World-Class Product Certification using Erlang

Ulf Wiger
Ericsson AB
S-126 25, Älvsjö
Sweden
+46 8 719 81 95
ulf.wiger@etx.ericsson.se

Gösta Ask
Ericsson AB
S-126 25, Älvsjö
Sweden
+46 8 757 11 18
Gosta.Ask@etx.ericsson.se

Kent Boortz
Ericsson AB
P.O. Box 1505
S-125 25, Älvsjö, Sweden
+46 8 727 57 55
Kent.Boortz@uab.ericsson.se

ABSTRACT

It is now ten years ago since the decision was made to apply the functional programming language Erlang to real production projects at Ericsson. In late 1995, development on the Open Telecom Platform (OTP) started, and in mid 1996 the AXD 301 project became the first user of OTP. The AXD 301 Multi-service Switch was released in October 1998, and later became "the heart of ENGINE", Ericsson's leading Voice over Packet solution.

In those early days of Erlang programming, high-level tools for development and testing were not really available, and programmers used mainly the Emacs editor and the Erlang shell. Still, anecdotal evidence suggested a 4-10x productivity increase compared to mainstream programming techniques.

Through the years, significant progress has been made, especially in the area of automated testing of Erlang programs. The OTP team designed an Erlang-based test suite execution environment, were developers can easily write their own automated test suites, and now performs nightly builds where more than one thousand test cases are executed on ten different platforms. OTP designers can view the outcome in web-based test reports as they come to work the next day. Each corrected bug results in a new test case that is incorporated into the ever-growing test suite. Thus, this world-class middleware is certified to telecom-class quality without a dedicated test team!

The AXD 301 project uses OTP's test environment, and executes more than 10,000 automated test cases before each major release. Designers and testers compose their own test suites, and the designers carry out function tests with little or no help from the Integration and Certification team. Each test case can be run both in a simulated environment on the designer's workstation and in the test lab on real hardware. In order to provide stimuli to the system, the testers often design their own traffic generators in Erlang.

To analyze the faults that occur, Erlang offers an increasing wealth of debugging options. Beyond the symbolic error messages, which are often sufficient to locate the fault, Erlang developers are able to dynamically turn on tracing on message passing, scheduling events, garbage collections, selected function calls, etc.

This paper demonstrates how Erlang's declarative syntax and pattern matching provide an outstanding environment for test suite development.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools (e.g., data generators, coverage testing), tracing, error handling and recovery.

General Terms

Measurement, Performance, Design, Reliability, Experimentation, Languages, Verification.

Keywords

Erlang, Testing

1. INTRODUCTION

In the late 90's Ericsson put much thought into designing general Test Tool Middleware for corporate-wide usage. It would adapt a Test Management System to the various Traffic generators that were being used in System Test Departments in Ericsson; TSS 2000, MGTS, HP BSTS, IGEN, Wandel & Goltermann to mention a few.

One driving force was to replace the Automation parts based on Autosis — a script-like language that was used in connection with mobile simulators like MSMS/MSTG in a predominantly AXE10 target environment. These thoughts were written down during the era of Business area Radio Automated Verification Environment (BRAVE), an initiative that for a while attracted large funding from Ericsson management.

The result was a Requirement Specification[1]. This was clearly a Top Down approach. It turned out to suffer from lack of practical experience to guide the work onwards. Some general ideas from the BRAVE era can still be used. On the whole it must be noted, however, that local Bottom Up approaches to Test Automation have been more successful in producing working results. The efforts to co-ordinate Test Automation Development have failed. Instead, a large marketplace has developed inside the company where different local solutions to the problem of Test Automation compete. This is not necessarily bad.

The ideas of the mid-90's, to be able to "verify" large systems and create "certification" (in the sense of guaranteeing fault-free operation by testing), are all but dead. Not that the ideas lacked merit, but the practical difficulties, and perhaps also the lack of understanding what "verification" actually implies in the context of large real-time systems, have drawn the ideas into disrepute. There is however work starting out again on a smaller scale with better hope of success. Model checking of parts of the Erlang-based AXD 301 component RCM, (specifically, a distributed read/write locker used by the cluster controller RCM) is reported in ref. [3]. Work by Prover Technology on the formal verification of certain characteristics of Hardware circuits also shows the possibilities, see ref. [4]. This is largely still a research activity.

Meanwhile, it is still perfectly true that testing is needed — more than ever actually, since switch control programs, both for telephony and for data, have grown enormously both in size and complexity. There is no other option than test. It is however important to understand the limitations of traditional testing, both from a theoretical and from a practical point of view. Isolated Function Test cannot find more than a fraction of all the faults that exist in any of the large products. Hence more efficient means to provoke error situations and thereby find faults are needed. The best practices so far consist of the generation of a massive and diversified traffic load on all external interfaces of the system under test. Let such testing be called Traffic Load Test.

There is a paradox in Traffic Load Test compared to Function Test (FT); you really do not know what kind of faults you are looking for. You hope for continuous error-free operation of the whole network, even during maximal utilisation of all available resources. At the same time you know from experience that it is precisely in a situation with massive traffic load on the network (the system) that dormant faults start to pop up and show their existence. It can be that timer constraints somewhere in the application programs are too tight, or data faults may appear in some protocol data that is not used until a congestion situation occurs. The faults were there all along, but FT test cases could not find them. To find such faults by systematic testing over all possible values on the input data streams is simply not feasible. The state space is too large. Nevertheless, faults of this nature are what our customers fear most of all. They will blame us when such faults appear during operation, creating a heavy burden on our maintenance staff. It is therefore imperative that we try and flush as many dormant faults as possible out of the systems before the products are released. Automated System Test with Traffic Load is the method.

Waiting for the general solutions, testers are always faced with the necessity to automate everyday tasks. Stability tests of large networks with a multitude of different tools generating traffic as Background Activity Traffic can be run manually, but the cost is high. The risk of human error is large, causing delays. Stability tests are usually run over extended periods of time and must be possible to restart frequently if some errors occur in the network. Customers also require regression tests on system updates.

All too often, testing is seen as a completely separate activity from design, and the ability of the design environment to prepare for, and help streamline, the testing phase is often ignored. Indeed, it is common in large industrial projects for developers to simply hand off their software to a testing department, which then tries to find faults in it. One obvious argument in favor of such a procedure is that it is difficult to test your own software, much like it is difficult to proofread your own documents. The obvious counter argument is that it is extremely difficult to completely test a component without rather detailed knowledge of how it was designed. Invariably, the testers in this scenario will stumble upon faults that they cannot analyze, and will call on the designers for support; the designers, not being familiar with the testing procedure, may find it difficult to reproduce the error.

With design and test thus separated, it is vital to create detailed specifications of the components, lest the testers be forced to read the source code of the entire system in order to find out how it is supposed to work. As traditional mainstream programming languages make pretty poor specification tools, new notations are invented to bridge the gap, and a whole industry has grown out of the need to support the parallel development of specification and program.

Within functional language research, much effort is put into designing programming languages that offer extensive support for finding faults already at compile time. This is achieved mainly by designing the programming language so that it becomes useful as a specification language as well, and then making the compiler capable of verifying as many of the specification properties as possible. As not many examples of complex industrial systems designed in such languages exist, it remains uncertain how much of the traditional testing process can be reduced.

The Erlang programming language strikes a compromise between traditional industrial languages and functional languages. Its syntax puts it at the same level as common specification languages like SDL[5], but it is not as strict as many of its cousins among functional programming languages. It does not necessarily reduce the amount of testing needed, but as it turns out, seems to appeal to testers and designers alike. This allows for breaking down the barriers between design and test, and blurring the line where programming ends and testing begins. We will try to show in this paper why this is desirable.

2. Evolution of Testing in OTP and AXD 301

2.1 The Setting

When the AXD 301 project started, the Erlang programming language had only been used in projects of significantly smaller size and complexity. The Open Telecom Platform was a concept, based on experiences from previous Erlang projects, but the only concrete implementation available was that of BOS – the predecessor of OTP. AXD 301 had to start conducting experiments based on BOS and await the first version of OTP.

Several things were unknown: whether Erlang/OTP would work in a project of AXD 301's size; whether performance would be adequate; and how, indeed, to proceed with design and testing.

It was decided that we would not use any modeling tool intended for another programming paradigm (e.g. UML, SDL) before we knew more about what programming method would be most appropriate. Pre-studies indicated that programming in Erlang would present altogether different challenges to a modeling tool than e.g. C/C++, due to dynamic typing and pattern matching semantics, native support for concurrency and distribution, etc.

Similarly, investments in test equipment were careful, awaiting more experience of testing in Erlang-based projects.

The product itself, the AXD 301 ATM Switch (which later evolved into a Multi-service Switch and Voice Media Gateway), defined the task to be solved: a datacom product with the same management profile and robustness normally associated with a traditional telephone switch. This means non-stop operation, in-service upgrade and expansion, excellent overload protection, and an abundance of diagnostics functionality. The sheer complexity of control systems for telephony has become quite staggering: The Lucent Definity PBX contains 4 million lines of code[7], and the Nortel DMS-500 local and long-distance telephony switch consists of 26 million lines of code[8].

Telecom-class quality can be illustrated another way: A subscriber to AT&T's private line service with an Enhanced Reliability Option (ERO)[1] is guaranteed less than a cumulative 5 minutes of service outage each month; otherwise, AT&T will refund the subscription fee for the month in question[9]. This naturally includes planned downtime. 5 minutes in one month corresponds to 99,9999% availability for each subscriber line. Assuming 64 kbps/subscriber (a typical telephone line) and e.g. 31*32 = 992 subscriber lines on one interface board in the switch[2], more than 5 minutes service outage on one interface board means that AT&T would lose the equivalent of 992 subscriber months, or >82 years, of revenue. Furthermore, assuming one control processor (without redundancy) for one subrack (16 interface boards), a control processor crash causing more than 5 minutes of downtime would by itself result in a loss of 15872 subscriber months, or >1300 years, of revenue. A subscription rate of $30/month would mean that $476,000 would have to be refunded. It is easy to imagine that faults requiring a service technician to drive to the switch and manually replace a board, or rebooting a control processor, would result in much longer outages than 5 minutes. More seriously, of course, losing the ability to service emergency calls could even lead to people dying.

Several tricks are used to ensure that the provided service meets the stringent quality requirements. The most important design feature is of course redundancy — all critical components duplicated — but great care must also be taken to make sure that the system automatically recovers from its own errors as often as possible. But most of all, the product must be subjected to an enormous battery of tests in order to weed out problems before the first delivery to a customer.

2.2 The Learning Phase

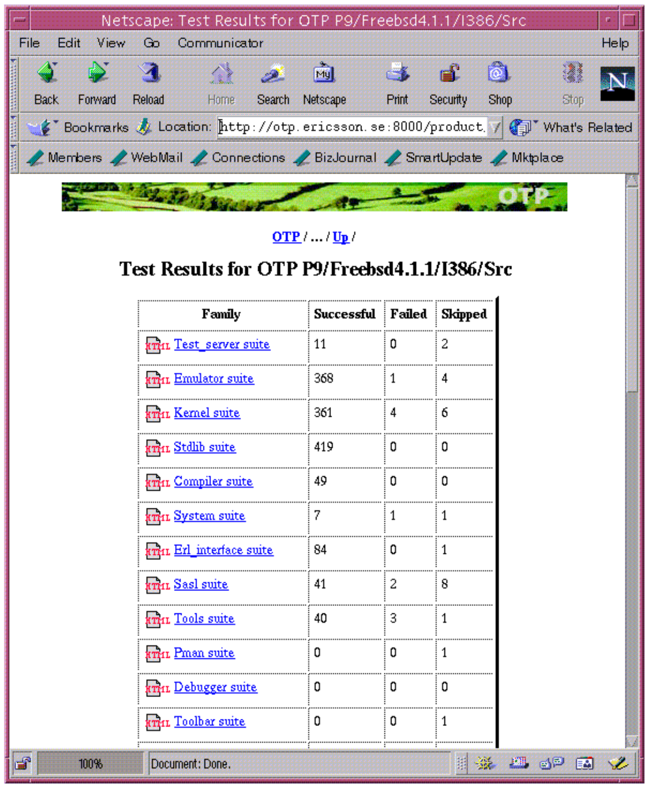
AXD 301 was developed incrementally, and the first increment was a *learning* increment. The main goal was to learn how to deliver software to the test lab, and how to integrate and start a basic system.

Much was learned about what needed to be improved in Erlang/OTP, and many suggestions on how to improve error reporting were put forth by testers experienced in certifying the AXE 10 telephony switch.

As the incremental development progressed, testers began designing their own test tools, traffic generators, etc. Some of the tools proved more useful than quite expensive third party equipment.

2.3 The OTP Test Server

During these early stages, the Erlang/OTP design team started developing an automated test server. The OTP test server relies heavily on the core aspects of the Erlang language and runtime system. Test cases are written in plain Erlang, and test results are presented in a user-friendly HTML format (see Figure 1).


Netscape: Test Results for OTP P9/Freebsd4.1.1/I386/Src
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://otp.ericsson.se:8000/product. What's Related
Members WebMail Connections BizJournal SmartUpdate Mktplace
OTP
100%
Document: Done.

OTP / ... / Up /

Test Results for OTP P9/Freebsd4.1.1/I386/Src

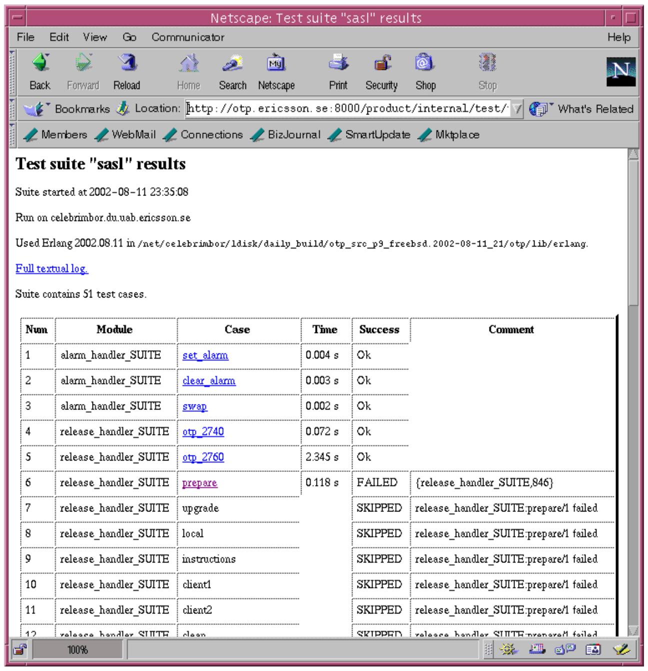
Family	Successful	Failed	Skipped
Test_server suite	11	0	2
Emulator suite	368	1	4
Kernel suite	361	4	6
Stdlib suite	419	0	0
Compiler suite	49	0	0
System suite	7	1	1
Erl_interface suite	84	0	1
Sasl suite	41	2	8
Tools suite	40	3	1
Pman suite	0	0	1
Debugger suite	0	0	0
Toolbar suite	0	0	1

Figure 1: Test run overview produced by OTP's Test Server

From the high-level overview of a test run, the designer can drill down into each test suite and further inspect the results. For each test case, the result, execution time and possible error identification are presented, along with a link to the test case output and source code (see Figure 2).

[1] AT&T does not actually use AXD 301 to deliver this service; it is simply an example to illustrate the point. AXD 301 was designed to provide this type of service quality.

[2] As each interface board in the AXD 301 is capable of handling 622 Mbps, it is entirely possible to service many more subscribers than this on one board.



http://otp.ericsson.se:8000/product/internal/test/

Test suite "sasl" results

Suite started at 2002-08-11 23:35:08

Run on celebrimbor.du.uab.ericsson.se

Used Erlang 2002.08.11 in `/net/celebrimbor/ldisk/daily_build/otp_src_p9_freebsd.2002-08-11_21/otp/lib/erlang`.

Full textual log.

Suite contains 51 test cases.

Num	Module	Case	Time	Success	Comment
1	alarm_handler_SUITE	set_alarm	0.004 s	Ok	
2	alarm_handler_SUITE	clear_alarm	0.003 s	Ok	
3	alarm_handler_SUITE	swap	0.002 s	Ok	
4	release_handler_SUITE	otp_2740	0.072 s	Ok	
5	release_handler_SUITE	otp_2760	2.345 s	Ok	
6	release_handler_SUITE	prepare	0.118 s	FAILED	{release_handler_SUITE,846}
7	release_handler_SUITE	upgrade		SKIPPED	release_handler_SUITE:prepare/1 failed
8	release_handler_SUITE	local		SKIPPED	release_handler_SUITE:prepare/1 failed
9	release_handler_SUITE	instructions		SKIPPED	release_handler_SUITE:prepare/1 failed
10	release_handler_SUITE	client1		SKIPPED	release_handler_SUITE:prepare/1 failed
11	release_handler_SUITE	client2		SKIPPED	release_handler_SUITE:prepare/1 failed
12	release_handler_SUITE	clean		SKIPPED	release_handler_SUITE:prepare/1 failed

Figure 2: Test suite overview

A process evolved: for each bug found in the OTP middleware, a test case was to be written that reproduced the error (see e.g. Figure 2 above, test cases otp_2740 and otp_2760). Once the bug was fixed, the solution should be verified using the new test case, and regression-tested using other test cases. The new test case was then included in the standard test suite for future test runs. The OTP test suites now comprise more than 3,000 test cases that are executed nightly on 5–10 different platforms.

The AXD 301 project eventually based its test automation on the OTP test server. To date, more than 10,000 test cases have been developed and now form a collection of automated test suites for AXD 301.

2.4 AXD 301's Custom Tools

Erlang proved a highly useful vehicle for designing lab tools. Testers have designed tools for administering the test labs, automatically configuring the test systems, generating different traffic patterns, and running customized long-term stability tests.

Design teams rely on Erlang's excellent support for executing[3] distributed systems in a workstation environment, and run the same installation procedures on their own workstation as those run in a target environment. Many test suites execute just as well in the design environment as they do in the target system, and designers are even able to run through software upgrade tests on their own workstation.

System management uses Erlang to design custom profiling and debugging tools for their own purposes. Many of these have made their way into the actual system and are available via the web-based operator interface.

[3] Using the word "simulating" is hardly appropriate, as the software executes in almost exactly the same manner as in the target environment; apart from using UDP instead of aal5 for some drivers, there is no extra emulation layer.

2.5 Putting it Together

In 1996, when the AXD 301 and OTP projects were started, no tools were available to augment design and testing. Still, anecdotal evidence indicated a significant productivity increase compared to mainstream technology[10].

Through the years, tools have been developed by designers, testers and system management that raise the level of productivity even further and increase the precision in the development and testing cycle. Erlang's tracing support and error reporting have improved significantly.

Using the OTP test server, designers write their own test cases that are then used for block testing, then put together into a larger subsystem test suite, all performed in the interactive design environment. Moving into the target environment, the test organization performs initial integration, after which the designers' test suites are reused for function testing – carried out by the designers themselves. After this, the test organization performs system tests, stress tests, long-term stability tests, etc., reusing the original test suites again in new combinations and adding their own test cases written by testers.

All considered, Erlang/OTP, the OTP test server, and custom tools form an integrated whole where the line between design and test becomes blurred. Designers experience increased productivity and become an integral part of the testing process (as it should be), and testers quickly become proficient programmers in their own right, designing test tools and test suites.

Finally, Erlang's excellent symbolic error reports and tracing facilities are powerful aides in locating and correcting faults that occur. Even here, testers become empowered by the close integration between test tools and the applications, and often assist in profiling and optimizing the system. This serves to reduce lead times and increase delivery precision. Furthermore, designers and testers are able to take great pride in their work.

3. Fault localization in Erlang

Taking a closer look at how fault localization is performed in Erlang/OTP, we will start with intrinsic language features, while describing some simple design techniques. Moving on to actual faults, we describe how errors are presented by the system, and how designers and testers can work to find the problem.

3.1 Language features

3.1.1 Pattern matching

Erlang programming is based in large part on describing patterns that are then matched and enforced by the Erlang runtime system.

Example 1: Pattern matching in function head

```
factorial(N) when integer(N), N>0 ->
   N * factorial(N-1);
factorial(0) ->
   1.
```

The function in Example 1 above will only accept an integer as input. Anything else will immediately trigger an exception. The same goes for expressions inside a function body. In Example 2 below, the function write_to_file/2 can be relied upon to either output Data to Filename, or exit:

Example 2: Pattern matching within function body

```
write_to_file(Filename, Data) ->
   {ok, Fd} = file:open(Filename, write),
   ok = io:fwrite(Fd, "~p.~n", [Data]),
   ok = file:close(Fd).
```

The write_to_file/2 function above contains a series of checks:

1. The function file:open/2 returns {ok, Fd} if, and only if, the file given by Filename could be opened with write access.
2. The function io:fwrite/3 will nowadays exit upon failure, so asserting that it returns 'ok' is actually redundant.
3. Finally, it is asserted that the file referenced by file descriptor Fd was properly closed.

This is a very common way to write Erlang functions, and proves highly useful when designing test cases. Assertions are implicit in the Erlang syntax, and a test case will either run to completion passing all built-in assertions, or fail with a verbose error description. In Example 3 below, taken from the OTP test suite for built-in storage structures, pattern-matching and concurrency support are used to verify that the access rights work as they should (indentation modified to fit publication format):

Example 3: Test case from OTP's 'ets' test suite

```
privacy(doc) ->
    ["Privacy check. Check that a "
     "named(public/private/protected) table "
     "cannot be read by the wrong process(es)."];
privacy(suite) -> [];
privacy(Config) when list(Config) ->
    ?line process_flag(trap_exit,true),
    ?line Nosy =
            spawn_link(?MODULE,nosy,[self()]),
    receive
        {'EXIT',Nosy,Reason} ->
            ?line exit({privacy_test,Reason});
        ok ->
            ok
    end,

    %% check read rights
    ?line [] = ets:lookup(pub, foo),
    ?line [] = ets:lookup(prot,foo),
    ?line {'EXIT',{badarg,_}} =
            (catch ets:lookup(priv,foo)),

    %% check write rights
    ?line true = ets:insert(pub, {1,foo}),
    ?line {'EXIT',{badarg,_}} =
            (catch ets:insert(prot,{2,foo})),
    ?line {'EXIT',{badarg,_}} =
            (catch ets:insert(priv,{3,foo})),

    %% check that it really wasn't written, either
    ?line [] = ets:lookup(prot,foo),

    Nosy ! die,
    receive
        {'EXIT',Nosy,_} ->
            ok
    end,
    ok.
```

```
nosy(Boss) ->
    Pub  = ets:new(pub, [public,named_table]),
    Prot = ets:new(prot,[protected,named_table]),
    Priv = ets:new(priv,[private,named_table]),
    Boss ! ok,
    receive
        die ->
            ok
    end.
```

3.1.2 Process supervision

All execution of Erlang programs takes place in dedicated lightweight threads – processes. An exception causes the process in question to die, not necessarily affecting other processes in the system. Processes communicate with each other using asynchronous message passing, and the receiving process can apply pattern matching on the message queue to facilitate selective message reception and prioritization of messages.

Furthermore, processes can be linked. When a process dies, all processes linked to it will also die ("cascading exit"), unless they have been configured to trap exits, in which case they will receive an exit notification as a regular message. This mechanism is used to implement special supervisor processes, which make sure that the system is able to heal from partial failure. A supervisor will normally restart a crashed process a (configurable) number of times, before failing in its turn, escalating the problem to the next higher level.

Since processes crashing do not bring down the rest of the system, faults can be trapped by error reporting software and presented in a user-friendly fashion.

3.2 Error reports

Erlang is a *symbolic* language, and data structures can thus easily presented in a very accessible format. This greatly simplifies interpretation of error reports and inspection of storage structures.

A standard error report contains information about type of fault, the name of the function and identity of the process where the fault occurred, and a call chain helping the designer to further clarifying the root cause. See examples in chapter 3.4 below.

3.3 Patching, code change

Once an error has been located to a certain module, that module can be modified, recompiled and inserted into a running system. Processes executing within that same module can continue running the old version of the code, either until completion or until they elect to migrate smoothly to the new version. This allows for both interactive fault localization and rapid correction of minor faults.

3.4 Small Example

A small example will serve to illustrate many of the above mentioned features. The following program implements a simple adder, using OTP's generic server behavior.

Example 4a: Simple client-server program

```
-module(adder).
-behaviour(gen_server).

%% Only exported functions are callable from
%% outside the module
-export([start_link/1,
         add/2,
         read/1]).

%% gen_server-specific exports
-export([init/1,
         handle_call/3,
         code_change/3,
         terminate/2]).

start_link(InitialValue) ->
    gen_server:start_link(Module = adder,
                          InitialValue,
                          Options = []).

%% Client functions
add(Adder, Value) ->
    gen_server:call(Adder, {add, Value}).

read(Adder) ->
    gen_server:call(Adder, read).

%% Server callback functions
init(InitialValue) ->
    {ok, InitialValue}.

handle_call({add, N}, From, Value) ->
    NewValue = Value + N,
    {reply, NewValue, NewValue};
handle_call(read, From, Value) ->
    {reply, Value, Value}.

terminate(Reason, Value) ->
    ok.
```

The following Erlang session illustrates a way to compile and test the above example by hand. The indentation of the output has been slightly edited to fit the page layout and comments have been inserted to clarify the sequence:

Example 4b: Interactive compile-and-test session

```
Erlang (BEAM) emulator version ... [threads:0]
Eshell V...  (abort with ^G)
```

First, start the SASL application[4] for descriptive error and progress reporting:

```
1> application:start(sasl).
ok
2>
```

Some output snipped; the progress reports tell of processes that have been started. The reports include process identifier, start function (module, function and arguments, or "mfa"), as well as some process supervision options.

```
=PROGRESS REPORT==== 8-Jul-2002::22:02:17 ===
    supervisor: {local,sasl_sup}
       started: [{pid,<0.47.0>},
                 {name,release_handler},
                 {mfa,{release_handler,
                       start_link,[]}},
                 {restart_type,permanent},
                 {shutdown,2000},
                 {child_type,worker}]

=PROGRESS REPORT==== 8-Jul-2002::22:02:17 ===
   application: sasl
    started_at: nonode@nohost

2> cd("c:/dev/erlang/sigplan/src").
c:/dev/erlang/sigplan/src
ok
```

Compile the adder module:

```
3> c(adder).
{ok,adder}
```

The module is now compiled and loaded. Now start the server process, and try the client functions from the shell prompt. The shell uses the same pattern matching syntax as the Erlang language:

```
4> {ok,Adder} = adder:start_link(17).
{ok,<0.66.0>}
5> adder:add(Adder,3).
20
6> adder:add(Adder,-3).
17
7> adder:read(Adder).
17
```

So far— so good. Now, try an illegal client request:

```
8> adder:add(Adder,b).

=ERROR REPORT==== 8-Jul-2002::22:17:12 ===
** Generic server <0.66.0> terminating
** Last message in was {add,b}
** When Server state == 17
** Reason for termination ==
** {badarith,[{adder,handle_call,3},
             {proc_lib,init_p,5}]}
** exited: {badarith,[{adder,handle_call,3},
                      {proc_lib,init_p,5}]} **
```

The above error report, generated by the Generic Server framework, warns that the server is about to terminate, then presenting some useful information, such as a function call trace, the internal state of the server, and the last message received by the server. This shows us that the server crashed in the function adder:handle_call/3 (in

[4] SASL = System Architecture Support Libraries, part of OTP

order to handle the message {add, b}). The reason for the crash is listed as ‘badarith’ — clearly from trying to add ‘b’ to 17. What follows below is the standard SASL crash report, generated by the supervisor once it detects the death of a child. Here, we are given additional information, such as process meta-data for the child and its neighbors.

```
9>
=CRASH REPORT==== 8-Jul-2002::22:17:12 ===
  crasher:
    pid: <0.66.0>
    registered_name: []
    error_info: {badarith,
                 [{adder,handle_call,3},
                  {proc_lib,init_p,5}]}
    initial_call: {gen,init_it,
                   [gen_server,<0.63.0>,
                   <0.63.0>,adder,17,[]]}
    ancestors: [<0.63.0>]
    messages: []
    links: [<0.63.0>]
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 610
    stack_size: 23
    reductions: 139
  neighbours:
    neighbour: [{pid,<0.63.0>},
                {registered_name,[]},
                {initial_call,
                 {shell,evaluator,3}},
                 {current_function,
                  {gen,wait_resp_mon,3}},
                {ancestors,[]},
                {messages,[]},
                {links,[<0.22.0>,<0.66.0>]},
                {dictionary,[]},
                {trap_exit,false},
                {status,waiting},
                {heap_size,233},
                {stack_size,22},
                {reductions,293}]
```

The main bug in this program was of course allowing the client to send invalid data to the server. At the very least, the client should die instead of the server. A quick fix is to add a type guard to the client function:

Example 4c: Modification of client-server program

```
add(Adder, Value) when integer(Value) ->
    gen_server:call(Adder, {add, Value}).
```

This will give another type of behavior:

Example 4d: Compile and test of modified program

```
33> adder:add(Adder,b).

=ERROR REPORT==== 23-Jul-2002::21:44:37 ===
Error in process <0.94.0> with exit value:
{function_clause,[{adder,add,
                   [<0.88.0>,b]},
                  {erl_eval,expr,3},
                  {erl_eval,exprs,4},
                  {shell,eval_loop,2}]}
```

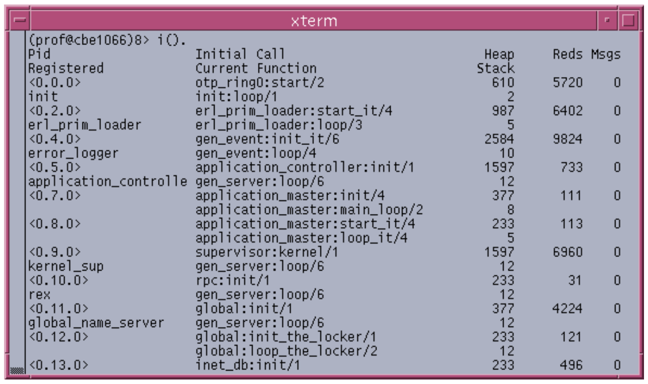
The error report shows that there was no matching pattern for the function call adder:add(<0.88.0>, b). It also shows a function call chain — this one is typical for the shell evaluator.

```
** exited: {function_clause,
            [{adder,add,[<0.88.0>,b]},
             {erl_eval,expr,3},
             {erl_eval,exprs,4},
             {shell,eval_loop,2}]} **
```

This is perfectly in line with the philosophy of Erlang programming: It is acceptable to let the process crash due to a programming error (in this case, faulty input, presumably from another program module). The OTP supervision framework (not used in the above example) takes care of restarting the crashed process, and the detailed error reports make it straightforward to find and correct most errors. Through thoughtful use of pattern matching, the program can be made to clearly describe — and at the same time assert — the correct behavior. This greatly reduces the need for error handling code in the program flow. Programmers accustomed to other languages may find it difficult to accept this style of programming, but tend to find that it leads to shorter and clearer programs and greatly simplifies testing.

3.5 Diagnosing faults

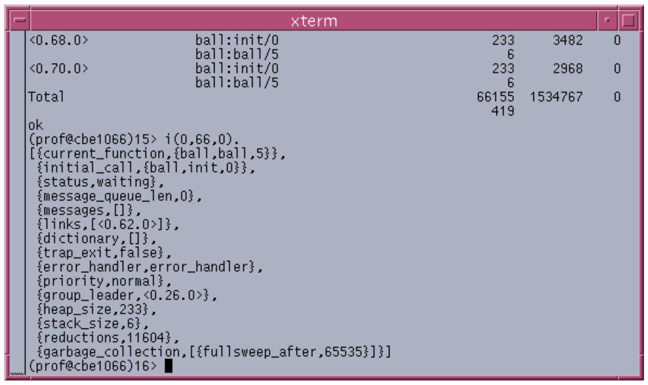
Programming faults are usually relatively easy to locate, as long as error reports are generated. A more difficult class of problem is when a process consumes too much memory or processing power, or bottlenecks appear in the system. In the following, a few of the most basic, and most widely used, diagnostic tools are described.



```
(prof@cbe1066)8> i().
Pid                    Initial Call                           Heap     Reds Msgs
Registered             Current Function                      Stack
<0.0.0>                otp_ring0:start/2                       610     5720    0
init                   init:loop/1                               2
<0.2.0>                erl_prim_loader:start_it/4              987     6402    0
erl_prim_loader        erl_prim_loader:loop/3                    5
<0.4.0>                gen_event:init_it/6                    2584     9824    0
error_logger           gen_event:loop/4                         10
<0.5.0>                application_controller:init/1          1597      733    0
application_controlle  gen_server:loop/6                        12
<0.7.0>                application_master:init/4               377      111    0
                       application_master:main_loop/2            8
<0.8.0>                application_master:start_it/4           233      113    0
                       application_master:loop_it/4              5
<0.9.0>                supervisor:kernel/1                    1597     6960    0
kernel_sup             gen_server:loop/6                        12
<0.10.0>               rpc:init/1                              233       31    0
rex                    gen_server:loop/6                        12
<0.11.0>               global:init/1                           377     4224    0
global_name_server     gen_server:loop/6                        12
<0.12.0>               global:init_the_locker/1                233      121    0
                       global:loop_the_locker/2                 12
<0.13.0>               inet_db:init/1                          233      496    0
```

Figure 3: Process listing in Erlang

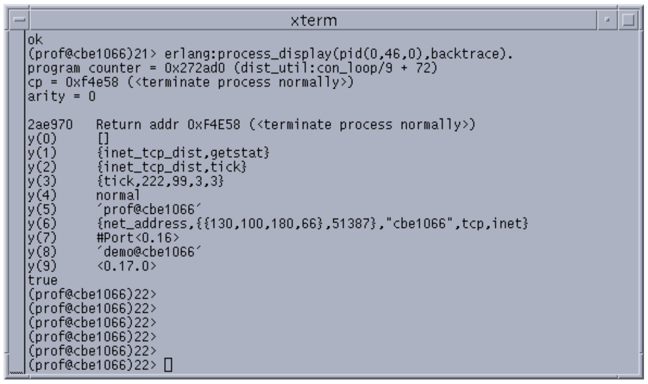
Figure 3 illustrates one of the many helper commands available in the Erlang shell: listing all active processes together with registered name, current function, message queue length, etc.



```
<0.68.0>              ball:init/0                               233     3482     0
                      ball:ball/5                                 6
<0.70.0>              ball:init/0                               233     2968     0
                      ball:ball/5                                 6
Total                                                         66155  1534767     0
                                                                419
ok
(prof@cbe1066)15> i(0,66,0).
[{current_function,{ball,ball,5}},
 {initial_call,{ball,init,0}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.62.0>]},
 {dictionary,[]},
 {trap_exit,false},
 {error_handler,error_handler},
 {priority,normal},
 {group_leader,<0.26.0>},
 {heap_size,233},
 {stack_size,6},
 {reductions,11604},
 {garbage_collection,[{fullsweep_after,65535}]}]
(prof@cbe1066)16> █
```

Figure 4: Individual process information

Figure 4 illustrates how any given process can be inspected, revealing plenty of process meta-data. As most processes in a massively concurrent system are idle most of the time, it often pays to look at the current function and focus on processes that seem to wait for something (could be a sign of deadlock). Being able to inspect the process stack may, even though it is difficult to interpret, provide a clue (Figure 5):



```
ok
(prof@cbe1066)21> erlang:process_display(pid(0,46,0),backtrace).
program counter = 0x272ad0 (dist_util:con_loop/9 + 72)
cp = 0xf4e58 (<terminate process normally>)
arity = 0

2ae970   Return addr 0xF4E58 (<terminate process normally>)
y(0)     []
y(1)     {inet_tcp_dist,getstat}
y(2)     {inet_tcp_dist,tick}
y(3)     {tick,222,99,3,3}
y(4)     normal
y(5)     ´prof@cbe1066´
y(6)     {net_address,{{130,100,180,66},51387},"cbe1066",tcp,inet}
y(7)     #Port<0.16>
y(8)     ´demo@cbe1066´
y(9)     <0.17.0>
true
(prof@cbe1066)22>
(prof@cbe1066)22>
(prof@cbe1066)22>
(prof@cbe1066)22>
(prof@cbe1066)22>
(prof@cbe1066)22> ▯
```

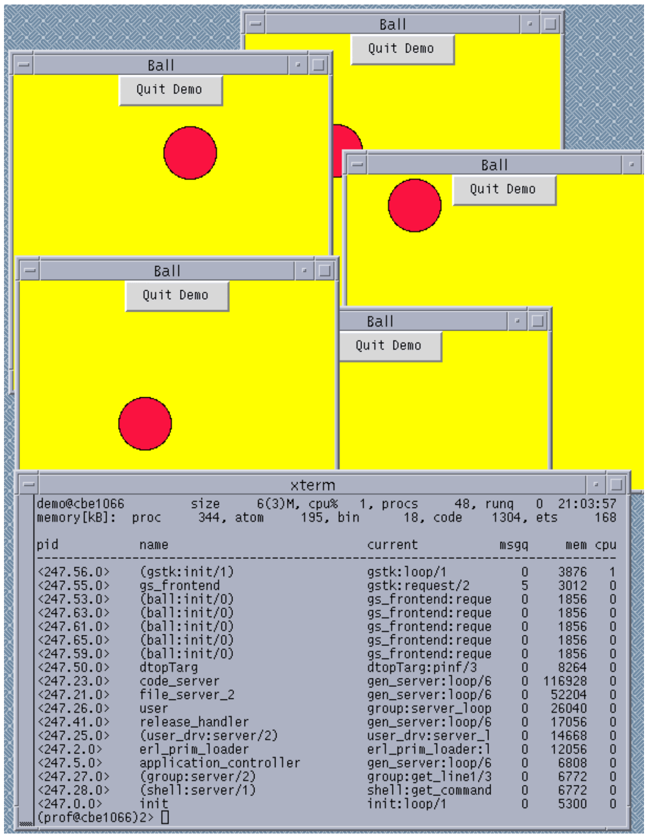
Figure 5: Example of process stack backtrace

In this case, nothing to worry about. The data structures shown in the output mainly represent state variables.

Quite often, it is important to locate the "hot spots": processes that execute often, or consume large amounts of memory. Working interactively in the shell of the Erlang VM being studied tends to disturb the real-time characteristics one wishes to study. One of the tools developed by AXD 301 System Management is a small "top-like" program that remotely inspects and displays the status of the most active processes in an Erlang node (see Figure 6).

More complex tools monitor garbage collection or function calls, making it possible to do advanced profiling with little effort. Erlang has extensive support for tracing, and trace filters can be set interactively in order to limit the trace output. Even sequence trace exists, allowing a "trace token" to follow a message from process to process. In our experience, tools that activate a set of pre-programmed trace functions tend to be favored by our testers and designers, as they provide much information rather easily.

While a more conventional graphical debugger with breakpoints and stepwise execution exists, it is often of limited value when debugging real-time applications.



```
demo@cbe1066          size     6(3)M, cpu%   1, procs     48, runq   0  21:03:57
memory[kB]:  proc     344, atom     195, bin      18, code   1304, ets      168

pid             name                           current              msgq      mem cpu
-------------------------------------------------------------------------------------
<247.56.0>      (gstk:init/1)                  gstk:loop/1             0     3876   1
<247.55.0>      gs_frontend                    gstk:request/2          5     3012   0
<247.53.0>      (ball:init/0)                  gs_frontend:reque       0     1856   0
<247.63.0>      (ball:init/0)                  gs_frontend:reque       0     1856   0
<247.61.0>      (ball:init/0)                  gs_frontend:reque       0     1856   0
<247.65.0>      (ball:init/0)                  gs_frontend:reque       0     1856   0
<247.59.0>      (ball:init/0)                  gs_frontend:reque       0     1856   0
<247.50.0>      dtopTarg                       dtopTarg:pinf/3         0     8264   0
<247.23.0>      code_server                    gen_server:loop/6       0   116928   0
<247.21.0>      file_server_2                  gen_server:loop/6       0    52204   0
<247.26.0>      user                           group:server_loop       0    26040   0
<247.41.0>      release_handler                gen_server:loop/6       0    17056   0
<247.25.0>      (user_drv:server/2)            user_drv:server_l       0    14668   0
<247.2.0>       erl_prim_loader                erl_prim_loader:l       0    12056   0
<247.5.0>       application_controller         gen_server:loop/6       0     6808   0
<247.27.0>      (group:server/2)               group:get_line1/3       0     6772   0
<247.28.0>      (shell:server/1)               shell:get_command       0     6772   0
<247.0.0>       init                           init:loop/1             0     5300   0
(prof@cbe1066)2> ▯
```

Figure 6: Erlang-based "distributed top"

4. Characteristics of Test Automation Programs

Test automation is predominately about running a series of "black box" test sequences, continuously verifying that the system's response to given stimuli falls within specifications.

Regardless of which implementation techniques are selected, some common requirements can be identified:

- Program control of stimuli on the network is essential
- A lot of activities must be controlled concurrently
- Supervision of tools is needed
- Co-ordination of traffic and synchronisation of timers
- Collection of logs from several sources
- Monitoring the liveness of the System Under Test

4.1 Requirements on Test Automation Tools

4.1.1 General middleware stuff

A framework is needed where several different types of control data can be handled, in the worst case one type for each tool that is connected to the System Under Test. This means that a group of protocol converters must be programmed. The framework must also be open in the sense that one more protocol converter (for a new tool) can be added with only minor effects on the existing converters. In the same vein it must be possible to change existing converters without having to rewrite all middleware. A modular approach is preferable.

Several design decisions must be made. One concerns the balance between a set of library functions and a set of specialised programs, one for each individual task. This is probably even more relevant for programming of the Test Cases that are to be run by the Test Executor. Another important design decision concerns the choice of programming language. There is no obvious relation between the programming language in which the tools, including the Test Executor, are programmed, and the programming language in which the Test Cases are written. From a maintenance point of view it is an advantage if they are the same, provided that there is no penalty in terms of speed or other performance factors.

In the case of Erlang/OTP and AXD 301, the test environment is essentially the same as the development and execution environment for the product. The same language is used for product, testing framework, and test cases.

4.1.2 *Concurrency stuff*

It is mandatory that concurrency (both in execution and control) is provided from the start. The simplest way to achieve this is to utilise a programming language with native concurrency built-in. Two examples of such languages are TTCN and Erlang. If concurrency is not part of the language libraries have to be built that emulate native concurrency. Or scheduling has to be programmed from scratch, no small task!

An asynchronous model of communication between the Test Executor and the Test Tools (and likewise between the whole Test System and the System Under Test) is preferable. The reason is that it is generally more robust and also corresponds better to the real conditions of target telecom and datacom systems. A reactive system being exposed to stochastically varying traffic conditions.

Erlang was designed to support massive concurrency with thousands of threads, possibly distributed across multiple virtual machines. Asynchronous message passing and selective message reception offer excellent support for design of e.g. protocol handlers.

4.1.3 *Tool supervision stuff*

Many tools, like TSS2000, can be controlled and supervised remotely, most of such control is carried over TCP/IP. For some tools, like MGTS from Tekelec, which needs emunet, it is necessary to have additional tools that are used to handle connections in a network.

Generally it is necessary to have at least a set of Start and Stop commands available to the Test Executor for each tool. In such commands it is useful to be able to send a text string that specifies the name of the program to be executed, perhaps also additional parameters that may be needed. If such basic communication is not available a set of predefined programs must be used and the co-ordination task becomes less flexible. The best situation is that not only Start and Stop can be controlled but also all operation of the tool can be monitored continuously, and parameters be changed at runtime.

With Erlang, all aspects of the system can be controlled and supervised remotely. The normal way to test the AXD 301 switch is to attach a test server to the control system (2-32 control processors) using normal Distributed Erlang mechanisms. This allows the test server to remotely call any exported function in any module in the control system, as well as communicate with any process in the distributed control system.

4.1.4 *Traffic coordination stuff*

If “both-side-connection” model for a call is used both the A-side (originating side) and the B-side (terminating side) of the call must be co-ordinated. Someone must keep control over which resources are being seized and busy-mark them (or follow some other algorithm to deliver unused resources to new calls). This calls for complicated programs, perhaps involving a lot of internal signalling. This complication can be avoided to some extent by designating groups of A-subscribers and B-subscribers, as has been demonstrated by Ericsson Traffic Lab in Hungary[11]. A call is directed towards such a named group. Simple statistical methods decide if the call shall be answered or not.

A more radical way is to choose the “half-connection” model first suggested by Däcker et al. during the POTS experiments[12] that preceded the birth of Erlang. This is the model used in AXD 301.

4.1.5 *Log handling stuff*

A set of Text files with logs from the different tools must be collected, transformed into common notation so that useful data can be extracted concerning traffic load, link usage, number of successful calls, cell loss etc. These logs will have to be combined with data from Test Case Execution and perhaps also with collected internal data from the System Under Test concerning such data as control processor load, handling of overload and congestion.

Erlang/OTP provides comprehensive support for logging, including support for exporting and browsing logs. Ubiquitous libraries for efficient data storage, counters, etc, combined with highly developed tracing support simplify the collection of system data.

4.1.6 *Robustness stuff*

The classical problem of Testers and Designers fighting over who is to blame when an error has occurred (“There is a programming fault in the switch!” — “No, there is a fault in the Load Tester, it sends bad protocol data!”) should be avoided. The only way to avoid this is to look upon Test Programs as vital parts during Design, parts that must also be thoroughly tested and maintained, preferably under configuration control like Clearcase.

Regarding the whole Test System it should be based on sound principles of modularity and fail-safe operation. When faults appear in one Test Tool controlled by the Test Executor the other tools should not be affected. The Test Executor should be designed with extra safety measures so that it can handle weeklong stability tests. This probably involves Self-Repair and automatic restarts if too many faults occur.

This is certainly one of Erlang’s strongest points. It was designed from the ground up to simplify the construction of extremely robust, self-healing systems. In AXD 301, all test suites are version-controlled together with the applications.

4.1.7 *Maintenance stuff*

Configuration Control is key. But also Debug Control must be possible to add (perhaps by recompilation with additional debug flags set) if problems occur later, for example during regression testing.

Erlang/OTP has excellent support for tracing. Function calls, message passing, process scheduling and garbage collection can be monitored, and complex filters can be activated in order to limit the generation of trace messages. All trace output can be timestamped and directed to the screen, disk or network.

4.1.8 *Protocol extensibility stuff*

It is a great advantage if the language with which Test Cases are designed has support for the data description language ASN.1. That means that drivers and protocol converters can be created with a minimum of effort, using the protocol definitions directly to derive parsers. ASN.1 descriptions of many standard protocols can be found in the public domain or requested from standardisation bodies.

Erlang/OTP supports ASN.1, IDL and XML[5].

4.1.9 *Platform and implementation stuff*

UNIX (or Linux) is the preferred implementation platform due to uncontested stability. Multi-platform support is even better. It is an open question how tools should communicate internally and with the Test Executor. There were requirements during the early BRAVE days to use CORBA. For real-time control there is a penalty to pay with the overhead caused by CORBA.

Erlang runs on several UNIX dialects, as well as on Win32, VxWorks and (soon) OSE Delta. Object code and distribution protocol are identical on all platforms, making it possible to combine platforms into a heterogeneous distributed network.

5. Conclusion

The same features that make Erlang/OTP an excellent development platform for Telecom products also make it ideal for development of test automation systems. When using the same tools for product development and test automation, exciting synergies between designers and testers can be explored. Designers, often reluctant to get involved in test suites, suddenly find themselves writing test cases, and testers can get more actively involved in fault localization and profiling. Erlang's expressive power also helps reduce the need for specification languages, whose main purpose is to communicate intentions between system management, design and test.

The addition of powerful support for test automation can be expected to further increase the leverage provided by Erlang/OTP for designing complex carrier-class products.

[5] XML is supported through a selection of Open Source contributions.

6. REFERENCES

[1] 1/1056-FCPBT 103 104 Uen, "MAIN REQUIREMENT SPECIFICATION FOR A TEST MANAGEMENT SYSTEM", Simon Hoff's RS for TMS, including Test Automation

[2] Report on LXJ/Vs stuff earlier (Phase 7 PDC)

[3] Thomas Arts and Clara Benac Earle, "Development of a verified Erlang Program for Resource Locking", (is available at http://www.ericsson.com/cslab/~thomas/publ2.shtml)

[4] See http://www.prover.com/research/research-papers.xml

[5] Magnus Fröberg, "Automatic Code Generation from SDL to a Declarative Programming Language", http://www.ericsson.com/cslab/publications/sdl2erlang.ps

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

[7] Lucent Meeting 3/2/00, see http://www.slac.standford.edu/grp/trip/lucent-mar02-2000.html

[8] "Taqua in the news", http://www.taqua.com/news/press_telecomclick.asp

[9] Strata Incorporated, http://www.stratainc.com/private.htm

[10] Ulf Wiger, "Four-fold increase in productivity and quality", FemSYS 2001, München, http://www.erlang.se/publications/Ulf_Wiger.pdf

[11] ETH/RL-2001:0137, Rev A, "Automated System Test Environment for GSM on the Net"

[12] B. Däcker, N. Elshiewy, P. Hedeland, C.W. Welin, M.C. Williams, "Experiments with Programming Languages and Techniques for Telecommunication Applications" (6th Intl Conference on Software Engineering for Telecommunication Switching Systems, Eindhoven 1986)