My graphic notation could now be expressed in Prolog syntax and I wrote a report [1] about it. The algebra had predicates:

```
idle(N)   means the subscriber N is idle
on(N)     means subscribed N in on hook
...
```

And operators:

```
+t(A, dial_tone) means add a dial tone to A
```

Finally rules:

```
process(A, f) :- on(A), idle(A), +t(A,dial-tone),
                 +d(A, []), -idle(A), +of(A)
```

This had the following declarative reading:

```
process(A, f)     To process an off hook signal from
                  a subscriber A
:-                then
on(A)             If subscriber A is on-hook
,                 and
idle(A)           If subscriber A is idle
,                 and
+t(A, dial_tone)  send a dial tone to A
,                 and
+d(A, [])         set the set of dialled digits to []
,                 and
-idle(A)          retract the idle state
,                 and
+of(A)            assert that we are off hook
```

Using this notation, POTS could be described using fifteen rules. There was just one major problem: the notation only described how one telephone call should proceed. How could we do this for thousands of simultaneous calls?

3.2 Erlang conceived

Time passed and my Smalltalk machine was delivered, but by the time it arrived I was no longer interested in Smalltalk. I had discovered Prolog and had found out how to write a meta-interpreter in Prolog. This meta-interpreter was easy to change so I soon figured out how to add parallel processes to Prolog. Then I could run several versions of my little telephony algebra in parallel.

The standard way of describing Prolog in itself is to use a simple meta-interpreter:

```
solve((A,B)) :- solve(A), solve(B).
solve(A)     :- builtin(A), call(A).
solve(A,B)   :- rule(A, B), solve(B).
```

The problem with this meta-interpreter is that the set of remaining goals that is not yet solved is not available for program manipulation. What we would like is a way to explicitly manage the set of remaining goals so that we could suspend or resume the computation at any time.

To see how this works, we can imagine a set of equations like this:

```
x ->  a,b,c
a ->  p,{q},r
r ->  g,h
p ->  {z}
```

This notation means that the symbol `x` is to be replaced by the sequence of symbols `a`, `b` and `c`. That `a` is to be replaced by `p`, `{q}` and `r`. Symbols enclosed in curly brackets are considered primitives that cannot be further reduced.

To compute the value of the symbol `x` we first create a stack of symbols. Our reduction machine works by successively replacing the top of the stack by its definition, or if it is a primitive by evaluating the primitive.

To reduce the initial goal `x` we proceed as follows:

```
x               replace x by its definition
a,b,c           replace a by its definition
p,{q},r,b,c     replace p by its definition
{z},{q},r,b,c   evaluate z
{q},r,b,c       evaluate q
r,b,c           replace r by its definition
g,h,b,c         ...
...
```

The point of the reduction cycle is that at any time we can suspend the computation. So, for example, after three iterations, the state of the computation is represented by a stack containing:

```
{z},{q},r,b,c
```

If we want several parallel reduction engines, we arrange to save and store the states of each reduction engine after some fixed number of reductions. If we now express our equations as Prolog terms:

```
eqn(x,[a,b,c]).
eqn(a,[p,{q},r]).
eqn(r,[g,h]).
eqn(p,[{z}]).
```

Then we can describe our reduction engine with a predicate reduce as follows:

```
reduce([]).
reduce([{H}|T]) :-
    call(H),!,
    reduce(T).
reduce([Lhs|More]) :-
    eqn(Lhs, Rhs),
    append(Rhs,More,More1),!,
    reduce(More1).
```

With a few more clauses, we can arrange to count the number of reduction steps we have made and to save the list of pending goals for later evaluation. This is exactly how the first Erlang interpreter worked. The interested reader can consult [4] for more details.

Time passed and my small interpreter grew more and more features, the choice of which was driven by a number of forces. First, I was using the emerging language to program a small telephone exchange, so problems naturally arose when I found that interacting with the exchange was clumsy or even impossible. Second, changes suggested themselves as we thought up more beautiful ways of doing things. Many of the language changes were motivated by purely aesthetic concerns of simplicity and generality.

I wanted to support not only simple concurrent processes, but also mechanisms for sending message between processes, and mechanism for handling errors, etc. My interpreter grew and some of the other lab members became interested in what I was doing. What started as an experiment in *"adding concurrency to Prolog"* became more of a language in its own right and this language acquired a name "Erlang," which was probably coined by Bjarne Däcker. What did the name Erlang mean? Some said it meant "Ericsson Language," while others claimed it was named after Agner Krarup Erlang (1878 – 1929), while we deliberately encouraged this ambiguity.

While I had been fiddling with my meta-interpreter, Robert Virding had been implementing variants of parallel logic programming languages. One day Robert came to me and said he'd been looking at my interpreter and was thinking about making a few small minor changes, did I mind? Now Robert is incapable of making small changes to anything, so pretty soon we had two different Erlang implementations, both in Prolog. We would take turns in rewriting each other's code and improving the efficiency of the implementation.

As we developed the language, we also developed a philosophy around the language, ways of thinking about the world and ways of explaining to our colleagues what we were doing. Today we call this philosophy *Concurrency-Oriented Programming*. At the time our philosophy had no particular name, but was more just a set of rules explaining how we did things.

One of the earliest ways of explaining what Erlang was all about was to present it as a kind of hybrid language between concurrent languages and functional languages. We made a poster showing this which was reproduced in [12] and shown here in Figure 1.

3.3 Bollmora, ACS/Dunder

By 1987, Erlang was regarded as a new programming language that we had prototyped in Prolog. Although it was implemented in Prolog, Erlang's error-handling and concurrency semantics differed significantly from Prolog. There were now two people (Robert Virding and I) working on the implementation and it was ready to be tried out on external users. By the end of the year, Mike Williams managed to find a group of users willing to try the language on a real problem, a group at *Ericsson Business Communications AB*, which was based in Bollmora. The group was headed by Kerstin Ödling and the other members of the team were Åke Rosberg,

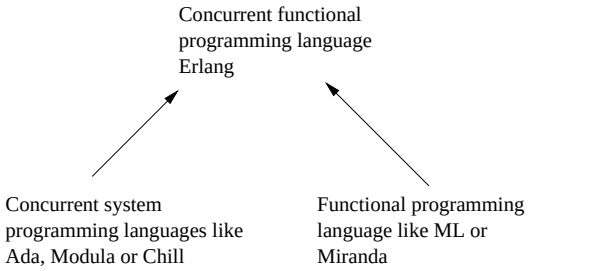


Figure 1. Relation of Erlang to existing languages.

Håkan Karlsson and Håkan Larsson. These were the first ever Erlang users.

The team wanted to prototype a new software architecture called ACS[3] designed for programming telephony services on the Ericsson MD110 PABX[4] and were looking for a suitable language for the project, which is how they got to hear about Erlang. A project called ACS/Dunder was started to build the prototype.

The fact that somebody was actually interested in what we were doing came as a great stimulus to the development and we entered a period of rapid development where we could actually try out our language ideas on real users.

3.4 Frenzied activity

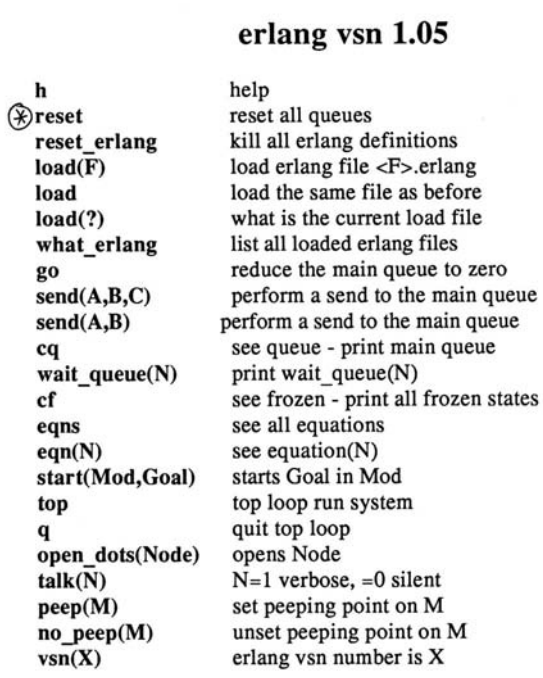
Erlang began to change rapidly. We now had two people working on the implementation (Robert and myself) and a large user community (three people). We would add features to the language and then try them out on our users. If the users or implementors liked the changes, they stayed in. If the users disliked the changes or if the implementation was ugly, the changes were removed. Amazingly, the fact that the language was changing under their feet almost every day didn't particularly bother our users. We met our Bollmora users once or twice a week for about six months. We taught them programming, they taught us telephony and both sides learned a lot.

Hardly anything remains from this period—most of the day-to-day changes to the language were not recorded and there is no lasting evidence of what those changes were. But fortunately, a few documents do remain: Figure 2 shows the entire Erlang 1.05 manual and Appendix A contains the comments at the start of the file erlang.pro (which was the main program for the Prolog interpreter). This is a change log documenting the changes made to the language in the nine-month period from 24 March 1988 to 14 December 1988. Like a child, Erlang took about nine months to develop. By the end of 1988, most of the ideas in Erlang had stabilized. While Robert and I implemented the system, the ideas behind the mechanism that we implemented often came from our users, or from the other members of the Lab. I always considered the morning coffee break to be the key forum where the brilliant ideas you had on the way to work were trashed and where all the real work was done. It was in these daily brainstormings that many a good idea was created. It's also why nobody can quite remember who thought of what, since everybody involved in the discussions seems to remember that it was *they* who had the key idea.

It was during this period that most of the main ideas in Erlang emerged. In the following sections I will describe Erlang as it was in 1988. Appendix B has a set of examples illustrating the language as it is today.

[3] Audio Communication System.

[4] Private Automatic Branch Exchange.



erlang vsn 1.05

h	help
reset	reset all queues
reset_erlang	kill all erlang definitions
load(F)	load erlang file <F>.erlang
load	load the same file as before
load(?)	what is the current load file
what_erlang	list all loaded erlang files
go	reduce the main queue to zero
send(A,B,C)	perform a send to the main queue
send(A,B)	perform a send to the main queue
cq	see queue - print main queue
wait_queue(N)	print wait_queue(N)
cf	see frozen - print all frozen states
eqns	see all equations
eqn(N)	see equation(N)
start(Mod,Goal)	starts Goal in Mod
top	top loop run system
q	quit top loop
open_dots(Node)	opens Node
talk(N)	N=1 verbose, =0 silent
peep(M)	set peeping point on M
no_peep(M)	unset peeping point on M
vsn(X)	erlang vsn number is X

Figure 2. The Erlang 1.05 manual.

3.5 Buffered message reception

One of the earliest design decisions in Erlang was to use a form of buffering selective receive. The syntax for message reception in modern Erlang follows a basic pattern:

```
receive
    Pattern1 ->
        Actions1;
    Pattern2 ->
        Actions2;
   ...
end
```

This means wait for a message. If the message matches `Pattern1` then evaluate the code `Actions1`. If the message matches `Pattern2` then evaluate the code `Actions2`, etc.

But what happens if some other message that matches neither `Pattern1` or `Pattern2` arrives at the processes? Answer—ignore the message and queue it for later. The message is put in a "save queue" and processed later and the `receive` statement carries on waiting for the messages it is interested in.

The motivation for automatically buffering out-of-order messages came from two observations. Firstly we observed that in the CCITT Specification and Description Language[5] (SDL), one of the most commonly used specification idioms involved queuing and later replaying out-of-order messages. Secondly we observed that handling out-of-order messages in a finite state machine led to an explosion in the state space of the finite state machine. This happens more often than you think, in particular when processing remote procedure calls. Most of the telecommunications programs we write deal with message-oriented protocols. In implementing the protocols we have to handle the messages contained in the protocol itself together with a large number of messages that are not part of the protocol but come from remote procedure calls made internally in the system. Our queuing strategy allows us to make an internal remote procedure call within the system and block until that call has returned. Any messages arriving during the remote procedure call are merely queued and served after the remote procedure call has completed. The alternative would be to allow the possibility of handling protocol messages in the middle of a remote procedure call, something which greatly increases the complexity of the code.

This was a great improvement over PLEX, where every message must be handled when it arrives. If a messages arrives "too early" the program has to save it for later. Later, when it expects the message, it has to check to see if the message has already arrived.

This mechanism is also extremely useful for programming sets of parallel processes when you don't know in which order the message between the processes will arrive. Suppose you want to send three messages M1, M2 and M3 to three different processes and receive replies R1, R2 and R3 from these three processes. The problem is that the reply messages can arrive in any order. Using our receive statement, we could write:

```
A ! M1,
B ! M2,
C ! M3,
  receive
    A ? R1 ->
      receive
        B ? R2 ->
          receive
            C ? R3 ->
               ... now R1 R2 and R3
                   have been received ...
```

Here `A!M` means send the message `M` to the process `A` and `A?X` means receive the message `X` from the process `A`.

It doesn't matter now in which order the messages are received. The code is written as if `A` replies first—but if the message from `B` replies first, the message will be queued and the system will carry on waiting for a message from `A`. Without using a buffer, there are six different orderings of the message to be accounted for.

3.6 Error handling

Error handling in Erlang is very different from error handling in conventional programming languages. The key observation here is to note that the error-handling mechanisms were designed for building fault-tolerant systems, and not merely for protecting from program exceptions. You cannot build a fault-tolerant system if you only have one computer. The minimal configuration for a fault-tolerant system has two computers. These must be configured so that both observe each other. If one of the computers crashes, then the other computer must take over whatever the first computer was doing.

This means that the model for error handling is based on the idea of two computers that observe each other. Error detection and recovery is performed on the remote computer and not on the local computer. This is because in the worst possible case, the computer where the error has occurred has crashed and thus no further computations can be performed on the crashed computer.

In designing Erlang, we wanted to abstract all hardware as reactive objects. Objects should have *"process semantics;"* in other words, as far as the software was concerned, the only way to interact with hardware was through message passing. When you send a message to a process, there should be no way of knowing if the process was really some hardware device or just another software process. The reason for this was that in order to simplify our programming model, we wanted to model everything as processes and

[5] SDL is widely used in the telecoms industry to specify communications protocols.

we wanted to communicate with all processes in a uniform manner. From this point of view we wanted software errors to be handled in exactly the same manner as hardware errors. So, for example, if a process died because of a divide by zero it would propagate an `{'EXIT',Pid,divideByZero}` signal to all the processes in its link set. If it died because of a hardware error it might propagate an `{'EXIT',Pid,machineFailure}` signal to its neighbors. From a programmer's point of view, there would no difference in how these signals were handled.

3.7 Links

Links in Erlang are provided to control error propagation paths for errors between processes. An Erlang process will die if it evaluates illegal code, so, for example, if a process tries to divide by zero it will die. The basic model of error handling is to assume that some other process in the system will observe the death of the process and take appropriate corrective actions. But which process in the system should do this? If there are several thousand processes in the system then how do we know which process to inform when an error occurs? The answer is the *linked* process. If some process `A` evaluates the primitive `link(B)` then it becomes linked to `A` . If `A` dies then `B` is informed. If `B` dies then `A` is informed.

Using links, we can create sets of processes that are linked together. If these are normal[6] processes, they will die immediately if they are linked to a process that dies with an error. The idea here is to create sets of processes such that if any process in the set dies, then they will all die. This mechanism provides the invariant that either all the processes in the set are alive or none of them are. This is very useful for programming error-recovery strategies in complex situations. As far as I know, no other programming language has anything remotely like this.

The idea of links and of the mechanism by which all processes in a set die was due to Mike Williams. Mike's idea was inspired by the design of the release mechanism used in old analogue telephones and exchanges. In the analogue telephones and in the early electromechanical exchanges, three wires called A, B and C were connected to the phones. The C wire went back to the exchange and through all the electromechanical relays involved in setting up a call. If anything went wrong, or if either partner terminated the call, then the C wire was grounded. Grounding the C wire caused a knock-on effect in the exchange that freed all resources connected to the C line.

3.8 Buffers

Buffers and the `receive` statement fit together in a rather non-obvious way. Messages sent between Erlang processes are always delivered as soon as possible. Each process has a "mailbox" where all incoming messages are stored. When a message arrives it is put in the mailbox and the process is scheduled for execution. When the process is next scheduled it tries to pattern match the message. If the match succeeds, message reception occurs and the message is removed from the mailbox and the data from the message works its way into the program. Otherwise the message is put into a "save" queue. When any message matches, the entire save queue is merged back into the mailbox. All buffering that takes place is then performed implicitly in either the mailbox or the save queue. In about 1988, this mechanism was the subject of intense debate. I remember something like a four-day meeting being devoted to the single topic of how interprocess communication should actually work.

The outcome of this meeting was that we all thought that processes should communicate through pipes and that the pipes should be first-class objects. They should have infinite[7] buffering capacity, they should be named and it should be possible to connect and disconnect them to and from processes. It should be possible to bend them and split them and join them, and I even created an algebra of pipes.

Then I tried to implement the pipe algebra but this turned out to be very difficult. In particular, pipe joining and coalescing[8] operations were terribly difficult to program. The implementation needed two types of message in the pipes: regular messages and small tracer messages that had to be sent up and down the pipes to check that they were empty. Sometimes the pipes had to be locked for short periods of time and what would happen if the processes at the end of the pipes failed was extremely difficult to work out. The problems all stem from the fact that the pipes introduce dependencies between the end points so that the processes at either end are no longer independent—which makes life difficult.

After two weeks of programming, I declared that the pipe mechanism now worked. The next day I threw it all away—the complexity of the implementation convinced me that the mechanism was wrong. I then implemented a point-to-point communication mechanism with mailboxes; this took a couple of hours to implement and suffered from none of the kind of problems that plagued the pipes implementation. This seems to be a rather common pattern: first I spend a week implementing something, then, when is is complete I throw everything away and reimplement something slightly different in a very short time.

At this point, pipes were rejected and mailboxes accepted.

3.9 Compilation to Strand

While the Prolog implementations of Erlang were being used to develop the language and experiment with new features, another avenue of research opened, work aimed at creating an efficient implementation of Erlang. Robert had not only implemented Erlang in Prolog but had also been experimenting with a Parlog compiler of his own invention. Since Erlang was a concurrent language based on Prolog it seemed natural to study how a number of concurrent logical programming languages had been implemented. We started looking at languages like Parlog, KL/1 and Strand for possible inspiration. This actually turned out to be a mistake. The problem here has to do with the nature of the concurrency. In the concurrent logic programming languages, concurrency is implicit and extremely fine-grained. By comparison Erlang has explicit concurrency (via processes) and the processes are coarse-grained. The study of Strand and Parlog led to an informal collaboration with Keith Clarke and Ian Foster at Imperial College London.

Eventually, in 1988, we decided to experiment with cross compilation of Erlang to Strand. Here we made a major error—we confidently told everybody the results of the experiment before we had done it. Cross compilation to Strand would speed up Erlang by some embarrassingly large factor. The results were lacklustre; the system was about five times faster but very unstable and large programs with large numbers of processes just would not run. The problem turned out to be that Strand was just too parallel and created far too many parallel processes. Even though we might only have had a few hundred Erlang processes, several tens of millions of parallel operations were scheduled within the Strand system.

Strand also had a completely different model of error handling and message passing. The Erlang-to-Strand compiler turned an Erlang function of arity N to a Strand process definition of arity N+8. The eight additional arguments were needed to implement Erlang's

[6] Process are either normal or system processes. System process can trap and process non-normal exit signals. Normal process just die.

[7] In theory.

[8] For example, if pipe X has endpoints A and B, and pipe Y has endpoints C and D, then coalescing X and Y was performed with an operation muff_pipes(B, C).

error-handling and message-passing semantics. More details of this can be found in chapter 13 of [14]. The problem with the implementation boiled down to the semantic mismatch between the concurrency models used in Strand and Erlang, which are completely different.

4. Part III: 1989 – 1997. Erlang grows

The eight years from 1989 to 1997 were the main period during which Erlang underwent a period of organic growth. At the start of the period, Erlang was a two-man project, by the end hundreds of people were involved.

The development that occurred during this period was a mixture of frantic activity and long periods where nothing appeared to happen. Sometimes ideas and changes to the language came quickly and there were bursts of frenetic activity. At other times, things seem to stagnate and there was no visible progress for months or even years. The focus of interest shifted in an unpredictable manner from technology to projects, to training, to starting companies. The dates of particular events relating to the organization, like, for example, the start of a particular project or the formation of Erlang Systems AB, are wellknown. But when a particular idea or feature was introduced into the language is not so wellknown. This has to do with the nature of software development. Often a new software feature starts as a vague idea in the back of implementors' minds, and the exact date when they had the idea is undocumented. They might work on the idea for a while, then run into a problem and stop working on it. The idea can live in the back of their brain for several years and then suddenly pop out with all the details worked out.

The remainder of this section documents in roughly chronological order the events and projects that shaped the Erlang development.

4.1 The ACS/Dunder results

In December 1989, the ACS/Dunder project produced its final report. For commercial reasons, this report was never made public. The report was authored by the members of the prototyping team who had made the ACS/Dunder prototype, in which about 25 telephony features were implemented. These features represented about one tenth of the total functionality of the MD 110. They were chosen to be representative of the kind of features found in the MD110, so they included both hard and extremely simple functions. The report compared the effort (in man hours) of developing these features in Erlang with the predicted effort of developing the same features in PLEX. The ACS/Dunder report found that the time to implement the feature in Erlang divided by the time to implement the feature in PLEX (measured in man hours) was a factor of 3 to 25, depending upon the feature concerned. The average increase in productivity was a factor of 8.

This factor and the conclusion of the report were highly controversial and many theories were advanced to explain away the results. It seemed at the time that people disliked the idea that the effect could be due to having a better programming language, preferring to believe that it was due to some "smart programmer effect." Eventually we downgraded the factor to a mere 3 because is sounded more credible than 8. The factor 3 was totally arbitrary, chosen to be sufficiently high to be impressive and sufficiently low to be believable. In any case, it was significantly greater than one, no matter how you measured and no matter how you explained the facts away.

The report had another conclusion, namely that Erlang was far too slow for product development. In order to use Erlang to make a real product, it would need to be at least 40 times faster. The fact that it was too slow came from a comparison of the execution times of the Erlang and PLEX programs. At this stage, CPU performance represented the only significant problem. Memory performance was not a problem. The run-time memory requirements were modest and the total size of the compiled code did not pose any problems.

After a lot of arguing, this report eventually led the way to the next stage of development, though the start of the project was to be delayed for a couple of years. Ericsson decided to build a product called the *Mobility Server* based on the ACS/Dunder architecture, and we to started work on a more efficient implementation of Erlang.

4.2 The word starts spreading

1989 also provided us with one of our first opportunities to present Erlang to the world outside Ericsson. This was when we presented a paper at the SETSS conference in Bournemouth. This conference was interesting not so much for the paper but for the discussions we had in the meetings and for the contacts we made with people from Bellcore. It was during this conference that we realised that the work we were doing on Erlang was very different from a lot of mainstream work in telecommunications programming. Our major concern at the time was with detecting and recovering from errors. I remember Mike, Robert and I having great fun asking the same question over and over again: *"what happens if it fails?"*—the answer we got was almost always a variant on *"our model assumes no failures."* We seemed to be the only people in the world designing a system that could recover from software failures.

It was about this time that we realized very clearly that shared data structures in a distributed system have terrible properties in the presence of failures. If a data structure is shared by two physical nodes and if one node fails, then failure recovery is often impossible. The reason why Erlang shares no data structures and uses pure copying message passing is to sidestep all the nasty problems of figuring out what to replicate and how to cope with failures in a distributed system. At the Bournemouth conference everybody told us we were wrong and that data must be shared for efficiency—but we left the conference feeling happy that the rest of the world was wrong and that we were right. After all, better a slow system that can recover from errors than a fast system that cannot handle failures. Where people were concerned with failure, it was to protect themselves from *hardware failures*, which they could do by replicating the hardware. In our world, we were worried by *software failures* where replication does not help.

In Bournemouth, we met a group of researchers headed by Gary Herman from Bellcore who were interested in what we were doing. Later in the year, in December 1989, this resulted in Bjarne, Mike, Robert and me visiting Bellcore, where we gave our first ever external Erlang lecture. Erlang was well received by the researchers at Bellcore, and we were soon involved in discussing how they could get a copy of the Erlang system. When we got back from Bellcore, we started planning how to release Erlang. This took a while since company lawyers were involved, but by the middle of 1990 we delivered a copy of Erlang to Bellcore. So now we had our first external user, John Unger from Bellcore.

4.3 Efficiency needed – the JAM

Following the ACS/Dunder report in December 1989, we started work on an efficient version of Erlang. Our goal was to make a version of Erlang that was at least 40 times faster than the prototype.

At this point we were stuck. We knew what we wanted to do but not how to do it. Our experiments with Strand and Parlog had led to a deadend. Since we were familiar with Prolog, the next step appeared to follow the design of an efficient Prolog machine. Something like the WAM [29] seemed the natural way to proceed. There were two problems with this. First, the WAM didn't support concurrency and the kind of error handling that we were interested

in, and second, we couldn't understand how the WAM worked. We read all the papers but the explanations seemed to be written only for people who already understood how the thing worked.

The breakthrough came in early 1990. Robert Virding had collected a large number of the descriptions of abstract machines for implementing parallel logic machines. One weekend I borrowed his file and took all the papers home. I started reading them, which was pretty quick since I didn't really understand them, then suddenly after I'd read through them all I began to see the similarities. A clue here, and hint there, yes they were all the same. Different on the surface, but very similar underneath. I understood—then I read them again, this time slowly. What nobody had bothered to mention, presumably because it was self-evident, was that each of the instructions in one of these abstract machines simultaneously manipulated several registers and pointers. The papers often didn't mention the stack and heap pointers and all the things that got changed when an instruction was evaluated because this was obvious.

Then I came across a book by David Maier and David Scott Warren [19] that confirmed this; now I could understand how the WAM worked. Having understood this, it was time to design my own machine, the JAM.[9] Several details were missing from the Warren paper and from other papers describing various abstract machines. How was the code represented? How was the compiler written? Once you understood them, the papers seems to describe the easy parts; the details of the implementation appeared more to be "trade secrets" and were not described.

Several additional sources influenced the final design. These included the following:

- *A portable Prolog Compiler* [9], that described how virtual machine instructions were evaluated.
- *A Lisp machine with very compact programs* [13], that described a Lisp machine with an extremely compact representation.
- *BrouHaHa – A portable Smalltalk interpreter* [22], that described some smart tricks for speeding up the dispatching instructions in a threaded interpreter.

When I designed the JAM, I was worried about the expected size of the resulting object for the programs. Telephony control programs were huge, numbering tens of millions of lines of source code. The object code must therefore be highly compact, otherwise the entire program would never fit into memory. When I designed the JAM, I sat down with my notepad, invented different instruction sets, then wrote down how some simple functions could be compiled into these instruction sets. I worked out how the instructions would be represented in a byte-coded machine and then how many instructions would be evaluated in a top-level evaluation of the function. Then I changed the instruction set and tried again. My benchmark was always for the "append" function and I remember the winning instruction set compiled append into 19 bytes of memory. Once I had decided on an instruction set, compilation to the JAM was pretty easy and an early version of the JAM is described in [5]. Figure 3 shows how the factorial function was compiled to JAM code.

Being able to design our own virtual machines resulted in highly compact code for the things we cared most about. We wanted message passing to be efficient, which was easy. The JAM was a stack-based machine so to compile `A ! B`, the compiler emitted code to compile `A` , then code to compile `B` , then a single byte send instruction. To compile `spawn(Function)`, the compiler emitted code to build a function closure on the stack, followed by a single byte spawn instruction.

[9] Modesty prevents me from revealing what this stands for.

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

{info, fac, 1}
 {try_me_else, label1}
        {arg, 0}
        {getInt, 0}
        {pushInt, 1}
        ret
 label1: try_me_else_fail
        {arg, 0}
        dup
        {pushInt, 1}
        minus
        {callLocal, fac, 1}
        times
        ret
```

Figure 3. Compilation of sequential Erlang to JAM code.

File	Lines	Purpose
sys_sys.erl	18	dummy
sys_parse.erl	783	erlang parser
sys_ari_parser.erl	147	parse arithmetic expressions
sys_build.erl	272	build function call arguments
sys_match.erl	253	match function head arguments
sys_compile.erl	708	compiler main program
sys_lists.erl	85	list handling
sys_dictionary.erl	82	dictionary handler
sys_utils.erl	71	utilities
sys_asm.erl	419	assembler
sys_tokenise.erl	413	tokeniser
sys_parser_tools.erl	96	parser utilities
sys_load.erl	326	loader
sys_opcodes.erl	128	opcode definitions
sys_pp.erl	418	pretty printer
sys_scan.erl	252	scanner
sys_boot.erl	59	bootstrap
sys_kernel.erl	9	kernel calls
18 files	4544	

Table 2. Statistics from an early Erlang compiler.

The compiler was, of course, written in Erlang and run through the Prolog Erlang emulator. To test the abstract machine I wrote an emulator, this time in Prolog so I could now test the compiler by getting it to compile itself. It was not fast—it ran at about 4 Erlang reductions[10] per second, but it was fast enough to test itself. Compiling the compiler took a long time and could only be done twice a day, either at lunch or overnight.

The compiler itself was a rather small and simple program. It was small because most of the primitives in Erlang could be compiled into a single opcode in the virtual machine. So all the compiler had to do was to generate code for efficient pattern matching and for building and reconstructing terms. Most of the complexity is in the run-time system, which implements the opcodes of the virtual machine. The earliest compiler I have that has survived is the erl89 compiler, which had 18 modules containing 2544 lines of code. The modules in the compiler were as in Table 2.

It was now time to implement the JAM virtual machine emulator, this time not in Prolog but in C. This is where Mike Williams

[10] One reduction corresponds to a single function call.

came in. I started writing the emulator myself in C but soon Mike interfered and started making rude comments about my code. I hadn't written much C before and my idea of writing C was to close my eyes and pretend it was FORTRAN. Mike soon took over the emulator, threw away all my code and started again. Now the Erlang implementor group had expanded to three, Mike, Robert and myself. Mike wrote the inner loop of the emulator very carefully, since he cared about the efficiency of the critical opcodes used for concurrent operations. He would compile the emulator, then stare at the generated assembler code, then change the code compile again, and stare at the code until he was happy. I remember him working for several days to get message sending just right. When the generated code got down to six instructions he gave up.

Mike's emulator soon worked. We measured how fast it was. After some initial tweaking, it ran 70 times faster than the original Prolog emulator. We we delighted—we had passed our goal of 40 by a clear margin. Now we could make some real products.

Meanwhile, the news from the Bollmora group was not good: *"We miscalculated, our factor of 40 was wrong, it needs to be 280 times faster."*

One of the reviewers of this paper asked whether memory efficiency was ever a problem. At this stage the answer was no. CPU efficiency was always a problem, but never memory efficiency.

4.4 Language changes

Now that we have come to 1990 and have a reasonably fast Erlang, our attention turned to other areas. Efficiency was still a problem, but spurred by our first success we weren't particularly worried about this. One of the things that happened when writing Erlang in Erlang was that we had to write our own parser. Before we had just used infix operators in Prolog. At this point the language acquired its own syntax and this in its turn caused the language to change. In particular, `receive` was changed.

Having its own syntax marked a significant change in the language. The new version of Erlang behaved pretty much like the old Prolog interpreter, but somehow it *felt* different. Also our understanding of the system deepened as we grappled with tricky implementation issues that Prolog had shielded us from. In the Prolog system, for example, we did not have to bother about garbage collection, but in our new Erlang engine we had to implement a garbage collector from scratch.

Since our applications ran in the so-called soft real-time domain, the performance of the garbage collector was crucial, so we had to design and implement garbage-collection strategies that would not pause the system for too long. We wanted frequent small garbage collections rather than infrequent garbage collections that take a long time.

The final strategy we adopted after experimenting with many different strategies was to use per-process stop-and-copy GC. The idea was that if we have many thousands of small processes then the time taken to garbage collect any individual process will be small. This strategy also encouraged copying all the data involved in message passing between processes, so as to leave no dangling pointers between processes that would complicate garbage collection. An additional benefit of this, which we didn't realise at the time, was that copying data between processes increases process isolation, increases concurrency and simplifies the construction of distributed systems. It wasn't until we ran Erlang on multicore CPUs that the full benefit of non-shared memory became apparent. On a multicore CPU, message passing is extremely quick and the lack of locking between CPUs allows each CPU to run without waiting for the other CPUs.

Our approach to GC seemed a little bit reckless: would this method work in practice? We were concerned about a number of problems. Would large numbers of processes decide to garbage collect all at the same time? Would programmers be able to structure their applications using many small processes, or would they use one large process? If they did use one large process, what would happen when it performed a garbage collection? Would the system stop? In practice our fears were unfounded. Process garbage collections seem to occur at random and programmers very rarely use a single large process to do everything. Current systems run with tens to hundreds of thousands of processes and it seems that when you have such large numbers of processes, the effects of GC in an individual process are insignificant.

Erlang in 1988

```
# wait_first_digit(A) ->
    receive 10 {
        A ? digit(D) =>
            stop_tone(A),
            received_digit(A,[],D);
        A ? on_hook =>
            stop_tone(A),
            idle(A);
        timeout =>
            stop_tone(A),
            wait_clear(A);
        Other =>
            wait_first_digit(A)
    }.
```

Erlang today

```
wait_first_digit(A) ->
    receive
        {A, {digit, D}} ->
            stop_tone(A),
            received_digit(A,[],D);
        {A, on_hook} ->
            stop_tone(A),
            idle(A);
        Other ->
            wait_first_digit(A)
        after 10 ->
            stop_tone(A),
            wait_clear(A)
        end.
```

Figure 4. Erlang in 1988 and today.

4.5 How receive changed and why

The early syntax of Erlang came straight from Prolog. Erlang was implemented directly in Prolog using a careful choice of infix operators. Figure 4 adapted from [2] shows a section of a telephony program from 1988 and the corresponding program as it would be written today. Notice there are two main changes:

First, in the 1988 example, patterns were represented by Prolog terms, thus `digit(D)` represents a pattern. In modern Erlang, the same syntax represents a function call and the pattern is written `{digit,D}`.

The second change has to do with how message reception patterns were written. The syntax:

```
Proc ! Message
```

means send a message, while:

```
receive {
  Proc1 ? Mess1 =>
        Actions1;
```

```
rpc(Pid, Query) ->
        Pid ! {self(), Query},
        receive
         {Pid, Reply} ->
                Reply
        end.
```

Client code

```
server(Data) ->
        receive
          {From, Query} ->
                {Reply,Data1} = F(Query,Data),
                From ! {self(), Reply},
                server(Data1)
        end.
```

Server Code

Figure 5. Client and server code for a remote procedure call.

```
        Proc2 ? Mess2 =>
              Actions2;
              ...
      }
```

means try to receive a message `Mess1` from `Proc1`, in which case perform `Actions1`; otherwise try to receive `Mess2` from `Proc2`, etc. The syntax `Proc?Message` seemed at first the obvious choice to denote message reception and was mainly chosen for reasons of symmetry. After all if `A!B` means send a message then surely `A?B` should mean receive a message.

The problem with this is that there is no way to hide the identity of the sender of a message. If a process `A` sends a message to `B`, then receiving the message with a pattern of the form `P?M` where `P` and `M` are unbound variables always results in the identity of `A` being bound to `P`. Later we decided that this was a bad idea and that the sender should be able to choose whether or not it reveals its identity to the process that it sends a message to. Thus if `A` wants to send a message `M` to a process `B` and reveal its identity, we could write the code which sends a message as `B!{self(),M}` or, if it did not wish to reveal its identity, we could write simply `B!M`. The choice is not automatic but is decided by the programmer.

This leads to the common programming idiom for writing a remote procedure call whereby the sender must alway include its own Pid[11] (the Pid to reply to) in the message sent to a server. The way we do this today is shown in Figure 9.

Note that we can also use the fact that processes do not reveal their identity to write secure code and to fake the identity of a process. If a message is sent to a process and that message contains no information about the sender, then there is no way the receiver of the message can know from whom the message was sent. This can be used as the basis for writing secure code in Erlang.

Finally, faked message Pids can be used for delegation of responsibilities. For example, much of the code in the IO subsystem is written with `{Request,ReplyTo,ReplyAs}` messages, where `Request` is a term requesting some kind of IO service. `ReplyTo` and `ReplyAs` are Pids. When the final process to perform the operation has finished its job, it sends a message by evaluating `ReplyTo!{ReplyAs,Result}`. If this is then used in code in the RPC programming idiom in Figure 5, the code essentially fakes the Pid of the responding process.

Now the point of all this argument, which might seem rather obscure, is that a seemingly insignificant change to the surface syntax, i.e. breaking the symmetry between `A!B` and `A?B`, has profound consequences on security and how we program. And it also explains the hour-long discussions over the exact placement of commas and more importantly what they mean.[12]

4.6 Years pass ...

The next change was the addition of distribution to the language. Distribution was always planned but never implemented. It seemed to us that adding distribution to the language would be easy since all we had to do was add message passing to remote processes and then everything should work as before.

At this time, we were only interested in connecting conventional sequential computers with no shared memory. Our idea was to connect stock hardware through TCP/IP sockets and run a cluster of machines behind a corporate firewall. We were not interested in security since we imagined all our computers running on a private network with no external access. This architecture led to a form of all-or-nothing security that makes distributed Erlang suitable for programming cluster applications running on a private network, but unsuitable for running distributed applications where various degrees of trust are involved.

1990

In 1990 Claes (Klacke) Wikström joined the Lab—Klacke had been working in another group at Ellemtel and once he became curious about what we were doing we couldn't keep him away. Klacke joined and the Erlang group expanded to four.

ISS'90

One of the high points of 1990 was ISS'90 (International Switching Symposium), held in Stockholm. ISS'90 was the first occasion where we actively tried to market Erlang. We produced a load of brochures and hired a stall at the trade fair and ran round-the clock demonstrations of the Erlang system. Marketing material from this period is shown in Figures 6 and 7.

At this time, our goal was to try and spread Erlang to a number of companies in the telecoms sector. This was viewed as strategically important—management had the view that if we worked together with our competitors on research problems of mutual interest, this would lead to successful commercial alliances. Ericsson never really had the goal of making large amounts of money by selling Erlang and did not have an organisation to support this goal, but it was interested in maintaining a high technical profile and interacting with like-minded engineers in other companies.

1991

In 1991, Klacke started work on adding distribution to Erlang, something that had been waiting to be done for a long time. By now, Erlang had spread to 30 sites. The mechanisms for this spread are unclear, but mostly it seems to have been by word-of-mouth. Often we would get letters requesting information about the system and had no idea where they had heard about it. One likely mechanism was through the usenet mailing lists where we often posted to comp.lang.functional. Once we had established a precedent of releasing the system to Bellcore, getting the system to subsequent users was much easier. We just repeated what we'd done for Bellcore. Eventually after we had released the system to a dozen or so users, our managers and lawyers got fed up with our pestering and let us release the system to whomever we felt like, provided they signed a non-disclosure agreement.

[11] Process Identifier.

[12] All language designers are doubtless familiar with the phenomenon that users will happily discuss syntax for hours but mysteriously disappear when the language designer wants to talk about what the new syntax actually means.

Figure 6. Early internal marketing – the relationship between Erlang and PLEX.

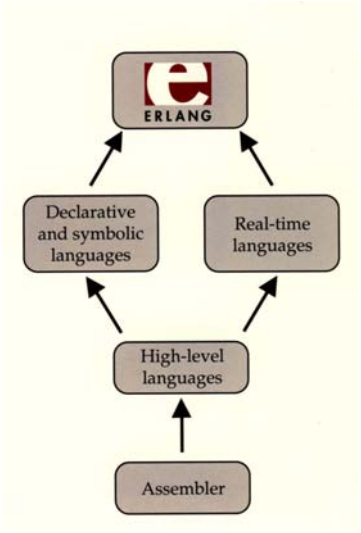


Figure 7. Erlang marketing – the relation of Erlang to other languages.

Also, Ericsson Business Communications ported Erlang to the FORCE computer and real-time OS VxWorks; these were our first steps towards an embedded system. This port was driven by product requirements since the mobility server at the time ran on VxWorks. We also ported Erlang to virtually all operating systems we had access to. The purpose of these ports was to make the language accessible to a large set of users and also to improve the quality of the Erlang system itself. Every time we ported the system to a new OS we found new bugs that emerged in mysterious ways, so porting to a large number of different OSs significantly improved the quality of the run-time system itself.

1992

In 1992, we got permission to publish a book and it was decided to commercialize Erlang. A contract was signed with Prentice Hall and the first Erlang book appeared in the bookshops in May 1993.

Even this decision required no small measure of management persuasion—this was definitely not how Ericsson had done things in the past; earlier languages like PLEX had been clothed in secrecy. Management's first reaction at the time was "if we've done something good, we should keep quiet about it", quite the opposite to today's reaction to the open-source movement.

The decision to publish a book about Erlang marked a change in attitude inside Ericsson. The last language developed by Ericsson for programming switches was PLEX. PLEX was proprietary: very few people outside Ericsson knew anything about PLEX and there were no PLEX courses in the universities and no external market for PLEX programs or programmers. This situation had advantages and disadvantages. The major advantage was that PLEX gave Ericsson a commercial advantage over its competitors, who were presumed to have inferior technologies. The disadvantages had to do with isolation. Because nobody else used PLEX, Ericsson had to maintain everything to do with PLEX: write the compilers, hold courses, everything.

AT&T, however, had taken a different approach with C and C++. Here, the burden of supporting these languages was shared by an entire community and isolation was avoided. The decision to publish an Erlang book and to be fairly open about what we did was therefore to avoid isolation and follow the AT&T/C path rather than the Ericsson/PLEX path. Also in 1992 we ported Erlang to MS-DOS windows, the Mac, QNX and VxWorks.

The Mobility Server project, which was based upon the successful ACS/Dunder study, was started about two years after the ACS/Dunder project finished. Exactly why the mobility server project lost momentum is unclear. But this is often the way it happens: periods of rapid growth are followed by unexplained periods when nothing much seems to happen. I suspect that these are the consolidation periods. During rapid growth, corners get cut and things are not done properly. In the periods between the growth, the system gets polished. The bad bits of code are removed and reworked. On the surface not much is happening, but under the surface the system is being re-engineered.

1993

In May, the Erlang book was published.

4.7 Turbo Erlang

In 1993, the Turbo Erlang system started working. Turbo Erlang was the creation of Bogumil (Bogdan) Hausman who joined the Lab from SICS.[13] For bizarre legal reasons the name Turbo Erlang was changed to BEAM.[14] The BEAM compiler compiled Erlang programs to BEAM instructions.

The BEAM instructions could either be macro expanded into C and subsequently compiled or transformed into instructions for a 32-bit threaded code interpreter. BEAM programs compiled to C ran about ten times faster than JAM interpreted programs, and the BEAM interpreted code ran more than three times faster than the JAM programs.

Unfortunately, BEAM-to-C compiled programs increased code volumes, so it was not possible to completely compile large applications into C code. For a while we resolved this by recommending a hybrid approach. A small number of performance-critical modules would be compiled to C code while others would be interpreted.

The BEAM instruction set used fixed-length 32-bit instructions and a threaded interpreter, as compared to the JAM, which had variable length instructions and was a byte-coded interpreter. The threaded BEAM code interpreter was much more efficient than the JAM interpreter and did not suffer from the code expansion that compilation to C involved. Eventually the BEAM instruction set and threaded interpreter was adopted and the JAM phased out.

[13] Swedish Institute of Computer Science.

[14] Bogdan's Erlang Abstract Machine.

I should say a little more about code size here. One of the main problems in many products was the sheer volume of object code. Telephone switches have millions of lines of code. The current software for the AXD301, for example, has a couple of millions lines of Erlang and large amounts of sourced C code. In the mid-'90s when these products were developed, on-board memory sizes were around 256 Mbytes. Today we have Gbyte memories so the problem of object code volume has virtually disappeared, but it was a significant concern in the mid-'90s. Concerns about object-code memory size lay behind many of the decisions made in the JAM and BEAM instruction sets and compilers.

4.8 Distributed Erlang

The other major technical event in 1993 involved Distributed Erlang. Distribution has always been planned but we never had time to implement it. At the time all our products ran on single processors and so there was no pressing need to implement distribution. Distribution was added as part of our series of ongoing experiments, and wasn't until 1995 and the AXD301 that distribution was used in a product.

In adding distribution, Klacke hit upon several novel implementation tricks. One particularly worthy of mention was the atom communication cache described in [30], which worked as follows:

Imagine we have a distributed system where nodes on different hosts wish to send Erlang atoms to each other. Ideally we could imagine some global hash table, and instead of sending the textual representation of the atom, we would just send the atom's hashed value. Unfortunately, keeping such a hash table consistent is a hard problem. Instead, we maintain two synchronised hash tables, each containing 256 entries. To send an atom we hashed the atom to a single byte and then looked this up in the local hash table. If the value was in the hash table, then all we needed to do was to send to the remote machine a single byte hash index, so sending an atom between machines involved sending a single byte. If the value was not in the hash table, we invalidated the value in the cache and sent the textual representation of the atom.

Using this strategy, Klacke found that 45% of all objects sent between nodes in a distributed Erlang system were atoms and that the hit rate in the atom cache was around 95%. This simple trick makes Erlang remote procedure calls run somewhat faster for complex data structures than, for example, the SunOS RPC mechanism.

In building distributed Erlang, we now had to consider problems like dropped messages and remote failures. Erlang does not guarantee that messages are delivered but it does provide weaker guarantees on message ordering and on failure notification.

The Erlang view of the world is that message passing is unreliable, so sending a message provides no guarantee that it will be received. Even if the message were to be received, there is no guarantee that the message will be acted upon as you intended. We therefore take the view that if you want confirmation that a message has been received, then the receiver must send a reply message and you will have to wait for this message. If you don't get this reply message, you won't know what happened. In addition to this there is a link mechanism, which allows one process to link to another. The purpose of the link is to provide an error monitoring mechanism. If a process that you are linked to dies, you will be sent an error signal that can be converted to an error message.

If you are linked to a process and send a stream of messages to that process, it is valid to assume that no messages will be dropped, that the messages are not corrupted and that the messages will arrive at that process in the order they were sent or that an error has occurred and you will be sent an error signal. All of this presumes that TCP/IP is itself reliable, so if you believe that TCP/IP is reliable then message passing between linked processes is reliable.

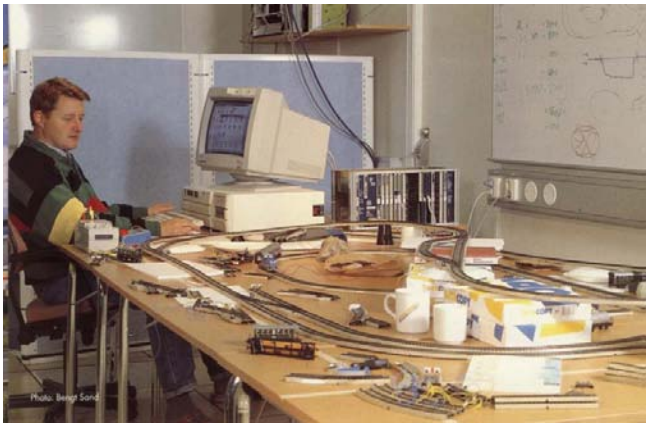


Figure 8. Robert Virding hard at work in the lab (1993).

4.9 Spreading Erlang

In April 1993, a new company called *Erlang Systems AB* was formed, which was owned by *Ericsson Programatic*. The goal was to market and sell Erlang to external companies. In addition, Erlang Systems was to take over responsibility for training and consulting and the production of high-quality documentation.

Erlang Systems also provided the main source of employment for the "Uppsala boys". These were former computer science students from the University of Uppsala who had completed their Master's thesis studies with an Erlang project. Many of these students started their careers in Erlang Systems and were subsequently hired out to Ericsson projects as internal consultants. This proved a valuable way of "kick-starting" a project with young and enthusiastic graduate students who were skilled in Erlang.

Another memorable event of 1993 was the Erlang display at the trade fair held in October in Stockholm. The main demonstrator at the display was a program which simultaneously controlled a small telephone exchange[15] and a model train. Figure 8 shows Robert Virding hard at work in the Lab programming the model train. To the immediate right of the computer behind the train set is the MD100 LIM. Following the trade fair, for several years, we used the train set for programming exercises in Erlang courses, until the points wore out through excessive use. While the control software was fault-tolerant, the hardware was far less reliable and we were plagued with small mechanical problems.

4.10 The collapse of AXE-N

In December 1995, a large project at Ellemtel, called AXE-N, collapsed. This was the single most important event in the history of Erlang. Without the collapse of AXE-N, Erlang would have still remained a Lab experiment and the effort to turn it into a commercial-quality product would not have happened. The difference is the many thousands of hours of work that must be done to produce high-quality documentation and to produce and test extensive libraries. AXE-N was a project aimed at developing a new generation of switching products ultimately to replace the AXE10 system. The AXE-N project had developed a new hardware platform and system software that was developed in C++.

Following a series of crisis meetings the project was reorganised and re-started. This time the programming language would be

[15] MD110 LIM (Line Interface Module).

Erlang and hardware from the AXE-N project was salvaged to start the production of a new ATM[16] switch, to be called the AXD. This new project was to be the largest-ever Erlang project so far, with over 60 Erlang programmers. At the start of the AXD project, the entire Erlang system was the responsibility of half a dozen people in the Lab. This number was viewed as inadequate to support the needs of a large development project and so plans were immediately enacted to build a product unit, called OTP, to officially support the Erlang system. At this time all external marketing of Erlang was stopped, since all available "resources" should now focus on internal product development.

OTP stands for the "Open Telecom Platform" and is both the name of the Erlang software distribution and the name of an Ericsson product unit, which can be somewhat confusing. The OTP unit started in the Lab but in 1997 was formed as a new product unit outside the Lab. Since 1997, the OTP unit has been responsible for the distribution of Erlang.

4.11 BOS – OTP and behaviors

Alongside the Erlang language development, the question of libraries and middleware has always been important. Erlang is just a programming language, and to build a complete system something more than just a programming language is needed. To build any significant body of software you need not only a programming language but a significant set of libraries and some kind of operating system to run everything on. You also need a philosophy of programming—since you cannot build a large body of software in an ad-hoc manner without some guiding principles.

The collective name for Erlang, all the Erlang libraries, the Erlang run-time system and descriptions of the Erlang way of doing things is the OTP system. The OTP system contains:

- Libraries of Erlang code.
- Design patterns for building common applications.
- Documentation.
- Courses.
- How to's.

The libraries are organised and described in a conventional manner. They also have pretty conventional semantics. One reviewer of this paper asked how we integrated side effects with our language, for example what happens if a open file handle is sent in a message to two different processes. The answer is that side effects like this are allowed. Erlang is not a strict side-effect-free functional language but a concurrent language where what happens inside a process is described by a simple functional language. If two different processes receive a Pid representing a file, both are free to send messages to the file process in any way they like. It is up to the logic of the application to prevent this from happening.

Some processes are programmed so that they only accept messages from a particular process (which we call the owning process). In this case problems due to sharing a reference can be avoided, but code libraries do not necessarily have to follow such a convention.

In practice this type of problem rarely presents problems. Most programmers are aware of the problems that would arise from shared access to a resource and therefore use mnesia transactions or functions in the OTP libraries if they need shared access to a resource.

What is more interesting is the set of design patterns included in the OTP system. These design patterns (called behaviors) are the result of many years' experience in building fault-tolerant systems. They are typically used to build things like client-server models, event-handling systems etc. Behaviors in Erlang can be thought of as parameterizable higher-order parallel processes. They represent an extension of conventional higher-order functions (like `map` ,`fold` etc) into a concurrent domain.

[16] Asynchronous Transfer Mode.

```
-module(server).
-export([start/2, call/2, change_code/2]).

start(Fun, Data) ->
  spawn(fun() -> server(Fun, Data) end).

call(Server, Args) ->
  rpc(Server, {query, Args}}

change_code(Server, NewFunction) ->
  rpc(Server, {new_code, NewFunction}).

rpc(Server, Query) ->
  Server ! {self(), Query},
  receive
         {Server, Reply} -> Reply
  end.

server(Fun, Data) ->
  receive
        {From, {query, Query}} ->
          {Reply, NewData} = Fun(Query, Data),
          From ! {self(), Reply},
          server(Fun, NewData);
        {from, {swap_code, NewFunction} ->
          From ! {self(), ack},
          server(Data, NewFunction)
  end.
```

Figure 9. A generic client-server model with hot-code replacement.

The design of the OTP behaviors was heavily influenced by two earlier efforts. The first was a system called BOS.[17] BOS was an application operating system written at Bollmora in Erlang specifically for the Mobility Server project. The BOS had solved a number of problems in a generic manner, in particular how to build a generic server and how to build a generic kind of error supervisor. Most of this had been done by Peter Högfeldt. The second source of inspiration was a generic server implemented by Klacke.

When the OTP project started, I was responsible for the overall technology in the project and for developing a new set of behaviors that could be used for building fault-tolerant systems. This in turn led to the development of a dozen or so behaviors, all of which simplify the process of building a fault-tolerant system. The behaviors abstract out things like failure so that client-server models can be written using simple functional code in such a manner that the programmer need only be concerned with the functionality of the server and not what will happen in the event of failure or distribution. This part of the problem is handled by the generic component of the behavior.

The other two people who were heavily involved in the development of the behaviors were Martin Björklund and Magnus Fröberg. Unfortunately space limitations preclude a more extensive treatment of behaviors. Figure 9 has a greatly simplified sketch of a client-server behavior. Note that this model provides the normal functionality of a client-server and the ability to hot-swap the code. The rationale behind behaviors and a complete set of examples can be found in [7].

[17] Basic Operating System.

4.12 More language changes

During the period 1989 to 1998, i.e. from the existence of a stable JAM-based system and up to the release of Open Source Erlang, a number of changes crept into the language. These can be categorised as major or minor changes. In this context, minor means that the change could be categorised as a simple incremental improvement to the language. Minor changes suggest themselves all the time and are gradually added to the language without much discussion. Minor changes make the programmer's life a lot easier, but do not affect how we think about the programming process itself. They are often derived from ideas in conventional programming languages that are assimilated into Erlang.

The minor changes which were added included:

- Records.
- Macros.
- Include files.
- Infix notation for list append and subtract ("++" and "–").

The major changes are covered in the following sections.

4.13 Influence from functional programming

By now the influence of functional programming on Erlang was clear. What started as the addition of concurrency to a logic language ended with us removing virtually all traces of Prolog from the language and adding many well-known features from functional languages.

Higher-order functions and list comprehensions were added to the language. The only remaining signs of the Prolog heritage lie in the syntax for atoms and variables, the scoping rules for variables and the dynamic type system.

4.14 Binaries and the bit syntax

A binary is a chunk of untyped data, a simple memory buffer with no internal structure. Binaries are essential for storing untyped data such as the contents of a file or of a data packet in a data communication protocol. Typical operations on a binary often include dividing it into two, according to some criteria, or combining several small binaries to form a larger binary. Often binaries are passed unmodified between processes and are used to carry input/output data. Handling binary data efficiently is an extremely difficult problem and one which Klacke and Tony Rogvall spent several years implementing.

Internally binaries are represented in several different ways, depending upon how and when they were created and what has happened to them since their creation. Sending messages containing binaries between two processes in the same node does not involve any copying of the binaries, since binaries are kept in a separate reference-counted storage area that is not part of the stack and heap memory which each process has.

The bit syntax [27] is one of those unplanned things that was added in response to a common programming problem. Klacke and Tony had spent a long time implementing various low-level communication protocols in Erlang. In so doing, the problem of packing and unpacking bit fields in binary data occurred over and over again. To unpack or pack such a data structure, Tony and Klacke invented the bit syntax and enhanced Erlang pattern matching to express patterns over bit fields.

As an example, suppose we have a sixteen-bit data structure representing a ten-bit counter, three one-bit flags and a three-bit status indicator. The Erlang code to unpack such a structure is:

```
<<N:10,Flag1:1,Flag2:1,Flag3:1,Status:3>> = B
```

This is one of those simple ideas which after you have seen it makes you wonder how any language could be without it. Using the bit syntax yields highly optimised code that is extremely easy to write and fits beautifully with the way most low-level protocols are specified.

4.15 Mnesia ETS tables and databases

In developing large-scale telecommunications applications it soon became apparent that the "pure" approach of storing data could not cope with the demands of a large project and that some kind of real-time database was needed. This realization resulted in a DBMS called Mnesia[18] [24, 25, 21]. This work was started by Klacke but soon involved Hans Nilsson, Törbjörn Törnkvist, Håkan Matsson and Tony Rogvall. Mnesia had both high- and low-level components. At the highest level of abstract was a new query language called Mnemosyne (developed by Hans Nilsson) and at the lowest level were a set of primitives in Erlang with which Mnesia could be written. Mnesia satisfied the following requirements (from [21]):

1. Fast Key/Value lookup.
2. Complicated non real-time queries, mainly for operation and maintenance.
3. Distributed data due to distributed applications.
4. High fault tolerance.
5. Dynamic reconfiguration.
6. Complex objects.

In order to implement Mnesia in Erlang, one additional Erlang module had to be developed. This was the Erlang module `ets` , short for Erlang term storage. Ets provided low-level destructive term storage based on extensible hash tables. Although `ets` looks as if it had been implemented in Erlang (i.e. it is an Erlang module), most of its implementation is contained in the Erlang virtual machine implementation.

4.16 High-performance Erlang

The HiPE (High-Performance Erlang) project is a research project at the Department of Information Technology at the University of Uppsala. The HiPE team have concentrated on efficient implementation of Erlang and type checking systems for Erlang. This project runs in close collaboration with members of the OTP group. Since 2001, the HiPE native code compiler has been an integral part of the Open Source Erlang distribution.

4.17 Type inference of Erlang programs

Erlang started life as a Prolog interpreter and has always had a dynamic type system, and for a long time various heroic attempts have been made to add a type system to Erlang. Adding a type system to Erlang seems at first a moderately difficult endeavour, which on reflection becomes impossibly difficult.

The first attempt at a type system was due to an initiative taken by Phil Wadler. One day Phil phoned me up and announced that a) Erlang needed a type system, b) he had written a small prototype of a type system and c) he had a one year's sabbatical and was going to write a type system for Erlang and "were we interested?" Answer —"Yes."

Phil Wadler and Simon Marlow worked on a type system for over a year and the results were published in [20]. The results of the project were somewhat disappointing. To start with, only a subset of the language was type-checkable, the major omission being the lack of process types and of type checking inter-process messages. Although their type system was never put into production,

[18] The original name was Amnesia until a senior Ericsson manager noticed the name. "It can't possible be called Amnesia," he said, "the name must be changed" — and so we dropped the "a."

it did result in a notation for types which is still in use today for informally annotating types.

Several other projects to type check Erlang also failed to produce results that could be put into production. It was not until the advent of the Dialyzer[19] [18] that realistic type analysis of Erlang programs became possible. The Dialyzer came about as a side-effect of the HiPE project mentioned earlier. In order to efficiently compile Erlang, a type analysis of Erlang programs is performed. If one has precise information about the type of a function, specialised code can be emitted to compile that function, otherwise generic code is produced. The HiPE team took the view that complete information about all the types of all the variables in all the statements of an Erlang program was unnecessary and that any definite statements about types, even of a very small subsection of a program, provided useful information that could guide the compiler into generating more efficient code.

The Dialyzer does not attempt to infer all types in a program, but any types it does infer are guaranteed to be correct, and in particular any type errors it finds are guaranteed to be errors. The Dialyzer is now regularly used to check large amounts of production code.

5. Part IV: 1998 – 2001. Puberty problems — the turbulent years

1998 was an exciting year in which the following events occurred:

- The first demo of GPRS[20] developed in Erlang was demonstrated at the GSM World Congress in February and at CeBIT in March.
- In February, Erlang was banned inside Ericsson Radio Systems.
- In March, the AXD301 was announced. This was possibly the largest ever program in a functional language.
- In December, Open Source Erlang was released.
- In December, most of the group that created Erlang resigned from Ericsson and started a new company called *Bluetail AB*.

5.1 Projects Succeed

In 1998, the first prototype of a GPRS system was demonstrated and the Ericsson AXD301 was announced. Both these systems were written in a mixture of languages, but the main language for control in both systems was Erlang.

The largest ever system built in Erlang was the AXD301. At the time of writing, this system has 2.6 millions lines of Erlang code. The success of this project demonstrates that Erlang is suitable for large-scale industrial software projects. Not only is the system large in terms of code volume, it is also highly reliable and runs in realtime. Code changes in the system have to be performed without stopping the system. In the space available it is difficult to describe this system adequately so I shall only give a brief description of some of the characteristics.

The AXD301 is written using distributed Erlang. It runs on a cluster using pairs of processors and is scalable up to 16 pairs of processors. Each pair is "self contained," which means that if one processor in the pair fails, the other takes over. The take-over mechanisms and call control are all programmed in Erlang. Configuration data and call control data are stored in a Mnesia database that can be accessed from any node and is replicated on several nodes. Individual nodes can be taken out of service for repair, and additional nodes can be added without interrupting services.

[19] DIscrepancy AnaLYZer of ERlang programs.

[20] General Packet Radio Service.

The software for the system is programmed using the behaviors from the OTP libraries. At the highest level of abstraction are a number of so-called "supervision trees"—the job of a node in the supervision tree is to monitor its children and restart them in the event of failure. The nodes in a decision tree are either supervision trees or primitive OTP behaviors. The primitive behaviors are used to model client-servers, event-loggers and finite-state machines. In the analysis of the AXD reported in [7], the AXD used 20 supervision trees, 122 client-server models, 36 event loggers and 10 finite-state machines.

All of this was programmed by a team of 60 programmers. The vast majority of these programmers had an industrial background and no prior knowledge of functional or concurrent programming languages. Most of them were taught Erlang by the author and his colleagues. During this project the OTP group actively supported the project and provided tool support where necessary. Many in-house tools were developed to support the project. Examples include an ASN.1 compiler and in-built support for SNMP in Mnesia.

The OTP behaviors themselves were designed to be used by large groups of programmers. The idea was that there should be one way to program a client-server and that all programmers who needed to implement a client server would write plug-in code that slotted into a generic client-server framework. The generic server framework provided code for all the tricky parts of a client-server, taking care of things like code change, name registration, debugging, etc. When you write a client-server using the OTP behaviors you need only write simple sequential functions: all the concurrency is hidden inside the behavior.

The intention in the AXD was to write the code in as clear a manner as possible and to mirror the specifications exactly. This turned out to be impossible for the call control since we ran into memory problems. Each call needed six processes and processing hundreds of thousands of calls proved impossible. The solution to this was to use six processes per call only when creating and destroying a call. Once a call had been established, all the processes responsible for the call were killed and data describing the call was inserted into the real-time database. If anything happened to the call, the database entry was retrieved and the call control processes recreated.

The AXD301 [8] was a spectacular success. As of 2001, it had 1.13 million lines of Erlang code contained in 2248 modules [7]. If we conservatively estimate that one line of Erlang would correspond to say five lines of C, this corresponds to a C system with over six million lines of code.

As regards reliability, the AXD301 has an observed nine-nines reliability [7]—and a four-fold increase in productivity was observed for the development process [31].

5.2 Erlang is banned

Just when we thought everything was going well, in 1998, Erlang was banned within Ericsson Radio AB (ERA) for new product development. This ban was the second most significant event in the history of Erlang: It led indirectly to Open Source Erlang and was the main reason why Erlang started spreading outside Ericsson. The reason given for the ban was as follows:

> *The selection of an implementation language implies a more long-term commitment than the selection of a processor and OS, due to the longer life cycle of implemented products. Use of a proprietary language implies a continued effort to maintain and further develop the support and the development environment. It further implies that we cannot easily benefit from, and find synergy with, the evolution following the large scale deployment of globally used languages.* [26] quoted in [12].

In addition, projects that were already using Erlang were allowed to continue but had to make a plan as to how dependence upon Erlang could be eliminated. Although the ban was only within ERA, the damage was done. The ban was supported by the Ericsson technical directorate and flying the Erlang flag was thereafter not favored by middle management.

5.3 Open Source Erlang

Following the Erlang ban, interest shifted to the use of Erlang outside Ericsson.

For some time, we had been distributing Erlang to interested parties outside Ericsson, although in the form of a free evaluation system subject to a non-disclosure agreement. By 1998, about 40 evaluation systems had been distributed to external users and by now the idea of releasing Erlang subject to an open source license was formed. Recall that at the start of the Erlang era, in 1986, "open source" was unheard of, so in 1986 everything we did was secret. By the end of the era, a significant proportion of the software industry was freely distributing what they would have tried to sell ten years earlier—as was the case with Erlang.

In 1998, Jane Walerud started working with us. Jane had the job of marketing Erlang to external users but soon came to the conclusion that this was not possible. There was by now so much free software available that nobody was interested in buying Erlang. We agreed with Jane that selling Erlang was not viable and that we would try to get approval to release Erlang subject to an open source license. Jane started lobbying the management committee that was responsible for Erlang development to persuade it to approve an open source release. The principal objection to releasing Erlang as Open Source was concerned with patents, but eventually approval was obtained to release the system subject to a patent review. On 2 December 1998, Open Source Erlang was announced.

5.4 Bluetail formed

Shortly after the open source release, the majority of the original Erlang development team resigned from Ericsson and started a new company called Bluetail AB with Jane as the chief executive. In retrospect the Erlang ban had the opposite effect and stimulated the long-term growth of Erlang. The ban led indirectly to Open Source Erlang and to the formation of Bluetail. Bluetail led in its turn to the introduction of Erlang into Nortel Networks and to the formation of a small number of Erlang companies in the Stockholm region.

When we formed Bluetail, our first decision was to use Erlang as a language for product development. We were not interested in further developing the language nor in selling any services related to the language. Erlang gave us a commercial advantage and we reasoned that by using Erlang we could develop products far faster than companies using conventional techniques. This intuition proved to be correct. Since we had spent the last ten years designing and building fault-tolerant telecoms devices, we turned our attention to Internet devices, and our first product was a fault-tolerant e-mail server called the mail robustifier.

Architecturally this device has all the characteristics of a switching system: large numbers of connections, fault-tolerant service, ability to remove and add nodes with no loss of service. Given that the Bluetail system was programmed by most of the people who had designed and implemented the Erlang and OTP systems, the project was rapidly completed and had sold its first system within six months of the formation of the company. This was one of the first products built using the OTP technology for a non-telecoms application.

5.5 The IT boom – the collapse and beyond

From 1998 to 2000 there were few significant changes to Erlang. The language was stable and any changes that did occur were under the surface and not visible to external users. The HiPE team produced faster and faster native code compilers and the Erlang run-time system was subject to continual improvement and revision in the capable hands of the OTP group.

Things went well for Bluetail and in 2000, the company was acquired by Alteon Web systems and six days later Alteon was acquired by Nortel Networks. Jane Walerud was voted Swedish IT person of the year. Thus is was that Erlang came to Nortel Networks. The euphoric period following the Bluetail acquisition was short-lived. About six months after the purchase, the IT crash came and Nortel Networks fired about half of the original Bluetail gang. The remainder continued with product development within Nortel.

6. Part V: 2002 – 2005. Coming of age

By 2002, the IT boom was over and things had begun to calm down again. I had moved to SICS[21] and had started thinking about Erlang again. In 2002, I was fortunate in being asked to hold the opening session at the second Lightweight Languages Symposium (held at MIT).

6.1 Concurrency oriented programming and the future

In preparing my talk for LL2 I tried to think of a way of explaining what we had been doing with Erlang for the last 15 years. In so doing, I coined the phrase "concurrency oriented programming" —at the time I was thinking of a analogy with object oriented programming. As regards OO programming I held the view that:

- An OO language is characterised by a vague set of rules.
- Nobody agrees as to what these rules are.
- Everybody knows an OO language when they see one.

Despite the fact that exactly what constitutes an OO language varies from language to language, there is a broad understanding of the principles of OO programming and software development. OO software development is based first upon the identification of a set of objects and thereafter by the sets of functions that manipulate these objects.

The central notion in concurrency oriented programming (COP) is to base the design on the concurrency patterns inherent in the problem. For modelling and programming real-world objects this approach has many advantages—to start with, things in the real world happen concurrently. Trying to model real-world activities without concurrency is extremely difficult.

The main ideas in COP are:

- Systems are built from processes.
- Process share nothing.
- Processes interact by asynchronous message passing.
- Processes are isolated.

By these criteria both PLEX and Erlang can be described as concurrency oriented languages.

This is then what we have been doing all along. The original languages started as a sequential language to which I added processes, but the goal of this was to produce lightweight concurrency with fast message passing.

The explanations of what Erlang was have changed with time:

1. 1986 – Erlang is a declarative language with added concurrency.
2. 1995 – Erlang is a functional language with added concurrency.

[21] Swedish Institute of Computer Science.

3. 2005 – Erlang is a concurrent language consisting of communicating components where the components are written in a functional language. Interestingly, this mirrors earlier work in Eri-Pascal where components were written in Pascal.

Now 3) is a much better match to reality than ever 1) or 2) was. Although the functional community was always happy to point to Erlang as a good example of a functional language, the status of Erlang as a fully fledged member of the functional family is dubious. Erlang programs are not referentially transparent and there is no system for static type analysis of Erlang programs. Nor is it a relational language. Sequential Erlang has a pure functional subset, but nobody can force the programmer to use this subset; indeed, there are often good reasons for not using it.

Today we emphasize the concurrency. An Erlang system can be thought of as a communicating network of black boxes. If two black boxes obey the principle of observational equivalence, then for all practical purposes they are equivalent. From this point of view, the language used inside the black box is totally irrelevant. It might be a functional language or a relational language or an imperative language—in understanding the system this is an irrelevant detail.

In the Erlang case, the language inside the black box just happens to be a small and rather easy to use functional language, which is more or less a historical accident caused by the implementation techniques used.

If the language inside the black boxes is of secondary importance, then what is of primary importance? I suspect that the important factor is the interconnection paths between the black boxes and the protocols observed on the channels between the black boxes.

As for the future development of Erlang, I can only speculate. A fruitful area of research must be to formalise the interprocess protocols that are used and observed. This can be done using synchronous calculi, such as CSP, but I am more attracted to the idea of protocol checking, subject to an agreed contract. A system such as UBF [6] allows components to exchange messages according to an agreed contract. The contract is checked dynamically, though I suspect that an approach similar to that used in the Dialyzer could be used to remove some of the checks.

I also hope that the Erlang concurrency model and some of the implementation tricks[22] will find their way into other programming languages. I also suspect that the advent of true parallel CPU cores will make programming parallel systems using conventional mutexes and shared data structures almost impossibly difficult, and that the pure message-passing systems will become the dominant way to program parallel systems.

I find that I am not alone in this belief. Paul Morrison [23] wrote a book in 1992 suggesting that flow-based programming was the ideal way to construct software systems. In his system, which in many ways is very similar to Erlang, interprocess pipes between processes are first-class objects with infinite storage capacity. The pipes can be turned on and off and the ends connected to different processes. This view of the world concentrates on the flow of data between processes and is much more reminiscent of programming in the process control industry than of conventional algorithmic programming. The stress is on data and how it flows through the system.

6.2 Erlang in recent times

In the aftermath of the IT boom, several small companies formed during the boom have survived, and Erlang has successfully re-rooted itself outside Ericsson. The ban at Ericsson has not succeeded in completely killing the language, but it has limited its growth into new product areas.

[22] Like the bit pattern matching syntax.

The plans within Ericsson to wean existing projects off Erlang did not materialise and Erlang is slowly winning ground due to a form of software Darwinism. Erlang projects are being delivered on time and within budget, and the managers of the Erlang projects are reluctant to make any changes to functioning and tested software.

The usual survival strategy within Ericsson during this time period was to call Erlang something else. Erlang had been banned but OTP hadn't. So for a while no new projects using Erlang were started, but it was OK to use OTP. Then questions about OTP were asked: "Isn't OTP just a load of Erlang libraries?"—and so it became "Engine," and so on.

After 2002 some of the surviving Bluetail members who moved to Nortel left and started a number of 2nd-generation companies, including Tail-F, Kreditor and Synapse. All are based in the Stockholm region and are thriving.

Outside Sweden the spread of Erlang has been equally exciting. In the UK, an ex-student of mine started Erlang Consulting, which hires out Erlang consultants to industry. In France, Process-one makes web stress-testing equipment and instant-messaging solutions. In South Africa, Erlang Financial Systems makes banking software. All these external developments were spontaneous. Interested users had discovered Erlang, installed the open-source release and started programming. Most of this community is held together by the Erlang mailing list, which has thousands of members and is very active. There is a yearly conference in Stockholm that is always well attended.

Recently, Erlang servers have begun to find their way into high-volume Internet applications. Jabber.org has adopted the ejabberd instant messaging server, which is written in Erlang and supported by Process-one.

Perhaps the most exciting modern development is Erlang for multicore CPUs. In August 2006 the OTP group released Erlang for an SMP. In most other programming communities, the challenge of the multicore CPU is to answer the question, "How can I parallelize my program?" Erlang programmers do not ask such questions; their programs are already parallel. They ask other questions, like "How can I increase the parallelism in an already parallel program?" or "How can I find the bottlenecks in my parallel program?" but the problem of parallelization has already been solved.

The "share nothing pure message passing" decisions we took in the 1980s produce code which runs beautifully on a multicore CPU. Most of our programs just go faster when we run them on a multicore CPU. In an attempt to further increase parallelism in an already parallel program, I recently wrote a parallel version of map (pmap) that maps a function over a list in parallel. Running this on a Sun Fire T2000 Server, an eight core CPU with four threads per core, made my program go 18 times faster.

6.3 Mistakes made and lessons learnt

If we are not to make the same mistakes over and over again then we must learn from history. Since this is the history of Erlang, we can ask, "What are the lessons learnt? the mistakes made? what was good? what was bad?" Here I will discuss some of what I believe are the generic lessons to be learned from our experience in developing Erlang, then I will talk about some of the specific mistakes.

First the generic lessons:

The Erlang development was driven by the prototype

Erlang started as a prototype and during the early years the development was driven by the prototype; the language grew slowly in response to what we and the users wanted. This is how we worked.

First we wrote the code, then we wrote the documentation. Often the users would point out that the code did not do what the documentation said. At this phase in the development we told

them, "If the code and the documentation disagree then the code is correct and the documentation wrong." We added new things to the language and improved the code and when things had stabilized, we updated the documentation to agree with the code.

After a couple of years of this way of working, we had a pretty good user's manual [3]. At this point we changed our way of working and said that from now on the manuals would only describe what the language was supposed to do and if the implementation did something else then it was a bug and should be reported to us. Once again, situations would be found when the code and the documentation did not agree, but now it was the code that was wrong and not the documentation.

In retrospect this seems to be the right way of going about things. In the early days it would have been totally impossible to write a sensible specification of the language. If we had sat down and carefully thought out what we had wanted to do before doing it, we would have got most of the details wrong and would have had to throw our specifications away.

In the early days of a project, it is extremely difficult to write a specification of what the code is supposed to do. The idea that you can specify something without having the knowledge to implement it is a dangerous approximation to the truth. Language specifications performed without knowledge of how the implementation is to be performed are often disastrously bad. The way we worked here appears to be optimal. In the beginning we let our experiments guide our progress, Then, when we knew what we were doing, we could attempt to write a specification.

Concurrent processes are easy to compose

Although Erlang started as a language for programming switches, we soon realized that it was ideal for programming many general-purpose applications, in particular, any application that interacted with the real world. The pure message-passing paradigm makes connecting Erlang processes together extremely easy, as is interfacing with external applications. Erlang views the world as communicating black boxes, exchanging streams of message that obey defined protocols. This make it easy to isolate and compose components. Connecting Erlang processes together is rather like Unix shell programming. In the Unix shell we merely pipe the output of one program into the input of another. This is exactly how we connect Erlang processes together: we connect the output of one process to the input of another. In a sense this is even easier than connecting Unix processes with a pipe, as in the Erlang case the messages are Erlang terms that can contain arbitrary complex data structures requiring no parsing. In distributed Erlang the output of one program can be sent to the input of another process on another machine, just as easily as if it had been on the same machine. This greatly simplifies the code.

Programmers were heavily biased by what the language does and not by what it should do

Erlang programmers often seem to be unduly influenced by the properties of the current implementation. Throughout the development of Erlang we have found that programming styles reflected the characteristics of the implementation. So, for example, when the implementation limited the maximum number of processes to a few tens of thousands of processes, programmers were overly conservative in their use of processes. Another example can be found in how programmers use atoms. The current Erlang implementation places restrictions on the maximum number of atoms allowed in the system. This is a hard limit defined when the system is built. The atom table is also not subject to garbage collection. This has resulted in lengthy discussion on the Erlang mailing lists and a reluctance to use dynamically recreated atoms in application programs. From the implementor's point of view, it would be better to encourage programmers to use atoms when appropriate and then fix the implementation when it was not appropriate.

In extreme cases, programmers have carefully measured the most efficient way to write a particular piece of code and then adopted this programming style for writing large volumes of code. A better approach would be to try to write the code as beautifully and clearly as possible and then, if the code is not fast enough, ask for the implementor's help in speeding up the implementation.

People are not convinced by theory, only by practice

We have often said that things could be done (that they were theoretically possible) but did not actually do them. Often our estimates of how quickly we could do something were a lot shorter than was generally believed possible. This created a kind of creditability gap where we did not implement something because we thought it was really easy, and the management thought we did not know what we were talking about because we had not actually implemented something. In fact, both parties were probably incorrect; we often underestimated the difficulty of an implementation and the management overestimated the difficulty.

The language was not planned for change from the beginning

We never really imagined that the language itself would evolve and spread outside the Lab. So there are no provisions for evolving the syntax of the language itself. There are no introspection facilities so that code can describe itself in terms of its interfaces and versions, etc.

The language has fixed limits and boundaries

Just about every decision to use a fixed size data structure was wrong. Originally Pids (process identifiers) were 32-bit stack objects—this was done for "efficiency reasons." Eventually we couldn't fit everything we wanted to describe a process into 32 bits, so we moved to larger heap objects and pointers. References were supposed to be globally unique but we knew they were not. There was a very small possibility that two identical references might be generated, which, of course, happened.

Now for the specific lessons:

There are still a number of areas where Erlang should be improved. Here is a brief list:

We should have atom GC Erlang does not garbage collect atoms. This means that some programs that should be written using atoms are forced to use lists or binaries (because the atom table might overflow).

We should have better ways interfacing foreign code Interfacing non-Erlang code to Erlang code is difficult, because the foreign code is not linked to the Erlang code for safety reasons. A better way of doing this would be to run the foreign code in distributed Erlang nodes, and allow foreign language code to be linked into these "unsafe" nodes.

We should improve the isolation between processes Process isolation is not perfect. One process can essentially perform a "denial of service attack" on another process by flooding it with messages or by going into an infinite loop to steal CPU cycles from the other processes. We need safety mechanisms to prevent this from happening.

Safe Erlang Security in distributed Erlang is "all or nothing," meaning that, once authenticated, a distributed Erlang node can perform *any* operation on any other node in the system. We need a security module that allows distributed nodes to process remote code with varying degrees of trust.

We need notations to specify protocols and systems Protocols themselves are not entities in Erlang, they are not named and they can be inferred only by reading the code in a program. We need more formal ways of specifying protocols and run-time methods for ensuring that the agents involved in implementing a protocol actually obey that protocol.

Code should be first class Functions in Erlang are first-class, but the modules themselves are not first-class Erlang objects. Modules should also be first-class objects: we should allow multiple versions of module code[23] and use the garbage collector to remove old code that can no longer be evaluated.

6.4 Finally

It is perhaps interesting to note that the two most significant factors that led to the spread of Erlang were:

- The collapse of the AXE-N project.
- The Erlang ban.

Both of these factors were outside our control and were unplanned. These factors were far more significant than all the things we did plan for and were within our control. We were fortuitously able to take advantage of the collapse of the AXE-N project by rushing in when the project failed. That we were able to do so was more a matter of luck than planning. Had the collapse occurred at a different site then this would not have happened. We were able to step in only because the collapse of the project happened in the building where we worked so we knew all about it.

Eventually Ericsson did the right thing (using the right technology for the job) for the wrong reasons (competing technologies failed). One day I hope they will do the right things for the right reasons.

[23] Erlang allows two versions of the same module to exist at any one time, this is to allow dynamic code-upgrade operations.

A. Change log of erlang.pro

24 March 1988 to 14 December 1988

```
/* $HOME/erlang.pro
 * Copyright (c) 1988 Ericsson Telecom
 * Author: Joe Armstrong
 * Creation Date: 1988-03-24
 * Purpose:
 *   main reduction engine
 *
 *
 * Revision History:
 *   88-03-24     Started work on multi processor version of erlang
 *   88-03-28     First version completed (Without timeouts)
 *   88-03-29     Correct small errors
 *   88-03-29     Changed 'receive' to make it return the pair
 *                msg(From,Mess)
 *   88-03-29     Generate error message when out of goals
 *                i.e. program doesn't end with terminate
 *   88-03-29     added trace(on), trace(off) facilities
 *   88-03-29     Removed Var :_ {....} , this can be achieved
 *                with {..}
 *   88-05-27     Changed name of file to erlang.pro
 *                First major revision started - main changes
 *                Complete change from process to channel
 *                based communication here we (virtually) throw away all the
 *                old stuff and make a bloody great data base
 *   88-05-31     The above statements were incorrect much better
 *                to go back to the PROPER way of doing things
 *                long live difference lists
 *   88-06-02     Reds on run([et5]) = 245
 *                Changing the representation to separate the
 *                environment and the process - should improve things
 *                It did .... reds = 283 - and the program is nicer!
 *   88-06-08     All pipe stuff working (pipes.pro)
 *                added code so that undefined functions can return
 *                values
 *   88-06-10     moved all stuff to /dunder/sys3
 *                decided to remove all pipes !!!!!! why?
 *                mussy semantics - difficult to implement
 *                This version now 3.01
 *                Changes case, and reduce Rhs's after =>
 *                to allow single elements not in list brackets
 *   88-06-13     added link(Proc), unlink(Proc) to control
 *                error recovery.
 *                a processes that executes error-exit will send
 *                a kill signal to all currently linked processes
 *                the receiving processes will be killed and will
 *                send kill's to all their linked processes etc.
 *   88-06-14     corrected small error in kill processing
 *                changed name of spy communications(onloff)
 *                to trace comms(onloff)
 *   88-06-16     added load(File) as an erlang command
 *                added function new_ref - retruns
 *                a unique reference
 *   88-06-22     added structure parameter io_env
 *                to hold the io_environment for LIM communication
 *                changes required to add communication with lim ...
 *                no change to send or receive the hw will just appear
 *                as a process name (eg send(tsu(16),...)
 *                note have to do link(...) before doing a send
 *                change to top scheduler(msg(From,To,...))
 *                such that message is sent to Hw if To represents Hw
 *                following prims have been added to prims.tel
 *                link _hw(Hw,Proc) - send all msgs from Hw to Proc
```

```
*               unlink _hw(Hw) - stop it
*               unlink _all hw -- used when initialising
*               added new primitive
*               init hw causes lim to be initialised
*               load-magic ... links the c stuff
*               simulate(onloff) -- really send to HW
*   88-06-27    Bug is bind _var ..
*               foo(A,B) := zap,...
*               failed when zap returned foo(aa,bb)
*   88-07-04    port to quintus --- chage $$ varaibles
*               change the internal form of erlang clauses
*               i.e. remove clause(...)
*   88-07-07    changed order in receive so that first entry is
*               in queu is pulled out if at all possible
*   88-07-08    exit(X) X <> normal causes trace printout
*   88-09-14    ported to SICSTUS -
*               changed load to eload to avoid name clash with
*               SICSTUS
*   88-10-18    changed the return variable strategy.
*               don't have to say void := Func to throw away the
*               return value of a function
*   88-10-18    If we hit a variable on the top of the reduction
*               stack, then the last called function did not return
*               a value, we bind the variable to undefined
*               (this means that write,nl,... etc) now all return
*               undefined unless explicitly overridden
*               <<< does case etc return values correctly ?>
*               [[[ I hope so ]]]
*   88-10-18    send just sends doesn't check for a link
*   88-10-19    reworked the code for link and unlink
*               multiple link or unlink doesn't bugger things
*               make link by-directional. This is done by linking
*               locally and sending an link/unlink messages to
*               the other side
*   88-10-19    add command trap exit(Arg) .. Arg = yes I no
*               Implies extra parameter in facts/4
*   88-10-19    Changed the semantics of exit as follows:
*               error_exit is removed
*               exit(Why) has the following semantics
*               when exit(Anything) is encountered an exit(Why)
*               message is sent to all linked processes the action
*               taken at ther receiving end is as follows:
*               1) if trap exit(no) & exit(normal)
*               message is scrapped
*               2) if trap exit(no) & exit(X) & X <> normal
*               exit(continue) is send to all linked processes
*               EXCEPT the originating process
*               3) if trap_exit(yes) then the exit message
*               is queued for reception with a
*               receive([
*                 exit(From,Why) =>
*                  ...
*                  statement
*   88-10-19    send_sys(Term) implemented .. used for faking
*               up a internal message
*   88-10-24    can now do run(Mod:Goal) ..
*   88-10-25    fixed spawn(Mod:Goal,Pri) to build function name
*               at run time
*   88-10-27    All flags changes to yes I no
*               i.e. no more on,off true, false etc.
*   88-11-03    help command moved to top.pl
*   88-11-03    changed top scheduler to carry on
*               reducing until an empty queu is reached and then stop
*   88-11-08    changed 1111 to 'sys$$call
*   88-11-08    added lots of primitives (read the code!)
```

```
*   88-11-08     removed simulate(on) ... etc.
*                must be done from the top loop
*   88-11-17     added link/2, unlink/2
*   88-11-17     added structure manipulating primitives
*                atom_2_list(Atom),list_2_atom(List),
*                struct_name(Struct), struct_args(AStruct),
*                struct_arity(Struct), make struct(Name,Args)
*                get arg(Argno,Struct), set arg(Argno,Struct,Value)
*   88-11-17     run out of goals simulates exit(normal)
*   88-11-17     and timeout messages
*   88-12-14     added two extra parameters to facts
*                Save-messages(yeslno) and alias
*/
```

B. Erlang examples

B.1 Sequential Erlang examples

Factorial

All code is contained in modules. Only exported functions can be called from outside the modules.

Function clauses are selected by pattern matching.

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

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

We can run this in the Erlang shell.[24]

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

Binary Trees

Searching in a binary tree. Nodes in the tree are either `nil` or `{Key,Val,S,B}` where `S` is a tree of all nodes less than `Key` and `G` is a tree of all nodes greater than `Key`.

Variables in Erlang start with an uppercase letter. Atoms start with a lowercase letter.

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

Append

Lists are written `[H|T]`[25] where `H` is any Erlang term and `T` is a list. `[X1,X2,..,Xn]` is shorthand for `[X1|[X2|...|[Xn|[]]]]`.

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

Sort

This makes use of list comprehensions:

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

`[X || X <- T, X < Pivot]` means the list of `X` where `X` is taken from `T` and `X` is less than `Pivot`.

Adder

Higher order functions can be written as follows:

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

[24] The Erlang shell is an infinite read-eval-print loop.

[25] Similar to a LISP cons cell.

B.2 Primitives for concurrency

Spawn

```
Pid = spawn(fun() -> loop(0) end).
```

Send and receive

```
Pid ! Message,
.....

receive
    Message1 ->
        Actions1;
    Message2 ->
        Actions2;
      ...
    after Time ->
      TimeOutActions
end
```

B.3 Concurrent Erlang examples

"Area" server

```
-module(area).
-export([loop/1]).

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.
```

"Area" client

```
Pid = spawn(fun() -> area:loop(0) end),
Pid ! {self(), {square, 10}},
receive
    Area ->
        ...
end
```

Global server

We can register a `Pid` so that we can refer to the process by a name:

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

B.4 Distributed Erlang

We can spawn a process on a remote node as follows:

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

B.5 Fault tolerant Erlang

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 and throw

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

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

Links and trapping exits

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

B.6 Hot code replacement

Here's the inner loop of a 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
```

To do a code replacement operation do something like:

```
Server ! {change_code, fun(I, J) ->
                         do_something(...)
                       end}
```

B.7 Generic client-server

The module `cs` is a simple generic client-server:

```
-module(cs).
-export([start/3, rpc/2]).

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

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

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

Parameterizing the server

We can parameterize the server like this:

```
-module(test).
-export([start/0, add/2, lookup/1]).

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.
```

Here's a test run:

```
> test:start().
true
> test:add(xx, 1).
ok
> test:add(yy, 2).
ok
> test:lookup(xx).
{found,1}
> test:lookup(zz).
error
```

The client code (in `test.erl`) is *purely sequential*. Everything to do with concurrency (`spawn`, `send`, `receive`) is contained within `cs.erl`.

`cs.erl` is a simple *behavior* that hides the concurrency from the application program. In a similar manner we can encapsulate (and hide) error detection and recovery, code upgrades, etc. This is the basis of the OTP libraries.

Acknowledgments

I have mentioned quite a few people in this paper, but it would be a mistake to say that these were the only people involved in the Erlang story. A large number of people were involved and their individual and collective efforts have made Erlang what it is today. Each of these people has their own story to tell but unfortunately the space here does not allow me to tell these stories: all I can do is briefly mention them here. Thanks to all of you and to all the others whose names I have missed.

Mangement – Bjarne Däcker, Catrin Granbom, Torbjörn Johnson, Kenneth Lundin, Janine O'Keeffe, Mats Persson, Jane Walerud, Kerstin Ödling.

Implementation – Joe Armstrong, Per Bergqvist, Richard Carlsson, Bogumil (Bogdan) Hausman, Per Hedeland, Björn Gustavsson,Tobias Lindahl, Mikael Pettersson, Tony Rogvall, Kostis Sagonas Robert Tjärnström, Robert Virding, Klacke (Klacke) Wikström, Mike Williams.

Tools – Marcus Arendt, Thomas Arts, Johan Bevemyr, Martin Björklund, Hans Bolinder, Kent Boortz, Göran Båge, Micael Carlberg, Francesco Cesarini, Magnus Fröberg, Joacim Grebenö, Luke Gorrie, Richard Green, Dan Gudmundsson, John Hughes, Bertil Karlsson, Thomas Lindgren, Simon Marlow, Håkan Mattsson, Hans Nilsson, Raimo Niskanen, Patrik Nyblom, Mickaël Rémond, Sebastian Strollo, Lars Thorsén, Torbjörn Törnvist, Karl-William Welin, Ulf Wiger, Phil Wadler, Patrik Winroth, Lennart Öhman.

Users – Ingela Anderton, Knut Bakke, Per Bergqvist, Johan Bevemyr, Ulf Svarte Bagge, Johan Blom, Maurice Castro, Tuula Carlsson, Mats Cronqvist, Anna Fedoriw, Lars-Åke Fredlund, Scott Lystig Fritchie, Dilian Gurov, Sean Hinde, Håkan Karlsson, Roland Karlsson, Håkan Larsson, Peter Lundell, Matthias Läng, Sven-Olof Nyström, Richard A. O'Keefe, Erik Reitsma, Mickaël Rémond, Åke Rosberg, Daniel Schutte, Christopher Williams.

References

[1] J.L. Armstrong. *Telephony Programming in Prolog.* Report T/SU 86 036 Ericsson Internal report. 3 March 1986.

[2] J.L. Armstrong and M.C. Williams. *Using Prolog for rapid prototyping of telecommunication systems.* SETSS '89 Bournemouth 3-6 July 1989

[3] J.L. Armstrong, S.R. Virding and M.C. Williams. *Erlang User's Guide and Reference Manual. Version 3.2* Ellemtel Utveklings AB, 1991.

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

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

[6] Joe Armstrong. *Getting Erlang to talk to the outside world.* Proceedings of the 2002 ACM SIGPLAN workshop on Erlang.

[7] Joe Armstrong. *Making reliable distributed systems in the presence of errors.* Ph.D. Thesis, Royal Institute of Technology, Stockholm 2003.

[8] Staffan Blau and Jan Rooth. *AXD 301 – A new generation ATM switching system* Ericsson Review No. 1, 1998

[9] D.L. Bowen, L. Byrd,and W.F. Clocksin, 1983. *A portable Prolog compiler.* Proceedings of the Logic Programming Workshop, Albufeira, Portugal, 74-83.

[10] B. Däcker, N. Eishiewy, P. Hedeland, C-W. Welin and M.C. Williams. *Experiments with programming languages and techniques for telecommunications applications.* SETSS '86. Eindhoven 14-18 April, 1986.

[11] Bjarne Däcker. *Datologilaboratoriet De första 10 åren.* Ellemtel Utvecklings AB, 1994.

[12] Bjarne Däcker. *Concurrent Functional Programming for Telecommunications. A case study of technology introduction:* October 2000. Licentiate thesis. ISSN 1403-5286. Royal Institute of Technology. Stockholm, Sweden.

[13] L. Peter, Deutsch, *A Lisp machine with very compact programs.* Proceedings of 3rd IJCAI, Stanford, Ca., Aug. 1973.

[14] Ian Foster and Stephen Taylor. *Strand – New Concepts in Parallel Programming.* Prentice hall, 1990.

[15] A.S. Haridi. *Logic Programs Based on a Natural Deduction System.* Ph.D. Thesis Royal Institute of Technology, Stockholm, 1981.

[16] Bogumil Hausman. *Turbo Erlang: Approaching the speed of C.* In *Implementations of Logic Programming Systems*, Kluwer Academic Publishers, 1994.

[17] S. Holmström. *A Functional Language for Parallel Programming.* Report No. 83.03-R. Programming Methodology Lab., Chalmers University of Technology, 1983.

[18] Tobias Lindahl and Konstantinos Sagonas. *Detecting software defects in telecom applications through lightweight static analysis: A War Story.* APLAS 2004.

[19] David Maier and David Scott Warren: *Computing with Logic: Logic Programming with Prolog.* Benjamin/Cummings 1988.

[20] Simon Marlow Philip Wadler. *A practical subtyping system for Erlang.* ICFP 1997.

[21] Håkan Mattsson, Hans Nilsson and Claes Wikström: *Mnesia – A distributed bobust DBMS for telecommunications applications.* PADL 1999.

[22] Eliot Miranda. *BrouHaHa – A portable Smalltalk interpreter.* SIGPLAN Notices 22 (12), December 1987 (OOPSLA '87).

[23] Paul Morrison. *Flow-Based Programming: A New Approach to Application Development.* Van Nostrand Reinhold, 1994.

[24] Hans Nilsson, Torbjörn Törnkvist and Claes Wikström: *Amnesia – A distributed real-time primary memory DBMS with a deductive query language.* ICLP 1995.

[25] Hans Nilsson and Claes Wikström: *Mnesia – An industrial DBMS with transactions, distribution and a logical query language.* CODAS 1996.

[26] Tommy Ringqvist. *BR Policy concerning the use of Erlang.* ERA/BR/TV-98:007. March 12, 1998. Ericsson Internal paper.

[27] Tony Rogvall and Claes Wikström. *Protocol programming in Erlang using binaries.* Fifth International Erlang/OTP User Conference. 1999.

[28] T. Sjöland. *Logic Programming with LPL0 – An Introductory Guide.* CSALAB The Royal Institute of Technology, Stockholm, Sweden. October 1983.

[29] D.H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[30] Claes Wikström. *Distributed programming in Erlang.* PASCO'94. First International Symposium on Parallel Symbolic Computation.

[31] Ulf Wiger. *Four-fold increase in productivity and quality – industrial strength functional programming in telecom-class products.* Workshop on Formal Design of Safety Critical Embedded Systems. March 21-23, 2001, Munich.