ERICSSON

Four-fold Increase in Productivity and Quality

— Industrial-Strength Functional Programming in Telecom-Class Products

FEmSYS 2001
Deployment on distributed architectures

Revision C

Ulf Wiger
Senior System Architect
Strategic Product & System Management
Ericsson Telecom AB
Data Backbone & Optical Networks Division

Abstract

The AXD 301 ATM Switch is the flagship in Ericsson's line of Datacom products. A fault tolerant and highly scalable backbone ATM switch, AXD 301 has enabled Ericsson to take the lead in the migration of public telephony networks to becoming true multi-service networks, offering both quality voice and broadband data services on the same backbone.

Telecom-class products exhibit some very special characteristics, which are fast becoming essential even in other products, such as traditional datacom products, trading systems and application servers:

- Non-stop operation (99.999% availability)
- Event-driven operation with massive concurrency
- High capacity/processing power demand
- Maintainability, with frequent updates and long service life

This paper demonstrates how the development of such systems is supported by the Erlang/OTP technology.

The Erlang programming language was developed by Ericsson specifically for the purpose of building fault tolerant, distributed and complex systems. In the interest of Open Systems technology, Erlang is publicly available, both as a commercial product and as Open Source software, and competing implementations from other sources have emerged.

The paper demonstrates how Erlang supports the characteristics mentioned, while offering unusually high productivity.

Primary features of Erlang/OTP are:

- Intuitive programming model
- Very high abstraction level
- Pointer-free programming
- Short learning curve
- Very short lead times
- Massive concurrency
- Distributed processing
- Powerful error detection and recovery
- Excellent support for incremental design and prototyping.

The AXD 301 was developed from scratch into a full-fletched commercial product in less than three years, and soon won prestigious contracts. Studies of programmer productivity and performance in the field indicates at least a four-fold increase in both productivity and product quality. Erlang/OTP was a key factor in this success.

Table of Contents

1 Introduction

While research on functional programming dates back to the 1960s, functional programming languages have not made much impact on industrial programming. The reasons for this are sometimes subject to vivid debate in forums like the comp.lang.functional news group, and papers have been written to try to explain the phenomenon. Philip Wadler [wadler] lists Erlang as one of the few functional languages that can actually be found in significant industrial products. This paper describes the development of one such product: the Ericsson AXD 301 ATM multi-service switch.

The AXD 301 development project was highly successful, and a very complex product was developed from scratch in less than three years. The AXD 301 has since its introduction become a centerpiece in Ericsson's ENGINE concept, offering telephony operators a smooth migration path from today's vertical networks towards the vision of a single "all-IP" network in the future.

Initially positioned as a flexible ATM switch with telecom profile, AXD 301 has become a vital component in Ericsson's ENGINE concept. ENGINE – Ericsson's next-generation network solution – allows existing circuit-switched networks to evolve smoothly and rapidly into packet-based multi-service networks [engine].

Several major operators have chosen ENGINE: among them are British Telecom, Vodaphone, Telia and Diveo.

As product requirements have changed quite significantly compared to the initial specification of the product, AXD 301 has also demonstrated its strength as a flexible and open development platform. The hardware and software architecture, as well as the implementation technology, allowed us to adapt very quickly to new requirements and capitalize on exciting opportunities as they arose in the marketplace.

This paper describes the development of AXD 301, and tries to offer some explanations as to why the project succeeded. We believe that the answer lies in a successful mix of project-, process-, organization- and technology choices, and a healthy measure of pragmatism. It is not enough to rely on one or a few "silver bullets": one must look carefully at how all components interact.

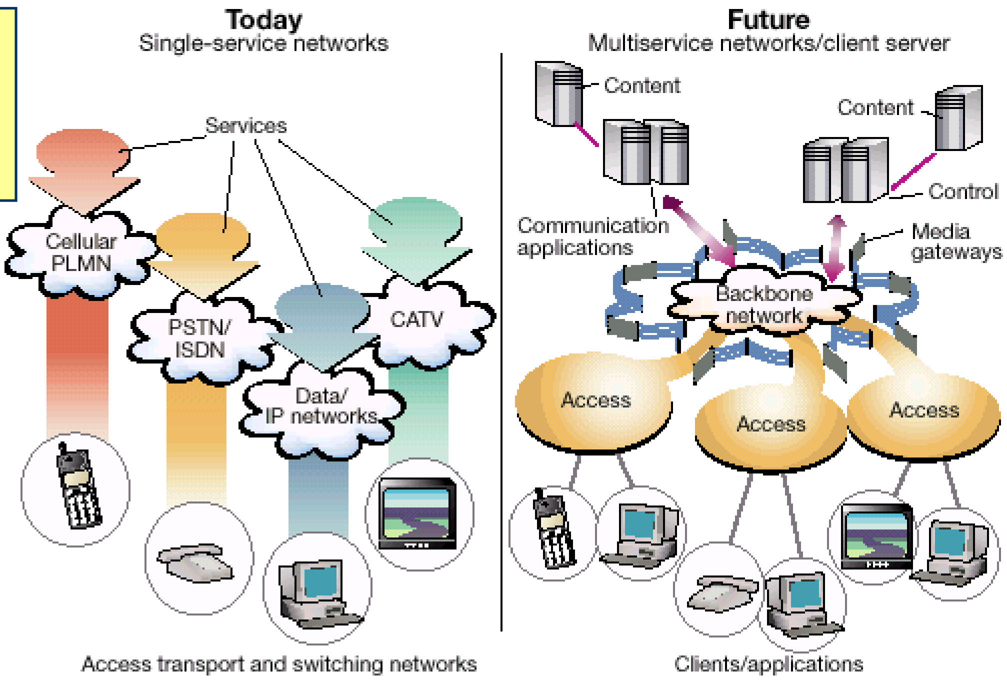
Specifically, this paper tries to show how the Erlang programming language and the OTP middleware platform fit exceptionally well into this way of developing complex software.

2 The AXD 301 Switch

The AXD 301 is a new asynchronous transfer mode (ATM) switching system from Ericsson. Combining features associated with data communication, such as compactness and high functionality, with features from telecommunications, such as robustness and scalability, the AXD 301 is a very flexible system that can be used in several positions in a network [blau]. The AXD 301 supports standardized ATM service categories, including constant bit rate (CBR), variable bit rate (VBR) and unspecified bit rate (UBR). The system supports ITU-T and ATM Forum signaling specifications.

Abbreviations:

PLMN	Public Land Mobile Network
PSTN	Public Switched Telephone Network
IP	Internet Protocol
CATV	Cable Television



Picture 1: Evolution of the network architecture [engine]

3 Carrier-Class Functionality

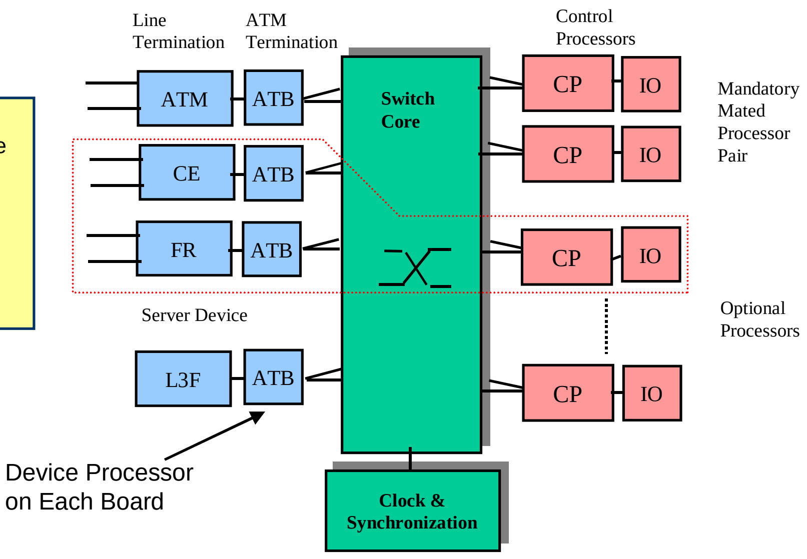
”Carrier-Class” is quickly becoming an abused term, as nearly all vendors in the telecom- and datacom world turn it inside out to find an angle that applies to their product. Simply put, Carrier-Class implies a measure of reliability and scalability which is necessary for the types of network that carry other networks and services. The operators of these networks are usually called ”carriers”; hence the term ”Carrier-Class”. However, as these networks evolve from traditional telecom networks to multiservice networks for both data and voice, it becomes less clear what level of service in network equipment is enough to warrant the label “Carrier-Class”. For example, as long as the requested service can be well served using a best-effort, connectionless network (e.g. the traditional Internet), then redundancy can be handled by the network, and individual network nodes can afford a system failure now and again. With the current strong trend towards real-time multimedia services over the Internet, the promised quality of service cannot be maintained unless the network nodes stay up at all times. This is Ericsson’s interpretation of the term “Carrier-Class”: using carrier-class network nodes, the network should offer the same quality of service as we’ve come to expect from traditional public telephony networks.

In this paper, “Carrier-Class”, “Telecom-Class” and “Telecom Profile” are used interchangeably.

3.1 Fault Tolerant Architecture

Abbreviations:

ATM	Asynchronous Transfer Mode
CE	Circuit Emulation
ATB	ATM Termination Board
FR	Frame Relay
L3F	Layer 3 Forwarding
CP	Control Processor
IO	I/O Board



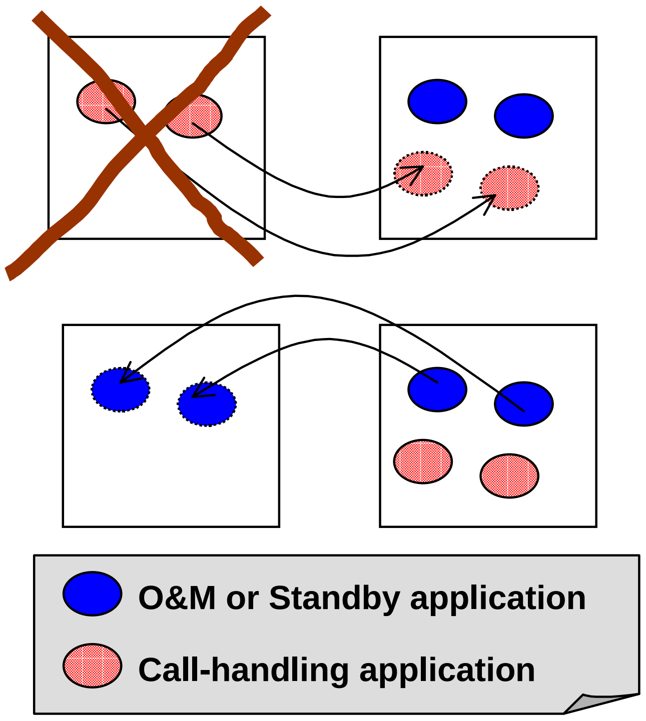
Picture 2: Logical hardware design

The AXD 301 was designed according to common practices for telecom systems. Work is divided between device processors that handle reasonably simple, real-time critical tasks close to the physical links, and control processors that take care of the more complex operations. In the AXD 301 release 3.2, the device processors are Motorola 68360 processors, running the VRTX operating system, and the control processors are SUN UltraSPARC 2i processors, running the Solaris operating system.

Switch core and clock boards are duplicated, and the control processors operate in mated-pair configurations. For maximum flexibility and economy, processing logic and I/O functions are placed on separate boards. This way, one ATM board can be used for a wide variety of physical interfaces. This is commonly referred to as a “mid-plane design”: boards for processing logic are found on the front side of the switch, while I/O boards with all physical interfaces are found on the rear side.

Control of device processors is divided among control processors according to a predefined or user-configured scheme. One pair of control processors will manage connections for a specific subset of the physical interfaces (dotted red line in Picture 2.) Connection setup is thus a fully distributed program, where the originating and terminating sides of a connection can be handled by different control processors. In order to survive control processor failure, connection state is replicated to the standby processor in each mated pair, as soon as a connection reaches a stable state.

The mated pairs in the AXD 301 share the load during normal operation: one takes care of traffic-handling tasks, while the other takes care of Operation & Maintenance tasks, or standby replication, or (in the case of the mandatory first mated pair) both.



Picture 3: **Mated-pair distributed error recovery**

If a control processor failure occurs, for example of the traffic-handling processor (picture 3), the standby processor will assume responsibility for traffic handling parts (called a failover). As the failed processor returns, the Operation & Maintenance and/or standby applications will move to the returned processor (called a takeover), in order to resume load-sharing mode. As traffic handling applications are more difficult to move, due to their real-time nature, AXD 301 always moves the least critical applications. During a takeover, applications have an opportunity to transfer state and log files to the new processor.

3.2 Availability

A common slogan for carrier-class availability is “5 nines”, i.e. 99,999% availability. This equates to roughly 5 minutes/year service downtime. Assuming no network redundancy, this should include planned outages. It follows that the system must support live upgrades and live expansion, while being fully resilient both in hardware and software. Also assuming that most of these systems are unattended, they must be able to recover automatically from any serious error that might yet occur – having to send a repair man to manually bring the system back on its feet will surely ruin any hopes of “5 nines” availability.

3.3 Scalability

A vital aspect of carrier-class systems is scalability. Actually, the system should be able to accomodate the future growth of the operator. This means being able to increase the number of interfaces, bandwidth, processing power, and number of services – all without stopping the system.

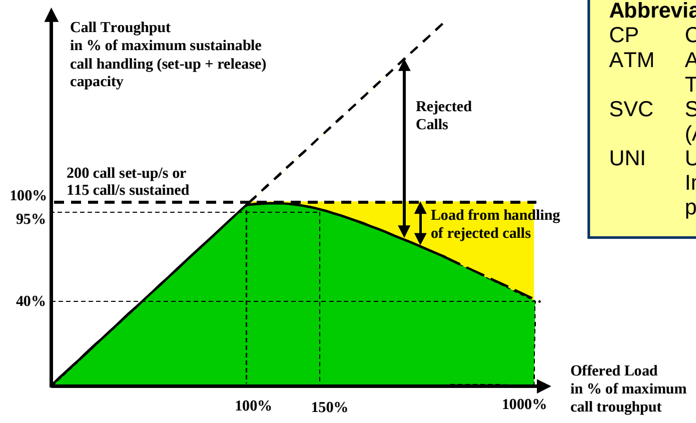
The AXD 301 is designed to scale from a 10 GBps to a 160 GBps system in-service. Meanwhile, call-handling capacity scales linearly as the number of control processors gradually increases from 2 to 32.

3.4 Load Tolerance

An often forgotten aspect of resilience is load tolerance. Network-level failures can cause recovery situations where a network node becomes severely overloaded. Therefore, load regulation is essential. A typical requirement is that a system should be able to provide 90% throughput at 150% load.

The AXD 301 manages 95% throughput at 150% load. Throughput then drops linearly to 40% at 1000% sustained load. The AXD 301 also senses the response time of other network nodes during recovery and slows down if the network node on the receiving end shows signs of overload.

Call Handling Throughput for one CP - AXD 301 release 3.2
Traffic Case: ATM SVC UNI to UNI



Abbreviations:

CP	Control Processor
ATM	Asynchronous Transfer Mode
SVC	Switched Virtual (ATM) Channel
UNI	User-Network Interface signaling protocol

Picture 4: Call-handling throughput

3.5 In-Service Upgrade and Expansion

In today's fast-paced market, with ever increasing requirements for new functionality, both in hardware and software, being able to upgrade and expand a system in-service becomes a magnificent challenge. The AXD 301 was designed from the start to support this, both in hardware and software. Even the switch core can be replaced in-service, without losing data. The control system supports a variety of strategies for upgrade, ranging from a fully synchronous smooth upgrade, to more complex scenarios, selectively rebooting individual control processors and/or temparily disabling non-critical services. Straightforward software upgrades are executed with minimal effort on behalf of the designers: the system will detect what software is to be upgraded, and automatically figures out how to do it.

3.6 Implementation Details

The AXD 301 control system, release 3.2, consists of roughly:

- 1 million lines of Erlang source code
- 400,000 lines of C/C++ code
- 13,000 lines of Java code

(excluding comments and whitespace)

In addition, some 500,000 lines of C code reside on device processors. In the control system, about 100,000 lines of C code constitutes 3rd party software which is started, configured and supervised by corresponding (tiny) Erlang programs.

The above numbers do not include the actual Erlang/OTP product, which contains 240,000 lines of Erlang, 460,000 lines of C/C++ and 15,000 lines of Java.

4 The Erlang Programming Language

4.1 Background

The Ericsson Computer Science Laboratory (CSLab) was established in 1984, and carried out a series of experiments with different programming languages and paradigms: [däcker]

- Imperative programming languages: Concurrent Euclid and Ada
- Declarative programming languages: PFL and Prolog
- Rule based programming: OPS4
- Object oriented languages: Frames and CLU

The target application was a control program for a private telephony exchange, and the purpose was to learn how to raise the level of programming using better language technology.

The experiments led to the conclusion that the telecommunications systems of the future couldn't be programmed with one language using one methodology. Rather, a number of different techniques might be used in different parts of the system. [däcker86]

One of the goals of the experiments was to find which style of programming led to the shortest and most beautiful programs closest to the level of formal specifications. These features have a great impact on productivity, since it has been shown that programmer productivity in number of error free lines of code per day is largely independent of the programming language. It was found in the first round of experiments that there was no clear winner among existing programming languages.

A second round of experiments started by adding concurrency to Prolog. Soon, a new experimental language started taking form, with a more functional style than Prolog. The new language was called Erlang after the Danish mathematician Agner Krarup Erlang.

Erlang can be described as a concurrent functional language, combining two main traditions:

- Concurrent programming languages: Modula, Chill, Ada, etc., from which Erlang inherits modules, processes, and process communication.
- Functional and logic programming languages: Haskell, ML, Miranda, Lisp, etc. from which Erlang inherits atoms, lists, guards, pattern matching, catch and throw etc.

Significant design decisions behind the development of Erlang were:

- It should be built on a virtual machine which handles concurrency, memory management etc., thus making the language independent of the operating system, and thus more portable.
- It should be a symbolic language with garbage collection, dynamic typing, and with data types like atoms, lists and tuples.
- It should support tail recursion, so that even infinite loops can be handled by recursion.
- It should support asynchronous message passing and a selective receive statement.
- It should enable default handling of errors, enabling an aggressive style of programming.

During 1988 and 1989, Erlang was tested in a prototype project for a new improved architecture of a private telephony exchange, called Audial Communication System (ACS). Based on the ACS architecture and the Erlang programming language, the prototype showed a significant improvement in design efficiency over current technology.

More prototypes followed, and Erlang evolved, with syntax changes and radical performance improvements. Support for distributed processing was added in 1993 [wik]. A book, *Concurrent Programming with Erlang*, was published in 1993 [Arm93], and a second edition, including a chapter on distributed Erlang, came in 1996 [arm96].

The first commercial release of Erlang, v4.1, appeared in October 1993. In 1996, the first prototype of the *Open Telecom Platform* (OTP) middleware was delivered.

Erlang was released as Open Source technology in 1998 (http://www.erlang.org). Since then, interest in Erlang has steadily grown, and the Open Source Erlang web site registered 140,000 user requests/month in late 2000. [däcker]

4.2 Sequential Erlang

The first Erlang program taught to students is usually the factorial function.

The standard definition of the factorial function (n!) is recursive:

$$x! = \begin{cases} 1 & \text{if } x = 0 \\ x*(x-1)! & \text{if } x \geq 1 \end{cases}$$

An Erlang program implementing this function could look like this:

```
1  -module(mathlib).
2  -export([factorial/1]).
3
4  factorial(0) ->
5     1;
6  factorial(N) when integer(N), N >= 1 ->
7     N * factorial(N-1).
```

Example 1: the factorial function

The first line defines a module, `mathlib`. A module encapsulates a set of functions into a loadable unit.

Line 2 is an export definition, making it possible to call the function factorial with one argument from outside the `mathlib` module. Only functions exported in this fashion can be called from other modules. A call to such a function would look like `mathlib:factorial(N)`.

Lines 4-7 implement the actual `factorial` function. Erlang relies on pattern matching and alternative function clauses to identify the correct branch to execute. Variables in Erlang start with uppercase. And the last expression evaluated in a function also becomes the return value of the function.

The first function clause, lines 4-5, is executed if the input argument is exactly 0. By definition, the factorial of 0 equals 1.

The second function clause, lines 6-7, matches any integer ≥ 1, and ends in a recursive call to itself. Recursion is the only way to create a loop in Erlang. The `when...` expression is called a guard, and allows for detailed inspection of the type and value of the input arguments.

This `factorial` function has the desireable property that it will either return a correct value or generate an exception. If the runtime system fails to find a matching function clause, it will generate a runtime error, which will cause the process executing the function to crash. More about handling runtime errors later.

Erlang variables have the following properties:

- All variables are local – there are no global variables
- A variable can be bound only once, trying to redefine an already bound variable causes a run-time error
- Variables are dynamically typed. Except for integer/float arithmetic, which behaves as expected, type coersion is not allowed. The dynamic type checking cannot be subverted.
- No explicit memory management. Variables are created by simply writing them down, memory allocation and de-allocation are handled automatically by the run-time system.

Erlang functions are inherently polymorphic, in that multiple definitions of the same function can be created, and selection of the right function definition is a result of pattern match on the structure, type, and contents of the input variables.

Erlang also has higher-order functions, which allow for very succinct specifications of complex behaviour. Below is an example of the quicksort algorithm expressed in Erlang, using list comprehensions:

```
1  qsort([Head|Tail]) ->
2     First = qsort([X || X <- Tail, X =< Head]),
3     Last = qsort([Y || Y <- Tail, Y > Head]),
4     First ++ [Head|Last];
5  qsort([]) ->
6     [].
```

Example 2: quicksort

The pattern in line 1 matches “a list of objects, where the first object is bound to the variable `Head`, and the list of remaining objects is bound to the variable `Tail`”. This also illustrates how pattern matching is used not only to identify data structures, but also to decompose them at the same time, identifying elements contained within the data structure and binding them to variables, so that they can be addressed directly.

The list comprehension in line 2 (`[X || X <- Tail, X =< Head]`), should be read as “all objects `X` taken from the list `Tail`, where `X` is ≤ `Head`”.

4.3 Concurrent Erlang

Erlang was designed from the beginning with explicit concurrency and distribution in mind. Its concurrency constructs were chosen to match the type of concurrency seen in telecom systems, with lightweight threads and asynchronous message passing[1]. A receiving process can filter the message queue using pattern matching. Only the first message matching a pattern will be removed from the message queue; other messages are automatically buffered.

[1] One result from the language experiments was that synchronous message passing does not adapt well to distributed programming. On the other hand, synchronous dialogues may be implemented on top of asynchronous message passing, when needed.

4.3.1 Client-server example

```
1  %% client-side function
2  call(Server, Request, Timeout) when pid(Server) ->
3     Reference = erlang:monitor(process, Server),
4     Server ! {request, self(), Reference, Request},
5     await_reply(Reference, Timeout).
6
7  await_reply(Reference, Timeout) ->
8     receive
9        {Reference, Reply} ->
10          erlang:demonitor(Reference),
11          Reply;
12       {‘DOWN’, Reference, process, Pid, Reason} ->
13          exit({server_died, Reason})
14    after Timeout ->
15       exit(timeout)
16    end.
17
18 %% server-side function
19 server_loop(State) ->
20    receive
21       {request, Client, Reference, Request} ->
22          {Reply, NewState} =
23             handle_call(Request, State),
24          Client ! {Reference, Reply},
25          server_loop(NewState)
26    end.
```

Example 3: a client-server example

The above program illustrates a safe client-server implementation. First of all, message sending is denoted by the ‘`!`’ operator, while message reception takes place within a ‘`receive ... end`’ expression, with an optional ‘`after Time -> ...`’ timeout guard. The client side uses a selective receive statement to wait for the right message. The ‘`Reference`’ returned from the function `erlang:monitor/2` is unique, so we use that to uniquely identify the dialogue. The call to `erlang:monitor/2` also tells the runtime system to send a special message (line 12) if the process in question (‘`Server`’) dies before responding.

A quick note on the recursion pattern, when implementing a non-terminating loop. Erlang uses a technique called Last Call Optimization in order to reuse the stack frame of the calling function, if there is no outstanding computation.

When using Erlang in a distributed setting, each instance of the Erlang runtime system is called an “Erlang node”. Distributed communication can be either transparent to a program, or made explicit. Explicit distribution is necessary for example when a number of Erlang nodes control different hardware units. It is also needed when controlling the exact distribution of processing tasks in a multi-processor system.

In the example above, the client and server may well be running on different Erlang nodes. The physical location of a process can be derived from its process identifier, but the majority of functions that operate on messages never need to worry about this.

4.3.2 Finite State Machine

One of the most attractive aspects of Erlang is perhaps programming of finite state machines. This is perhaps not too surprising, as the target application – telephony – is full of finite state machines.

```
1  ringing_a(A, B) ->
2     receive
3        {A, on_hook} ->
4           A ! {stop_tone, ring},
5           B ! terminate,
6           idle(A);
7        {B, answered} ->
8           A ! {stop_tone, ring},
9           switch ! {connect, A, B},
10          conversation_a(A, B)
11    after 30000 ->
12       A ! {stop_tone, ring},
13       B ! terminate,
14       idle(A)
15    end.
```

Example 4: Telephony example: state Ringing, A-side

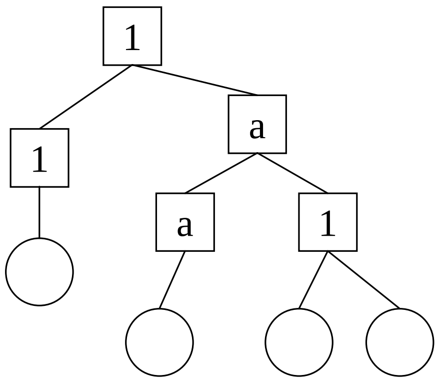
In Erlang, a finite state machine is simply an Erlang process. The states are represented by functions, and events are signalled via messages. In the example above, a telephony half-call, A-side (calling party) has entered state ringing (the telephone at the receiving end is ringing). Three things can happen:

- The calling party gives up and hangs up (line 3):
  we stop the ringing; next state will be idle (lines 4-6)
- The called party (B-side) answers:
  we stop the ringing, connect the call; next state is conversation;
- The called party does not answer; after a while, the protocol will time out:
  we stop the ringing; next state will be idle.

The fact that the state machine has its own Erlang process – a natural thread of control – makes it possible for the programmer to focus completely on describing the state machine specification.

4.3.3 Supervisor processes

The ability of an Erlang process to monitor other processes is also used to implement a special “supervisor” function. A supervisor is a process whose sole purpose it is to start, supervise, and possibly restart other erlang processes. For each supervisor, it is possible to specify the strategy to use when restarting processes (one-for-all, one-for-one, etc) and a limit on the number of times a process can be restarted.

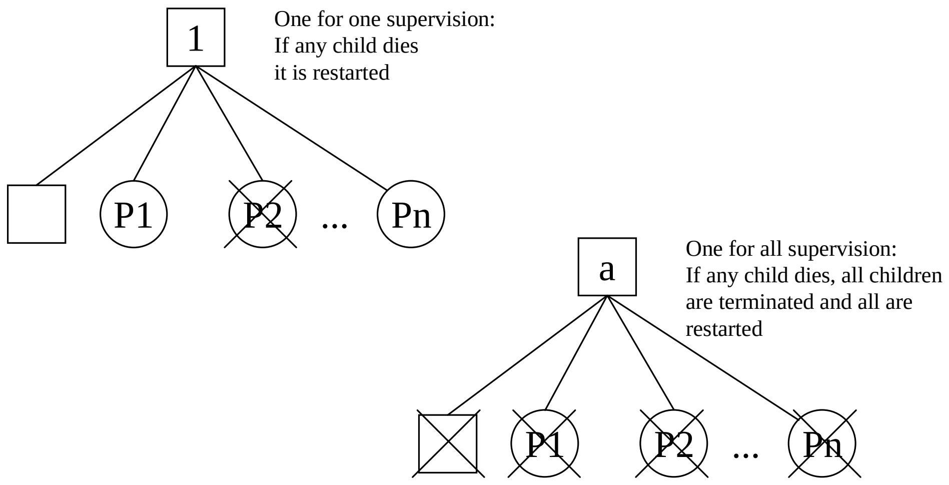


Abbreviations:

a	One-for-all supervision
1	One-for-one supervision

Picture 5: A supervision tree

In Picture 5, a supervision tree is illustrated. Square boxes indicate supervisor processer, while circles indicate worker processes. An ‘a’ stands for “one-for-all” supervision, while a ‘1’ stands for “one-for-one” supervision.



Picture 6: One-for-one and one-for all supervision

A process crash caught by a supervisor leads to a rather verbose description of the event, which gives significant support in fault localization.

```
1  =CRASH REPORT==== 14-Mar-2001::10:48:47 ===
2    crasher:
3      pid: <0.102.0>
4      registered_name: ccv_server
5      error_info: {{badmatch,{error,enoent}},
6                   [{ccv_views,get_otp_version,3},
7                    {ccv_views,make_views,4},
8                    {ccv_server,init,1},
9                    {gen_server,init_it,6},
10                   {proc_lib,init_p,5}]}
11     initial_call: {gen,init_it,
12                        [gen_server,
13                         <0.74.0>,
14                         <0.74.0>,
15                         {local,ccv_server},
16                         ccv_server,
17                         [],
18                         []]}
19     ancestors: [ccv_supervisor,<0.72.0>]
20     messages: []
21     links: [<0.74.0>]
22     dictionary: []
23     trap_exit: true
24     status: running
25     heap_size: 17711
26     stack_size: 23
27     reductions: 80774
28   neighbours:
```

Example 5: a crash report

The above standard crash report was snipped from an Erlang-based source code browser. The error (line 5) indicates a missing file (the POSIX code ‘`enoent`’). The ‘`badmatch`’ token means that a pattern match failed: the term `{error, enoent}` was returned, when something else was expected. The function, in which the failed pattern match occurred, was (line 6) `ccv_views:get_otp_version/3`. A call chain (lines 6-10) is also provided, to further clarify the context.

Looking at the other information elements, most of the known information about the process is included, making it easy to identify its role in the system. All such crash reports are also logged to disk, and can be browsed using a special tool.

Having identified the process, and the reason why it died, we can quickly get on with the job of fixing the problem, in this case, the function `ccv_views:get_otp_version/3`.

Looking at the function in question:

```
1  get_otp_version(CS, CSF, OTPPaths) ->
2      {ok, Bin} = file:read_file(CSF),
3      Opts =
4        ccv_cs:parse_cs_contents(
5           binary_to_list(Bin), CS, OTPPaths),
6      {value,{_,OTP}} =
7         lists:keysearch(otpVsn, 1, Opts),
8      OTP.
```

Example 6: programming for the correct case

there are only two possible candidates: `file:read_file/1` (line 2) and `lists:keysearch/3` (line 7). Of the two, only `file:read_file/1` can return `{error, enoent}`. While it would have helped even more to also find out the name of the missing file, the error in question was easily identified and corrected within minutes, even though the function contained no code for error checking, apart from the patterns describing the correct behaviour.

In large Erlang systems, supervision hierarchies are built in order to encapsulate errors and support restart escalation. With a little thought, one can build a structure where the worker processes rely on the supervision infrastructure for fault containment and recovery. The worker process itself can focus on asserting that functions return “useable” values, throwing an exception if something unexpected happens, and re-initialize itself after a restart. This, in combination with the use of pattern matching, leads to a style of programming that is often referred to as “programming for the correct case” or “ the Let it Crash philosophy.” The latter is of course a bit provocative, but seems to serve the purpose of countering many programmers’ instinctive urge to save the life of their process at all cost.

4.4 Built-in functions

Erlang has a number of built-in functions, which implement things that are difficult to do (or at least difficult to do efficiently) in Erlang. One prominent example is the Erlang Term Storage (ets) functions. Ets implements access structures for long-lived storage. While functionally equivalent structures could be (and in fact have been) implemented in Erlang, using processes, message passing, and standard Erlang data types, the speed of the built-in implementation is roughly 30 times higher. The available ets structures are hash tables (bag or set semantics) and ordered set. Tables can be named or anonymous, and can be either hidden from other processes, accessible as read-only, or fully read-write to all processes. All accesses are thread safe and fast – in the order of microseconds. When a process dies, all ets tables created by that process are automatically removed, as one would expect.

4.5 External interfaces

In order to interface with the outside world, Erlang provides “ports”, which behave very much like Erlang processes. An Erlang process can communicate with a port via message passing, and is notified if the port is closed for some reason. A port can be an interface to a file, a inter-process communication pipe, or a user-defined C program, which is dynamically linked into the runtime system for minimum overhead.

When developing and testing software, ports are easily emulated using an Erlang process with stub functionality.

4.6 Dynamic code loading

Erlang modules can be loaded at any time into a running system. The runtime system allows processes running in the old version of a module to finish, unless instructed to delete the old module, in which case all processes executing in it are killed. All fully qualified function calls (i.e. `Module:Function(...)`) will always enter the newest available version of a module. This way a process executing an endless loop in a module can switch to a newer version upon a given signal:

```
1  server_loop(State) ->
2     receive
3        {request, Client, Reference, Request} ->
4           case Request of
5              {change_code, VersionInfo} ->
6                 NewState =
7                    ?MODULE:transform_state(
8                       State, VersionInfo),
9                 Client ! {Reference, ok},
10                ?MODULE:server_loop(NewState);
11             _ ->
12                {Reply, NewState} =
13                   handle_call(Request, State),
14                Client ! {Reference, Reply},
15                server_loop(NewState)
16          end
17    end.
```

Example 7: client-server example with code change

In the above example, we've altered the code of Example 3 (the client-server program) to make the server change to a newer version of a module in an orderly fashion. Since the server handles one request at a time, we know that it will be in a stable state every time it reads a new request from the message queue (line 3). In this example we use the Erlang `case ... end` expression to identify the request as a `change_code` instruction. The second branch (lines 12-15) is identical to lines 22-25 in Example 3. As it ends with a "local" recursive call (the module name excluded), the function will loop within the same version of the module every time. The first branch (lines 6-10) calls a special function (lines 6-7) to transform the internal state representation. This function call is fully qualified[2], which will cause the newest implementation of transform_state/2 to be executed. It returns to server_loop/1 which is still executing in the old module. Finally (line 10), a recursive call to server_loop/1 is made, but this one also fully qualified. Thus, the server enters the new version of the module, after transforming its internal state in an orderly fashion. Pending requests are automatically buffered, and will be served as usual.

[2] ?MODULE is a special Erlang pre-processor macro identifying the current module

5 Open Telecom Platform

The Open Telecom Platform (OTP) is a middleware product for Erlang development. It was written primarily in Erlang[3], but also offers support libraries for C and Java programs that need to interface with Erlang. A wide range of components needed for telecom-class systems are included:

- Support libraries for error handling, reporting and logging
- SNMP agent
- An in-memory, distributed real-time database management system
- CORBA
- HTTP server, FTP, SSL, TCP/IP components
- ASN.1 compiler
- Productivity tools, static source code analysis, profiling, debugging

The database management system, Mnesia, deserves special attention. It operates directly on Erlang data objects and in the same memory space as Erlang applications. Therefore, there is no semantic gap, and very little processing overhead when operating on the data. Tables can be replicated on demand to several processors, and can be kept fully in RAM, in RAM and on disk, or on disk only. Mnesia provides full transaction support, but also allows “dirty” operations to operate on database objects without the overhead of transactions. This means that Mnesia can be used even in applications where data access times must be in the order of microseconds – a requirement for real-time operation.

[3] As seen in Chapter 3.6, Erlang/OTP contains twice as much C as Erlang source code, but most of the functionality is still implemented in Erlang. Erlang source code is much more concise than C source code.

6 Development of AXD 301

6.1 Background

In 1996, Ericsson canceled a high-profile project, aimed at developing nation-wide Broadband ISDN systems. The reasons were of course many and often complex, but perhaps most importantly, the market had moved in a different direction. The AXD 301 project was started in an effort to capture some of the knowledge gained in the previous project and produce a marketable product[4].

A decision was made to develop an ATM switch with features from both the telecom world and the datacom world. In other words: small and compact, but with carrier-class features. As the ATM switch market was already well established in 1996, it was imperative not only to reach the market quickly, but also to somehow leapfrog the competitors, producing a first entry that would surpass their second or third entries. Clearly, a radical development strategy was needed.

On a more general note, the rather costly failure of a large, highly publicized project served as a useful reminder that the Software Crisis identified in 1980 was far from over. A 1995 study of 365 Information Systems professionals made by PC Week, found the following statistics on software development projects:

- 16% successful
- 53% operational (but less than successful)
- 31% cancelled

When looking at development of complex software, the situation is even worse: according to Butler Group (1997), five out of six large development projects fail, and in companies with more than $300 million turnover, only one large project out of ten succeeds.

6.2 Assumptions

The following assumptions were made at the start of the project:

- Most of the project members had only experienced “traditional” large software development projects, and were eager to try a different approach.
- Apart from the pure fundamentals of project management, few other things could be taken for granted.
- Close attention would be paid to the progress from the start, with empirical data based on experimentation to support the decision making process.
- To minimize administrative overhead, all 200 project members were put under one roof[5]. Consequently, the project came to be known within Ericsson as the “roof project”.

[4] In fact, several such activities led to successful products a few years after the “large project”, as there was a virtual grab-bag of excellent technology to build on.
[5] As it happened, a building just large enough became available when the large project was cancelled.

It was also clear from the beginning that there was no commercially available middleware platform that could provide significant leverage. A quick study of available alternatives was performed, and Erlang/OTP was selected as the most promising technology. The OTP middleware was under development at the time, so the AXD 301 project would have to start prototyping and modelling based on OTP's predecessor, BOS. However, experiences from other Ericsson projects using Erlang gave such strong positive indicators that this seemed like the least risky approach.

6.3 Project

The AXD 301 project was built around the following tenets:

- Early phases under one roof
- Powerful architecture team
- Clear chain of command
- Rapid decisions (within days)
- Organize according to product structure

6.3.1 Early phases under one roof

One of the painful lessons from previous projects was that attempting large-scale industrial design with geographically separated design centers incurs a large administrative overhead. Therefore, it was decided that at least the early phases of development should be done in one place. Thus, both hardware and software development was located at Varuvägen in Älvsjö, outside Stockholm. We quickly found this extremely convenient, as designers, testers, system management and product management were only a few strides away from each other. It was easy to form special task teams to solve complex problems, and brain storming sessions often took place around the coffee machines in the hallway.

6.3.2 Powerful architecture team

Our system management team consisted of some of Ericsson's most experienced architects and subject matter experts – even a couple of leading Erlang experts. As a bi-product from the previous project, many designers also had significant experience with ATM protocols and support functions, so much of the design work could be handled by the design teams themselved. The central architecture team could focus on maintaining a dialogue with the design teams, publishing and communicating design rules, and working very much hands-on with characteristics measurements of the system.

6.3.3 Clear chain of command

While design and implementation responsibility was mostly decentralised, a clear chain of command was established from the start. Thus, whenever difficult decisions had to be made, there would be no doubt about who had the last word in each matter.

6.3.4 Rapid decisions

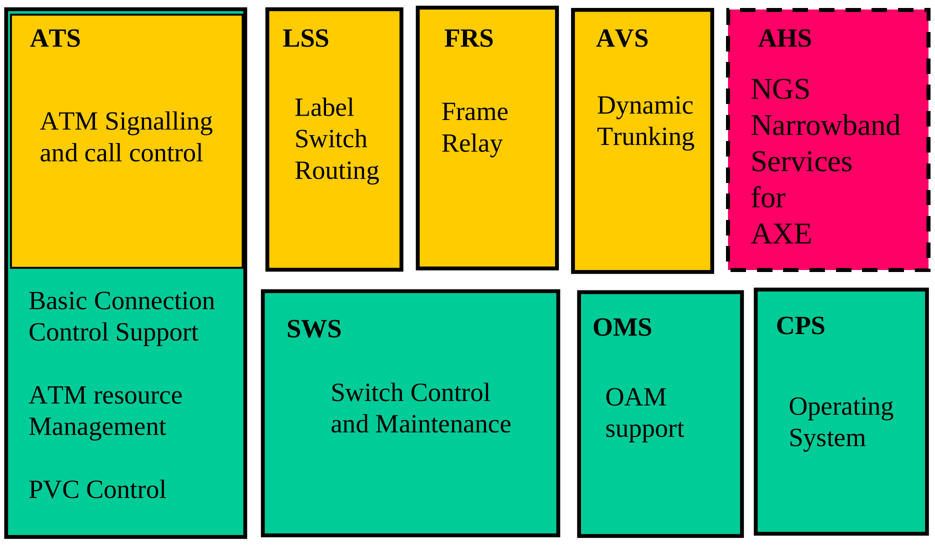
An expressed goal was to deal with matters as they came up, and not let things fester. One person at system management spent nearly all his time making sure that problems were identified, discussed and resolved as quickly as possible.

6.3.5 Organize according to project structure

As Ericsson is quite used to building telecom-class systems, the basic building blocks needed are fairly well known. The organization was built around a tested functional decomposition into subsystems and blocks, and design sections were formed to match this structure.

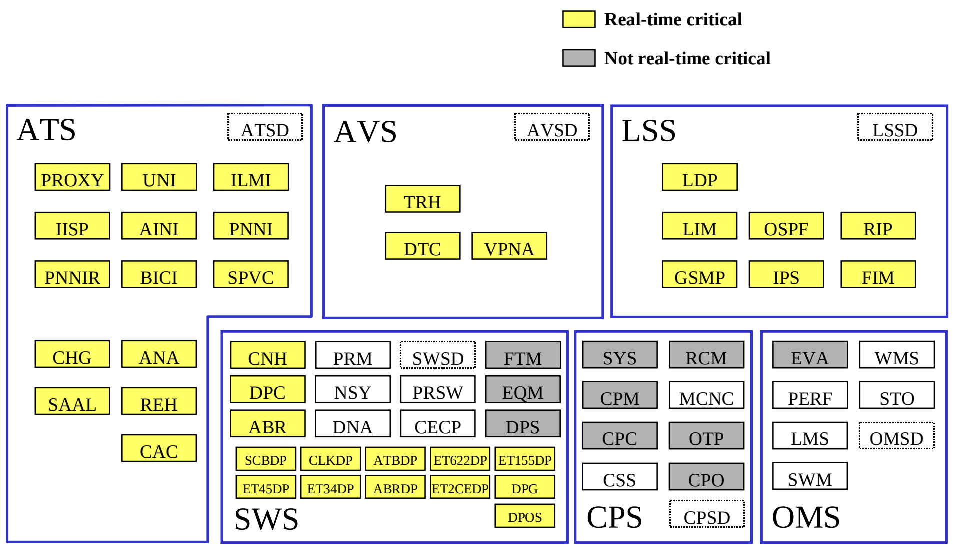
Abbreviations:

ATM	Asynchronous Transfer mode
PVC	Permanent Virtual (ATM) Channel
OAM	Operation & Maintenance



Picture 7: **Identify subsystems**

Abbreviations:
... AXD 301 internal block names; the meaning of the abbreviations is not important in this context.



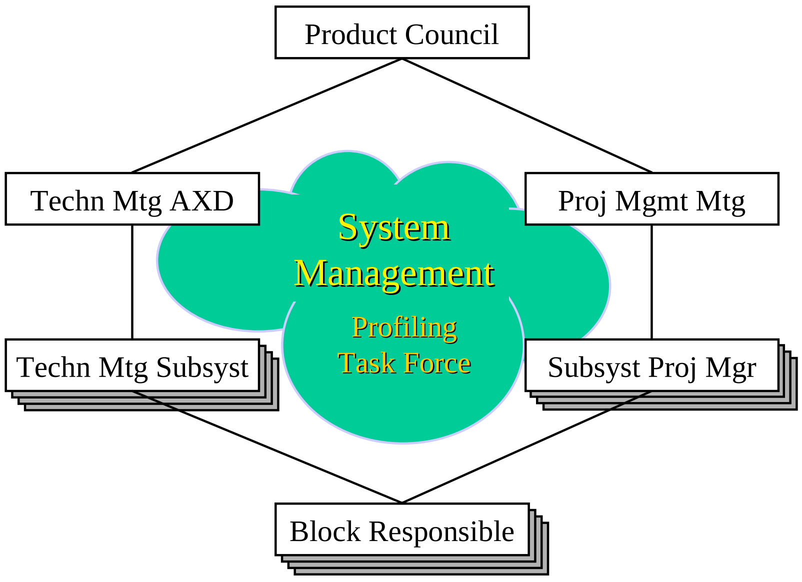
Picture 8: **Indentify blocks within subsystems**

Great care was taken to define intuitive and stable block interfaces. Interface definitions were pre-released and subjected to peer review and formal approval. Any changes to a block interface would be subjected to a formal change procedure, with new peer review and project cost estimates.

Responsibility for the internal block implementation, however, was delegated as much as possible to the design team in charge of the block. Formal code review was usually only done if the designers requested it, or if system testing and profiling identified strange or inappropriate behaviour of certain blocks.

Technical decision forums were created for each subsystem, and a central technical forum was established for the whole system, with representatives from each subsystem, system management, project management and product management.

As functionality was planned for different increments, sub-projects were formed, one per subsystem and increment. Decision forums similar to the technical forums were established.



Picture 9: Formal control structures

The technical decision structure, representing the design organization, and the project decision structure, representing the projects, formed two parallel control structures with different focus. To facilitate efficient communication between the two control structures, some people would take part on both sides. Furthermore, system management acted as a glue between these two control structures and the designers.

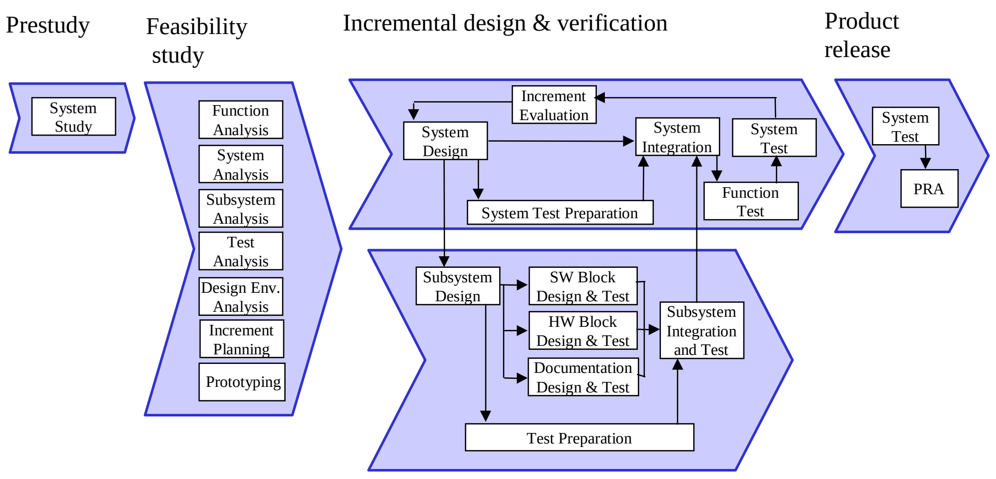
System management played a very active part throughout the development. In order to encourage developers to talk to the architects at system management, it assumed the role as a designer help desk with an open door policy. This work included holding seminars and participating in trouble shooting activities. Extensive characteristics profiling was also done by the Erlang experts at system management, and optimization and robustness task forces were formed at times to improve characteristics along the way.

This approach relied heavily on human interaction, and was helped significantly by the fact that we were all gathered in one place. This highly interactive environment also allowed developers to mature with the system, and the organization also consistently scores very highly in employee satisfaction surveys.

6.4 Process

Ericsson has a well established project management method, called PROPS [props]. It is a generic and adaptable project management method, whose terminology and concepts are well known to Ericsson personnel. PROPS-based development projects usually define their own “PROPS adaptation”, which is a specialization of the generic concepts, defining which documents should be written at certain stages, which templates should be used, etc.

AXD 301 defined a PROPS adaptation called MIX. It takes into account the incremental development strategy.



Picture 10: The MIX Development process

The MIX process focuses on incremental design and verification, prototyping of critical parts, early characteristics feedback and gradual system “growth”, rather than a “big-bang” design effort.

6.4.1 Incremental design

The first commercial release of AXD 301 was developed in 8 major increments, each building upon the functionality and experience from its predecessor.

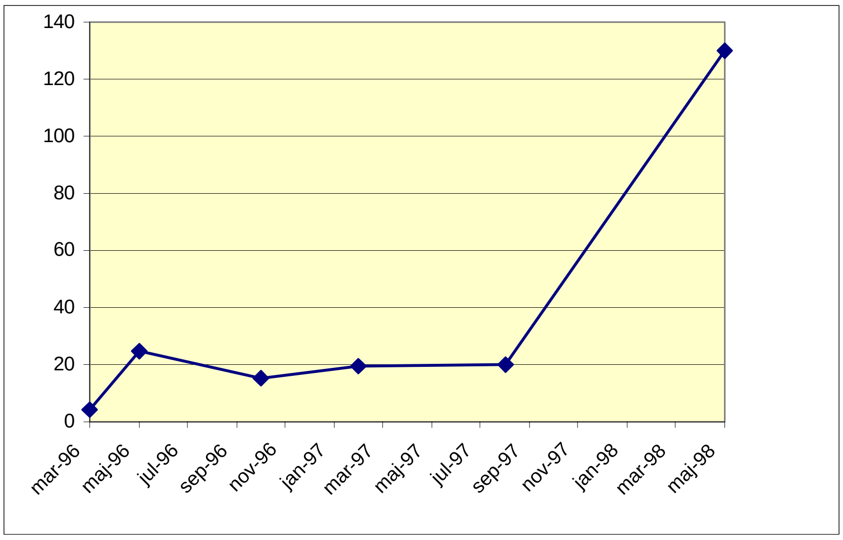
- The first increment consisted only of basic integration and system start – its main purpose being to learn how we should manage delivery and integration.
- Increment 2 supported switched and permanent channels, basic redundancy and fault management
- Starting with increment 3, the system was useable in customer demonstrations. Significant function growth was achieved.
- Increment 3.5 was trown in, initially as an experiment, in order to port the AXD 301 software to the latest version of Erlang/OTP. This work was carried out by 1-3 people over a two-week period. Since the results were so positive, further work became based on this increment.
- Increments 4-8 added functionality in both software and hardware, at the same time as core functions stabilized
- Increment 8 was certified as a full-blown commercial release.

Development of the first commercial release lasted from May ’96 to Jan ’99. Two of the major increments were split into 2 increments each, as we ported to a new chip set and switching architecture. On average, each increment took 4,5 months of design and testing.

The incremental design allowed us to “grow” the system in a controlled manner. Designers performed function tests on real hardware and were able to learn the system as it evolved.

Each increment was followed by “early system tests”, devised to give an indication of where we were heading. This helped us identify problem areas early, and resulted in several task forces to study the problems, review existing program code, and suggest improvements.

One such task force was the performance task force, started after profiling of the first bare-bones implementation of the UNI protocol. Although the CPU time required for one UNI-UNI call setup had improved from 250 ms in an early prototype to 65 ms in the first production version (Oct '96), we felt that we needed even more focus on performance in order to reach our goal. As a result of the optimization work led by the task force, the time was brought down to under 10 ms in increment 8, despite significant function growth in the protocol implementation.



Picture 11: UNI call handling performance progress (calls/sec)

Reliability and usability were two other areas where task forces were formed to capture experiences from early testing.

6.5 Documentation

Another lesson from previous projects was that excessive documentation can pose a real threat to productivity. Too much documentation tends to obscure the details rather than clarifying them, and make it difficult to impossible to identify the information elements that are vital for strategic decisions.

The AXD 301 project decided from the start to keep documentation down to a minimum. For the average designer, the following documents were the most important:

- The Requirement Specification
- Implementation Proposals
- Function Specifications
- Block Descriptions
- Interface Descriptions

The designer needed to maintain the block description and interface descriptions for the block. Other documents were written by teams, as part of the specification and modelling process. As a system stabilizes, documentation becomes more and more important, and should be accompanied by tutorials, seminars, and other means of describing the product. This should of course be part of the planning from the start, but the challenge lies in not introducing too much too soon.

Just like the system itself, the documentation must be allowed to grow and mature through the incremental development process.

6.6 Tools

The AXD 301 project uses a robust suite of configuration management, development and test tools, such as the ClearCase version management system, TTCM suites, traffic generators, and a number of special programs for automated build and product packaging.

The development of the AXD 301 control system software is somewhat unusual in that it started with a very primitive development environment. Some tools existed, but the initial development environment for Erlang designers consisted of a compiler, an interpreter, and an Emacs editor.

As Erlang/OTP has matured, several productivity tools have become available. One of the most critical tools for large projects is the Erlang test server. While not yet a supported product, it is very stable and quite powerful. Test cases are written in Erlang, taking full advantage of the pattern matching syntax. An adaptation layer takes care of making the test cases executable on a designer's workstation as well as in the target environment. An HTML page is generated, showing clearly the outcome of the test run.

As Erlang programs execute inside a virtual machine, creating simulated environments is quite straightforward. AXD 301 designers use a tool that automates the installation and start of a multi-processor simulation on the workstation. In this environment, the majority of test suites, and even upgrade tests, can be executed.

Erlang also comes with extremely powerful built-in profiling capabilities. It is possible to generate trace output from function calls, scheduler events, message passing, garbage collections, etc. without any special compiler directives or source code for debugging. All trace output can be timestamped, allowing for very accurate profiling and characteristics analysis.

7 Experiences

Comparisons between Ericsson-internal development projects indicate similar line/hour productivity, including all phases of software development, rather independently of which language (Erlang, PLEX, C, C++ or Java) was used. What differentiates the different languages then becomes source code volume.

Some concrete comparisons of source code volume have been made, as applications written in C++ have been rewritten in Erlang, resulting in a ten-fold reduction in the number of lines of uncommented source code. Other comparisons have indicated a four-fold reduction. A reasonable conclusion is that productivity increases by the same factor, given the same line/hour programmer productivity.

When measuring product quality in number of reported errors per 1000 lines of source code, the same relationship seems to exist: similar error density, but given the difference in code volume, roughly 4-10 times fewer errors in Erlang-based products.

Furthermore, experiences from maintaining some Erlang-based products in the field indicates that error corrections usually do not give rise to unwanted side-effects. This is in line with what one would expect from using a mostly side-effect free declarative language.

These comparisons do not pretend to hold up to scientific scrutiny, but they are collected from indicators which show a rather consistent pattern. They also seem consistent with opinions offered by non-Ericsson developers [one2one],[sendmail].

While AXD 301 consists of programs written in C, C++ and Java as well as Erlang, and each language is used where it fits best, practically all the complex work is done in Erlang. This is usually decided by the designers themselves – many of our most skilled C programmers prefer Erlang for complex tasks, and consider it the obvious choice for anything that involves concurrency or distribution.

Finally, we have seen that programmers – even those with no previous exposure to functional languages – learn Erlang quickly, and become productive within a matter of weeks.

8 A Word on Methodology

The AXD 301 project did not settle for any particular existing methodology, but rather developed one from scratch, based on the circumstances and available expertise. An expressed goal was to achieve a lightweight, adaptive methodology. Underneath lies the conviction that we are not ready to view software design as a strictly top-down activity, where formal design documents are carefully crafted, so that the actual programming becomes a predictable manufacturing process.

We do not seem to be alone in this thinking. As many efforts to apply “traditional engineering principles” to software engineering have failed, more and more people begin arguing that *all* activities in software design: high-level design, programming, and testing, are part of the creative process, and therefore difficult to predict or automate [fowler].

8.1 Design Abstractions

Indeed, a provocative comment on the state of software design today is that "the source code listing *is* the engineering design" [reeves]. Another, perhaps more diplomatic way to formulate it: "Programming is both a *top down* activity from requirements and overall structure of the system and a *bottom up* activity based on the *abstractions* provided by the programming language. Abstractions such as *modules, processes, higher order functions*, etc. are to the programmer like transistors, capacitors, resistors, etc. to the hardware designer. It is important that the abstractions be few, simple to understand and yet powerful and provide the needed functionality" [däcker].

A central part of our software design methodology was to select a programming language, Erlang, with powerful abstractions that were exceptionally well suited for designing our type of system. Indeed, for many parts of the system, the only really accurate and detailed design documentation is the source code listing. This is hardly unique to our system, but perhaps we are better off than most, since our programming language was designed from the beginning to be as close to a formal specification as possible.

8.2 Extreme Programming

Many of the tenets of the AXD 301 MIX methodology show striking similarity with Extreme Programming [xp]. The Extreme Programming approach was based on experiences from the SmallTalk community in the late '80s and early '90s, and was formalized after a Daimler Chrysler project started in March 1996. Four phrases sum up the central philosophy of Extreme Programming:

- Improve communication
- Seek simplicity
- Get feedback on how well you are doing
- Always proceed with courage

The more detailed rules of Extreme Programming may be well applicable to individual parts of a project as large as the AXD 301. When looking at a subsystem in AXD 301, designers move around, working on different blocks, and integration takes place on a daily basis. Erlang/OTP fits this mode of working exceptionally well.

Integrating the whole system cannot easily be done that frequently, partly because hardware delivery schedules must be taken into account as well; but also because perhaps the most important reason for integrating the whole system is to perform a rather staggering amount of tests in order to ensure that the system justifies the label Carrier-Class. Most of these tests must be performed on complete (rather expensive) systems, and to a large extent carried out manually. This makes too frequent system integrations hard to justify.

9 Conclusions

Subjective opinions from the AXD 301 project tell of great satisfaction with the turn-around time, and how easy it is to change the implementation of individual components without having to rewrite other parts of the system. The concise and readable format of Erlang programs makes it easy to reason about the implementation, and the excellent profiling support makes it easy to identify hotspots.

The increased productivity of the programmers is felt throughout the development project, as lead times are reduced, and one arrives much sooner at a point where a running system can be studied and compared against the initial requirements. This is gratifying to the programmers, as well as reassuring to project management.

The most striking argument, however, must be the track record of the AXD 301, which suggests a method to develop complex, mission-critical systems with high productivity and quality. While the method hinges on a holistic approach to software development, and individual components (like Erlang/OTP) can most likely be replaced by something else, Erlang/OTP has proven to be an almost perfect match for this approach.

10 References

[arm93] ”Concurrent Programming in Erlang” (Armstrong, Virding, Williams), Prentice-Hall, 1993, ISBN 0-13-285792-8, 1st Edition

[arm96] ”Concurrent Programming in Erlang” (Armstrong, Virding, Wikström, Williams), Prentice-Hall, 1996, ISBN 0-13-285792-8, 2nd Edition

[arts] Verifying Generic Erlang Client-Server Implementations (Thomas Arts, Thomas Noll) http://www.ericsson.se/cslab/~thomas/ifl2000.pdf

[axd301] Ericsson AXD 301: http://www.ericsson.com/datacom/products/wan_core/axd301/index.shtml

[blau] AXD 301: A new generation ATM switching system: http://www.ericsson.com/datacom/emedia/ericssonvol31iss6.pdf

[bt1] Ericsson wins groundbreaking GBP 270 million contract with BT http://www.ericsson.com/press/archive/1999Q1/19990121-0035.html

[bt2] Ericsson secures £110 million next-generation network order from BT http://www.ericsson.com/press/archive/1999Q4/19991012-0501.html

[carrier] Carrierclass Product Information: http://www.carrierclass.com/products.htm

[cescom] Ericsson to provide global VoIP solution for CESCOM http://www.ericsson.com/press/archive/1999Q2/19990621-0008.html

[cms6000] Convedia CMS-6000 Media Server: http://www.convedia.com/cms.htm

[comm] CommWorks Total Control 2000: http://commworks.com/svprovider/tc2000/tc2000_mainpage.html

[cont] Continuant: http://www.gnp.com/continuant/index.html

[diginet] Ericsson [...] to deliver first Pan-Latin America broadband wireless network http://www.ericsson.com/press/archive/1999Q2/19990630-0003.html

[däcker] Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction (Bjarne Däcker)

[däcker86] ”Experiments with Programming Languages and Techniques for Telecommunications Systems” (Däcker, Elshiewy, Hedeland, Welin, Williams), *Software Engineering for Telecommunication Switching Systems*, Eindhoven, April 14-18, 1986

[ekström] Design Patterns for Simulations in Erlang/OTP (Ulf Ekström) http://ftp.csd.uu.se/pub/papers/masters-theses/0178-ekstrom.pdf

[engine] ENGINE Server Network http://www.ericsson.se/review/2000_03/files/2000031.pdf (Hallenstål, Thune, Öster, Ericsson Review 3/2000)

[fowler] ”The New Methodology” http://www.martinfowler.com/articles/newMethodology.html (Martin Fowler)

[hinde] ”Use of Erlang/OTP as a service creation tool for IN services” (Sean Hinde, One2One) http://www.erlang.se/euc/00/one2one.pdf

[kpn] Ericsson signs Next Generation Network contract with Dutch KPN http://www.ericsson.com/press/archive/1999Q3/19990927-0002.html

[nuera] Nuera Softswitch: http://www.nuera.com/products/orca_gx_21.cfm

[nyteknik] ”Stora Datorprojekt Kraschar” (Ny Teknik no 21 1997)

[props] ”Common Concept” Ericsson EPMI http://www.ericsson.se/epmi/methods/index.shtml

[reeves] ”What is software Design” (Jack W. Reeves, C++ Journal 1992) http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm

[sendmail] ”Sendmail meets Erlang: Experiences Using Erlang for Email Applications”, (Fritchie, Larson, Christenson, Öhman) http://www.erlang.se/euc/00/euc00-sendmail.pdf

[telefonica] ”Ericsson next generation network solution, ENGINE chosen by Telefónica” http://www.ericsson.se/press/19991227-0005.html

[telia] Ericsson and Telia build world's first multi-service network aimed at enterprise users http://www.ericsson.com/press/archive/1999Q2/19990603-0003.html

[wadler] Functional Programming: Why no one uses functional languages http://cm.bell-labs.com/cm/cs/who/wadler/papers/sigplan-why/sigplan-why.ps (Philip Wadler)

[wik] ”Distributed Programming in Erlang” (Claes Wikström), *First International Symposium on Parallel Symbolic Computation*, Linz, Sep 26-29, 1994

[xp] ”Extreme Programming: A Gentle Introduction” http://www.extremeprogramming.org/