6th International Erlang/OTP User Conference

Stockholm, October 3, 2000

Proceedings

EUC'2000 `http://www.erlang.se/euc/00/`
Open Systems
Ericsson Utvecklings AB
P.O. Box 1505
SE-125 25 Älvsjö Stockholm
Sweden

ERICSSON

ERLANG

08.30 *Registration.*

Applications Session.

09.00 **Use of Erlang/OTP as a Service Creation Tool for IN Services.** Sean Hinde, one2one, UK. Intelligent Networking requires large databases with high performance, soft real time access, very high availability, ease of management, and minimum customisation. This talk describes the experiences of a GSM mobile operator in using Erlang/OTP as a service creation tool for Number Translation, VPN, Mini Pre-pay, and authentication services.

09.30 **Sendmail Meets Erlang: Experiences Using Erlang for Email Applications.** Scott Lystig Fritchie, Sendmail Inc, USA. Our experience developing a prototype of a bulk I/O daemon, based on ONC RPC both as a server and as a client, suggested a long and difficult development cycle. Erlang lived up to its promise to make the task easier, despite the learning curve, skeptical management, and the challenging nature of the project itself.

10.00 **MPowered by Erlang.** Per Bergqvist, CellPoint. Finder! offers multiple interfaces to the end-user, while providing a uniform look and feel for the service that is easy to use. The user is always in full control of their privacy. Finder! supports standard GSM phones and WAP phones, and can also be accessed via the Internet, providing for mass-market services today and in the future.

10.30 *Refreshments.*

11.00 **NETSim - Six Years with Erlang.** Bengt Tillman, Ericsson Radio. Ericsson develops many different applications for the controlling and configuring of telecommunication networks. Alarm handling and the generation of statistics files are two such applications which have to be tested on very large networks (hundreds of AXE exchanges). This talk explains how Erlang has been used in order to make a simulator which solves this huge testing task.

Implementations Session.

11.30 **A High Performance Erlang System.** Mikael Pettersson, Uppsala University. HiPE (*High Performance Erlang*) is an optimising native code compiler for Erlang under development at Uppsala University. The compiler offers selective compilation and produces significant speedups over emulated code. This talk will describe the architecture and main features of the HiPE system. HiPE is available open source at `www.csd.uu.se/projects/hipe/osh`.
In Proceedings of *ACM SIGPLAN. International Conference of Principles and Practices of Declarative Programming.*

12.00 **ECOMP - an Erlang Processor.** Robert Tjärnström and Peter Lundell, Ericsson Telecom. An Erlang processor has been built in an FPGA (i.e. programmable hardware). The JAM compiler has been changed to generate native code which allows Erlang programs to be run directly on the processor without any OS and with improved performance. Furthermore, results from experiments with call control software will be presented.

12.30 *Lunch.*

Developments Session I.

13.45 **An Erlang DTD.** Richard A. O'Keefe, Otago University. The Erlang DTD supports literate programming, so that the documentation can be automatically extracted from the master file, and supports better cross referencing. If the marked up source code gets through the SGML parser, there isn't much left for an Erlang parser to check. There is one program to convert Erlang source code to marked up SGML or XML, and another to extract source code from a marked up document.

14.30 **XMErl - Interfacing XML and Erlang.** Ulf Wiger, Ericsson Telecom. XML (*Extensible Markup Language*) is an emerging standard for handling structured data on the Internet. This talk describes how XML can be combined with Erlang in an almost seamless fashion. Examples will show how to parse XML documents, and how to generate true XML or HTML documents from native Erlang data structures.

15.00 **Extending Erlang with Structured Module Packages.** Richard Carlsson, Uppsala University. An Erlang system at present is a flat structure where all modules are on the same level. This causes difficulties for really large systems. This talk will present a proposal based on the package system from Java and shows how it can be introduced very smoothly into the current Erlang system.

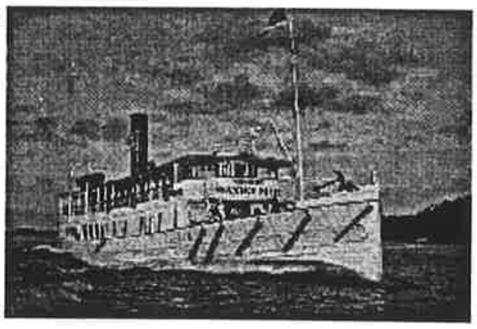
15.30 *Refreshments.*

Developments Session II.

16.00 **Highlights from Erlang 5.0/OTP R7B.** Kenneth Lundin, OTP Product Unit. The R7B release of Erlang/OTP contains a number of interesting new features and enhanced characteristics which will be presented here, such as the bit syntax, trace of local functions, CORBA services etc.

16.15 **COMET - An Erlang-to-COM Port.** Jakob Cederlund, OTP Product Unit. COMET is a library that enables Erlang to use COM-objects on the Windows platform. This gives possibilities for Erlang programs to use many services and applications on windows, such as Internet Explorer, Microsoft Office and Windows Scripting. Combining Erlang's functional semantics with COM and object-oriented systems in general will be discussed. COMET will be part of the next *open source* release of Erlang.

16.45 **The Bit Syntax - The Released Version.** Patrik Nyblom, OTP Product Unit. A powerful new feature in the new R7B release of Erlang/OTP is the *bit syntax* which enables efficient creation and matching of *binaries*. This is very useful when, for example, implementing protocol stacks in Erlang. A proposal for the *bit syntax* was presented at EUC'99 and this talk presents the actual implementation.



17.45 *Bus Transfer.*

Evening Session.

18.30 **Conference Dinner.** Conference dinner on board traditional steamship Waxholm III cruising out into the Stockholm archipelago.

Poster and Demo Session.

During refreshments and lunch.

A Toolf for Verifying Software Written in Erlang. Lars-Åke Fredlund, SICS.
Demonstration of the Erlang Verification tool to prove interesting properties of a program.

A Monitoring and Instrumentation Tool Developed in Erlang. Miguel Barreiro, Victor M. Gulias, Juan J. Sanchez, Universidade da Coruña. MONET is a monitoring and instrumentation tool developed using OTP applications which is being used in a video-on-demand server, under development at LFCIA lab, running on a Beowulf cluster. Status information is produced in XML and then transformed into different formats (HTML, WML, etc.) using Erlatron, a distributed XSLT processor also developed in Erlang using *expat* and *sablotron* libraries. .

Use of Erlang/OTP as a Service Creation Tool for IN services

Sean Hinde (One2One)

Abstract

This paper describes an approach taken to Intelligent Network (IN) service development using a combination of standard IN elements (SCP, SSF) and a set of Erlang/OTP nodes using the Mnesia distributed database to store customer records, and the Erlang language to provide complex service logic.

Introduction

The telecoms market has some unique and well documented requirements of its core network systems including:

- Soft real time
- Massively high availability - 99.999%, including planned outages!!
- Enormous complexity - GSM
- Great flexibility

One could probably say without too much fear of contradiction that these requirements are completely orthogonal! The telecoms industry has had to invent some pretty clever tools to allow the flexibility required without compromising the other requirements. Some of these have been hardware based with multiple processors locked together in some level of hot/warm standby, others have been in software engineering techniques, and yet others have been the approach taken by Intelligent Networks.

This approach is to take a high level abstraction of the core building blocks of (in this case) a telephone call. For example a simple freephone call consists of someone picking up the receiver, dialling a number, having this number translated by the network into another number, and routing the call to the destination. IN provides a set of building blocks which allow operators to intervene at each of these points and manipulate the call data at each one in a network safe manner. These building blocks are normally implemented as a gui tool for defining the logic, and normally run on a separate system to the actual telephone exchange.

The availaibility requirements are met by testing the toolset as thoroughly as practicable to ensure that even a misuse of the tools cannot bring down any part of the system, or indeed affect any call other than the one being controlled.

One of the drawbacks of this approach to date has been that the interfaces have never been fully standardised meaning that a vendors IN equipment is much more likely to work seamlessly with its own switches. At One2One we have an entirely Ericsson based GSM switching network and have Ericsson IN Service Control Points.

An architectural decision was taken at a very early stage in our IN depolyment to store the data in an external database and split the logic between the SCP and this database.

The SCP only contains the logic to drive the telecoms interfaces into the switched network and it hands off control of the "Business Logic" to the external database systems which hold the customer data. The SCPs and external databases are interfaced using c7 to TCP/IP protocol converters.

Experience of using a standard UNIX/commercial database.

Initial solutions to this problem of where to store the customer data were (and still are) based on a system written by a small in house team of IT programmers based on C code interfacing to a standard SQL database. Various problems were experienced with this system:

- One read/write failure lead to many others in a row. Solution was to do multi threading!
- Random explosions in the numbers of worker threads. Solution was a full time UNIX admin killing off zombies.
- Commercial Failover system deployed never worked correctly. Took at least 30 minutes to do a manual failover.
- CPU load extremely high. Adding more processors made it worse - turned out to be bug in some spinlock C code.
- Still slowed down periodically - Full time team of DBAs to tune and optimise.

We now have a team of 20 full time C programmers hacking code as fast as they can to try to introduce resiliency into this system.

The Problem

The conclusion drawn from this project by the telecoms designers at One 2 One was that the tools used in the IT software industry are not up to the task of fast and safe development of systems with the properties required in a telecoms network without access to the vast teams of people and months of continuous development and testing available to the telecoms vendors (Surely not - an admission that these things cost a lot to produce?!).

At the time there didn't appear to be many alternatives which were comparable with the existing GUI based IN Service Creation Environments.

The Solution?

Then one day in 1998 while browsing the Ericsson web site for fun the author stumbled across Erlang/OTP. The most noticable quality of this system was that for the first time here was a product which claimed to have almost all the properties we had been looking for in a platform which runs on standard commercial hardware. It proved a little more difficult to get throught the hype and really figure out what the platform was and exactly how it could be used. Permission was obtained to spend some time on an invesigation into whether this platform was the answer to our prayers.

It proved to be very easy to learn to program in a functional style even for a completely non programmer(!), and after a few weeks we had produced the bones of a multithreaded TCP/IP server which would receive a command from the SCP, invoke a thread to implement a piece of custom service logic (e.g. translate a number) and send the result back to the network.

The conclusion to this investigation was that there was minimal customisation required to the standard Erlang/OTP platform to fit in with the existing infrastructure. Namely:

SCP Interface – Socket based using a simple proprietary binary interface
Alarm Interface – to the existing Network Mgt System (socket based text interface)
Stats Interface – ets stats counters needed to be output to files in certain format every 15 mins
Provisioning Interface – Socket based using same protocol as the interface to the SCP
Event Logging – Socket based io of existing event_logger stream to the network management systems
Intranet based monitoring – Simple HTTP query interface to lookup a customer, view alarms, stats.

The effort required to understand the more esoteric parts of the OTP application structure took some more time and effort, but solutions to these customisations were quickly implemented and work started on the first IN service.

Applications

Two large and two small systems have been deployed to date. The first one to go live was the 4th one to be started and the 2nd one to go live was the 3rd one to be started!

Currently in service are:

- **Corporate VPN service.** This service allows short code dialling, global and per VPN black and white lists, Closed User Group functionality (On-net and Off-net calling). It will also present the short code of the calling mobile for a mobile to mobile call. Pretty much all the actual service logic ir written in Erlang rather than on the SCP. There is also a prototype WAP based telephone book service.

  Part of this system is a complete Intranet based customer care system based on the INETS web server included with Erlang/OTP. This has accounted for about 80% of the effort in writing the service. The HTML is currently being ported to a modifed version of the esp Erlang Server Pages system which allows for lists:map/2 loops containing raw HTML and local variables amongst the embedded Erlang code.

- **RADIUS Authentication Server for WAP services.** This system started out life as a database containing a record for all One 2 One customers which would include their IN service profiles. The first requirement turned out to be for authentication of Dial in WAP users so a front end was written in Erlang which implemented the RADIUS protocol. The system consists of 24 erlang nodes – 16 mnesia database servers, 2 O&M, 2 Radius nodes, 2 provisioning nodes, and 2 nodes providing other miscellaeneous functions.

  The database servers are configured as 8 pairs of nodes with data split amongst the nodes using the fragmented tables feature of MNESIA and each fragment replicated across a pair.

  So far the system has been in service for 5 months and has had a series of individual nodes fail for various software licence or hardware reasons but there has not been a single call lost.

Imminently in service:

- **Mini Prepayment system.** This consists of an IN Service which queries an Erlang/OTP based system at the start of the call to allow/disallow the call and at the end of the call to update the customer record with their usage.

  There is an Inets based query front end where one can query the database based on oldest record, or highest usage, or highest number of inbound calls first. It allows an agent to update the cutoff thresholds on an individual basis.

- **Number Translation Service.** This service was the original one planned and due to a whole host of reasons unrelated to Erlang is only now about to go into service.

Issues

In this section I present some of the difficulties and issues encountered along the way (Using Erlang/OTP R6B)

EVA Integration
Integration between the standard EVA application and built in alarm_handler alarms is non existent. In the end I gave up using this and now just send all alarms via SNMP traps using a slightly modified version of the standard alarm_handler. To be useable straight out of the box this would need to be tidied up quite a lot.

Potential for Partitioned Network
The weakest part of the deployed systems is the communication between the database nodes and the potentially serious consequences of a failure for the consistency of the database. A dual LAN arrangement has been set up using some commercial software but there is still a single point of failure in the stacked switched hub arrangement.

It would be nice to see some mechanism whereby multiple sockets could be set up between two nodes using different interfaces with load sharing/failover. Or there could be some retry mechanism before declaring the link to be down.

There must also be scope to perform some form of (perhaps assisted) recovery from a partitioned network situation in the mnesia database.

Memory Usage
The Erlang runtime system can get out of hand in terms of memory allocation. This has manifested itself during testing when a table transform of an mnesia table containing 100k small records exceeded the 1G beam process limit.

Future Work

Future work in One 2 One with Erlang/OTP is currently planned to remain solely in the telecoms domain. Projects currently underway or planned are:

Mass SMS sending tool.

Tuxedo Middleware Interface for query of systems.

Investigation into how the architecture may fit into the future 3rd Generation equivalents of Intelligent Networks

Resourcing

We have found that a single developer can develop and test all of the IN service logic and Erlang code for each of the systems mentioned above within 6 months (some more , some less).

Conclusion

The combination of using an Intelligent Network Service Control Point for driving the telecoms interfaces into a GSM network, and Erlang/OTP based systems to store customer records and provide the complex business logic of the service has proved itself to meet sufficient of the requirements of telecoms network components to be used in full commercial service.

The main reasons this is seen to be true are the existence of the mnesia database system to provide soft real time performance and real time data replication, and the suitability of the Erlang language for writing telecoms logic in such a way that any failures which do occur cannot affect other calls in progress.

Attention to the network resiliency problem and the susceptibility to excessive memory usage would result in a truly formidable system for this sort of database application.

Sendmail Meets Erlang: Experiences Using Erlang for Email Applications

Scott Lystig Fritchie, Jim Larson, and Nick Christenson: Sendmail, Inc.*
Debi Jones†
Lennart Öhman: Sjöland & Thyselius Telecom AB‡

October 3, 2000

Abstract

Our software engineering team needed to create a system that moves data from a set of legacy applications with diverse properties to data repositories scattered around the network. This system had to be highly concurrent, straightforward to extend, have high performance, and be coded rapidly by a small development staff. Because of these requirements, the authors embarked upon an experiment to write this application in Erlang. This paper describes what we did, why we did it, and what we learned over the course of our development effort. It is our hope that this chronicle may be useful to others thinking about coding in Erlang for the first time and to the incumbent Erlang community to hear an outsider's perspective on this fine language.

1 Introduction

In the fall of 1999, a development team from Sendmail, Inc. began developing a program that would act as an I/O request broker connecting a set of legacy applications to a set of distributed data repositories. Some of these legacy applications might operate under a UNIX `inetd`-like forking model, some might be multithreaded using one thread per connection, and some might be multithreaded using thread pools and event-driven programming techniques. The data passing through this program, which was creatively named the "Client Daemon", is multiplexed over several separate connections to remote data repositories. In essence, the Client Daemon acts as a "traffic cop" — marshaling, massaging, and redirecting data and data requests from a varied set of legacy applications to a set of distributed servers.

There were a significant number of additional constraints placed on this project:

- **Rapid Development.** This technology introduced some fairly radical notions of data movement and storage. The system needed to be demonstrated to work or to fail as soon as possible, to allow time for a redesign.
- **Early time to market.** Sendmail, Inc., like most software companies, is competing on "Internet time." We wanted to be able to get our system to market as quickly as possible.
- **Avoid proprietary, non-portable hardware and software.** Platforms such as VAXcluster [KLS86] or products from IBM, HP, and other vendors could achieve many of these goals using off-the-shelf proprietary solutions. However, this system must be straightforward to port to a variety of UNIX operating systems and perhaps even to Windows NT.

*`<scott@sendmail.com>`, `<npc@sendmail.com>`, and `<jim@sendmail.com>`, respectively.
†`<netbean@earthlink.net>`
‡`<lennart.ohman@st.se>`

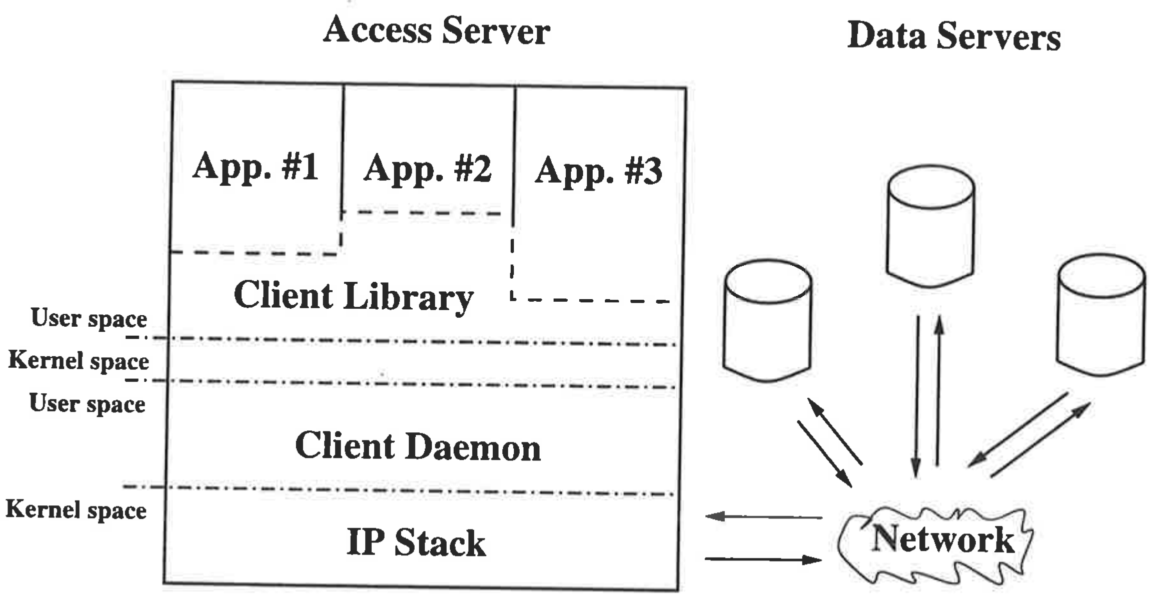


Figure 1: Process Architecture Sketch

- **High performance.** A distributed architecture will have higher overhead than a well-tuned architecture based on a single machine. Given that fundamental limitation, we want to make the overall system and each component as fast as possible. The Client Daemon initiates few data requests itself. The bulk of its work is taking data from one side (from the network or another application), massaging it, and passing it through to the other side. This means that its network interface and its IPC must be highly efficient.

- **Concurrency.** The Client Daemon will need to handle and track many simultaneous requests and responses from disparate sources. Each of these request/response pairs will take an unknown amount of time to complete.[1] In order to accommodate the various legacy applications, the Client Daemon will need to handle both synchronous and asynchronous communication modes, a notoriously difficult task in most programming languages.

- **Ease of Management.** There will be many nodes running a Client Daemon in a production environment. Therefore, we need some mechanism to tie all these systems together for monitoring and administration.

Figure 1 represents the overall architecture. The legacy applications are linked with a glue layer called the "Client Library". The Client Library uses IPC to move the data from these applications to the Client Daemon, and the Client Daemon moves the data over the network to remote Data Servers.

This paper describes the justification for and the procedures we went through to build the Client Daemon using Erlang. We chronicle our experiences, including what went well and what didn't. We provide some advice for other groups not terribly familiar with programming in Erlang and some suggestions for the Erlang community on how Erlang might be made more appealing to software development groups like ours.

[1]In fact, they may never complete. The Client Daemon must handle network timeouts in a context-sensitive manner.

2 Background

Since we and the company had developed many applications in C already, we had an intrinsic bias toward this language. Early on, we built a rudimentary prototype in C using the Libero [Lib] and SMT [SMT] tools by iMatix and using the familiar `rpcgen` and `libc`'s ONC RPC library to handle the data transport over the network to the remote Data Servers. The Libero/SMT finite-state machine approach seemed a viable way to manage the Client Daemon's growth if we decided we liked SMT, and we wouldn't lose much time if we didn't like it.

2.1 C Prototype

We confirmed our suspicions that the ONC RPC stub generator, `rpcgen`, generates some really ugly code. Furthermore, it assumes that all RPC calls are fully synchronous. Even modern `rpcgen` implementations capable of generating thread-safe stubs assume the calls are synchronous. As a result, we had to write our own RPC call mux/demux code, ignoring most of what `rpcgen` had created and bypassing much of the ONC RPC library's infrastructure.

After just over a month of coding, the Libero/SMT version of the Client Daemon was capable of interacting with both the applications and the Data Servers, although it was by no means ready for production use. At this point, we had learned a lot about RPC client stubs, RPC server stubs, XDR encoding, RPC fragment reassembly, and other RPC arcana, but the Client Daemon still had at least one obscure memory leak, its performance seemed slower than estimates predicted, and there were a great many core features that still needed to be implemented.

2.2 Erlang Discovered

Before starting the first prototype, we had conducted an exhaustive literature search to learn what lessons we could from the work of others. During this time, we came across references to the HTTP load balancer Eddie [Edd], which is written in Erlang. Erlang's built-in concurrency model was seductive: all the threading add-ons to C and C++ looked ugly by comparison. The "standard" Erlang libraries and OTP helped keep us from typing too much. Tony Rogvall gave us the source code to a full-featured ONC RPC library, allowing us to avoid a lot of XDR- and RPC-related drudge work.[2]

Also, around this time we had the opportunity to talk with a number of the software engineers at Bluetail A.B. who have written most of an email proxying system, called the Mail Robustifier [Blu], in Erlang. The fact that these folks had done significant work on an email-related project in Erlang increased our confidence in Erlang's viability. Their kindness to answer some general questions about coding such a project in Erlang helped us tremendously.

Two potential benefits of using Erlang were most appealing: higher programmer productivity and easier-to-maintain code. It was difficult to tell how many of the success stories from Ericsson and other companies using Erlang were worthwhile praise and how many were hype. Since our group was already familiar with a wide variety of arcane languages, we were quickly able to understand the reasons that these claims might be more than just smoke. We felt that if Erlang could live up to its promises, many of the goals of the project could be met much more easily with Erlang than with C, especially given our time-to-market concerns.

3 Implementation in Erlang

It's no surprise that software development managers are uncomfortable adopting new programming languages or techniques. Trying new and radical techniques with the industry's current time-to-market demands is usually a recipe for disaster. Sendmail, Inc.'s management was as skeptical as one would expect. Our dabbling with developing a second Client Daemon prototype in Erlang was greeted cautiously. Outside of our team,

[2]We have contributed our enhanced version of his package to the Erlang community. It should be available at `http://www.erlang.org/user.html` by the time of the conference.

nobody in our company had even heard of the language, much less knew anything about developing software in it.

3.1 The Second Prototype

The beginning of the learning curve was steep. The simple matter of writing a non-trivial program in a functional language is a radical change for people used to working with procedural languages such as C or Perl. We used parts of the OTP as best we understood them. The result was not pretty, but it worked.

Once we got over the startup costs of coming to terms with the new language, the Erlang prototype fell together quickly. After a month and a half, the Erlang Client Daemon had surpassed the Libero/SMT prototype in stability and feature set with comparable performance. We didn't discover any project-killing issues, so we recommended to our management that we complete our development of the Client Daemon in Erlang.

3.2 Convincing Management To Let Us Continue

Our team had the advantage of being small and closely-knit. The design and prototype development had been done in a "skunkworks"-like atmosphere: most of the company was preoccupied with other projects. Given the size of our team, the promise of writing a complex application using a relatively small number of lines of code, and our time-to-market constraints, our team agreed that the potential benefits were worth risking our project on Erlang.

Our management was skeptical about using Erlang for production code. First, they felt the discomfort normally associated with radical ideas. Second, no other developers knew it, which makes it more difficult to solicit advice or conduct code reviews in-house. Third, management felt that it might be hard to hire programmers for the team, since they would have to know both C and Erlang.

On the first and second points, we were still not entirely comfortable with the language ourselves, but both we and our management team were willing to set aside discomfort if the reasons for doing something new were compelling enough. On the issue of hiring, we pointed out that we had come up to speed fairly quickly. We felt that any programmer with an adventurous spirit could learn Erlang as quickly as we had (especially with our mentoring) and that our projections about time-to-market made this a worthwhile trade-off. Further, they were persuaded by our estimates about how quickly we could code and debug this application in Erlang as opposed to C. In the end, the rapid development schedule was able to compensate for the risk, since "If you're going to fail, fail early [Bro95]."

3.3 Further Development and Performance Tuning

We spent the next several months tightening up the code, improving Tony's RPC library, filling out features in the Client Daemon, improving the Client Library, and fixing bugs. While the core of the system was coming together, we had two outstanding concerns. The first was that we found the overall structure of the code to be vaguely unsettling, and the Client Daemon still didn't perform as well as we'd like.

Prior to starting work on this project, we had established performance goals. One of our really big concerns going into this project was that the Client Daemon would be doing a lot of data copying. There would be significant performance penalties if it could not do so efficiently. The default drivers in the Erlang emulator implement I/O via disk, pipes to spawned sub-processes, and TCP or UDP network connections. The only way to have direct communication with an unrelated UNIX process is through TCP or UDP sockets, which are not as efficient as other IPC mechanisms.

We created some crude programs that simulated how the Client Daemon might perform if it could interact with the outside world using UNIX domain sockets, through an `mmap()`-style DMA mechanism, and via other mechanisms. We saw some potential for improvement there, but we still didn't know what was consuming all of our systems' resources.

We used the Erlang profiler, `eprof`, to determine which processes were taking most of the CPU time. Unfortunately, it was difficult to measure CPU time (versus wall-clock time) consumed or a global context for the answer. We also had no insight as to how much time was spent in the runtime system for things like scheduling, memory management, linked-in drivers, and message-passing. To answer these questions, we compiled the virtual machine with `gprof` support. This gave us a global context for the performance data, but didn't give us any correlation with the Erlang code, only with the C code. For instance, we knew how much of the resources were taken up by garbage collection, but we did not know which processes or modules were producing the most garbage.

3.4 Outside Assistance

At this point, we felt that we could accelerate our progress with some outside help. Friends in the Erlang community recommended Lennart Öhman (one of the authors of this paper), an experienced Erlang developer and trainer.

After being briefed on what we were doing and why, Lennart first set out to explain current best practice regarding process hierarchy, including supervisor structure principles and process linking techniques.[3] Since the Client Daemon was not written using a strict top-down approach, but grew via more "organic" methods, we never looked at the entire process structure as a whole, and thus had a process hierarchy that was structured poorly. In hindsight, much of his advice seemed like common sense, and we probably would have figured it out eventually, but we never stumbled across it in our perusal of the current documentation and our inspection of other Erlang applications. Lennart's presentation was much more efficient than discovering the principles ourselves by trial and error.

We also received a great deal of training on parts of the OTP that we hadn't yet used. We learned quite a number of best practices for Erlang coding that we hadn't found documented anywhere or found only after we knew what to look for. Overall, this contributed a great deal to the organization of our code, making it more flexible, structured, and readable, and generally enabled us to think more clearly about its architecture.

The performance issues proved to be more difficult. We discussed several ideas on how to speed things up. One idea was to run the legacy applications as an Erlang I/O port under the Client Daemon, allowing communication with that process using a pipe rather than an IP socket. However, that technique won't work with those legacy applications that fork. We considered adding an Erlang driver that would allow zero-copy I/O of bulk data in and out of the virtual machine (in most cases), but the time to implement such a modification to the system ran against our time-to-market constraints and has been shelved for future consideration.

It was brought to our attention that there haven't been many Erlang projects that are both I/O intensive and have had externally-driven performance goals. Our application's workload seems atypical in today's Erlang usage, excepting Eddie and Bluetail's products. While it has been disappointing not to reach our *a priori* performance goals, detailed study suggests that our performance shortfall is probably related more to the UNIX process architecture and less to our language choice.

After performing code cleanup based on Lennart's suggestions, we prepared for an initial test release of the system during the summer of 2000. That release has been put on hold while we perform integration work with another complex legacy application and improve our monitoring system.

4 Lessons Learned

Working with Erlang over the past year has been educational. We've learned a number of lessons that we think are worth sharing with the rest of the community.

[3] "Imagine your application running for *ten years*. How many uncontrolled processes are you willing to tolerate?" This was a perspective we desperately needed.

4.1 Erlang is Quickly Learned

Even in isolation, a decent programmer can quickly come up to speed on the basics of Erlang, with greater ease than with many other more popular languages. With mentoring, we expect that a new person could be able to understand enough of our existing code to begin making non-trivial contributions in less than a month.

4.2 OTP Isn't So Quickly Learned

Proficiency with the OTP, however, is another matter. In our estimation, there simply isn't sufficient documentation to expect isolated programmers to make decent use of OTP on their own. We often found ourselves making up too many things as we went along. We'd make informal bets that such-and-such a problem had already been solved but we didn't know how. The `erlang-questions` mailing list [erlb] was invaluable — as long as we had a coherent question to ask — but it was hard to know what we might be missing.

If we'd been learning Erlang at Ericsson the same way we learned C, rubbing shoulders with much more experienced programmers and tackling small, self-contained projects, we would have had a much easier experience. Unfortunately, we had to immediately create the architecture for a major application. Without a mentor, the best place for education is by reading existing code of well-written applications. However, without commentary, we would expect the novice Erlang programmer to miss many subtle issues involved in employing it correctly. We certainly did. If mentoring isn't an option, we think a training course in the use of the OTP after the developers have some familiarity with the language is probably wise.

4.3 Erlang Is Good For Both Rapid Prototype and Production Code

Now that we're more proficient with Erlang, prototyping new ideas is a rapid process. Once the prototype is done, it can often be folded into production code with only small modifications.

The language, together with the standard and OTP libraries, provides an extremely useful framework. We're free to consider important operational issues from the beginning, knowing that many mundane details are already taken care of. The process linking concept, together with process supervision trees, is the most valuable, in our experience. The tools for event logging are a close runner-up. And the inter-node message-passing infrastructure is so easy to use it's hard to explain to programmers not familiar with the language.

4.4 Erlang Performs Well

Our first experiences with bulk I/O with Erlang were bad, since the original version of the RPC code moved all data as lists of bytes rather than binaries. Once we modified our code to be binary-friendly, we saw its performance increase by more than an order of magnitude. Without binary data types, though, the language's performance would have been abysmal for our application.

For a single application reading or writing a large file over the network, data throughput measures of the Erlang and C Client Daemons are identical: the performance of both is limited by network latency. When performing multiple concurrent bulk reads or writes through the same Client Daemon, the C prototype is faster, but only by a couple of percentage points. We were pleasantly surprised to find the difference so small. We have found that our performance shortfall, relative to C, is due to a small penalty per byte of bulk data in each RPC transaction, and a large penalty per transaction. We hope that the former can be addressed by a more advanced network driver or other IPC mechanisms, and that the latter can be addressed by further tuning our code.

4.5 We Would Use Erlang Again

Many advanced programming languages are useful for research but have severe deficiencies for production work. Erlang doesn't fit that pattern. Our increased productivity with Erlang more than offset the difficulties of learning the language. Aside from wanting access to UNIX domain sockets and shared memory, the language provided most of the tools we needed to develop the prototype and then expand it into a production-quality system. Indeed, we had difficulties with the packaging and distribution tools largely because they provided more functionality than those in the traditional C/UNIX environment.

We worried that the virtual machine might be too slow for our purposes. Instead, performance has not been a big issue: Erlang's performance is on par with our C prototype. The next big performance increase will come from changing the communication channel between the Client Daemon and its applications, probably using shared memory. If we were to use that scheme in both languages, we expect we would see comparable performance, and we expect the Erlang version would be finished sooner.

If we had to do this all over again, we'd still use Erlang.

5 What Erlang Needs

We've got a long wish list of things we'd like to see in future Erlang/OTP releases: enhancements to the virtual machine, new built-in functions, expanded libraries, more supported platforms, and better documentation. Fortunately, under the Erlang Public License [EPL], the source code is available for us to modify to suit our needs. Many of these things aren't tremendously difficult to do[4], and we may yet implement some of them, but code is almost always nicer if someone else writes and maintains it.

We realize many of these wishes may be fulfilled by the R7 release of Erlang/OTP. However, since we do not have that release at the time this document is being written, these wishes are based on the R6B release.

5.1 Better Documentation of Best Coding Practices

The Erlang book is the best reference we've found for learning the language. However, documentation on the Open Telecom Platform is confined to the reference material in the online documentation [Erla]. In fairness to the current OTP documentation, it is a good reference resource, but it's not a tutorial. You simply need to know where to look and to know if it's the right hammer to pound any particular nail.

Programmers coming from a C/UNIX background are accustomed to an edit-compile-debug work cycle inherited from batch-processing origins. Interactive editors and integrated development environments have accelerated the cycle but have not changed its fundamental character. The interactive interpreter was therefore a little puzzling — how should it be used in daily work? Being unable to learn at the knees of local experienced Erlang programmers, we experimented on our own. Having a detailed "user story" in the documentation, a low-level chronicle of a typical day programming Erlang, would have accelerated this process.

5.2 More, Better, and Faster IPC Mechanisms

Understanding that our desires are biased toward the applications we typically work on, we'd like to see UNIX domain sockets formally supported. We'd also love to dabble with shared memory, though it can be problematic in a system where memory management is hidden from the programmer. We'd love to see a more efficient TCP and UDP driver, one that makes use of the `outputv()` driver interface to allow use of vectored I/O primitives and other efficiency mechanisms.

We understand the portability concerns raised by supporting these admittedly platform-specific features. It can make "write once, run anywhere" code more difficult to write and maintain.[5] Such feature creep presents a slippery slope: what new features are platform-independent enough, or is customer demand great enough? In our opinion, the need for fast IPC warrants their use in the standard Erlang distribution.

As discussed in Section 4.5, Erlang is a surprisingly useful, practical language. Support for additional IPC mechanisms can only encourage other adventurous programmers to develop other Erlang applications that break yet more new ground.

[4]As an example, we've already experimented with having the TCP and UDP drivers allocate their buffers from a shared memory pool.

[5]The Client Daemon runs quite well on Erlang/OTP for Windows NT, despite intentionally ignoring NT during development.

5.3 Better Debugger

The debugger in Erlang/OTP R6 is useful, but it needs enhancements. We would love to see a binding watchpoint feature added. A more streamlined "compile, debug, edit, compile, debug newly-edited code in the same debugger environment" cycle, one that requires fewer mouse clicks, would be nice. Also, it would be handy to save breakpoint settings in a context-sensitive manner, attempting to maintain breakpoint locations despite adding or deleting lines of code prior to the breakpoint.

5.4 Better Profiling Tools

We've spent a good deal of effort trying to understand the performance characteristics of the BEAM VM in general and of our application running within it. The Erlang/OTP R6 profiling tool, `eprof`, is okay at best and utterly inaccurate at worst. We ended up working on a better tool,[6] but the effort has been limited by the VM's process trace output itself: it can fail to mention execution of some short functions, which throws off function call counts and can lead to misattribution of execution time.

The fundamental problem is the lack of a global context for the profiling results. If `eprof` profiling reveals a function to be the tall tent pole within a given process, it may still be insignificant if the profiled process is only a small fraction of the overall runtime. The user-contributed `top` tool [Top] is useful for getting a system-wide view of VM reductions, but it cannot account for reductions made by short-lived processes. Furthermore, there is no accurate correlation between VM reductions and either CPU or wall-clock time.

To give a global context for our performance, we used `gprof` to measure the BEAM VM as a C program. This didn't allow us to directly measure the execution of our program code, but we were able to see the relative weights of bytecode execution, message-passing, garbage collection, and linked-in drivers.

Within the Erlang code, we see the need for both process-oriented and function-oriented profiling. We also need the ability to create a `gprof`-style call graph. Lastly, we need both wall-clock and CPU-clock timing statistics.

5.5 Better String Handling

Being avid Perl hackers, it may be unfair to criticize Erlang for weak string handling features, but we'll do it anyway. The functions found in the standard `string` module are a good base. But Erlang's treatment of strings as lists of bytes is as elegant as it is impractical. The factor-of-eight storage expansion of text, as well as the copying that occurs during message-passing, cripples Erlang for all but the most performance-insensitive text-processing applications.

Erlang's treatment of binaries, by contrast, has so far proven to be a showcase for the language's features without a significant cost in performance. We'd much rather see a `string` library, parser-generator, etc., based on binaries, or on some new binary-like string representation, rather than the current list representation.

5.6 Multiprocessing Support and Memory Usage

Again, on this topic Erlang is caught between a rock and a hard place. On multiprocessing machines, we'd really like to see the virtual machine take advantage of as many CPUs as are available, especially since moving data between Erlang instantiations via IPC currently requires several data copies and is therefore expensive. We see no reason why SMP support would require any change to the language or its libraries. It would require a massive redesign of the virtual machine and possibly large sections of platform-specific code to get the best performance. However, if Erlang wants to be considered for applications like ours on high-end hardware, SMP support is necessity.

We understand that with R7 we'll see the VM able to use up to 4 GBytes of RAM, but, again, for high end applications this isn't enough. We need to be able to run our system on 12+ processor machines with

[6]We were desperate enough to modify `eprof` for greater accuracy and to try to measure both wall-clock and CPU time. See `http://www.erlang.org/ml-archive/erlang-questions/200005/msg00052.html`.

12+ GBytes of RAM. For the foreseeable future, this means running multiple virtual machines per physical server, which is something we'd prefer not to do.

5.7 More Support For Writing Network Servers

Many important Internet applications, such as the Apache web server and the BIND naming daemon, are moving toward a multithreaded programming model. We shudder to think of using C/pthreads to achieve this. With the imminent release of the bit syntax, we see a great opportunity for Erlang to be the premiere language for serious Internet server development.

We'd like to see more support for writing servers providing TCP- and UDP-based protocols. While there are a few sample applications to learn from, more could be done to assist programmers communicating via an IP network to non-Erlang-based clients. Items on this wish list include:

- More documentation and examples of non-trivial clients and servers of popular protocols.
- Allow a listening TCP socket to operate in an "active" mode, i.e. by allowing `gen_tcp:accept()` to send a message to the listening process rather than as a blocking function call.[7] This would allow a `gen_server` to directly accept new connections without blocking.
- Streamline/simplify the process for passing ownership of a newly-accepted socket descriptor process to another process. The most natural way to write a per-session process's `start()` entry point is to have the TCP connection socket as an argument and to give the new process's PID as a return value. However, there is a race between the new process using the socket (e.g. printing a prompt or greeting) and the listener process changing the socket ownership. This race can be resolved with some message-passing for synchronization, but it has occurred often enough in our code to be annoying.
- Avoid needless data copies across the driver boundary.
- The binary syntax will help immensely with network byte-order conversions and (un)packing encoded data structures.

With a few clean-ups and an eye towards attracting a wider audience, we feel that Erlang can make significant strides as a network application language.

6 Conclusion

We were surprised at the extent to which Erlang fulfilled its promises. It took some effort, but it was straightforward for developers to come up to speed on the language, and its use significantly reduced the time it took to produce working code. While we still believe that Erlang has some deficiencies, it has demonstrated itself as a first class prototyping language, and we have no qualms about shipping a production application based upon it. A number of Erlang's features are novel and compelling. As we look at other projects written in C, we often find ourselves thinking about solving their problems with Erlang. While there are a great number of places in which the language and its environment could be significantly improved, Erlang is already a fascinating language that deserves a wider audience than it currently has.

7 Acknowledgments

The authors of this paper would like to thank our management for allowing us to get ourselves into this mess. Thank you as well to those fine folks at Ericsson who came up with Erlang in the first place. Also, a

[7] Some thought should be given to a rate-limiting mechanism for these automatic accepts, although much existing connection accepting code has no such mechanism, either.

special thanks to the folks at Bluetail A.B. for their early encouragement. We especially appreciate the work of Tony Rogvall and his ONC RPC library: it saved us a great deal of work. Finally, we want to especially thank everyone on the `erlang-questions` mailing list for helping a bunch of neophytes come up to speed with their nice language. You've helped us more than you'll probably ever know.

References

[Blu] The Bluetail Mail Robustifier. See `http://www.bluetail.com/products/bmr/`.

[Bro95] Frederick P. Jr. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 20th anniversary edition, 1995.

[Edd] Eddie, an HTTP load balancer. See `http://www.eddieware.org/`.

[EPL] Erlang Public License. See `http://www.erlang.org/EPLICENSE`.

[Erla] Erlang online documentation. Available at the Open Source Erlang distribution site: `http://www.erlang.org/download.html` and in browsable form at: `http://www.erlang.org/doc.html`.

[erlb] The `erlang-questions` mailing list archive. See `http://www.erlang.org/ml-archive/erlang-questions/`.

[KLS86] N. Kronenberg, H. Levy, and W. Strecker. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[Lib] Libero, a finite-state machine-based, programming language-independent code generation tool. See `http://www.imatix.com/html/libero/`.

[SMT] SMT, the Simple Multi-Threading kernel. See `http://www.imatix.com/html/smt/`.

[Top] `top-1.0`, a UNIX `top`-like tool. See `http://www.erlang.org/user.html`.

MPowered by Erlang

Per Bergqvist
per@cellpt.com

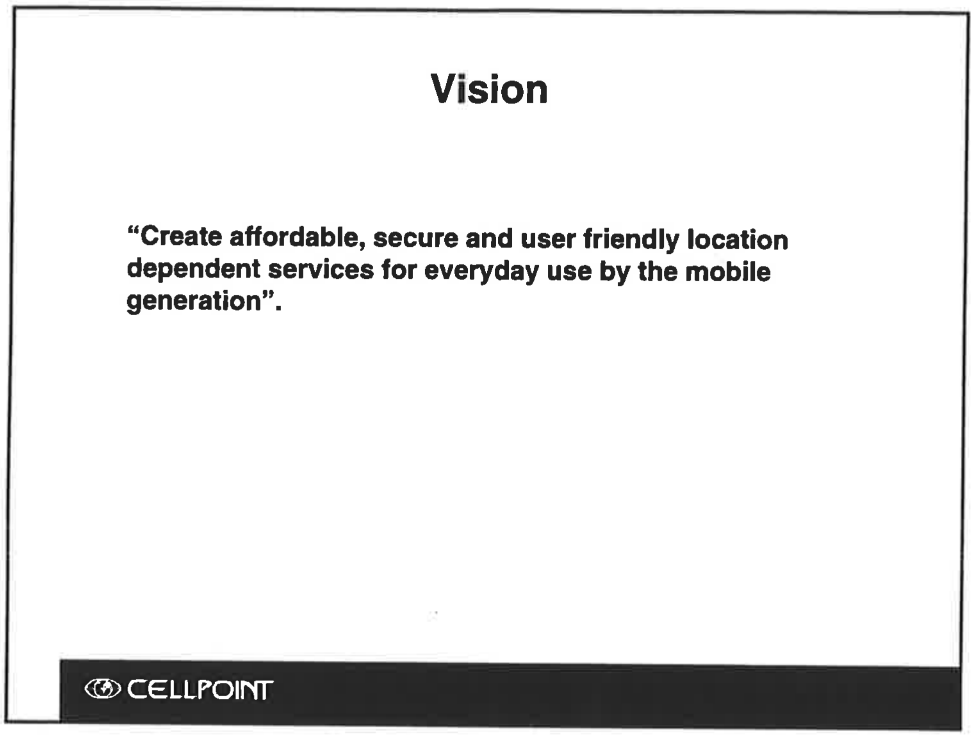
Date: October 3rd 2000 Place: Erlang User Conference 2000

CELLPOINT

CellPoint Systems

- World leading in location dependent services based on standard GSM terminals
- >90 persons in Marketing and Sales, R&D and Operations, expanding very rapidly
- Stockholm, London and Johannesburg, SA
- 4 years experience of commercial GSM based positioning systems
- Complete solutions ready to implement, launch and bill

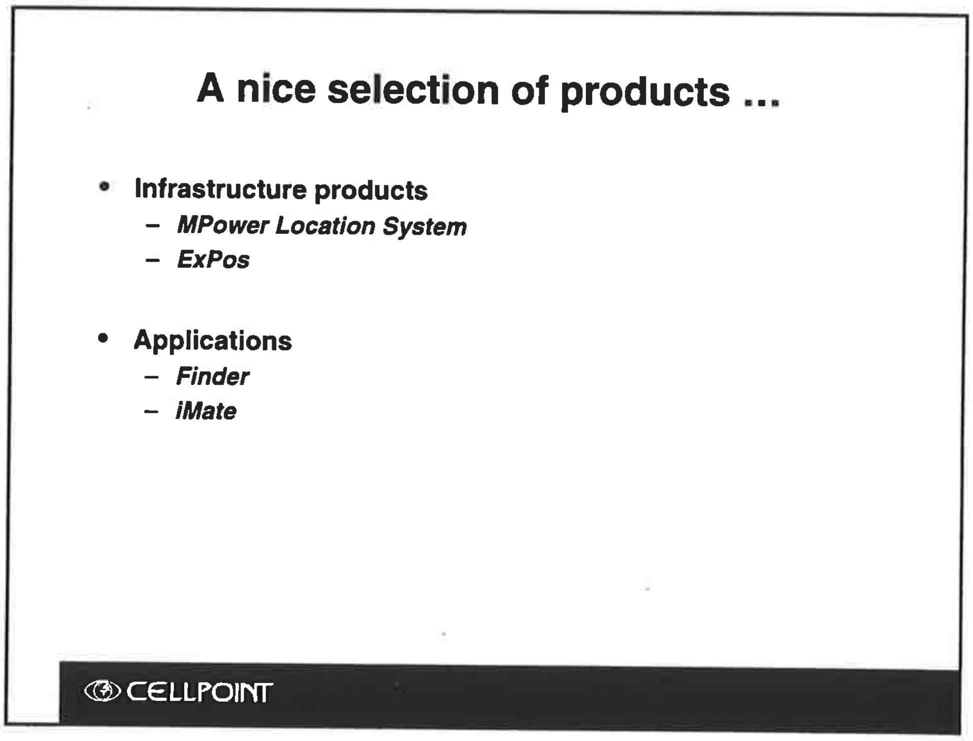
CELLPOINT



Vision

"Create affordable, secure and user friendly location dependent services for everyday use by the mobile generation".

CELLPOINT

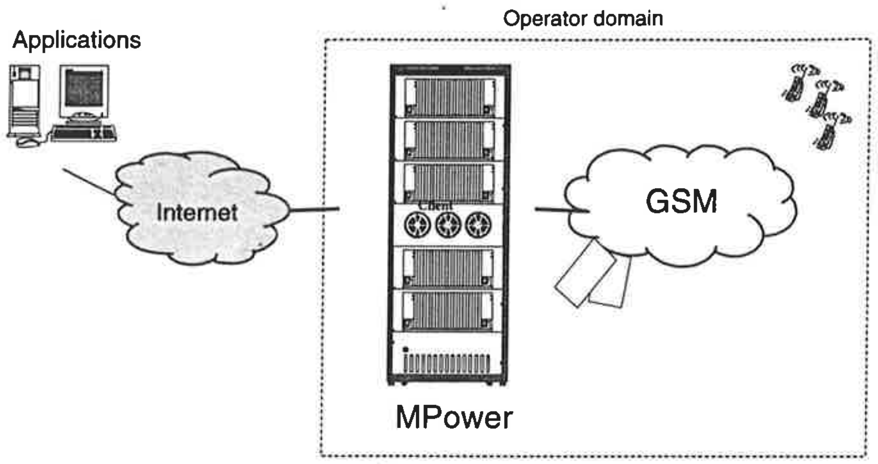


A nice selection of products ...

- Infrastructure products
  - *MPower Location System*
  - *ExPos*
- Applications
  - *Finder*
  - *iMate*

CELLPOINT

MPower Location System



CELLPOINT

iMate demo

CELLPOINT

Taming Windows ...

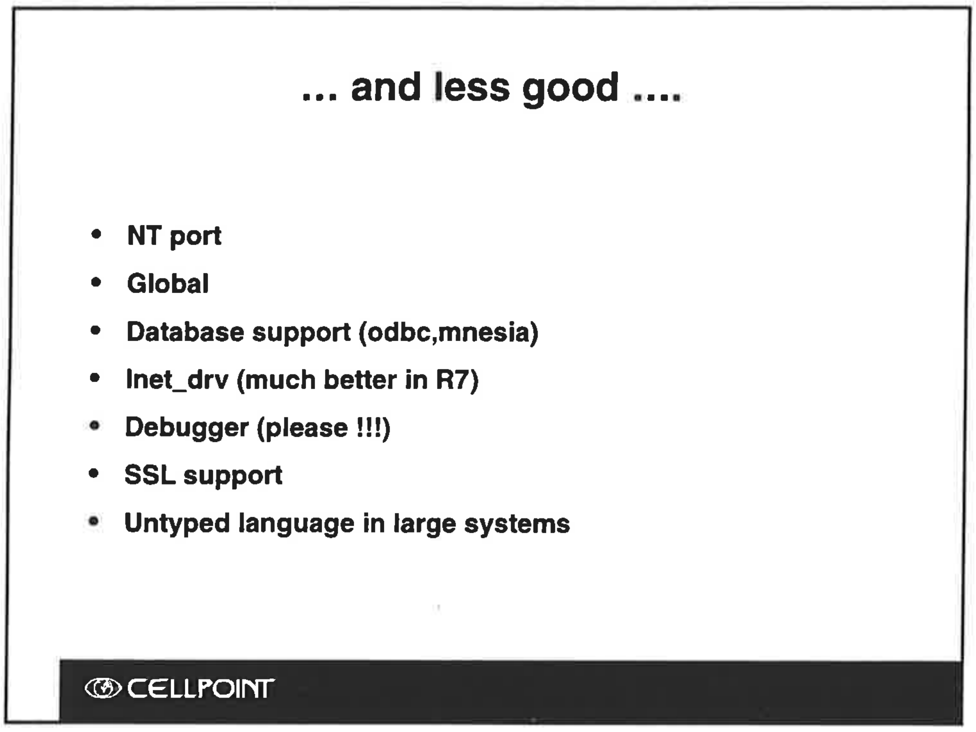
- Development environment NT
  - *Historic, office integration, price reasons*
- Portability
  - *Windows NT/2000*
  - *Solaris*
- Tools
  - *Cygwin*
  - *CVS*
- Performance
  - *NT/Pentium beats Solaris/Sparc (both absolute and bang-for-the-buck)*

CELLPOINT

Very good experiences ...

- As we all already know
  - *Robustness*
  - *Less code*
  - *Easy to learn*
- Performance
  - *Excellent for truly massive parallel applications*

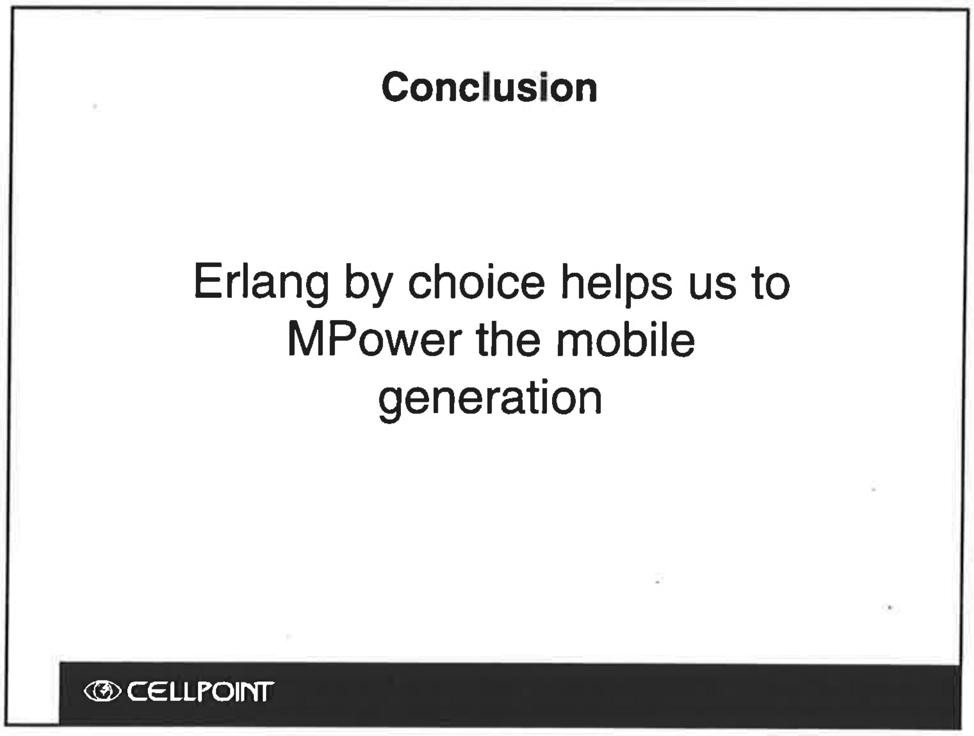
CELLPOINT



... and less good ....

- NT port
- Global
- Database support (odbc,mnesia)
- Inet_drv (much better in R7)
- Debugger (please !!!)
- SSL support
- Untyped language in large systems

CELLPOINT



Conclusion

Erlang by choice helps us to MPower the mobile generation

CELLPOINT

NETSim - 6 years with Erlang

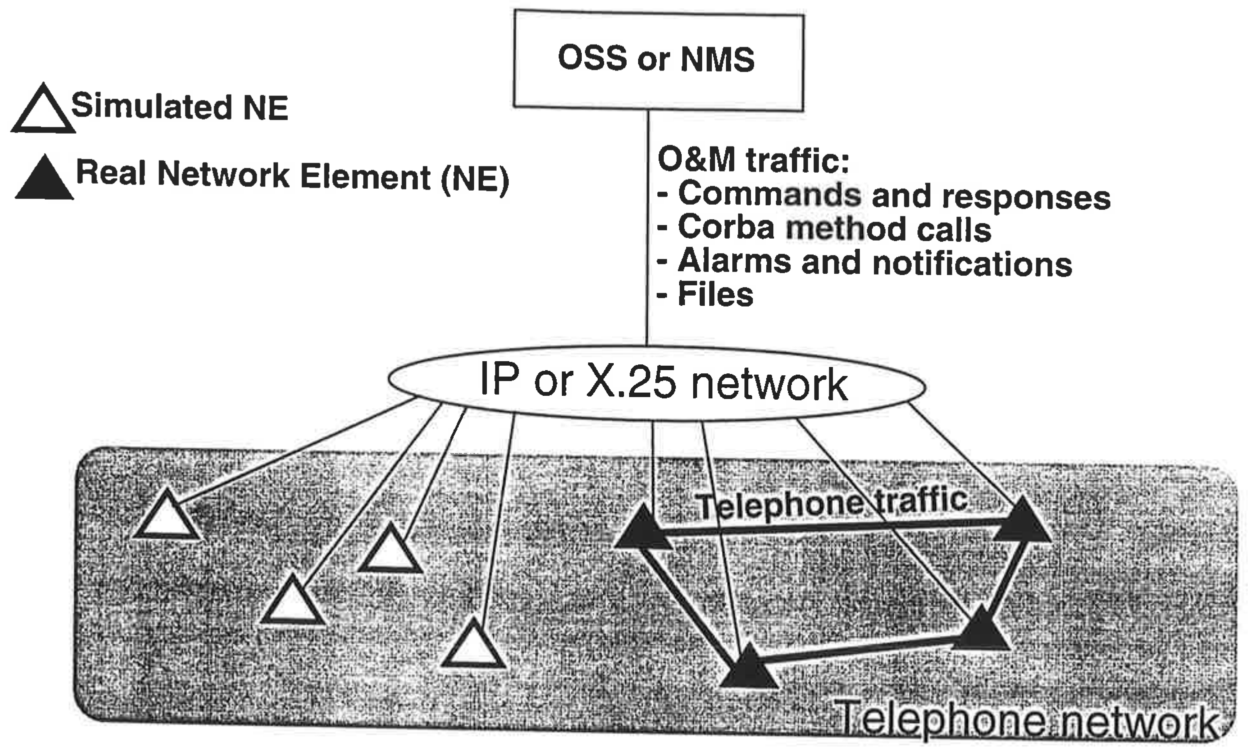
Bengt Tillman
ERA Linköping

bengt.tillman@era.ericsson.se
http://www.lmera.ericsson.se/tss/ (Ericsson only)

NETSim - six years with Erlang

- NETSim - a Network Element Test Simulator
- What is NETSim?
- Technical challenges and their solution with Erlang/OTP
- Administrative challenges and their solution with Erlang/OTP
- Summary

NETSim in a telephone network



NETSim

- NETSim simulates the operations and maintenance (O&M) functions of network elements (exchanges) in a telephone network
- Customers:
  - Developers and testers of Operation Support Systems (OSS) and Network Management Systems (NMS)
  - Trainers of OSS and NMS operators
- Installations:
  - Ericsson development, test and education departments all over the world
- Scope:
  - Many different O&M protocols (MTP, IIOP, ftp, X29, MIP, telnet...)
  - Many different types of simulated NEs
  - Many different versions of each type of simulated NE
  - Many different O&M functions (MML commands, Corba methods)
  - Many simultaneous simulated NEs
  - Many simultaneous users
  - Heavy O&M traffic

Tech. challenge 1 - Product structure

- Different products for different markets
  - GSM
  - PDC (Pacific Digital Cellular)
  - UMTS
  - General AXE simulation
  - User developed products

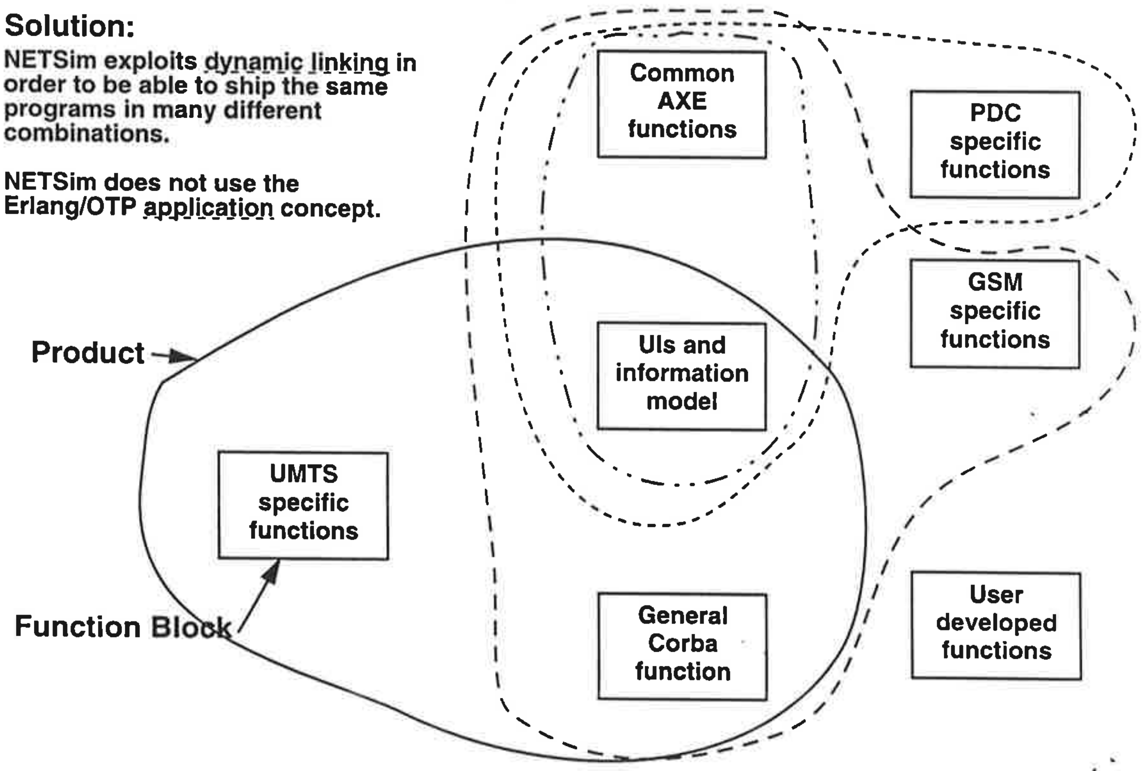
- Many functions in more than one product
  - General AXE simulation
  - General Corba functions
  - User interfaces
  - NETSim information model

Tech. challenge 1 - Product structure

Solution:

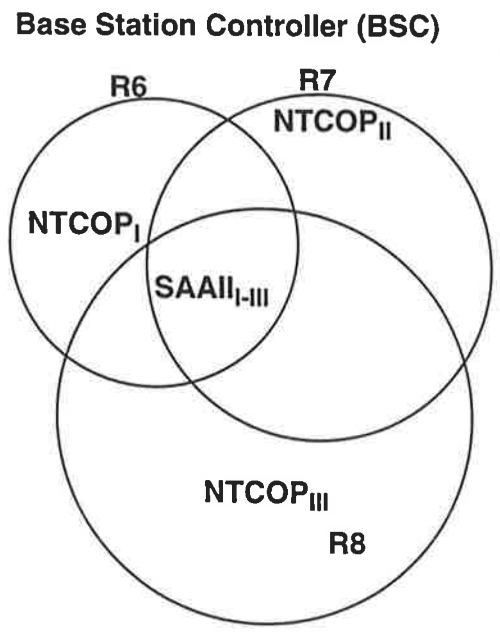
NETSim exploits dynamic linking in order to be able to ship the same programs in many different combinations.

NETSim does not use the Erlang/OTP application concept.



Tech. challenge 2 - Many "commands"

NETSim shall simulate many different "commands" in many different versions of many different types of NEs,

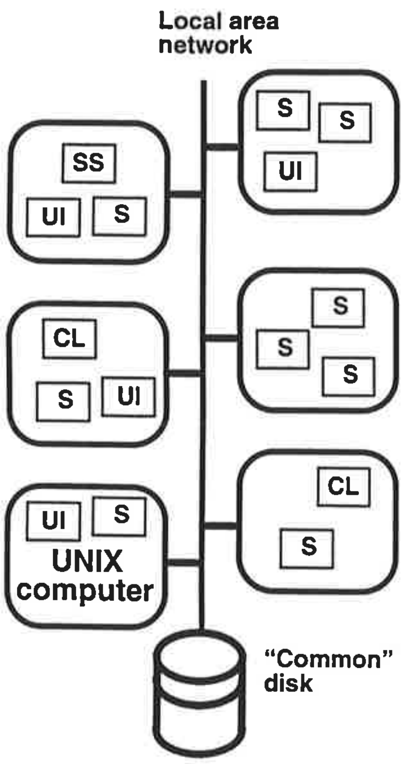


- Example of commands
  - AXE commands (CACLP, SAAII, NTCOP)
  - Corba methods (basic_create_MO)
- Solution: One Erlang <u>module</u> per command
  - Unique module names:
    - ntcop_bscr6.erl
    - ntcop_bscr7.erl
    - ntcop_bscr8.erl
  - ~ 1100 modules (Erlang/OTP runtime 738)
  - Search database per NE type which maps command to Erlang module
  - Structured modules would have been nice

Tech. challenge 3 - Many NEs

- Many NEs must be active simultaneously
  - System test - 300 to 600 simulated NEs - > 4000 in the future
  - Function test - a number of testers (10) have one network each (10-50 NEs)
- Other requirements
  - Each NE shall have its own database where the following functions shall be possible independently and at any time:
    - save database (10 - 30,000 records)
    - restore database
    - copy database to another simulated NE
  - Load distribution over several computers must be possible
- Solution: NETSim database implementation:
  - <u>ets</u> tables with a wrapper around (mmldb, netsimdb) [{{Table, Key}, [{Tag, Value}, ...]}...]
  - <u>Mnesia</u> too rigid - we have looked at it
- Solution: NETSim load distribution implementation:
  - <u>Distributed Erlang</u>, see next page

Tech. challenge 3 - Many NEs



Erlang nodes in NETSim

- SS - superserver - administers licenses and keeps track of all other nodes ("fixed point")
- CL - client - administers the networks of simulated NEs for one user
- UI - user interface - graphical (Java) or command line interface
- S - server - executes simulated NEs. Limitations to consider:
  - Number of ets tables per Erlang node => max ~ 256 NEs per server
  - Number of unix file descriptors per unix process
  - Memory size of computers
  - One Erlang ORB (Orber) per NETSim server
- Very dynamic environment - nodes come and go all

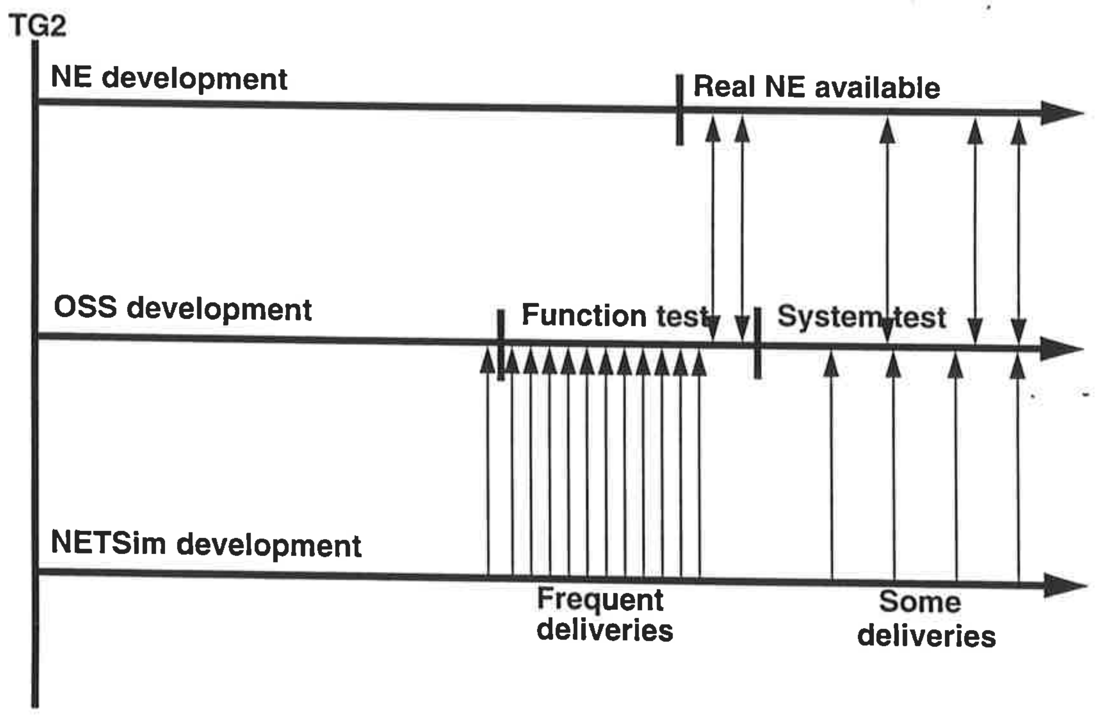
Challenge 4 - Technological frontline

- First Erlang (R2?) in NETSim had no ets, no behaviours...
- Until 1998 Erlang grew and NETSim followed
- With new GUI (Java) 1997: Erlang - Java coupling through jinterface and ic (first through Orber) - unfortunately gs is not good enough.
- With UMTS (3rd gen mobile systems) NETSim is pushing Erlang:
  - Corba - Orber
  - Corba - CosNotifications
  - Many R7 changes caused by NETSim requirements
- Very good cooperation with and support from Erlang/OTP:
  - Niclas Eklund and Babbis Xagorarakis

Tech. challenge 5 - parallelism

- Requirements:
  - Many NEs executing at the same time
  - Several parallel command sessions per NE
  - Many UIs open at the same time
- Solution:
  - Erlang processes for commands and NEs
  - Distributed Erlang for parallel UIs

Adm. challenge - Lead times and freq. deliveries



Adm. challenge 1 - Lead times

- Solution: The outstanding productivity in Erlang/OTP due to
  - loading of code into an executing system
  - behaviours (both faster coding and repeated loading)
  - its high level
  - This gives
    ==> real short turn-around times (a few seconds)
    ==> development in a "real" NETSim installation
    ==> incremental development: "write 3 lines and then test them"
    ==> development, basic test and integration test at once
    ==> time is spent on making functions - not on debugging
    ==> compact source code - easy to read and easy to maintain
- Solution: Erlang/OTP is easy to learn
  - we have had many persons writing "commands" over the years and no one has known Erlang in advance

Adm. challenge 2 - Frequent deliveries

- Problem:
  - Incremental deliveries so test can start early
  - Frequent changes
  - Often parallel support to different customers
- Solution (all implemented as Erlang programs)
  - Daily build (also used as part of system test)
  - Administration tool (about 32 different windows implemented in about three weeks using gs)
  - Automatic test records
  - This gives:
    ==> The status of the source code known at all times
    ==> Anyone can do configuration management tasks
    ==> Beta delivery with 24 hours notice
    ==> Alpha delivery with 3 hours notice
    ==> Patch delivery with returning mail (seldom needed)

Adm. challenge 2 - Frequent deliveries
Daily build process

- Build all function blocks (2 hours)
- Install function blocks of one product
- Perform black-box testing of all functions directly available to the customer (without programming). (1 - 2 hours/product)
- Install test programs (dynamic linking again)
- Perform black-box testing of application programmer interfaces and white-box testing of internal interfaces (NOTE: the product we deliver to our customers is NOT contaminated with test programs). (1 - 2 hours/product).
- Go back and install the next product
- For all products tested generate a test record (which is automatically published on the intranet)

Summary

- Without the following properties of Erlang/OTP it would have been impossible to develop and maintain NETSim with the small team we have been (2 - 15 persons):
  - so easy to learn - and still so powerful
  - so fast to develop in because of loading of code into an executing system
  - so flexible because it allows distribution and dynamic linking
  - so powerful because of its light-weight processes
- Some other nice features which have saved us a lot of time:
  - behaviours, ets
  - the tools tv, pman and debugger => debugging at customer sites
  - ASN.1, SNMP, Orber - so far no 3rd party products needed
- My personal experiences in programming:
  - I programmed assembler, Forth and Pascal 1966 - 1990 and enjoyed it
  - I learned C++ in 1990 and disliked it
  - In 1993 I thought about working with project management and system design rather than to program in C++
  - Since 1993 I have worked with system design and produced more programs than ever - in Erlang - and I still enjoy it.

A High Performance Erlang System

Erik Johansson
Computing Science
Department, Uppsala
University, Sweden
happi@csd.uu.se

Mikael Pettersson
Computing Science
Department, Uppsala
University, Sweden
mikpe@csd.uu.se

Konstantinos Sagonas
Computing Science
Department, Uppsala
University, Sweden
kostis@csd.uu.se

ABSTRACT
Erlang is a concurrent functional programming language designed to ease the development of large-scale distributed soft real-time control applications. It has so far been quite successful in this application domain, despite the fact that its currently available implementations are emulators of virtual machines. In this paper, we improve on the performance aspects of Erlang implementations by presenting HiPE, an open-source native code compiler for Erlang. HiPE is a complete implementation of Erlang, offers flexible integration between emulated and native code, and efficiently supports features crucial for Erlang's application domain such as concurrency. As our performance evaluations show, HiPE is currently the fastest among all Erlang implementations.

1. INTRODUCTION
The concurrent functional programming language Erlang was designed by Ericsson to address the needs of large-scale distributed soft real-time control applications [2]. Such applications routinely arise in products developed by the telecommunications industry. Erlang caters for these needs with a run-time system that provides many features often associated with an operating system rather than a programming language. These features include scheduling of light-weight concurrent processes, automatic memory management, networking, protection from deadlocks and programmer errors, and support for continuous operation even when performing software upgrades.

After around a decade of existence, Erlang is generally considered as a "success-story" in declarative programming languages; see e.g. [25]. Users experience that Erlang allows telecommunication systems to be programmed with less effort and fewer errors than by using conventional programming language technology [1, 5]. It is worthwhile to note that such systems typically consist of several hundred thousand lines of source code (the size is partly due to the complexity of the telecommunication protocols), and rely heavily upon the concurrency capabilities of Erlang.

The industry, besides Ericsson, is showing a growing interest in Erlang, but there is a very limited choice of compilers, partly due to Erlang's—until recently exclusive—"in-house" development. Also, as an implementor of these compilers publicly admits [1]: 'performance has always been a major problem' and 'we are (even) considering adding imperative features to the language to solve these (performance) problems'. Indeed, the performance of current implementations of Erlang is inferior to that of good implementations of other functional programming languages; see also [8, 11]. In the competitive market of telecommunications, however, the need for a high-performance implementation is sometimes pressing.

As one such example, consider AXD 301, a new generation ATM switching system from Ericsson [5]. The major part of AXD 301's software is written in Erlang; it consists of about 480,000 lines of Erlang code, with about 95,000 of them constituting the time-critical modules of the system. Speeding up this time-critical part would be more than welcome by the AXD 301 engineering team, let alone Ericsson, because this speedup directly corresponds to the ATM switch being capable of servicing more requests; see also [5].

Currently, complete implementations of Erlang are based on emulators of virtual machines. This gives them good portability, but emulation incurs a performance penalty to Erlang programs which some users wish—and in some cases need to—avoid. Ways to avoid the performance problems caused by emulation are: 1) compile to a sufficiently low-level and fast language such as C or 2) use the recently proposed C-- [19] as a portable assembly language, 3) use a retargetable code generator such as ML-RISC [18] or 4) the gcc back-end [21], or 5) compile directly to native code. Each of these implementation choices has well-known pros and cons but one can roughly expect a decrease in portability and an increase in performance and implementation effort for a higher choice number; see also the above references and the references therein.

Perhaps another issue deserves attention: byte-code emulators usually result in smaller object code size than C-based or native code compilers. Although object code size is becoming less and less of a concern nowadays, it is still a potential problem when the source code of the application consists of several hundred thousand lines. Also, not paying attention to code size can result in poor I-cache behaviour leading to significant performance degradation. Ideally, a system should provide a seamless integration of emulated and native code execution, and allow its user to choose the execution mode individually for each application component based on the various space/time trade-offs that are involved. Information about these trade-offs should also be something that the system provides to its user.

This paper presents our approach to the efficient execution of Erlang. We have developed a system, HiPE, which combines the performance characteristics of a native code compiler with the benefits of an emulated implementation. HiPE currently uses the JAM emulator [1] as a basis and allows selective compilation of whole modules or individual functions into native code, which is then executed directly by the underlying hardware. Besides fully describing the architecture of HiPE and the changes to the JAM run-time system needed to support native code execution, we discuss various technical issues that this emulated/native code integration entails in the context of Erlang and how we dealt with them. More specifically:

- we pay special attention to supporting *hot-code loading* (see next section) and error handling in HiPE;
- we describe a method for performing tail call optimization in a mixed mode of execution where a separate stack is used for each mode—this method is probably folklore but, to the best of our knowledge, it has not been reported in the literature;
- we compare and analyse the performance of existing Erlang implementations on "standard" small benchmarks and on large programs from actual industrial Erlang applications and show that HiPE's performance is superior.

To make this paper self-contained, we begin by reviewing the characteristics of Erlang (Section 2). Sections 3 and 4 form the main part of this paper and describe the basic characteristics of HiPE, its architecture, and the integration of native and emulated execution within the same run-time system. Section 5 contains a performance comparison of HiPE against other implementations of Erlang, both industrial and academic, and against other functional languages. Section 6 contains additional measurements on some of the benchmark programs used in our performance evaluation that give more insight on HiPE's performance. We end this paper with some concluding remarks.

2. A BRIEF INTRODUCTION TO ERLANG

Erlang is a dynamically typed, strict, concurrent functional programming language. It is possible to create closures in Erlang, but typical Erlang programs are mostly first-order. Erlang's basic data types are atoms, numbers (integers with arbitrary precision and floats), process identifiers, and references; compound data types are lists and tuples. There is no destructive assignment of variables and the first occurrence of a variable is its binding instance. Function selection happens using pattern matching. Erlang's design inherits some ideas from concurrent constraint logic programming languages such as the use of flat guards in function clauses.

Processes in Erlang are extremely light-weight, their number in typical applications is quite big, and their memory requirements vary dynamically. Erlang's concurrency primitives—`spawn`, "`!`" (send), and `receive`—allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any data value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs though pattern matching. There is no shared memory between processes and distribution is almost invisible in Erlang. To support robust systems, a process can register to receive a message if another one terminates.

Erlang applications typically consist of a number of modules: an Erlang module defines a number of functions. Only explicitly exported functions may be called from other modules. Calling functions in different modules, called *remote calls*, is done through supplying name of the module of the called function. During execution of functions, tail call optimization is performed. As in other functional languages, Erlang's memory management is automatic through garbage collection. The real-time concerns of the language call for bounded-time garbage collection techniques; see [24, 16]. In practice, garbage collection times are usually small as most processes are short-lived or small in size.

To perform system upgrading while allowing continuous operation, an Erlang system needs to cater for the ability to change the code of a module while the system is running, so called *hot-code loading*. Processes that execute old code can continue to run, but are expected to eventually switch to the new version of the module by issuing a remote call (which will always invoke the most recent version of that module). Erlang provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

The Erlang language was purposely designed to be small, but comes with libraries containing a large set of *built-in functions* (known as *BIFs*). With the Open Telecom Platform (OTP) middleware [23], Erlang is extended with standard solutions to common requirements of telecommunication applications (servers, state machines, process monitors, load balancing), standard interfaces (CORBA), and standard communication protocols such as HTTP and FTP.

3. JAM: THE BASIS OF HIPE

HiPE is based on the bytecode emulated JAM implementation of Erlang, to which it adds the ability to compile and execute Erlang as native machine code. HiPE is a new component (currently 30,000 lines of Erlang code and 3,000 lines of C and assembly code) added to an otherwise mostly unchanged JAM system; only the JAM emulator and the garbage collector have been extended to be aware of native code. Because of this tight integration, we describe relevant aspects of the basic JAM system here; Section 4 continues with HiPE-specifics.[1]

[1]HiPE is publicly available as open-source. The current release is based on Ericsson's Open Source Erlang 47.4.1. See http://www.csd.uu.se/projects/hipe.

3.1 The JAM system

The JAM-based Erlang implementation uses JAM: a virtual stack machine whose primitive operations closely correspond to the Erlang language. For example, since Erlang is a dynamically-typed language, JAM uses a self-describing data representation with tagged values [10], and the emulator's instructions operate on tagged values, never on raw machine values.

3.1.1 The JAM compiler

The JAM compiler is a non-optimising compiler which performs a straightforward translation to the JAM virtual stack machine. The object files contain bytecodes and relocation entries which describe all symbolic references that must be resolved by the loader.

3.1.2 The JAM loader

The JAM loader translates JAM bytecodes from the external format to the internal format expected by the JAM emulator.

Erlang atoms are symbolic constants, like atoms in Prolog or symbols in Lisp. The internal representation of an atom is its position in the atom table, which is not known until runtime. Therefore, the first task of the loader is to replace symbolic references to atoms by their current internal representation.

A remote (non-local) function call is represented by a triple (module name, function name, function arity). First the names are translated to internal atoms. Then special cases are identified, such as calls to the **erlang** module which become calls to C functions in the runtime system. Finally, the JAM instruction at this call site is patched to reflect the result.

After a module has been loaded, a global symbol table is updated with information about the module, its exported functions, and the code addresses at which those functions start.

3.1.3 The JAM emulator

The JAM emulator is a single C function which executes JAM instructions represented as bytecodes. The JAM implementation is indirectly threaded [4, 17]. The emulator is invoked on a runnable Erlang process, and executes code for that process until it blocks. A process blocks either when it attempts to read a message and its message queue is empty, or when its "time slice" has expired. Time slices are represented by work budgets, which are explicitly decremented and checked at specific points in the emulator.

Each Erlang process is described by a process control block (PCB), a stack, a heap, and a set of pointer registers:

`sp`	next word on the stack
`fp`	start of current function's activation record
`ap`	first actual parameter
`pc`	current bytecode instruction
`cc`	debug information about the current function

The stack discipline is simple but unoptimised. At a call, the parameters are pushed in left-to-right order, followed by a 4-word continuation record containing the caller's `fp`, `ap`, `pc`, and `cc`. Then `fp` is set to point to the start of this record, `sp` to the first word after the record, and `ap` to the first parameter (derived from `fp` and the callee's arity); see Figure 1. (Note that `ap` now has the same value as the caller's `sp` had before the call.) At return, `sp` is reset to `ap`, then `cc`, `pc`, `ap`, and `fp` are restored from the frame, and the return value is pushed onto the stack.

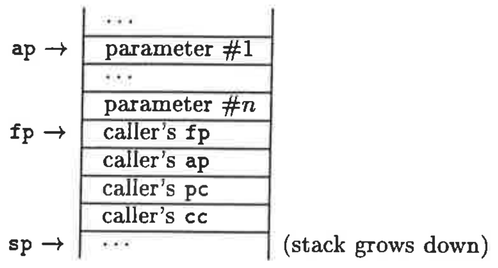


Figure 1: JAM stack on entry to a function

In JAM, tail calls are complicated by the fact that the four-word continuation frame is pushed *after* the parameters instead of before. Since the continuation frame is adjacent to the parameter area, it must be relocated whenever there is a tail call and the caller and callee have different number of parameters. At a tail call, the JAM emulator copies the frame into temporary variables, then copies the outgoing parameters from the bottom of the stack to the parameter area, and then (if necessary) moves the copy back to the stack.[2]

Exception handling is implemented by *dynamic tracking* [3]. On entry to a protected code block, a 2-word catch frame is pushed onto the stack, containing a pointer to the previous catch frame and the address of the first instruction after the protected block. The address of this frame is saved in the PCB. To raise an exception, the stack is unwound one call-frame at a time, until the activation record containing the current catch frame is found. The unwinding process simultaneously restores the `sp`, `fp`, `ap`, and `cc` registers.

3.2 Processes and memory management

An Erlang *node* is an instance of the Erlang runtime system executing on a given machine. On Unix, this is a single Unix process. Within a node, Erlang processes are created dynamically and execute as coroutines. A C procedure acts as scheduler, continuously selecting a runnable process and passing it to the emulator for execution.

Each Erlang process has a PCB, a stack, and a *private* heap for the data structures it creates. An Erlang process starts with small stack and heap areas, which are grown when needed. Compared to a typical implementation of Posix threads in Unix, which would allocate in the order of one megabyte of virtual memory for each thread's stack, Erlang

[2]Performance could be improved by shrinking the frame to a minimum: `ap` is redundant and `cc` can be computed from `pc` when needed. If the continuation was pushed *before* the parameters, it could remain in place during tail calls [7, Section 4.6.1].

processes are extremely lightweight. An Erlang node is expected to handle hundreds or thousands of Erlang processes with relative ease.

The garbage collector is of a standard two-generational stop-and-copy type [16]. It does, however, have one interesting design feature: each process' heap is strictly private to that process, and no references are allowed from one process to another's heap. The advantage of this arrangement is that it simplifies memory management. Since references from other processes cannot occur, garbage collection is a strictly local activity, and when a process terminates, its memory can be reclaimed immediately. This is believed to reduce total memory management costs, since the majority of Erlang processes are expected to be short-lived or to have small amounts of live data. The disadvantage, however, is that message passing must be implemented by data copying, which reduces sharing and increases the pressure on the memory caches.[3] Messages are expected to be small, however.

4. HIPE: SYSTEM OVERVIEW

The HiPE compiler is called as an ordinary Erlang function, within a running Erlang system. The smallest unit of compilation is a function: given the name of an existing Erlang function, the compiler translates that function's JAM bytecodes to SPARC V9 machine code, and then a linking phase updates the system state so that future calls invoke the native code; see Section 4.2. In this respect, HiPE resembles a (user-invoked) JIT compiler. Alternatively, the HiPE compiler can compile a whole Erlang module, and the result of the compilation can be saved in a file, as symbolic SPARC code. This file can later be loaded explicitly into the HiPE system.

Some Erlang applications are delivered only in bytecode format and no sources are available. For example, before Erlang became open-source, Erlang libraries were only present as JAM bytecode files. Therefore, HiPE was designed to be able to take already-loaded functions in JAM bytecode format instead of Erlang source code as input. Although this means that HiPE cannot perform some high-level optimizations, this compilation scheme offers more freedom to its users: users have the possibility to identify (using HiPE's profiling tools) those functions and call paths that would benefit most from compilation to native code and selectively compile them. This way, HiPE combines the performance of a native code system with the code compactness of a bytecode system. This integration is very tight: code compiled by HiPE uses the same runtime system and the same built-in functions as the JAM emulator.

4.1 The HiPE compiler

The compiler has four intermediate representations: an internal representation of JAM bytecode, a high level intermediate language (ICode), a general register transfer language (RTL), and a machine-specific assembly language, currently only SPARC; see Figure 2. ICode, RTL, and SPARC are represented as control flow graphs of basic blocks.

[3]We are planning to measure memory subsystem performance by implementing a shared-heap runtime system and comparing it against the current runtime system.

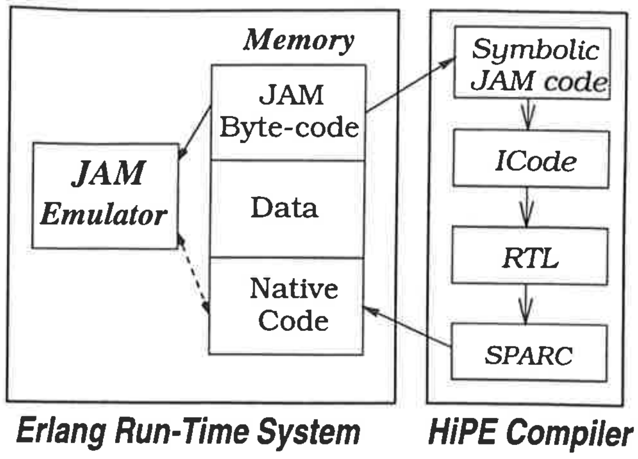


Figure 2: Intermediate representations in HiPE.

The JAM bytecodes are translated to symbolic form by a straightforward process. Internal atom numbers are converted to real atoms, and branches to local functions are translated to call instructions with symbolic function names.

ICode is based on a register-oriented virtual machine for Erlang. Arguments and temporaries are located in an infinite number of registers, and all values are proper Erlang terms. The call stack is implicit, and calls preserve registers. Bookkeeping operations, such as heap overflow checks, context switching, and time-slice decrements, are implicit. The translation from JAM bytecode to ICode uses a simulated stack to map JAM stack slots to ICode registers. To simplify dependency analysis in later compilation passes, a renaming post-pass ensures that independent live ranges use different registers.

The ICode is then optimised with standard compiler optimizations such as copy and constant propagation, and constant folding. This is done in one pass over all extended basic blocks. Dead code removal is then performed to remove assignments to dead temporaries.

Unreachable code is removed by the translation to RTL, since only reachable basic blocks are inserted in the RTL control flow graph. Operations on Erlang values are expanded to make data tagging and untagging explicit.

The optimisations that were performed on ICode are applied again to the RTL code. Heap overflow tests, call stack management, and the saving and restoring of registers around calls are made explicit, and the standard optimisations are applied again. In order to limit the number of heap overflow tests, they are propagated backwards as far as possible, and adjacent tests are merged.

The RTL code then is translated to abstract SPARC code, and registers are assigned using a graph-colouring register allocator similar to the one of Briggs *et al.* [6]. Finally, symbolic references to atoms and functions are replaced by their values in the running system, memory is allocated for the code, and the code is linked into the system.

4.2 The HiPE linker

As described before, Erlang requires the ability to upgrade code at runtime, without affecting processes currently executing the old version of that code.

The JAM system maintains a global table of all loaded modules. Each module descriptor contains a name, a list of exported functions, and the locations of its *current* and *previous* code segments. The exported functions always refer to the current code segment. At a remote function call, `module:function(parameters...)`, the JAM emulator first performs a lookup based on module and function name. If the function is found, the emulator starts executing its bytecodes. Otherwise, an error handler is invoked.

In native code, each function call is a direct machine-level call to an absolute address. When the caller's code is being linked, the linker initialises the call to directly invoke the callee. If the callee has not yet been loaded, the linker will instead direct the call to a stub which performs the appropriate error handling. If the callee exists, but only as emulated bytecode, the linker directs the call to a stub which in turn will invoke the JAM emulator.

In order to handle hot-code loading and dynamic compilation at runtime, the linker also maintains information about all call sites in native code. This information is used for *dynamic code patching*, as follows:

- When a module is updated with a new version of the emulated code, all remote function calls from native code to that module are located. These call sites are then patched to call the new emulated code, via new native-to-emulated code stubs.
- When an emulated function is compiled to native code, each native code call site which refers to this function is patched to call the new native code. The first instruction in the bytecode is also replaced by a new instruction which will cause the native code version to be invoked. Finally, the native-to-emulated stub used to invoke it from native code is deallocated.
- When a module is unloaded and its memory is freed, all native code call sites referring to this module are patched to instead invoke an error handling stub. All native code call sites within this now non-existent module are also removed from the linker's data structures, to prevent future attempts to update them.

Both the standard Erlang system and HiPE support load-on-demand of modules. When invoked, the error handler for undefined function calls will attempt to load the JAM bytecodes for that module from the file system. If this is successful, the call continues as normal. As a side-effect of loading the JAM module, the HiPE linker will patch native code call sites as described above.

4.3 Native code calling conventions

In the HiPE runtime system, an Erlang process can execute both emulated JAM code and native SPARC code. To facilitate data sharing, HiPE uses the same data representation as JAM. However, the JAM calling convention is inappropriate for native code, since JAM passes all parameters on the stack and uses large call frames containing redundant information. Instead, native code passes the return address and the first five parameters in registers, remaining parameters (if any) on the native stack, and shrinks the fixed portion of stack frames to a single word for preserving the previous return address. Currently, HiPE does not use the SPARC's register windows; instead, registers are saved and restored as needed around function calls.

HiPE uses two stacks for each process, one for emulated code (the *estack*) and one for native code (the *nstack*). An earlier version of HiPE used only one stack, but that scheme was quickly abandoned as it was found to be quite complex and difficult to implement correctly. Our current dual-stack approach has disadvantages and advantages too:

– As described in Section 3.1.3, the JAM emulator implements exception handling by creating a linked list of catch frames on the stack. Native code uses the same strategy, which means that each stack may contain pointers to the other. If the runtime system relocates one stack (to increase its size), then the other stack must be traversed so that the catch frame links can be updated.

+ By separating the stacks the stack-scanning code in the garbage collector is kept simple. With a single-stack approach, the scanning code would have to know when to switch "mode", in order to correctly deal with the different stack frame layouts. This is certainly doable, but would require more effort to implement correctly.

Foreign Function Interface

Erlang programs can and often do call C functions: standard procedures (BIFs) in the runtime system are written in C, as are the I/O modules. For each BIF, there is a machine-code stub which is directly callable from native code. The stub takes care of saving the native-code state registers before invoking the C function on the C stack. The stub also checks for exceptions at return, and either returns to its caller or invokes the current Erlang exception handler.

However, the Erlang runtime system is not designed to permit arbitrary recursive calls between Erlang and C functions. Whether the Erlang code is emulated or native makes no difference. A detailed explanation is beyond the scope of this paper, but the limitation stems mainly from the non-reentrancy of the Erlang process scheduler (a C function) and the way in which the runtime system periodically polls I/O channels. (It would not be overly difficult to modify the runtime system to eliminate this limitation, however.)

4.4 Mode-switching

In HiPE, a *mode-switch* occurs whenever there is a transfer of control from native code to emulated code, or vice-versa. We made the design decision that the mere presence of multiple execution modes should not impose any runtime overheads, as long as no mode-switches occur. This design requirement calls for great care when implementing mode-switches, not only for performance, but also for correctness.

4.4.1 *Where do switches occur?*

Since HiPE compiles individual functions to native code, a mode-switch must occur whenever there is a flow of control from one function to another, and the two functions are in different modes. Thus, mode-switches occur at call and return sites. Erlang's exception mechanism also introduces mode-switches, viz. when an exception is thrown from code executing in one mode, and the most recent catch handler is in a different mode. We will refer to these cases as *call*, *return*, and *throw* events, respectively.

4.4.2 *When do switches occur?*

How does the system discover that a particular instance of a call, return, or throw event must perform a mode-switch?

Call Events

Older Lisp systems often use a dynamic test at calls to determine the mode of the callee (e.g. compiled or emulated), and then perform the appropriate action.

Another common choice is to use a fixed mode for calls, usually native code. Emulated functions are represented as small native code stubs which invoke the emulator when called. The advantage of this approach is that no dynamic type test is needed when both caller and callee are in native code. The disadvantage is that calls between emulated-mode functions are penalised since they have to make conversions to and from the native-code calling conventions.

HiPE uses a pseudo-static approach in which calls always use the mode of the caller. As described in Section 4.2, if a native-code caller refers to an emulated-mode callee, then the linker redirects the call instruction to instead invoke a native-code stub, which in turn causes a switch to emulated mode. If an emulated function is compiled to native code, then the start of the original bytecodes is overwritten with a special emulator instruction which causes a switch to native mode. (The asymmetry between these cases is due to the fact that the HiPE linker only has knowledge about call sites in native code.)

Return Events

Whenever a recursive function call causes a mode-switch, the return sequence must be augmented to perform the inverse mode-switch. Like calls, this is often implemented by dynamic tests, a fixed-mode convention, or by same-mode stubs.

If dynamic tests are used, then the return address is tagged to indicate that a mode-switch is required when the callee returns. Typically, either the lowest or the highest bit is used. The lowest bit is available if return addresses are aligned on some $2^i > 1$ boundary. Using the highest bit imposes a limit to the usable address space.

If a fixed-mode convention is used for calls, then the same mode is also used for returns. If that mode is native code, then a call from an emulated-mode function must push a "stub" native-mode return address which causes control to flow back into the emulator.

HiPE uses a same-mode convention for returns. When a call causes a mode switch, a new continuation (stack frame) is created in the mode of the callee. The return address in this continuation points to code which causes a switch back to the caller's mode. For returns from native to emulated code, the return address points to machine code in the runtime system. For returns from emulated to native code, the return address points to a special emulator instruction. We made this choice in HiPE because it causes no overhead except during mode-switches, and it minimised the amount of changes we had to made to the existing JAM emulator.

Throw Events

HiPE deals with exception throws in the same way as it deals with function returns: a same-mode convention augmented with mode-switching stack frames. When a call causes a mode-switch, a new exception catch frame is created in the mode of the callee. The handler address in this catch frame points to code which switches back to the caller's mode, and then re-throws the exception. Thus, when a call causes a mode-switch, *two* frames are pushed: first a catch frame, then a return frame. The code at the return address in the return frame knows that it also has to remove the catch frame beneath it before switching mode.

4.4.3 *Maintaining tail-recursion*

The same-mode calling convention with mode-switch stack frames is efficient and easy to implement. For many programming languages, this would be enough. However, like most other functional programming languages, Erlang relies on tail-recursive function calls for expressing iteration. Consider the following sequence of tail calls, where each f_i^e is an emulated function, and each f_j^n is a native code function:

$$f_1^e \overset{tail}{\rightarrow} f_2^n \overset{tail}{\rightarrow} f_3^e \overset{tail}{\rightarrow} f_4^n \overset{tail}{\rightarrow} \cdots$$

A correct implementation of Erlang is expected to execute such a sequence in bounded stack space, regardless of its length. Unfortunately, at each call, a new mode-switch stack frame is pushed, to make the return perform the inverse mode-switch. Thus, stack space usage will grow linearly with the length of the sequence of tail calls, and tail-recursion optimisation is lost.

HiPE solves this problem as follows. The return address in a mode-switch stack frame will always have a known value: either the address of the return mode-switch routine (in native mode), or the address of the return mode-switch instruction (in emulated mode). Thus, a simple runtime test is able to distinguish mode-switch stack frames from normal stack frames. Now, consider the following call sequence:

$$f^e \rightarrow g^n \overset{tail}{\rightarrow} h^e$$

When f^e calls g^n, it pushes two mode-switch frames on the native-code stack: first a catch frame, then a return frame. When g^n tail calls h^e, the system would normally push two new mode-switch frames, on the emulated-code stack. Instead, HiPE implements a mode-switch call event as follows:

1. if the current return frame is a mode-switch frame, then

   (a) remove the mode-switch return frame from the caller's stack

   (b) remove the mode-switch catch frame from the caller's stack

   (c) invoke the callee;

   otherwise

2. push a mode-switch catch frame on the callee's stack

3. push a mode-switch return frame on the callee's stack

4. invoke the callee

By preventing adjacent mode-switch frames from being created, the test restores proper tail-recursive behaviour. The test itself is not expensive, and is only executed when there is a mode-switch call. This implementation refutes a statement in [9, Section 4.4.3], where it was claimed that the use of mode-switch stack frames loses tail-recursion optimisation. Although similar methods to maintaining proper tail-recursion in the context of mixed mode execution have been used in some Prolog implementations (e.g. BIM Prolog, SICStus Prolog) and perhaps elsewhere, these methods are probably folklore: to the best of our knowledge, they have never been in print.

4.4.4 Mode-switching in suspend/resume events

In addition to the call, return, and throw events described above, HiPE may also need to perform mode-switches when a process is suspended or resumed.

The scheduler in the Erlang runtime system has no knowledge about the current mode of a process. It assumes, implicitly, that each process is executed by the JAM emulator. Therefore, when a process is created or resumed, the scheduler simply passes the process' PCB to the JAM emulator for execution.

When a process is suspended while executing in native code, HiPE sets the resume address in the PCB to point to a special emulator instruction. When the scheduler resumes the process, the JAM emulator executes this instruction, which transfers control to the suspended native code.

4.5 Modifications to the JAM emulator

As described in Sections 4.2–4.4, we have modified the JAM emulator to support mixing native and emulated code. In summary, these modifications are:

- The JAM loader registers the location of each function's bytecodes with the HiPE linker.
- A native-code stack has been added to the PCB, together with a few native-code variables (stack pointer, resume address).
- The garbage collector has been extended to scan the native-code stack, and to repair catch frame links when either stack is relocated.
- A small number of instructions have been added to the JAM emulator, to support mode-switching between emulated and native code.

A previous version of HiPE used dynamic tests in the JAM emulator instead. At each call, a check was made if the target also had a native-code version, and at each return, throw, and resume, a check was made if the return address was zero, which was interpreted as a signal to switch mode. That design required changes to many different locations in the emulator, complicated the mode-switch stack frame management, imposed runtime overheads on emulated code, and was generally ugly and difficult to maintain.

In contrast, our current design requires only a small localised extension of the emulator, and imposes no runtime overheads except during mode-switches.

4.6 Performance instrumentation features

To support performance analysis and benchmarking, and to help users identify which parts of their code could benefit most from compilation to native code, we have added two kinds of performance instrumentation features to HiPE's runtime system: *software event counters* and *hardware performance counters.*

The software event counters keep track of how often various interesting operations are performed. These include: the number of times each Erlang function is called, the number of times each built-in library function is called, how many times each JAM instruction is executed, and how many times control is passed between emulated and native code.

The hardware performance counters are event counters inside the UltraSPARC processor[4]. They are used to measure the number of clock cycles spent in code regions, and to provide hardware-specific information, for example the amount of time lost due to stalls and cache misses. The reason for a stall can also be determined: data cache miss, instruction cache miss, external cache miss, or a branch misprediction. For more details on the instrumentation facilities, the reader is referred to [14].

5. PERFORMANCE EVALUATION

We conducted our performance comparison on a 143 MHz single-processor Sun UltraSPARC 1/140 with 128 MB of primary memory running Solaris 2.6. Although slow by today's standards, this machine was used because we could ensure it was lightly loaded during the benchmark runs, and it had enabled user-level access to the UltraSPARC's performance instrumentation facilities [22]. The latter was needed for the performance results presented in the following section. When using HiPE, all functions actually called in the programs were compiled to native code.

5.1 Erlang systems used in the comparison

Besides HiPE (version 0.90), three other Erlang systems were used in this comparison: JAM, BEAM, and Etos. The JAM and BEAM systems used in our measurements are from Ericsson's Erlang 47.4.1 (upon which HiPE 0.90 is based). The version of Etos used is 2.3.[5] We describe BEAM

[4]Many modern processors have hardware performance counters, although their functionality and the level of OS support varies.

[5]Versions of Etos & HiPE used in [8] are significantly older than those used here.

and Etos below; HiPE and JAM have been fully described earlier in this paper.

BEAM

The BEAM [12] is a register-based abstract machine, influenced by the Warren Abstract Machine (WAM) [26] used in many Prolog implementations. Compared to JAM, the translation of Erlang code to BEAM abstract machine instructions is more advanced. For example, the treatment of pattern matching is better in the BEAM system, even though a full pattern match compiler (like that in e.g. [20]) is not implemented. Also, BEAM uses a direct-threaded emulator [4, 17] using `gcc`'s first-class labels extension [21]: instructions in the abstract machine code are addresses to the part of the emulator that implement the instruction.

Etos

Etos, described in [8], is a system based on the Gambit-C Scheme compiler. It translates Erlang functions to Scheme functions which are then compiled to C. The translation from Erlang to Scheme is fairly direct. Thus, taking advantages of the similarities of the two languages, many optimizations in Gambit-C are effective when compiling Erlang code. Among these optimizations are inlining of function calls (currently only *within* a single module) and unboxing of floating-point temporaries. Etos also performs some optimizations in its Erlang to Scheme translation, for example, simplification of pattern-matching. Process suspension in Etos is done using `call/cc` implemented using a lazy copying strategy; see [13]. When a process is suspended, the stack is "frozen" so that no frame currently on the stack can be deallocated. When control returns to a suspended process, its stack frames are copied to the top of the stack. When the stack overflows, the garbage collector moves all reachable frames from the stack to the heap. In general, suspending and resuming a process will require its stack to be copied at least once. In contrast, the JAM/BEAM/HiPE runtime systems handle processes explicitly; saving or restoring the state of a process involves storing or loading only a small number of registers. The Etos compiler is work under progress, and it is not yet a full Erlang implementation. We have therefore been able to run only relatively small benchmarks on Etos.

5.2 Time on benchmark programs

We start our performance comparison using the following set of "standard" small sequential benchmarks also used in [8]:

fib A recursive Fibonacci function. Calculates `fib(30)` 50 times.

huff Huffman encoder. Compresses and uncompresses a short string 5000 times.

length A tail-recursive list length function finding the length of a 2000 element list 100,000 times.

nrev Naive reverse of a 100 element list 20,000 times.

qsort Ordinary quicksort. Sorts a short list 50,000 times.

smith The Smith-Waterman DNA sequence matching algorithm. Matches one sequence against 100 others; all of length 32. This is done 30 times.

Benchmark	HiPE	Etos	JAM	BEAM
fib	33.8	31.8	281.4	120.6
huff	11.9	12.1	234.7	69.2
length	22.7	17.2	375.6	98.9
nrev	18.5	24.4	241.3	56.9
qsort	12.3	11.0	208.1	97.6
smith	11.4	11.6	114.6	53.9
tak	13.5	12.8	140.1	100.2
decode	22.8	52.4	67.8	49.0

Table 1: Times (in secs) for small sequential benchmarks.

tak Takeuchi function, uses recursion and integer arithmetic intensely. Calculates `tak(18,12,6)` 1000 times.

and a medium-sized one (≈ 400 lines):

decode Part of a telecommunications protocol. Decodes an incoming `binary` message 500,000 times.

Table 1 contains the results of the comparison. In all benchmarks, HiPE and Etos are the fastest systems: in small programs they are between 7 to 20 times faster than JAM and 3 to 8 times faster than the BEAM implementation; see also Figure 3. Excluding **length** and **nrev** where HiPE and Etos show complementary behaviour, the performance difference between these two systems on small programs is not significant. In **decode**, where it is probably more difficult for Etos to optimize operations and pattern matching on `binary` objects (i.e. on immutable sequences of binary data), HiPE is more than 2 times faster than Etos. HiPE is faster than JAM and BEAM, but not to the same extent as for the other benchmarks.

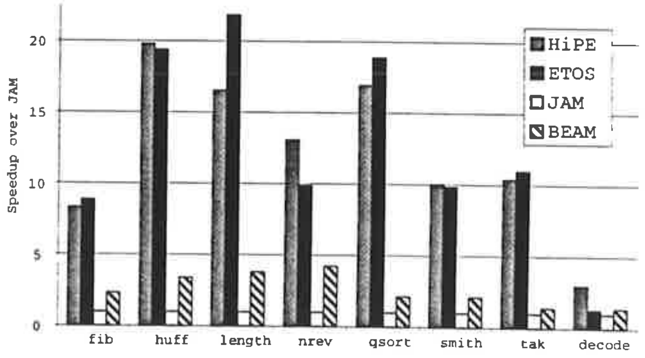


Figure 3: Speedup compared to JAM for small benchmarks.

Next, we compare the Erlang implementations on concurrent programs. As mentioned in the introduction, most Erlang programs rely heavily on the concurrency primitives of the language. Thus, these programs call for special attention in good Erlang implementations. The benchmark programs we used are:

ring Creates a ring of 10 processes and sends 100,000 messages. The benchmark is executed 100 times.

Benchmark	HiPE	Etos	JAM	BEAM
ring	37.1	76.0	101.6	72.5
stable	12.8	27.9	37.8	19.5
life	5.6	20.1	13.4	8.7

Table 2: Times (in secs) for small concurrent benchmarks.

stable Solves the stable marriage problem for 10 men and 10 women 5000 times.

life Executes 1000 generations in Conway's game of life on a 10 by 10 board where each square is implemented as a process.

Table 2 contains the results of the comparison. Once again, HiPE is the fastest system: it is around 2.5 times faster than JAM, 55% faster than BEAM (95% on **ring**), just over 2 times faster than Etos on **ring** and **stable** and more than 3.5 times on the **life** benchmark; see also Figure 4. In fact, Etos 2.3 does not seem to be significantly faster than JAM and is slower than BEAM when processes enter the picture. We suspect that Etos' implementation of concurrency via `call/cc` is not very efficient.

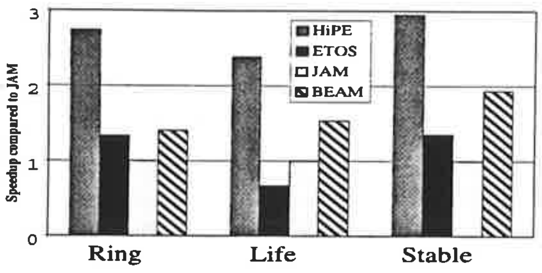


Figure 4: Performance speedup for concurrent benchmarks.

5.3 Time on real programs

To us, it was very unclear whether performance experiences gathered from the study of small or medium-sized benchmarks are applicable to real-life applications of Erlang. We thus also compared the performance of HiPE on quite large Erlang programs. The programs used in this endeavour were:

JAM Compiler This "application" is incestuous, but large nevertheless. The used portion of the compiler consists of 30 modules totalling $\approx 18,000$ lines of Erlang code. The benchmark is to compile 11 of these modules using the JAM compiler compiled in each of the systems.

Eddie An HTTP parser which handles 30 complex http-get requests. Excluding the OTP libraries used, it consists of 6 modules for a total of 1,882 lines of Erlang code. The benchmark is executed 1,000 times.

AXD/SCCT This is the time-critical software part of the AXD 301 switch mentioned in the introduction. Not counting standard libraries, it consists of about 95,000 lines of Erlang code. This actual benchmark of the ATM switch sets up and tears down a number of connections 100 times; 501 functions are used in the benchmark which amount to $\approx 15\%$ of the code. The remaining code provides other ATM services or handles errors that can occur in practice (but not during the benchmark run!). These parts were not compiled to native code.

Program	HiPE	JAM	BEAM
JAM Compiler	5.4	17.2	5.9
Eddie	18.8	93.6	40.0
AXD/SCCT	68.0	109.9	84.5

Table 3: Execution times (in seconds) for large programs.

Table 3 shows the results of this comparison.[6] HiPE is once again the fastest system. However, as we move from benchmarks to real-world applications of Erlang, programs tend to spend more and more of their execution time in built-ins from the standard library. For example, the **AXD/SCCT** program extensively uses the built-ins to access the shared database on top of the Erlang term storage; see the data presented in the Section 6 and in [14].

As the implementation of these built-ins is currently shared by all three systems, the percentage of execution spent in these builtins becomes a bottleneck and HiPE's speedup is less than before. Still, HiPE is 24% faster than BEAM on the largest benchmark, and considerably faster than the JAM implementation on which it is based; see also Figure 5.

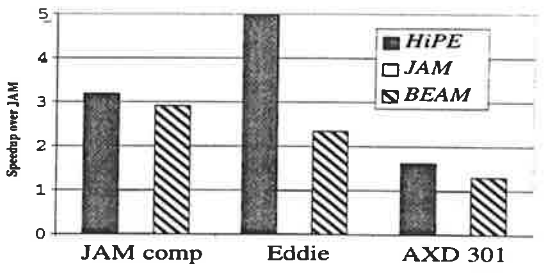


Figure 5: Performance speedup for large programs.

5.4 Code size

During benchmarking, all called functions of programs were compiled to native code. This makes perfect sense in small programs but might not be the best approach in all cases. As mentioned, users of HiPE can selectively compile time-critical parts of their applications to native code and use the emulator for the remaining parts. Still, HiPE's space overhead is not prohibitive and compilation of whole applications to native code is a valid option.

Table 4 provides some evidence to that effect. For JAM, BEAM, and HiPE it presents the size (in bytes) of the *loaded* code that is *used* by the benchmarks in this section. In all cases but **AXD/SCCT** all functions of loaded modules are

[6]Etos is not included here; it currently cannot handle these large programs.

Program	JAM	BEAM	HiPE	Etos†
fib	616	1937	4080	17808
huff	2530	7691	38728	90744
length	678	2085	4400	22424
nrev	750	2145	4932	24444
qsort	850	3049	6308	24784
smith	1416	4709	13616	71760
tak	656	2057	4712	23684
decode	1933	8641	14012	100132
ring	784	2737	6920	29380
life	1400	5069	15280	63772
stable	1299	4701	14008	59264
Eddie	21717	81075	170308	N/A
AXD/SCCT	83370	—	244936	N/A

Table 4: Code-size (in bytes) for the programs used.

used in the benchmark; in **AXD/SCCT** only a fraction of the functions is used. As the BEAM-based Erlang system does not provide any means of obtaining the size of individual functions (only of whole modules), we could not report the corresponding number for BEAM. For small programs, the code size increase of HiPE compared to JAM is between 7 to 15 times. Compared to BEAM, HiPE requires 2 to 3 times more code space (excluding **huff**). In large programs, the space overhead is on the low end; see e.g **AXD/SCCT**. For comparison, we also present the sizes of stripped `.o1` files produced by the Etos compiler; they are quite big partly due to the aggressive inlining performed by Etos 2.3. It is not clear to us to what extent Etos benefits from its use of inlining. Note that the code size reported for Etos might differ slightly from the code size of these programs when loaded; however, not by much. Table 4 denotes this by a †.

5.5 Erlang vs. other functional languages

We have compared the performance of HiPE with that of several other functional language implementations (Bigloo 2.1c, CML from SML/NJ 110.0.6, CLEAN 1.3.2) on a few small benchmarks (versions of the **qsort**, **fib**, **huff**, and **ring** benchmarks described previously).

On the small sequential benchmarks, CLEAN was consistently the fastest system. The other three systems showed considerable variance in their performance, with no clear winner. Compared to CLEAN, CML was from 1.75 to 4.4 times slower, Bigloo 1.33 to 13 times slower, and HiPE 1.88 to 14.8 times slower. HiPE and Bigloo had comparable performance, except on **qsort** where HiPE was more than twice as fast. On the concurrent **ring** benchmark, HiPE was 1.3 times slower than CML, while the JAM emulator was 1.7 times slower than CML.

See [15] for further details about this comparison.

6. MORE DETAILED MEASUREMENTS

To shed more light on HiPE's performance characteristics compared to the other two Erlang systems from Ericsson, we present some additional measurements on three benchmarks of different sizes: **length**, **Eddie**, and **AXD/SCCT**. These measurements were obtained by enabling the performance instrumentation features of the three systems (see Section 4.6) during their installation; this explains slight mismatches with performance numbers reported in the previous section. A more thorough analysis and comparison of these systems can be found in [14].

length is a small sequential benchmark consisting of two nested loops, one that traverses a list and one that iterates a number of times.

On this benchmark HiPE is 16 times faster than JAM and over 3 times faster than BEAM; see Table 5. The byte-code emulation overhead is evident in this benchmark where HiPE executes only 6 million SPARC instructions while JAM and BEAM execute 49 and 10 million instructions respectively. JAM and BEAM also have problems (10% and 14% of their run time, respectively) with pipeline stalls from branch mispredictions; see Table 7. The same table shows that all three systems have some problems with load stalls (a value is needed before it has been completely loaded into a register) but none of the systems suffers from instruction cache misses.

Eddie is a mildly concurrent (only 5 messages are sent) benchmark which parses HTTP requests. The parser consists of four modules, another two are added for the benchmarking purposes, 4.7% of the calls are to built-in C functions, and some Erlang/OTP standard libraries are heavily used. The benchmark consists of 159 different Erlang functions that are called a total of $\approx 30,000$ times.

On this benchmark, HiPE is 6.3 times faster than JAM and over 2 times faster than BEAM. Here the improved BEAM compiler almost makes up for the emulation overhead: BEAM executes 4 million SPARC instructions which is not much more than HiPE's 3 million and a lot less than JAM's 13 million. All three systems have about the same percentage of pipeline stalls: 36% for JAM and HiPE, and 37% for BEAM.

AXD/SCCT is a time critical part of Ericsson's AXD 301 ATM switch. **SCCT** is responsible for setting up and tearing down connections in the switch. The code we have used is from version 6 of AXD 301, an earlier version than what is used in the product today. This benchmark is more concurrent than **Eddie**, with several processes and over 3000 messages sent. It also uses the built-in functions heavily: about 32% of the execution time for JAM is spent in built-ins. In absolute terms, this time is the same in HiPE; in relative terms, it corresponds to 50% of HiPE's execution time and built-ins start becoming a bottleneck. **SCCT** is a large benchmark: the total size of the 501 called functions is in native code 244,936 bytes (see also Table 4).

	JAM	BEAM	HiPE
length	1	4.5	16.1
Eddie	1	2.6	6.3
AXD/SCCT	1	1.4	1.6

Table 5: Speedups for BEAM and HiPE compared to JAM calculated from the total execution times in clock cycles.

		Mc	*Mi*	*CPI*
length	**JAM**	63	49	1.30
	BEAM	14	10	1.45
	HiPE	4	6	0.70
Eddie	**JAM**	18	13	1.39
	BEAM	7	4	1.54
	HiPE	3	3	1.07
AXD/SCCT	**JAM**	153	101	1.52
	BEAM	111	62	1.79
	HiPE	97	62	1.56

Table 6: Measurements for the benchmarks. The *Mc* column is rounded average execution times in million of cycles. The *Mi* column is the average number of millions executed instructions. The *CPI* column is the number of cycles per instruction (the lower the better).

On this benchmark HiPE is only about 1.6 times faster than JAM and 1.15 times faster than BEAM. Here the improved BEAM compiler completely makes up for the emulation overhead. This is indicated by the fact that BEAM and HiPE both execute about 62 million SPARC instructions on this benchmark; see Table 6.

The execution times in millions of clock cycles (*Mc*) and millions of executed instructions (*Mi*) for each system and benchmark are shown in Table 6.

The high speedup on **length** for HiPE compared to JAM has two reasons:

1. Because of compilation to native code and various optimizations, HiPE executes less than 12% of the number of SPARC instructions that JAM executes.
2. By utilizing the pipeline better, HiPE executes 1.86 times as many instructions per cycle as JAM.

With fewer instructions executed and better pipeline utilization, HiPE gets a speedup of 16 (8.6*1.86) times over JAM; this is shown in Table 5.

On **Eddie**, as on **length**, HiPE and BEAM execute considerably less instructions than JAM, and HiPE has a lower *CPI* than JAM and BEAM. This benchmark uses built-in functions and concurrency. This together with the use of different types of calls probably is the reason that the speedup for HiPE is not as great as it was on **length**. On **Eddie**, all three systems spend more than 35% of the time stalling, mainly on loads. HiPE has significantly higher instruction cache stalls than the other two system, while JAM and BEAM suffer more from branch misprediction stalls; see Table 7.

Because of the large size of the generated native code, on **AXD/SCCT** the absolute number of instruction cache stalls is about twice as many for HiPE (22% of 97 million) as for JAM (6% of 153 million). But taken together, the number of stalls from mispredictions and stalls from instruction cache misses is about the same for JAM and HiPE. All three systems spend about 40% of their total execution time stalling

		%L	*%M*	*%IC*	*%Total*
length	**JAM**	12	10	0	22
	BEAM	9	14	1	23
	HiPE	16	0	2	18
Eddie	**JAM**	23	10	2	36
	BEAM	23	11	3	37
	HiPE	18	4	14	36
AXD/SCCT	**JAM**	23	12	6	41
	BEAM	25	8	7	40
	HiPE	16	5	22	43

Table 7: Percentage of the execution time spent on the most common types of pipeline stalls. The column *%L* shows the percentage of load stalls, *%M* shows the percentage of branch misprediction stalls, *%IC* shows the percentage of instruction cache stalls, and finally the *%Total* column shows the total percentage of stalls. All numbers are based on the number of executed machine cycles.

on this benchmark. This indicates that even though HiPE runs into problems with the instruction cache, HiPE does not suffer more from this than JAM or BEAM suffer from other types of stalls. One reason why HiPE does not achieve the same speedup as on smaller benchmarks is that much of the time is spent in code currently outside HiPE's control, such as built-in functions and garbage collection. Ignoring the time spent in built-in functions, the garbage collector, and the operating system, HiPE is 2 times faster than JAM on this benchmark.

7. CONCLUSIONS AND FUTURE PLANS

This paper has described the architecture and implementation of HiPE, a native code compiler for Erlang. HiPE offers flexible, user-controlled integration between interpreted and native code, is a complete implementation of Erlang, and supports features crucial for telecommunication applications such as concurrency, error handling, and hot-code loading. As our performance evaluation shows, HiPE is the fastest of current Erlang implementations.

HiPE is publicly available as open-source since March 2000. The current release is based on Erlang 47.4.1. Since the Ericsson releases of open-source Erlang will exclusively use the BEAM implementation from now on, we plan to port HiPE to the BEAM run-time system. At present HiPE only runs on one platform, the UltraSPARC. To improve the usefulness of the HiPE system, we are currently developing a code generator for the x86 processor family. We are also developing a new front-end for HiPE so that compilation does not necessarily rely on the outcome of a previous compilation to JAM (or to BEAM). Instead, the new front-end will be based on Core Erlang,[7] an intermediate representation for Erlang developed recently. Since Core Erlang is a fairly high-level functional language, we expect that it should be easier to include optimizations, for example efficient pattern-match compilation, at that level.

Our benchmark results indicate that real-world applications spend a large fraction of their time in built-in standard pro-

[7]See `http://www.csd.uu.se/projects/hipe/corerl`.

cedures. We will therefore investigate this issue further, and extend the scope of HiPE accordingly. This may require improving the compiler, tuning the runtime system, or both.

Acknowledgements

HiPE has greatly benefited from the prior involvement of Christer Jonsson and from various design discussions with Thomas Lindgren, Sven-Olof Nyström, and Richard Carlsson. We also thank some anonymous reviewers for comments which helped improve the presentation of this paper. This research has been supported in part by ASTEC (Advanced Software Technology) competence center.

8. REFERENCES

[1] J. Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, June 1997.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] T. P. Baker and G. A. Riccardi. Implementing Ada exceptions. *IEEE Software*, 3(5):42–51, Sept. 1986.

[4] J. R. Bell. Threaded code. *Communications of the ACM*, 16(8):370–373, June 1973.

[5] S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.

[6] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.*, 16(3):428–455, May 1994.

[7] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report TR87-011. Available from: http://www.cs.indiana.edu/scheme-repository/.

[8] M. Feeley and M. Larose. Compiling Erlang to Scheme. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in LNCS, pages 300–317. Springer-Verlag, Sept. 1998.

[9] A. D. Gordon. How to breed hybrid compilers/interpreters. Technical Report ECS-LFCS-88-50, Department of Computer Science, University of Edinburgh, 1988.

[10] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, Oct. 1993.

[11] P. H. Hartel et al. Benchmarking implementations of functional languages with "pseudoknot", a float intensive program. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[12] B. Hausman. Turbo Erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[13] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[14] E. Johansson, S.-O. Nyström, T. Lindgren, and C. Jonsson. Evaluation of HiPE, an Erlang native code compiler. Technical Report 99/03, ASTEC, Uppsala University, 1999.

[15] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. Technical report, ASTEC, Uppsala University, 2000. In preparation.

[16] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons, 1996.

[17] P. Klint. Interpretation techniques. *Software – Practice and Experience*, 11(9):963–973, Sept. 1981.

[18] G. Lal. MLRISC: Customizable and reusable code generators. Unpublished technical report available from: http://www.cs.bell-labs.com/~george, 1996.

[19] S. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In G. Nadathur, editor, *Principles and Practice of Declarative Programming: Proceedings of International Conference PPDP'99*, number 1702 in LNCS, pages 1–28. Springer-Verlag, Sept. 1999.

[20] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[21] R. M. Stallman. Using and porting gcc. Technical report, The Free Software Foundation, 1993.

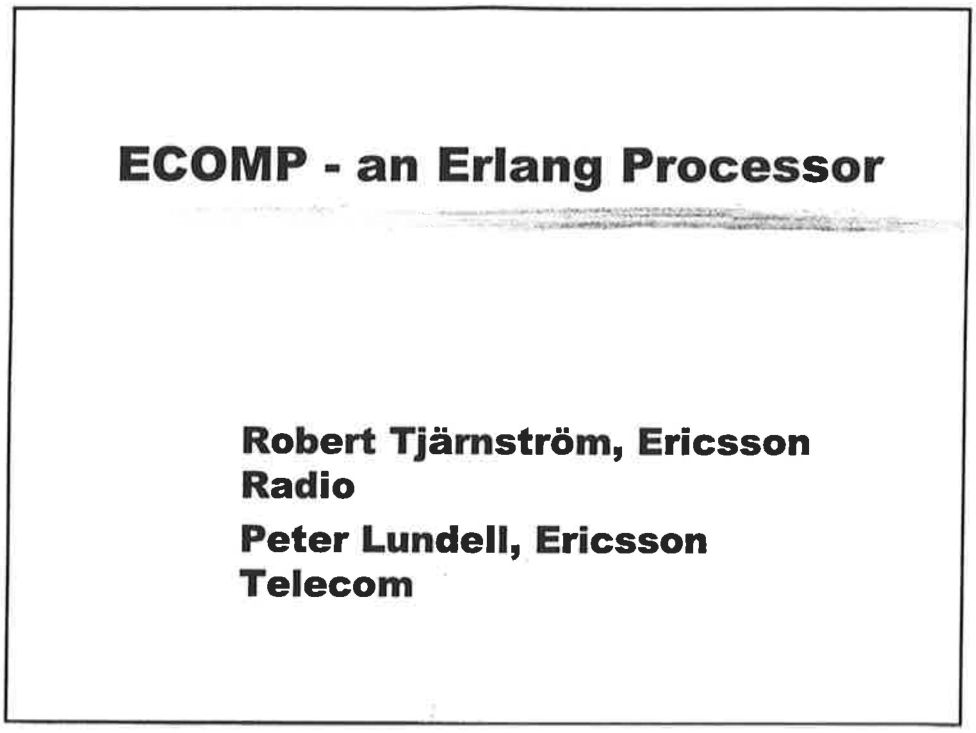
[22] Sun Microsystems. UltraSPARC^{TM} User's Manual. Technical report, Sun Microelectronics, Palo Alto, CA, 1997.

[23] S. Torstendahl. Open telecom platform. *Ericsson Review*, 75(1):14–17, 1997. See also: http://www.erlang.se.

[24] R. Virding. A garbage collector for the concurrent real-time language Erlang. In H. G. Baker, editor, *Proceedings of IWMM'95: International Workshop on Memory Management*, number 986 in LNCS, pages 343–354. Springer-Verlag, Sept. 1995.

[25] P. Wadler. An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, Feb. 1998.

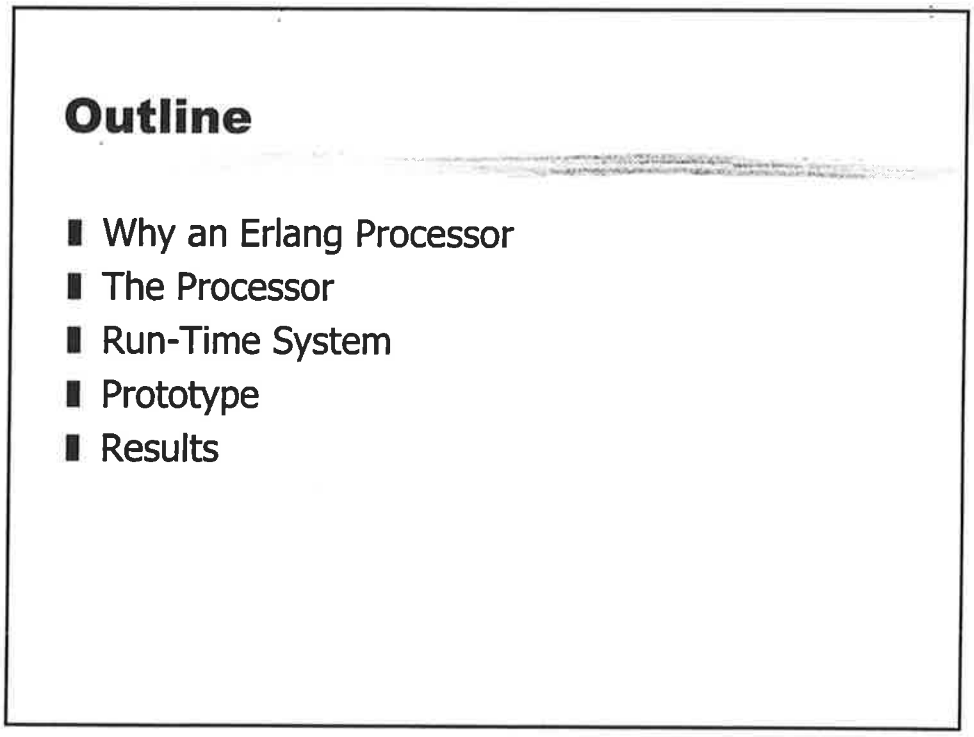
[26] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.



ECOMP - an Erlang Processor

Robert Tjärnström, Ericsson Radio

Peter Lundell, Ericsson Telecom

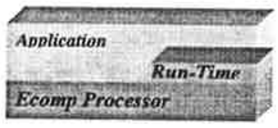


Outline

- Why an Erlang Processor
- The Processor
- Run-Time System
- Prototype
- Results

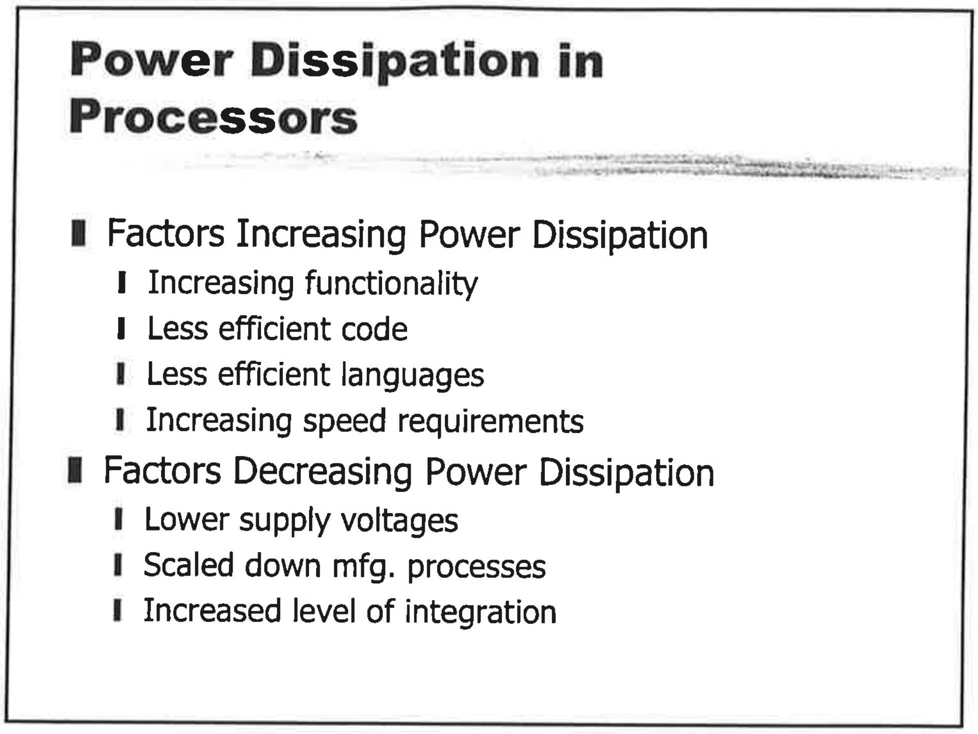
What Is an Erlang Machine

- A (micro) processor dedicated for execution of Erlang.
  - Executes compiled Erlang code.



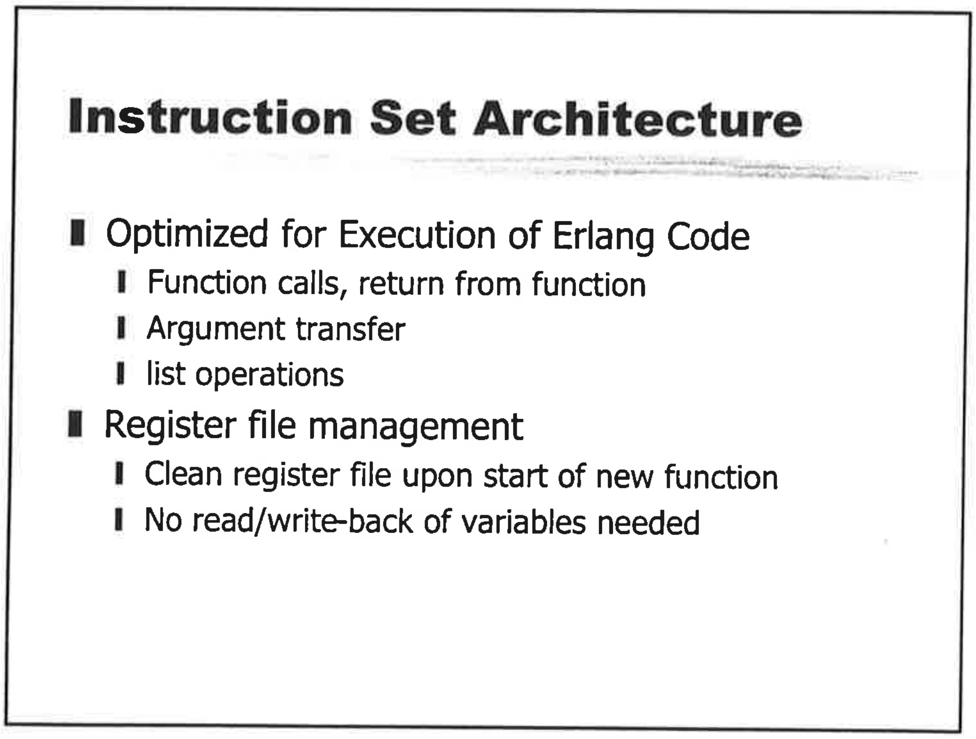
Why a Dedicated Erlang Processor

- Increased use of Erlang
- Eliminating Performance and Power Dissipations Concerns
  - Low Power Important in Embedded Control
- Simplify use of Erlang for Embedded Control
- Eliminate cost for Real-Time Operating System
  - Provide run-time functionality



Power Dissipation in Processors

- Factors Increasing Power Dissipation
  - Increasing functionality
  - Less efficient code
  - Less efficient languages
  - Increasing speed requirements
- Factors Decreasing Power Dissipation
  - Lower supply voltages
  - Scaled down mfg. processes
  - Increased level of integration



Instruction Set Architecture

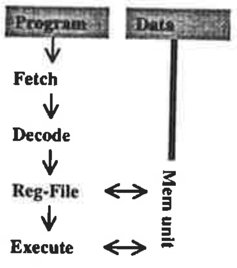
- Optimized for Execution of Erlang Code
  - Function calls, return from function
  - Argument transfer
  - list operations
- Register file management
  - Clean register file upon start of new function
  - No read/write-back of variables needed

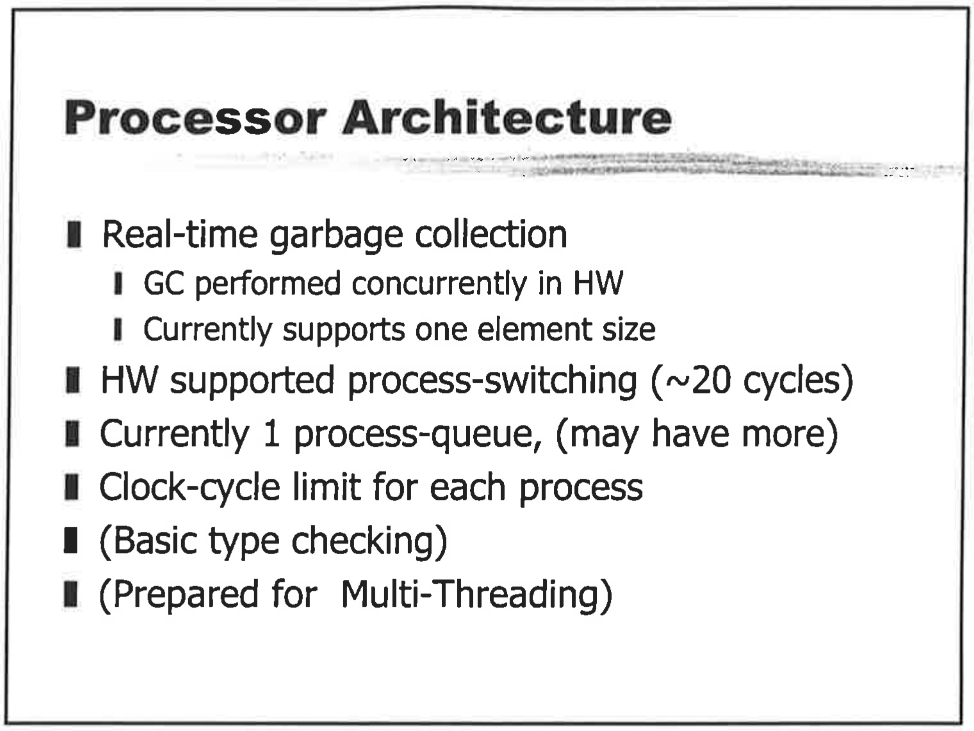
Instruction Set Architecture

- Supports processes
- Supports local scope
- Three sub-instructions in each machine instruction
  - Sub-instructions for garbage collections

Processor Architecture

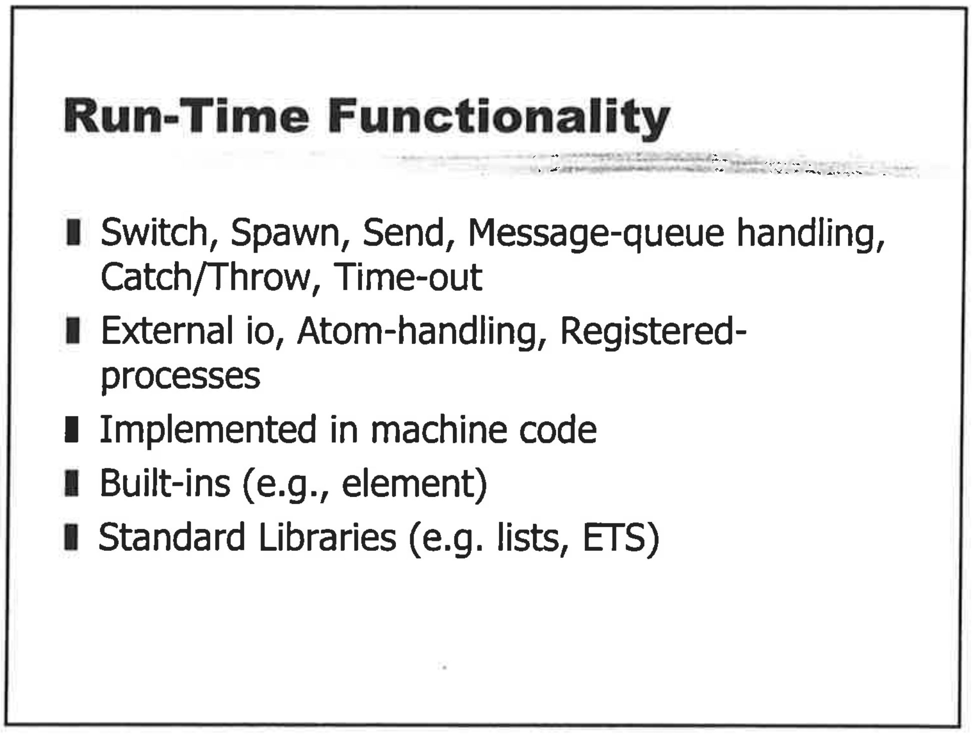
- Much in common with conven-
- tional architectures
  - RISC
  - LIW
  - Harvard
  - Pipelined (3-5 stages)
- No complex (advanced) features
  - Not super-scalar
  - No OOO-execution or speculative execution
  - No branch-prediction (but will be added)





Processor Architecture

- Real-time garbage collection
  - GC performed concurrently in HW
  - Currently supports one element size
- HW supported process-switching (~20 cycles)
- Currently 1 process-queue, (may have more)
- Clock-cycle limit for each process
- (Basic type checking)
- (Prepared for Multi-Threading)



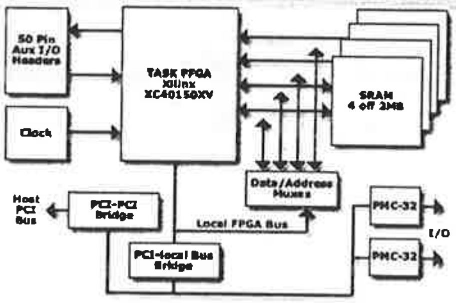
Run-Time Functionality

- Switch, Spawn, Send, Message-queue handling, Catch/Throw, Time-out
- External io, Atom-handling, Registered-processes
- Implemented in machine code
- Built-ins (e.g., element)
- Standard Libraries (e.g. lists, ETS)

Prototyping

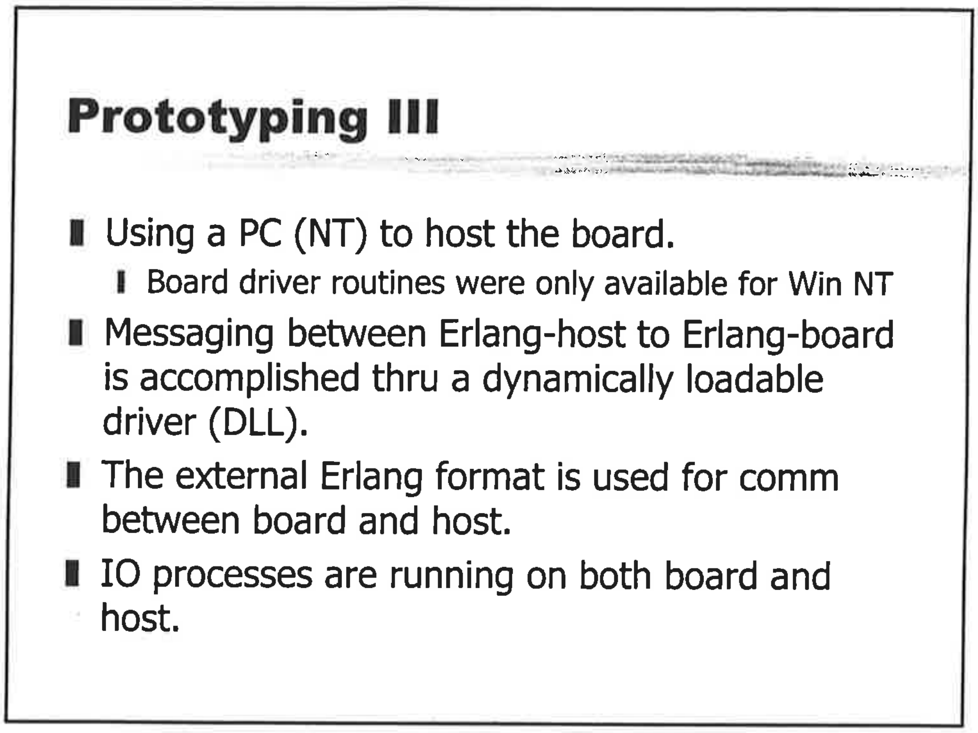
- HW model of the processor (developed in Erlang)
- VHDL implementation & test bench
- FPGA based demonstrator (VHDL-code)

Prototyping II



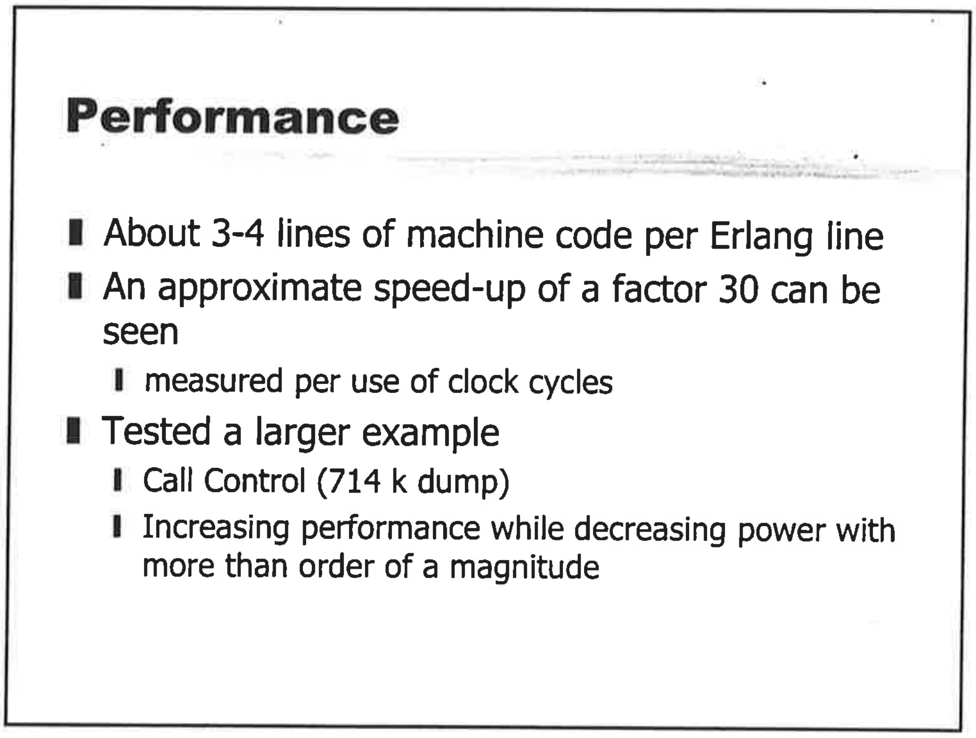
Functional Block Diagram Of RC1000-PP

- PCI Board with Xilinx 40150 FPGA and 4 banks of 2 MB SRAM each
- Board has PCI bridge (slows down communication)



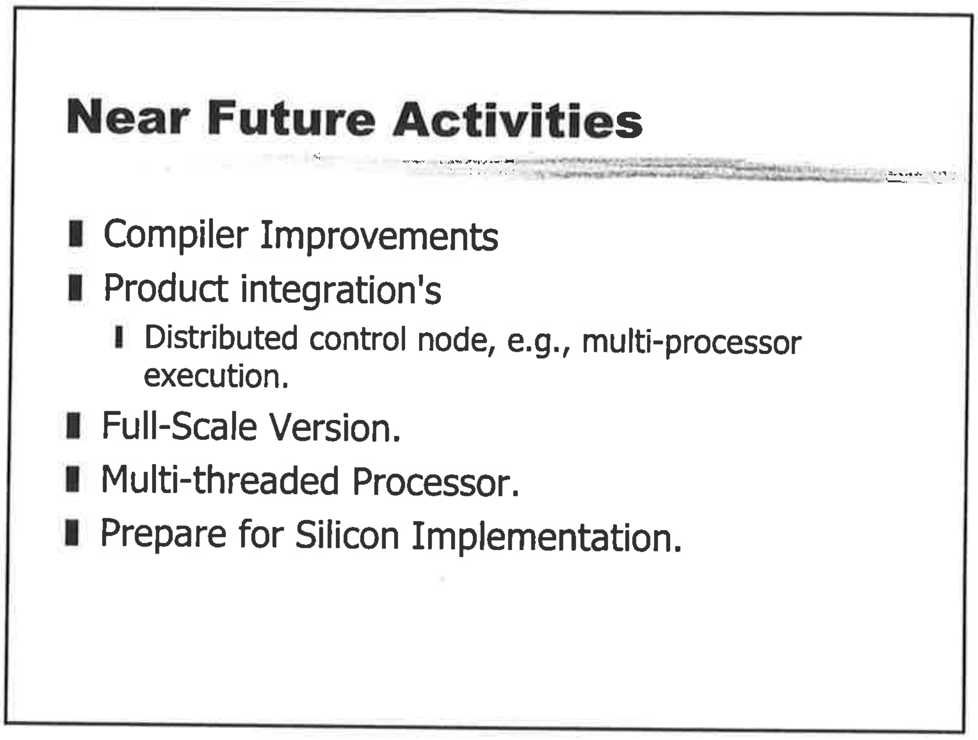
Prototyping III

- Using a PC (NT) to host the board.
  - Board driver routines were only available for Win NT
- Messaging between Erlang-host to Erlang-board is accomplished thru a dynamically loadable driver (DLL).
- The external Erlang format is used for comm between board and host.
- IO processes are running on both board and host.



Performance

- About 3-4 lines of machine code per Erlang line
- An approximate speed-up of a factor 30 can be seen
  - measured per use of clock cycles
- Tested a larger example
  - Call Control (714 k dump)
  - Increasing performance while decreasing power with more than order of a magnitude



Near Future Activities

- Compiler Improvements
- Product integration's
  - Distributed control node, e.g., multi-processor execution.
- Full-Scale Version.
- Multi-threaded Processor.
- Prepare for Silicon Implementation.

Erlang/OTP User Conference 2000

An Erlang DTD

Richard A. O'Keefe

CS, University of Otago

Erlang/SGML is not an SGML parser for or in Erlang.

If you want one, parse the ESIS output of an 'sgmls' or 'nsgmls' process (dead easy).

Or you could pick up Jan Wielemaker's sgml package for SWI Prolog (the parser core is in C; it calls foreign interface code to build Prolog data structures) and adapt that (considerably faster than nsgmls but less capable).

Erlang/SGML is not an XML parser either (I've done one 10-20% faster than expat in 950 lines of C).

Erlang/SGML is not primarily a literate programming tool for Erlang. Unlike most LP tools, it cannot generate multiple source files from a single document. Currently it does not handle fragments, which Erlang has little need of. It *does* handle cross-referencing.

Noweb, Nuweb, and Funnelweb work perfectly well with Erlang, so we don't need a new LP tool for Erlang.

If we did want one, there's Kristina Sirhuber's YERL.

It's all about document indexing to support the maintenance process.

Hypothesis:

> Maintenance programmers seeking information or trying to figure out the consequences of a change have to find related documents.
>
> Improving that step should help maintenance.

Isn't "maintenance" a rather old-fashioned way of thinking about a neat fast-development language like Erlang?

No. Faster development means that the program is useful sooner, which just means that the maintenance phase is longer. ("Longer" is not the same as "more expensive".)

Extreme Programming involves continuing redesign, which absolutely requires good tools for finding related places in related documents easily and quickly.

Erlang/SGML is part of the Large Scale Erlang project. Approaches that work well for 300 kSLOC do not scale to 100 MSLOC.

One major requirement is a water-tight module system. (I offered two papers this year; that would have been number 3.)

Another requirement is more contextual information, so that far less human filtering is required.

See the "SIF" problem later.

What are SGML and XML anyway?

SGML is a language in which one may define document grammars, with rules like

```
<!ELEMENT module - O
    (metadata,
     (function|pattern|constant|
      typedef|protocol|p)*,
     section*,glossary?,index*)>
<!ELEMENT section - -
    ((function|pattern|constant|
      typedef|protocol|p)*,
     section*,glossary?,index*)>
```

Abstractly, a document is a tree, where each node is either a text node or a labelled node; a labelled node has a label, a set of attribute=value pairs, and a possibly empty sequence of children. Nodes may be given unique identifiers, which are used for cross-references within a tree.

Developing a document grammar is basically an exercise in information modelling, rather like the conceptual modelling phase of the Unified Process.

There are several semi-formal models of the information content of a document. The SGML standard has Element Structure Information Sets; HyTime and DSSSL have Graph Representations Of property ValuEs. XML has several not quite mutually compatible ones, but is like SGML.

SGML	XML
writable	need machine help
readable	need machine help
concise	amazingly bulky
standard, stable	W3C, grows fast (Namespaces, XBase, ...)
tricky to parse	trivial to parse
easy to process	harder to process

The Erlang SGML DTD and the Erlang XML DTD both include the same grammar file; the SGML version enables human readability features such as tag omission and short reference strings.

Erlang source: 39k; stripped: 21k;
as SGML: 43k; as XML: 104k.

Erlang/SGML expresses

- text (based on HTML with strong influence from TEI, but reasonable restraint)
- tables (based on HTML4 but not as powerful)
- mathematics (not ISO 12083, not HTML 3.0, most certainly not MathML which is not meant for human beings, but home brew; *nearly* as concise as LaTeXbut stricter)
- pictures (images via Encapsulated Postscript, diagrams via Pic)
- ... continued ...

Erlang/SGML expresses

- Erlang source code (enforces all syntactic constraints except for what's allowed in guards; could do that too)
- glossaries
- indices
- Dublin core metadata
- examples

Status

- Erlang source code $\rightarrow$ markup: done in Prolog. Complete.
- markup $\rightarrow$ Erlang source code: done in Prolog. Complete.
- parser: SP 'nsgmls' (a wee bit too complex for 'sgmls'), SWI 'sgml'.
- SGML to XML conversion: done by 'nsgmls'
- document searching: LT XML toolkit (good) or sgrep (poor).
- ...

. . . Status:

- editing:
  by hand (own editor) or Emacs SGML mode (psgml). Free XML editors are disappointing.
- formatting:
  patchwork of AWK scripts; to be redone in Prolog.
- manual:
  in LaTeX; woefully incomplete but slowly growing.

Why Prolog? Because I have Prolog on my home Macintosh but not Erlang. Also because of SWI Prolog/SGML kit. Should migrate to Erlang when complete.

The context problem: is 'SIF'

- the name of a Goddess?
- part of a word ('SIFTER' perhaps)?
- a file extension (Smalltalk Interchange Format)?
- some other acronym?
- a tag used inside some data structure?
- a module name?
- a function name without its module and arity?
- a type name?

If I'm looking for references to a module, I do not want to be distracted by Norse goddesses.

The context solution. Use a document grammar which is based on Erlang syntax but enriches it with information people need and compilers don't.

<name myth/SIF/	goddess
<text/SIF/	text fragment
<atom u=ext/SIF/	file extension
<acr/SIF/	acronym (→<glossary>)
<atom u=tag/SIF/	data structure tag
<modname/SIF/	module name
<funcname/SIF/	local function name
<funcname m=x/SIF/	imported function name
<typename/SIF/	type name
<pidname/SIF/	PID name

<name>, <acr>, <abbr>. <modname>, <funcname>, <patname>, <pidname> and <type already existed for use in text; <text> already existed for use in mathematical formulas. Allowing them to be used where an otherwise unclassified <sym> would have been used was easy.

There is no way to predict all possible uses for an atom.

```
<!ELEMENT atom - - (#PCDATA)>
<!ATTLIST atom u NAME #IMPLIED>
```

says that the 'u' attribute of a <sym> can be any name.

<text> normally allows all sorts of markup inside it including mathematical formulas; that doesn't make a lot of sense in atoms. SGML lets us write

```
<!ELEMENT expr (...)
-(em|strong|math|...)>
```

The exclusion here says that certain tags are not allowed anywhere inside an <expr> element even if the rest of the grammar (such as the definition of <text> says that they are. XML does not have exclusions, more's the pity (XHTML really needs them).

What a system can't figure out, it can be told. The u= attribute lets us tell Erlang/SGML how an atom is being used.

If a system can figure something out, it should. This is very like type checking, but it goes beyond current Erlang type systems. It should be possible to automatically propagate usage information.

But no type checker will analyse the Erlang atoms that appear in the explanatory text, and we do need to find them during maintenance.

Nothing in Erlang syntax expresses a relationship between documents. Directives such as

```
-module(this).
-export_to(that, [a/1,b/2]).
-import(the_other, [c/3]).
```

express relationships between *this* document and some modules. The links are completed at run time by loading files.

No official annotations express relationships between source files and other documents such as standards, requirements, test plans, user documentation, you name it.

There are not even any clear suggestions about how to use -author -vsn and so on.

There *is* an official set of document annotations for cataloguing and indexing purposes: the Dublin Core.

Dublin Core p1

- <identifier/unambiguous formal reference for this resource/
- <title/Name by which humans know this resource/
- <subject/keywords,key phrases,ACM codes,wh
about,for indexing/
- <description/Reasonably full description (lengthy abstract)/
- <coverage/scope, *e.g.*, which standards supported/
- <date/yyyy-mm-dd/
When this resource became available

Dublin Core p2

- <language/en_NZ/
one for each language used; I extend this
- <type/software/
but useful in descriptions of other files
- <format>text/SGML</format>
but useful to describe other files
- <creator/Repeat with name of everyone taking major part in creation (-author?)
- <contributor/Repeat with name of each contributor (minor maintenance?)/

Dublin Core p3

- <publisher/Who caused this to be released/
- <rights/State or cite rights held—
  like -copyright but more possibilities/
- <source/unambiguous formal reference for base of derivation/
- <relation/unambiguous formal reference to related document/

I propose adding a new directive to Erlang:

-dc(Attribute, Value[, Qualifier(s)]).

```
-dc(creator, "Karl Marx", [{type,architect}]).
-dc(creator, "Groucho Marx", [{type,coder}]).
-dc(contributor, "Greasy Marks").
-dc(rights, "Copyright (c) 2001 FuBar Ltd").
-dc(rights, "See licence.txt").
```

The difficult thing about doing this is

studying all the relevant literature to glean techniques and ideas.

- Should paragraphs be like HTML or TEI? (TEI)

- What should tables be like?

- How to express mathematics in SGML so that a human can type formulas with useful syntax checking but without dying of exhaustion? (I have used a point-and-click equation editor and it *stank*. Even typing MathML by hand was faster, which says a lot.)

- How best to use the SHORTREF feature to make typing Erlang easy?

SGML lets you attach macros to strings contextually. So

- "abc" →
  <str>abc</str>

- 'abc' →
  <sym>abc</sym>

- ['a','b','c'] →
  <elst><sym>a</sym><expn>
  <sym>b</sym></expn><expn>
  <sym>c</sym></expn></elst> in expressions

- ['a','b','c'] →
  <plst><sym>a</sym><patn>
  <sym>b</sym></patn><patn>
  <sym>c</sym></patn></plst> in patterns

Makes Erlang code much more readable and writable; not available in XML.

Erlang/SGML has type declarations and annotations. When generating Erlang source code, they can simply be ignored. Erlang/SGML has protocol declarations; a protocol is a data type for a process's message queue elements (as in OCCAM). Purpose is documentation and maintenance. When generating Erlang source code, they are ignored. When you want to know what P!{frazzle,X} means, it's nice to have

```
<import m=nutwork><protoname p=nutty></import>
...
<send p=nutty>@P,{'frazzle',@X}</send>
```

so we know at once where to look for the documentation.

For references to documents, file names are too fragile, and in practice URIs are too. SGML's Formal Public Identifiers are a time tested way to provide stable names for documents. Catalogues provide local mappings.

79 formal public identifier = owner identifier, "//", text identifier

80 owner identifier = ISO owner identifier | registered owner identifier | unregistered owner identifier

82 registered owner identifier = "+//IDN ", domain name | "+//", other data

83 unregistered owner identifier = "-//", data

84 text identifier = public text class, " ", unavailable text indicator?, public text description, "//", public text language designation, ("//", public text display version)?

E.g., `"+//IDN cs.otago.ac.nz//DOCUMENT Erlang/SGML report//EN"`

FPI's are text telling people who the owner is and what the document is called; ask them where it is, update your catalogue.

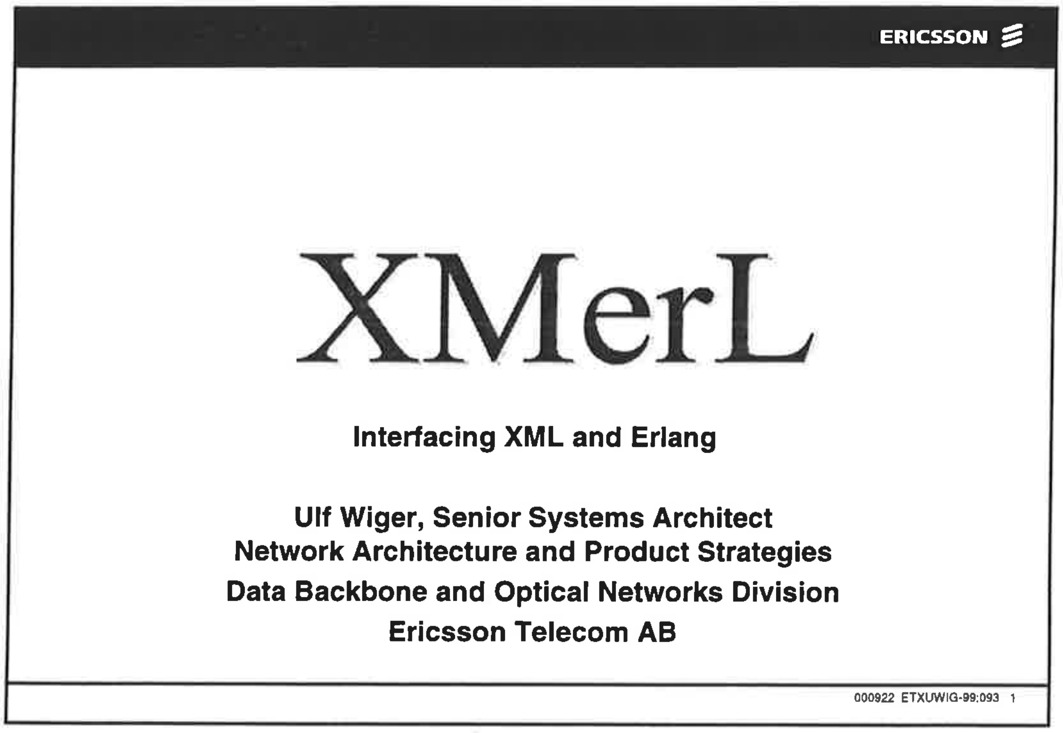
FPI's may also be internet URN's.

Kristina Sirhuber found that Ellemtel and Uppsala people

- Did not like the tangling step.
  Not a problem: an Erlang compiler could work straight from the Erlang/SGML source (no reordering or fragments in the Erlang DTD)

- Did not like the idea of having to learn yet another language
  is a problem; if you want to produce well documented programs you *have* to learn a language other than Erlang. But the Erlang manual set is written using SGML, and it's rather simpler than LaTeX.

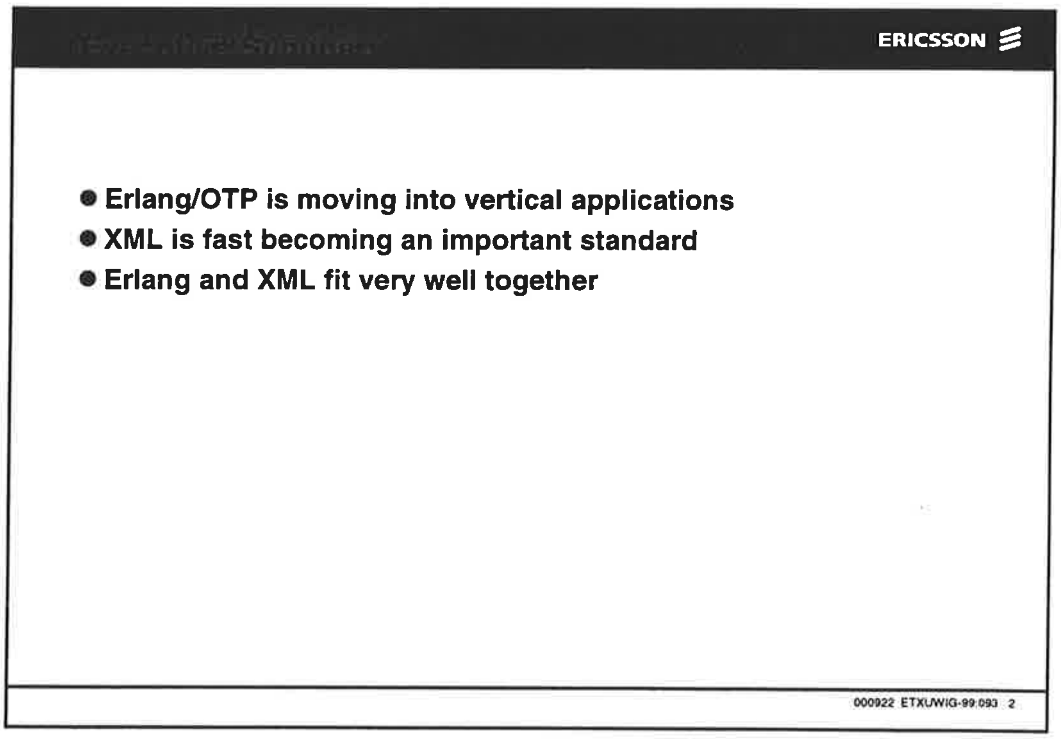
Of course, "Bird tracks" would be even simpler, but would not solve the context problems.


ERICSSON

XMerL

Interfacing XML and Erlang

Ulf Wiger, Senior Systems Architect
Network Architecture and Product Strategies
Data Backbone and Optical Networks Division
Ericsson Telecom AB


Executive Summary

ERICSSON

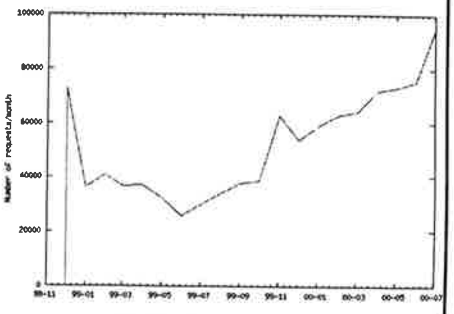
- Erlang/OTP is moving into vertical applications
- XML is fast becoming an important standard
- Erlang and XML fit very well together

[illegible]

ERICSSON

- Interest in Erlang is growing
- No longer just for embedded systems
- New interfaces must evolve
  - Powerful GUI components
  - Data exchange (COM, ODBC, XML, ...)
- XML is a logical addition to OTP
  - (ASN.1, HTTP, IDL, CORBA, ...)

- Real reason:
  - I bought a book and became curious



Number of Requests to www.erlang.org

[illegible]

ERICSSON

- "A Stricter HTML"
- "A Simpler SGML"
- Relatively Easy to Parse
- Content Oriented

- XML springs mostly from SGML
  - All non-essential SGML features have been removed
  - Web address support taken from HTML, HyTime and TEI
  - Some new functionality added
    - Modularity
    - Extensibility through powerful linking
    - International (Unicode) support
    - Data orientation

ERICSSON

- **Large Web sites**
  - HTML is generated via special (XSL) stylesheets
  - Internet Explorer has built-in support for XML
- **Document management**
  - When machines must be able to read the documents
- **Machine-to-machine communication**
  - XML RPC, SOAP
  - XML processors exist in many languages (even Erlang!)

ERICSSON

```
<?xml version="1.0"?>
<home.page title="My Home Page">
    <title>
        Welcome to My Home Page
    </title>
    <text>
        <para>
            Sorry, this home page is still under
            construction. Please come back soon!
        </para>
    </text>
</home.page>
```

- All elements must have a start tag and an end tag (exception: <empty.tag/>)
- An element can have a list of attributes

Erlang analogy:
{Tag, Attributes, Content}

ERICSSON

XML

```
<?xml version="1.0"?>
<home.page title="My Home Page">
   <title>
      Welcome to My Home Page
   </title>
   <text>
      <para>
         Sorry, this home page is still under
         construction. Please come back soon!
      </para>
   </text>
</home.page>
```

Erlang

```
{'home.page', [{title, "My Home Page"}],
  [{title, "Welcome to My Home Page"},
   {text,
     [{para,
         "Sorry, this home page is still under "
         "construction. Please come back soon!"}
     ]}
  ]}.
```

Almost equivalent

ERICSSON

- **XML is more complex than that**
  - **External DTDs**
  - **Global namespace**
  - **Language encoding**
  - **Structural information should be optimized for queries**
- **To parse XML properly, we use records**
- **To output to XML (or similar), we may use the simple form**

Example record definition

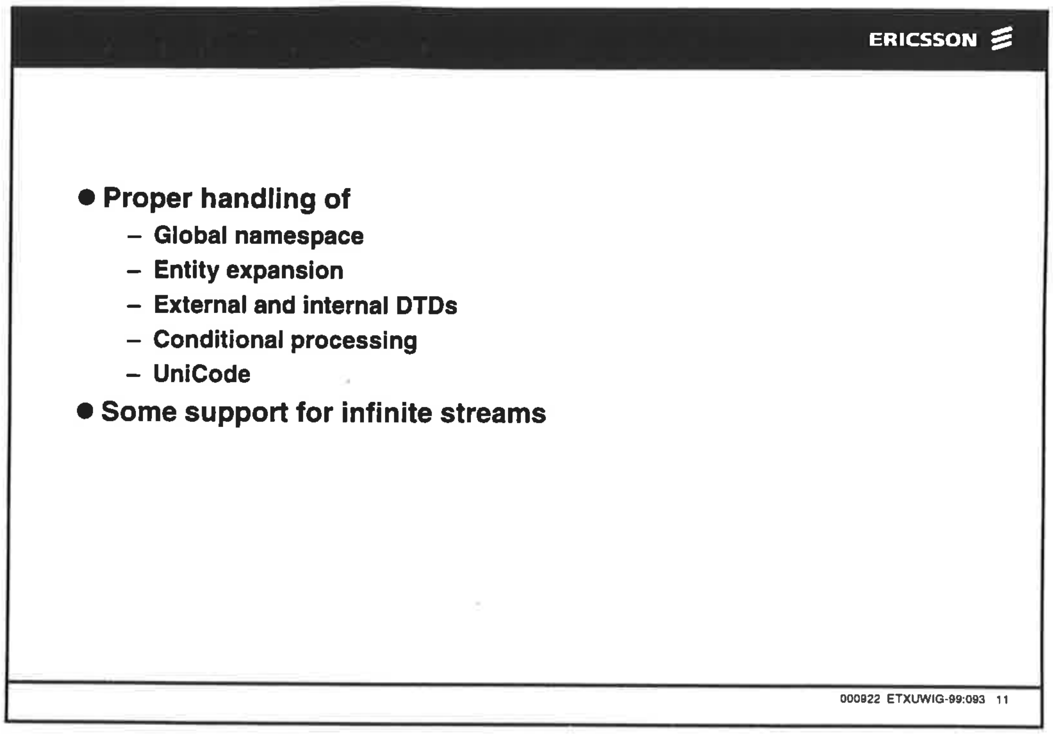
```
%% XML Element
-record(xmlElement, {
        name,
        parents = [],
        pos,
        attributes = [],
        content = [],
        language = [],
        expanded_name = [],
        nsinfo = [],  % {Prefix, Local} | []
        namespace = #xmlNamespace{}
       }).
```

ERICSSON

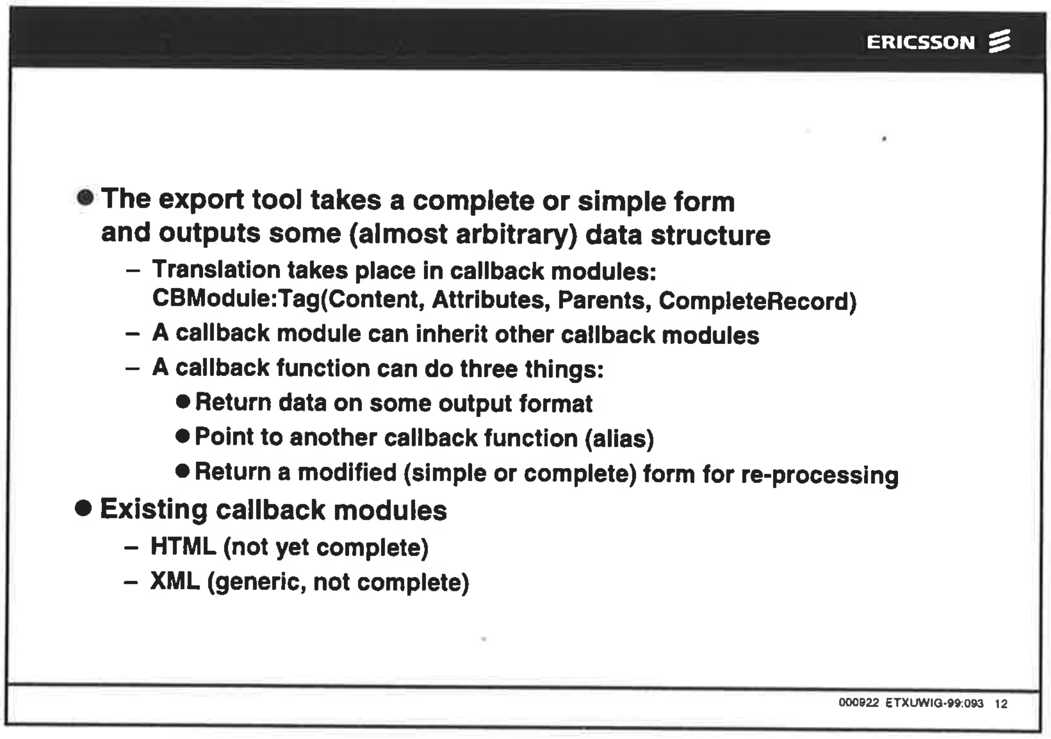
- A fast XML processor produces an Erlang representation of the XML document
  - Let's call this representation a "complete form"
- Erlang programs can use an XML-like representation
  - Let's call this a "simple form"
- An export tool can take either form and output almost anything
- Plans to support XML Stylesheets (XSL, more on that later)
- Basic support for XPATH (needed for XSL, Xlink, Xpointer, ...)

ERICSSON

- Vsn 0.6 is a single-pass scanner/parser implementing XML 1.0
- Has been tested on thousands of XML documents
  - Appears to handle lots of different documents
  - Appears to be fast and flexible
- There are two ways to process an XML document:
  - Tree-based parsing; the whole document at once
  - Event-based parsing; one element at a time
- The XMerL processor can do either
  - The behaviour is specified through higher-order functions ("funs")
  - Validation can also be carried out in funs



- Proper handling of
  - Global namespace
  - Entity expansion
  - External and internal DTDs
  - Conditional processing
  - UniCode
- Some support for infinite streams



- The export tool takes a complete or simple form and outputs some (almost arbitrary) data structure
  - Translation takes place in callback modules: CBModule:Tag(Content, Attributes, Parents, CompleteRecord)
  - A callback module can inherit other callback modules
  - A callback function can do three things:
    - Return data on some output format
    - Point to another callback function (alias)
    - Return a modified (simple or complete) form for re-processing
- Existing callback modules
  - HTML (not yet complete)
  - XML (generic, not complete)

ERICSSON

```
foo() ->
    xmerl:export_simple(simple(), xmerl_html, [{title, "Doc Title"}]).

foo2() ->
    xmerl:export_simple(simple(), xmerl_xml, [{title, "Doc Title"}]).


simple() ->
    {document, [{title, "Doc Title"}, {author, "Ulf Wiger}],
     [
      {section, [{heading, "heading1"}],
       [{'P', "This is a paragraph of text."},
        {section, [{heading, "heading2"}],
         [
          {'P', "This is another paragraph."},
          {table, [{border, 1}],
           [{heading,
             [{col, "head1"},
              {col, "head2"}]},
            {row,
             [{col, "col11"},
              {col, "col12"}]},
            {row,
             [{col, "col21"},
              {col, "col22"}]}
            ]}
          ]}
        ]}
      ]}.
```

ERICSSON

```
foo() ->
    xmerl:export_simple(sim

foo2() ->
    xmerl:export_simple(sim

simple() ->
    {document, [{title, "Do
     [
      {section, [{heading,
       [{'P', "This is a pa
        {section, [{heading
         [
          {'P', "This is an
          {table, [{border,
           [{heading,
             [{col, "head1"
              {col, "head2"
            {row,
             [{col, "col11"
              {col, "col12"}]},
            {row,
             [{col, "col21"},
              {col, "col22"}]}
            ]}
          ]}
        ]}
      ]}.
```

Sample Code:

```
%%% section/3 is to be used instead of headings.
section(Data, Attrs, [{section,_}, {section,_}, {section,_} | _], E) ->
    opt_heading(Attrs, "<h4>", "</h4>", Data);
section(Data, Attrs, [{section,_}, {section,_} | _], E) ->
    opt_heading(Attrs, "<h3>", "</h3>", Data);
section(Data, Attrs, [{section,_} | _], E) ->
    opt_heading(Attrs, "<h2>", "</h2>", Data);
section(Data, Attrs, Parents, E) ->
    opt_heading(Attrs, "<h1>", "</h1>", Data).

opt_heading(Attrs, StartTag, EndTag, Data) ->
    case find_attribute(heading, Attrs) of
        {value, Text} ->
            [StartTag, Text, EndTag, "\n" | Data];
        false ->
            Data
    end.
```

ERICSSON

```
foo() ->
    xmerl:export_simple(simple(), xmerl_html, [{title, "Doc Title"}]

foo2() ->
    xmerl:export_simple(simple(), xmerl_xml, [{title, "Doc Title"}])

simple() ->
    {document, [{t
     [
      {section, [{
       [{'P', "Thi
        {section,
         [
          {'P', "T
          {table,
           [{headi
             [{col
              {col
            {row,
             [{col
              {col
            {row,
             [{col
              {col
             ]}
           ]}
        ]}
      ]}.
```

```
<?xml version="1.0"?>
<document title="Doc Title"
          author="Ulf Wiger">
  <section heading="heading1">
    <P>
      This is a paragraph of text.
    </P>
    <section heading="heading2">
      other paragraph.
      r="1">
          </row>
        </table>
      </section>
    </section>
</document>
```

Sample Code:

```
%% The '#root#' tag is called when the entire structure has
been exported.
%% It does not appear in the structure itself.
'#root#'(Data, Attrs, [], E) ->
    ["<?xml version=\"1.0\"?>\n", Data].

'#element#'(Tag, [], Attrs, Parents, E) ->
    TagStr = mk_string(Tag),
    ["<", tag_and_attrs(TagStr, Attrs), "/>\n"];
'#element#'(Tag, Data, Attrs, Parents, E) ->
    TagStr = mk_string(Tag),
    ["<", tag_and_attrs(TagStr, Attrs), ">\n",
     Data, opt_newline(Data),
     "</", TagStr, ">\n"].
```

ERICSSON

- **Stylesheet support is clearly needed**
- **Interpreting XML stylesheets is slow and cumbersome (lots of independent, heavy XPATH queries)**
- **Possible approach:**
  - **Read the stylesheets using the XMerL processor**
  - **Translate them into an Erlang program**
  - **Optimization opportunity: convert xsl:match statements into match criteria for a single scan function**
- **Lots more work is needed here...**

- Current xmerl version, 0.6, is on Open Source
- Thanks to the beta testers:
  - Mickael Remond
  - Luc Taesch

Extending ERLANG with structured module packages

Richard Carlsson
Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala, Sweden
richardc@csd.uu.se

December 2, 1999

Abstract

This article describes a way to extend ERLANG with structured program module packages, in a simple, straightforward and useful way.

1 Introduction

When ERLANG was conceived, it inherited a lot of its flavour from languages like Strand, Prolog and Parlog, which (at least in many implementations) have a similar concept of program modules: these are program files ("compilation units"), each assigned a globally unique name (in the system), and each declaring some or all of its functions as *exported.* Non-exported functions can only be referred to from within the same module, while exported functions are also accessible from any other module in the system. In ERLANG, files containing source or object code for a module must be given the same name as the module, plus a suffix which is ".erl" for source files, and *e. g.* ".jam" for object files for the JAM abstract machine.

The name space for modules in these languages is *flat*, *i. e.*, when a particular module is referred to, this is always done by its full name, regardless of the context in which the reference is made: there is no way to express a reference to another module in relation to the current module. Furthermore, since programmers like to keep names short, names such as "lists", "math", "queue", "shell", "random", etc., quickly become used (these examples all taken from the standard library). When code from different vendors is combined in the same system, each distribution possibly consisting of several hundred modules, the likelihood of one or more name clashes becomes large. Also, because of the meta-calls often used in ERLANG, it is not always an easy task in such cases to rename the clashing modules uniquely without introducing errors, even if the source code is available. To keep the risk of clashes down, some programmers resort to giving modules abbreviated names such as "gb", "rb" "dbg", etc., which is uninformative and could be considered bad programming style, or using prefixed names such as "snmp_supervisor", "snmp_error", "snmp_generic", etc., which more or less solves the problem, but in a way that is clumsy and limited by the maximum length of file names on the host operating system.

Section 2 of this article describes the package system of Java and introduces the basic workings of a similar system for ERLANG. Section 3 describes extensions to the ERLANG language in order to make life easier for the programmer using packages. Section 4 discusses possible pitfalls, and section 5 is a summary of the system and the necessary changes.

2 A package system

To solve this situation and bring order to the present chaos, I suggest a system of packages whose basic structure is shamelessly borrowed from that of Java [2].

In Java, each compilation unit is a publicly available *class*, which is similar to a module in ERLANG, the main difference being that ERLANG modules cannot have distinct *instances* and do not support inheritance. Java source and object files are, like ERLANG modules, given the name of the public class they contain plus the extensions ".java" and ".class", respectively.

In Java, however, classes can belong to a *package*: this is a way of structuring the *files*, and is orthogonal to the class hierarchy of Java. The same concept can therefore be applied to ERLANG, even though it is not an object-oriented language.

2.1 The structure of packages in Java

If a Java class file (having the suffix ".java") contains a *package declaration* stating a package name, then that class file belongs to the named package. Furthermore, the package name also indicates to the Java implementation where the object file is located.

A Java package name consists of a sequence of names separated by period characters, such as

```
java.rmi.server
```

The full name of the class `RemoteObject` in this package, then, is

```
java.rmi.server.RemoteObject
```

When the Java implementation tries to load the object file for a class by its full name, it uses its `CLASSPATH` setting. This is simply a sequence of file system paths in the host operating system, which are to be used for the search in the order they are given. The package name is then subdivided at each period character into a sequence of one or more names. This is interpreted as a relative path in the host operating system, in a way that is system-dependent: in a Unix-like system, the relative path corresponding to the above package name would be

```
./java/rmi/server/
```

An attempt is then made to, for each path `ROOT` listed in `CLASSPATH`, load the file whose name is the concatenation of `ROOT`, the relative path for the package, the class name, and the object file suffix (".class"); in this example:

```
ROOT/java/rmi/server/RemoteObject.class
```

until such a file is found for some ROOT, or all paths in CLASSPATH have been tried. With this approach, object files are thus located in a set of directory trees, rather than in a set of flat directories.

When a Java program refers to a class that is not defined in the same file, and by its class name only, the Java compiler will assume that the class is defined in the same package as that of the current file, and will not confuse it with classes of the same name in other packages. In this way, a package declaration creates a distinct name space for classes.

For simplicity, and often useful for testing or writing simple applets, if a Java source file does not contain a package declaration, it is automatically placed in the "unnamed default package", which is a flat name space just like the current ERLANG module system.

2.2 Packages in Erlang

Today, ERLANG module names seldom or never contain period characters; one reason for this is that since the module name declaration of an ERLANG module has the form

```
-module(<A>).
```

where <A> is an ERLANG atom,[1] such a module would have to be declared as *e. g.*

```
-module('foo.bar').
```

stating the module name within single-quotes. Therefore, it can be expected that the use of the period character as separator in module names can be adopted with few (if any) existing ERLANG programs needing rewriting.

My suggestion, then, is that ERLANG modules in packages be named similarly to full class names in Java: for instance, the full name of a module `m` in a package `a.b.c` should be `a.b.c.m`, while its object file would be named "m.jam" (we assume for simplicity from now on that all object files are for the JAM abstract machine) and reside in a directory ROOT/a/b/c/ for some ROOT in the search path of the ERLANG code server. The object file for a module whose full name does not contain any period characters, such as *e. g.* `io_lib`, is thus assumed to be located in some directory ROOT/, exactly as in the ERLANG implementations of today; for this example, the file would be ROOT/io_lib.jam.

To handle this first step, only the code server needs to be modified, and only when the name of a requested module does contain period characters will the behaviour differ from that of today. It should also be apparent that this convention is compatible with existing code: packaged code could pass a full module name (generally as an atom) to old-style code, which could use the name obliviously, even for making meta-calls, without errors; to the old code, a module name does not have structure.

Existing standard modules could easily be moved into packages without disturbing old code, by simply creating "stub" replacement modules in which all exported functions make a direct jump to the function of the same name in the corresponding packaged module; such stub modules will be small, and the extra

[1]Atoms are a primitive datatype in ERLANG; they can be seen as nullary constructors, and are identified by their print names. Unless surrounded by single-quotes, their names must begin with a lowercase letter, and not contain other characters than letters, digits, or underscore ('_'). Examples of atoms are `foo`, `mad_hatter` and `'foo@bar'`.

call is a relatively small cost. In particular, the package `erlang` and all its sub-packages should be reserved for standard library functions. A library module such as `lists` could *e. g.* be renamed `erlang.list` – thus note that this would present a very good opportunity to restructure (by renaming, splitting, moving individual functions, etc.) the existing standard modules, preferably according to the suggestion for a new set of standard modules made by Jonas Barklund [1]. However, details of such a structuring of existing code into packages is outside the scope of this article.

3 Extensions to the Erlang language

So far, I have only described a structured way of storing object files in relation to module names. If the ERLANG language itself remained unchanged, this convention would force the programmer to write the full module names, always within single quotes, in all situations. This would be cumbersome and ugly, and miss one of the main points with a structured name space: to be able to make references relative to the current package.

3.1 A new form of module declarations

It is a simple task to extend the ERLANG grammar to not only accept module name declarations on the form

$$\texttt{-module(<}A_1\texttt{>).}$$

where $\texttt{<}A_1\texttt{>}$ is an atom, but also more generally on the form

$$\texttt{-module(<}A_1\texttt{>.<}A_2\texttt{>}\cdots\texttt{.<}A_n\texttt{>).}$$

for $n \geq 2$, where all $\texttt{<}A_i\texttt{>}$, $i \in [1, n]$, are atoms. Each such atom could of course be individually stated within single-quotes, and it is therefore necessary to check that the atoms do not themselves contain period characters, and that they are not the empty string (`''`). It is also recommended that some other characters, such as *e. g.* '$' be reserved for future use in module names, for example for auto-generated object files for sub-modules, if such a concept would be shown to be useful.

We introduce a little terminology:

- The *full module name* is the concatenation of the print names of the atoms $\texttt{<}A_1\texttt{>}\cdots\texttt{<}A_n\texttt{>}$ and the separating period characters. A full module name should not contain two adjacent period characters.
- The *module name* is the atom $\texttt{<}A_n\texttt{>}$.
- The *package name* is the concatenation of the atoms $\texttt{<}A_1\texttt{>}\cdots\texttt{<}A_{n-1}\texttt{>}$ and the separating period characters.

for instance, in a declaration `-module(fee.fie.foe_fum).`, the full module name is given by the atom `'fee.fie.foe_fum'`, the package name by the atom `'fee.fie'` and the module name by `'foe_fum'`. For a module whose full name contains no period characters, such as `io_lib`, the package name is the empty string, and the module name is the same as the full module name; thus, the meaning of old-style declarations does not change.

3.2 Package-relative compilation

The main advantage with the extended form of module name declarations, however, is not to relieve the programmer from writing single-quotes: it is to *signal that the source file is part of a package*, and that module references within it may therefore be interpreted as relative to the same package.

Note that it is still legal to use a declaration such as `-module('foo.bar.baz')` to give a full module name, but that this does not enable package-relative compilation. Compilation of modules with such old-style name declarations is not affected by the transformations described in this section.

3.2.1 Explicit remote calls

When a remote call on the form

```
<A>:<F>(...)
```

is encountered in a packaged module, where `<F>` is any expression and `<A>` is an *atom whose print name does not contain period characters*, then that atom is interpreted as the name of a module in the same package as the current module.

In this case, the compiler will automatically replace `<A>` with the corresponding full module name, by prepending the package name to `<A>`, separated by a period character. For example, if the call `fred:f()` occurs in a module whose package name is `foo.bar`, it will be replaced by the call `'foo.bar.fred':f()`.

This allows the programmer to *e. g.* create a module named `lists` in a package, and refer to that module directly by that name without confusion with the standard module of the same name.

To simplify calling modules in specific packages, it is easy to extend the ERLANG grammar to allow remote calls on the form

$$\texttt{<A}_1\texttt{>.<A}_2\texttt{>.}\cdots\texttt{.<A}_n\texttt{>:<F>(...)}$$

for atoms `<A`$_i$`>`, $i \in [1, n]$, $n \geq 2$, and an expression `<F>` (with the same restrictions on the atoms as in a module name declaration). Thus, a programmer is not forced to write a full module name within single quotes, but still has the possibility. For example, the calls

```
foo.bar.baz:f()
```

and

```
'foo.bar.baz':f()
```

are equivalent. Thus note that if the module specifier is a single atom, then that atom may contain period characters, but not otherwise.

3.2.2 Imported functions

Import declarations in packaged modules should be handled analogously to remote calls. If an import statement

```
-import(<A>, [...]).
```

is encountered in such a module, and `<A>` is an *atom whose print name does not contain period characters*, then that atom is interpreted as the name of a module in the same package as the current module, and the full module name is substituted by the compiler.

The ERLANG grammar should also be extended analogously to allow period-separated full module names to be written without surrounding single quotes in import declarations; *i. e.*, both

```
-import('foo.bar.baz', [...]).
```

and

```
-import(foo.bar.baz, [...]).
```

should be allowed, and be equivalent.

3.2.3 Forcing absolute module references

It is quite possible that a packaged module could need to refer to a module that is not packaged (*i. e.*, whose package name is the empty string). For this purpose, it is necessary to be able to refer to a full module name using a leading period character, as in the call

```
.lists:reverse(X)
```

This form (more generally described as `.<A`$_1$`>`$\cdots$`.<A`$_n$`>`, for atoms `<A`$_i$`>`, $i \in [1, n]$, $n \geq 1$), should therefore be included in the ERLANG grammar for remote calls and import declarations.

The same effect could be achieved by giving the module name with a prepended period character within single quotes:

```
'.lists':reverse(X)
```

however, this is *not recommended* in general, since two atoms `'m'` and `'.m'` do not compare equal, but can be interpreted as references to the same object file.

3.3 Meta-call support

Meta-calls are often used in ERLANG, either for direct function calls as in the following examples:

```
<M>:<F>(...)
```

and

```
apply(<M>, <F>, [...])
```

where `<M>` and `<F>` are any expressions evaluating to atoms, or for the evaluation of a function call by a new process, as in:

```
spawn(<M>, <F>, [...])
```

```
spawn_link(<M>, <F>, [...])
```

or

```
spawn(<N>, <M>, <F>, [...])
```

where in addition, `<N>` is an expression evaluating to an atom that is taken to represent the name of an ERLANG node.

It must then be remembered that the atoms yielded by evaluating expressions `<M>` above must be *full module names*; the built-in functions `apply` and `spawn` (and their variants) *can and should not* be modified to interpret module names relative to the current module. As a simple example, consider a function

```
f(X) -> spawn(X, start, [...]).
```

defined in a packaged module `foo.m`, and a call

```
foo.m:f('my_server')
```

in some other module. It would then be impossible for the `spawn` in function `foo.m:f/1` to know if the module name `my_server` should be interpreted relative to package `foo` or if it is a full module name whose package name is the empty string. Thus, the `apply` and `spawn` functions should remain unchanged, always interpreting the given module name as a full module name.

3.3.1 Getting the module name

Since it is generally a source of errors to be forced to write things more than once, I suggest a new predefined (but not exported) function `this_module/0` which returns the *full module name* of the module in which it occurs. For instance, to ensure correct behaviour when spawning a process to execute a function `run` in the current module, one could write

```
spawn(this_module(), run, [...])
```

Another typical example is to pass the full name of the current module to some other function, for general use including making meta-calls, as in

```
gen_server:start(this_module(), ...)
```

The call `this_module()` could be defined as synonymous to

```
module_info(module)
```

using the already predefined function `module_info/1`.

It would also be possible to use the automatically defined preprocessor macro `MODULE` for the same purpose, but the use of the preprocessor for any purpose is strongly discouraged by this author.

3.3.2 Getting package-relative names

Where, in a situation similar to those above, it is necessary to refer to a module other than the current, but in the same package, it would be convenient to not have to specify the full package name. This could be accomplished by another predefined (not exported) function `this_package/1`. For example, assume that in a module `m` in package `foo.bar`, we are to spawn a function `start/3` in a module `server` in the same package. We could then write

```
spawn(this_package(server), start, [...]
```

which would be equivalent to

```
spawn('foo.bar.server', start, [...]
```

but not dependent on the actual name of the current package.

The call `this_package(<A>)` could be defined as a substitution of the print name of `<A>` for the last component of the value of `this_module()`, if `<A>` does not contain period characters and is not the empty string `''`.

3.3.3 No other predefined functions!

There should be no other additions to the set of predefined functions or ERLANG. It might for some purposes be necessary to find the package name (the full module name without the last segment) or the module name (the last segment of the full module name) of the current module or of another module, or to perform other operations on module names, such as concatenating a package and a module name, or to map a full module name or a package name onto a relative file path, but such functions would more suitably be placed in some separate support module.

It could also be argued that special versions of `call` and `spawn` should be added to handle this kind of name expansion automatically, but this is the wrong way to go. It would add extra predefined functions without solving the general problem when full module names need to be passed between functions, and is not even a big advantage. It is no doubt easier (but more opaque) to write, say

```
spawn(start, [...])
```

instead of

```
spawn(this_module(), start, [...])
```

but *e. g.*

```
spawn_this_package(server, start, [...])
```

is not really simpler than

```
spawn(this_package(server), start, [...])
```

Furthermore it can be argued that, in particular for `spawn`, local functions should if possible never be called via a meta-call, since this requires the target function to be exported from the module, even if it is not part of the official interface. A version of `spawn` which could initiate the evaluation of a named local function, *e. g.* `fun start/3`, or an anonymous local function (a so-called "fun expression") `fun (...) -> ... end`, by a new process, would be much cleaner.[2]

3.3.4 Passing names of modules in other packages

When the full name of a module that is not part of the same package as the current module is to be passed to some function for purposes as those described above, it should be given explicitly as an atom, within single-quotes if necessary. *E. g.*, if we were to give the full name of module `a.b.c` as argument to `gen_server:start/3`, from a module `x.y.z`, we would write

```
gen_server('a.b.c', ...)
```

[2]It has been hinted that this form of `spawn` will be included in a coming release of ERLANG.

The alternative would be to extend the syntax of period-separated atom sequences to be allowed as general expressions, so we simply could write

```
gen_server(a.b.c, ...)
```

That, however, is taking the idea too far; we would then have introduced a general kind of atom-concatenating operator in the language, which could be used regardless of context, but there is no really good reason for being able to write something like

```
{'R2'.'D2'}
```

where the period character could easily be mistaken for a comma, or what is worse, be mistakenly inserted instead of an intended comma.

3.4 Imported module names

The currently existing import declarations in ERLANG allow the programmer to use functions in other modules than the current by their function names only; *e. g.*, a declaration `-import(lists, [reverse/1]).` allows a function call `lists:reverse(X)` to be written more briefly `reverse(X)`, where the imported name overrides any locally defined function of the same name.

A more general form of such declarations would allow the definition of a *local alias* for a remote function, where also the actual name used locally for making a call could be individually selected by the programmer.

When a packaged module needs to refer to several modules that are not in the same package, it would then either have to specify the full module name in each call to those modules, or use import statements so that the individual functions can be called directly by name. However, there is then the possibility that the same function name exists in two distinct modules, where both modules have long full names, or that for some module, many functions are used but we do not wish to import them all, perhaps because of the risk of clashes with locally defined names.

3.4.1 Importing packaged modules

To support easier access to particular modules in cases such as the above, I suggest a new form of import declaration, on the following form:

```
-import(<M>).
```

where `<M>` is a *full module name.*

The occurrence of such a declaration in a module would allow the use of the *module name alone*, *i. e.*, without a package name, in calls to the imported module. For example, a declaration

```
-import(foo.bar.baz).
```

in module `a.b.c` would make a call

```
baz:f(...)
```

in the same module be synonymous to

```
foo.bar.baz:f(...)
```

However, the behaviour of `this_package(baz)` would not be affected, as its name suggests, yielding `'a.b.baz'`.

In particular, note that since the imported module name is always a full module name, a declaration `-import(lists).` would make a call `lists:f(...)` in the same module be synonymous to `.lists:f(...)` thus correctly referring to the module whose package name is the empty string.

It is important that all occurrences of this kind of import statement are processed *before any function-importing declarations are expanded*, since the latter should be interpreted relative to the former, as well as to the current package. Thus, the two declarations `-import(foo.bar.baz).` and `-import(baz, [fred/1]).` (given in any order) in a module `a.b.c` together make a call `fred(X)` synonymous to `foo.bar.baz:fred(X)`, and not to `a.b.baz:fred(X)`.

3.4.2 Similarity to imported packages in Java

Java has a similar form of import declaration, on the two forms

```
package.class;
package.*;
```

where the former imports a particular class, and the latter all classes in a particular package. This allows the programmer to refer directly to any imported class without its package name; however, if two classes with the same name are imported, then neither can be used without giving its package name.

Java is a statically typed language, and needs information about the types of all external classes referenced in a program file in order to compile that file. The Java compiler therefore searches for object files for such classes, recursively compiling source files where possible in order to produce any object files that are missing. This makes importing of all classes in a package possible, because the search order is well-defined, and all referenced classes must be present.

ERLANG, however, is dynamically typed, and the compiler never actually needs to examine other source files in order to compile a particular file. It would not be in line with ERLANG programming conventions to let the sets of existing object files in two distinct packages decide from which of these packages a particular module is imported; *e. g.*, if we would import all modules from packages `foo` and `bar`, then a reference to a module `m` would be resolved to either `foo.m` or `bar.m` depending on which package actually defines a module `m`. Therefore, a full module name must be given in the `-import(<M>).` declaration described above, and not just a package name.

3.5 Why no package-relative package references?

The reader may wonder why, in a "structured" package system, there are no language constructs that allow the programmer to refer to a module in a sub-package of the current package by a relative name, instead of by its full name.

For example, in a module `a.b.m1`, it would certainly be possible to let a call

```
c.m2:f()
```

be interpreted as equivalent to

```
'a.b.c.m2':f()
```

There are however several problems with this approach. Most importantly, it adds to the complexity of the package system, making programs difficult to understand and being prone to errors. For instance, all calls to functions in packaged modules that are *not* in subpackages of the current package would have to be written as

```
.x.y.z:f()
```

with a leading period character. This is annoying and could often result in mistakes.

When there is a need to refer to modules in subpackages by a short name, the general mechanism for importing packaged modules suggested above should be sufficient, and results in perspicuous programs. The same design decision was apparently made in Java.

4 Caveats

4.1 Program transformations

If a source code transformation should want to rewrite a meta-call expression such as

```
<M>:<F>(...)
```

if it can show that `<M>` will evaluate to a specific atom `<A>`, to

```
<A>:<F>(...)
```

(doing a so-called *constant propagation*), then it is important to keep in mind that the result of `<M>` should be interpreted as a *full module name*, and never relative to the package of the module containing the code.

In this case, either the resulting program must not be compiled in a package-relative way (this can be done by expanding all package-relative references and substituting an old-style module name declaration, which states the full module name within single quotes), or the program performing the transformation must be aware of this interpretation and instead substitute the expression

$$.\texttt{<A'}_1\texttt{>}.\texttt{<A'}_2\texttt{>}\cdots\texttt{<A'}_n\texttt{>:<F>(...)}$$

where the $\texttt{<A'}_i\texttt{>}$, $i \in [1, n]$, are the period-separated (nonempty) segments of `<A>`. Note the leading period character, which explicitly indicates a full module name.

4.2 Other possible problems

There exist some modules in the ERLANG distribution which make assumptions about the present way of storing object files in the file system, notably the module `filename`. Such modules may need to be updated to handle the tree substructure of the new object file storage.

5 Summary

I have described a system of structured module packages for ERLANG, which is conceptually simple and easy to implement, and which should be backwards compatible with practically all existing code. I have suggested straightforward extensions to the ERLANG language for easier programming with packages, all of which could be removed by a preprocessor pass if so desired. I have also discussed why no further extensions should be necessary, or even motivated.

In brief, the following things need to be implemented in order to support the package system as described:

- Extend the code server to analyse module names in order to find the search path substructure for an object file.
- Add the new, package-relative compilation enabling form of module declarations to the grammar.
- Allow period-separated full module names in remote calls and in import declarations, including names with a leading period. (Note that this is not dependent on package-relative compilation, and thus should be allowed regardless of the form of module name declaration being used.)
- Extend the compiler to, for packaged modules, expand package-relative remote calls to full module names.
- Add the predefined functions `this_module/0` and `this_package/1`.
- The preprocessor `epp` must be extended to handle the new form of module name declarations, in order to correctly support the automatically defined macro `MODULE`.
- Add `-import(<M>).` declarations for full function names `<M>`, and make the compiler expand these before ordinary function imports are processed.
- The stdlib function `filename:find_src` will probably also need to be made aware of the new structure of object file search paths.

References

[1] Jonas Barklund et al., *Proposal 15: Built-in functions of Erlang.* ERLANG specification project, June 1998, `http://www.ericsson.se/cslab/~rv/Erlang-spec/index.shtml`.

[2] David Flanagan, *Java in a Nutshell.* O'Reilly & Associates, Inc., 1997.

Highlights

Erlang 5.0/OTP R7B

This document points at the most important improvements in Erlang 5.0/OTP R7B. For more detailed information, please check the release notes for the respective applications.

General

Some of the major new features in this version of Erlang/OTP are:

- The *bit syntax*, a language addition which can be used to construct binaries and to match binary patterns.
- The new CORBA services cosNotification and cosTime.
- The new very capable cross reference tool Xref.

- The new application Comet which makes it possible to call COM objects from Erlang on the Windows platform.
- A new interopability guide whose purpose is to give the reader an orientation of the different interoperability mechanisms that can be used when integrating a program written in Erlang with a program written in another programming language, from the Erlang programmer's point of view.

Note also that the Jive application has now been removed.

ERTS-5.0.1

- Several changes has been made to the garbage collector and internal memory allocation routines to reduce memory consumption and memory fragmentation.
- The bit syntax, a language addition for constructing and matching binaries. For more infomation, see **Extensions to Erlang**. Also, the handling of small binaries (up to 64 bytes) has been optimized, as well as splitting of binaries.
- Call tracing is greatly improved. It is now possible to trace local function calls (and calls to local functions) set up with the `erlang:trace_pattern` BIF. Trace compilation for local call is no longer needed.
- There is now support for operating system threads in drivers. The file driver has been modified to make use of this, which means that lengthy file operations no longer cause everything on the node to pause. Currently, only supported on Solaris and Windows. The number of threads in the thread pool can be set with the emulator system flag `-A`. The default number of threads is 0, which means that this feature is turned off. The number of threads in a node can be obtained by the call `erlang:info(thread_pool_size)`.
- The BIF `erlang:monitor(process,Proc)` has been extended to accept named processes, local and remote. See the documentation for `erlang:monitor/2` and `erlang:demonitor/1`.
- The inet driver (`inet_drv`, i.e the driver for (TCP&UDP)/IP) has been replaced, as well as a lot of supporting Erlang code in the Kernel application. The new code eliminates one data relay process per socket and thereby gives a throughput boost.
- The Erlang emulator now uses a 2 bit tag scheme internally. This change is not visible at the language level, but it means that the Erlang emulator can now address the entire 4Gb address space (OTP R6 could only address 1Gb).

IC-4.0.5

- Limited support for multiple file module definitions.
  The current version supports multiple file module definitions for all backends except the C oriented backends.
- The following new datatypes are now handled by IC:
  - long long
  - unsigned long long
  - wchar
  - wstring

Kernel-2.6.1

- On Unix, `os:cmd/1` previously used a dedicated subprocess, which meant that calls to `os:cmd/1` were serialized. This is no longer the case.
- The `gen_` family (mainly `gen_server` and `gen_fsm`) and `rpc` has been rewritten to use the new BIF `erlang:monitor/2` as much as possible. This should improve performance and remove some possibilities of hanging gen_server calls. In particular, `gen_server:multi_call/2..4` and `rpc:multi_server_call/2,3` should now never hang, at least not when all nodes are of this release.
  See also the documentation for `gen_server` and `rpc`.
- The inet driver (`inet_drv`, i.e the driver for (TCP&UDP)/IP) has been replaced, as well as a lot of supporting Erlang code in the Kernel application. The new code eliminates one data relay process per socket and thereby gives a throughput boost.

STDLIB-1.9.1

- The `win32reg` module has now been documented and made official. The module makes it possible to access and manipulate the registry on Windows platforms.
- Calls using `gen_server`, `gen_fsm` and `gen_event` to a process on another node no longer hang indefinitely if the timeout is `infinity` and the process doesn't exist. The exit reason in that case is `noproc` (for local processes, this change was made already in OTP R5).
- The new module `beam_lib` reads data from BEAM files.

Appmon-2.0.1

- There is now an Appmon User's Guide.
- The main window is now similar to that of the other tools, showing only one node at a time.

Asn1-1.2.9.3

- The ASN.1 language feature `COMPONENTS OF` is now implemented.
- The encoding and decoding of ENUMERATED and INTEGER with NamedNumbers is made more efficient and thus faster in runtime.
- The compiler now also supports AUTOMATIC TAGS for BER (previously only for PER).

Comet-1.0

Comet is a new application.

Comet, a COM client for Erlang, is a way to call any COM-service in Windows from an Erlang program. It's a combination of a gen_server and a port program (or port driver) that enables Erlang programs to call almost any COM-server.

Comet uses a gen_server in the module `erl_com`, together with a port program (or port driver), to call COM-servers. Both the late-binding interface IDispatch and early-binding virtual interfaces can be used. Erlang types are converted to COM types and parameters are returned.

COM stands for Component Object Model, (or sometimes Common Object Model), and is MicroSoft's technique for component-based programming. It allows programs on Windows systems to call other programs and libraries across language boundaries. It is a competitor to Corba, but has other functionality too.

COM is available on all 32-bit Windows systems, such as NT 4, Windows 95,Windows 98 and Windows 2000. Comet can be used on any of these.

With Comet, an Erlang application can use (almost) any COM-service on Windows from Erlang. Examples of what can be done include:

- Opening webpages with Internet Exporer (or Netscape)
- Reading and writing data from Excel Worksheet
- Reading and writing from Word
- Calling C-code-libraries in an efficient way, without the hassle of creating a port-driver.
- Executing scripts in VBScript or JavaScript

Compiler-3.0.1

- The compiler builds lists and tuples in a smarter way than previous versions did, meaning that literal lists and tuples of practically any size can now be built. The previous limit was 1024 elements in a literal list or tuple.
- Several changes has been made to the garbage collector and internal memory allocation routines to reduce memory consumption and memory fragmentation. Also, the compiler now makes sure that references to any data that will not be used again will be killed, so that the garbage collector can discard it as soon as possible.
- It is now allowed to use an expression within a pattern, if the expression uses only numeric or bitwise operators, and can be evaluated to a constant at compile-time. E.g., `case X of {1+2, T} -> T end.`
- The bit syntax, a language addition for constructing and matching binaries. For more infomation, see **Extensions to Erlang**. Also, the handling of small binaries (up to 64 bytes) has been optimised, as well as splitting of binaries.
- The compiler now supports inlining within a module, see the compiler documentation. Numerous known problems and limitations have been corrected, and the optimisation is better. For instance, tuples used in cases for grouping are no longer built.
  The earlier compiler versions v1 (R5) and v2 (R6) have been discontinued because there are no longer any reason to use them.

Inets-2.5.3

- It is now possible to run more than one HTTP server in an Erlang node listening to the same port but different addresses.

Jinterface-1.2

- A new class, AbstractConnection, has been added to deal with most of the aspects of the Erlang communication protocol, and which can be subclassed in order to provide different levels of support to the application as necessary. OtpConnection is now a subclass to AbstractConnection.
- OtpCookedConnection is a new subclass to AbstractConnection, which together with OtpNode provides an intuitive mailbox-based communication mechanism. By using this interface, applications are no longer required to manage connections explicitly, since the OtpNode now opens and manages connections to remote nodes as needed. Outgoing messages are sent through mailboxes and automatically dispatched through the correct connections to the destination node, while incoming messages are queued in the destination mailbox. This allows parts of an application to communicate with several peers simultaneously without the need to sort and dispatch incoming messages. Additionally, mailboxes can be linked with Erlang processes or with each other, in much the same manner that Erlang processes can be linked together.

Mnesia-3.9.2

- Access to non-local tables avoids disc_only_replicas if possible.
- A new configuration parameter `fallback_error_function` has been introduced to let the user handle the case when Mnesia has a fallback installed and another Mnesia goes down. The default behavior is as it always been: to kill itself to avoid inconsistencies. The user can now start Erlang with `-mnesia fallback_error_function '{UserMod,UserFunc}'`.

Orber-3.1.8

- Earlier, Orber did not use the IIOP/GIOP version specified in an external object key when invoking an intra-ORB request.
- The OMG standard now supports an Interoperable Naming Service. Initially, there where two proposals of which Orber earlier supported one of them. Now both standards are supported.
- It is now possible to start Orber as lightweight.
- It is now possible to create pseudo objects, i.e., not server objects.

cosNotification-1.0.2

cosNotification is a new application which implements the CORBA (OMG) Notification service.

cosTime-1.0.1

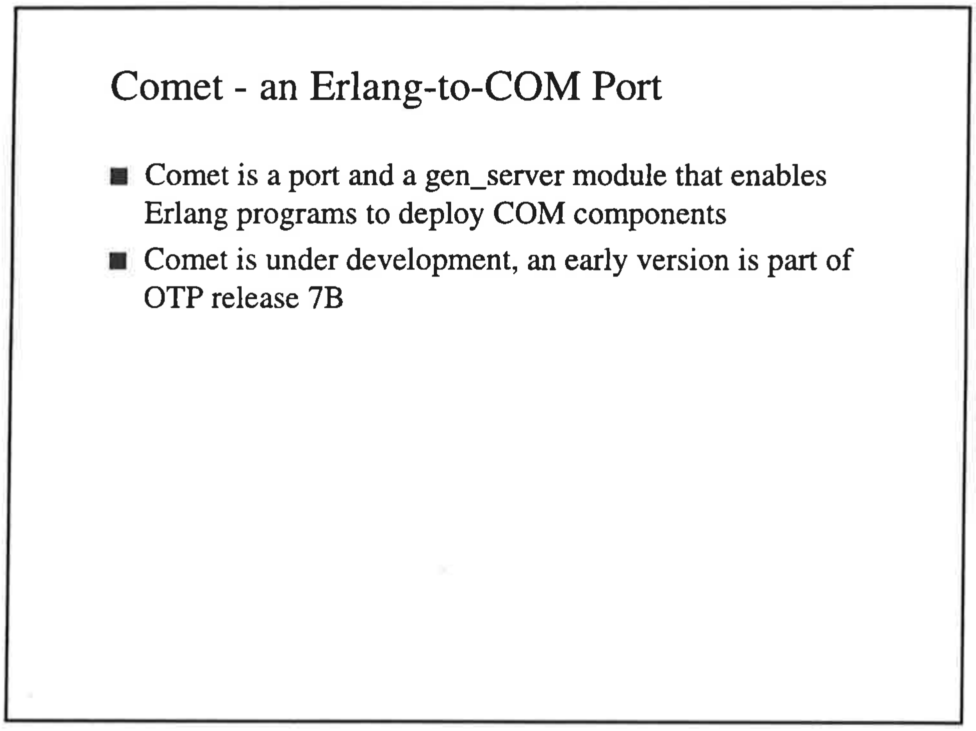
cosTime is a new application which implements the CORBA (OMG) Time service.

SNMP-3.2.1

- Debugging has been improved. It is now possible to debug all named processes (individually) of the SNMP application. See the documentation for the `snmp` module for details.
- Filter (audit trail) logs on timestamp.
- The MIB-compiler has been improved. It is now possible to include Description-field into compiled MIB.

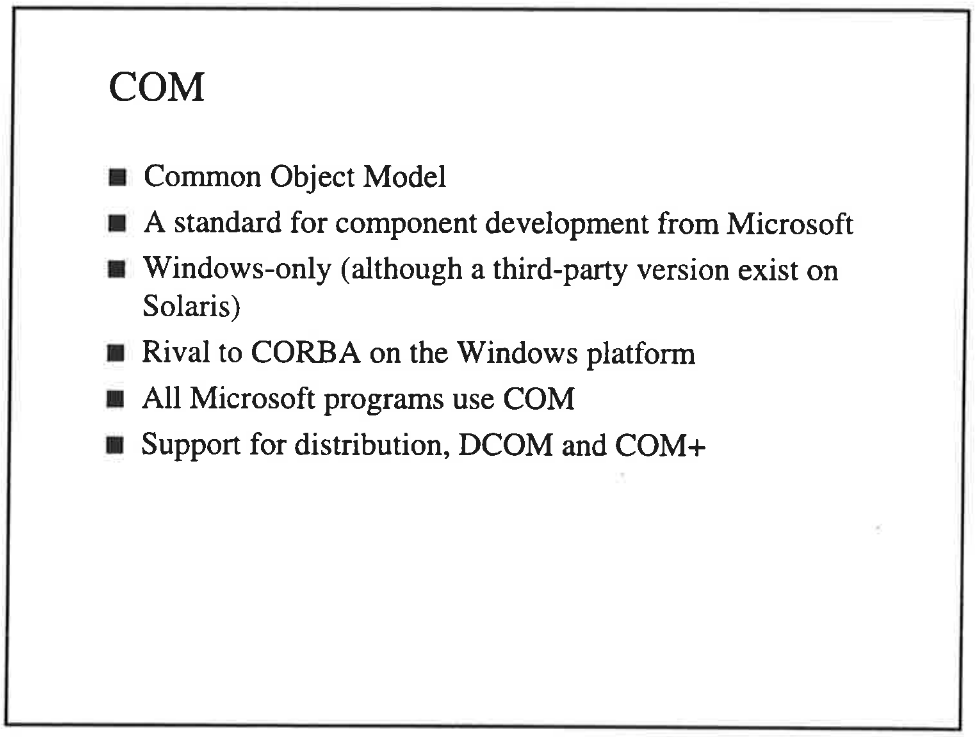
Tools-1.6

- Xref is a new very capable cross reference tool which works with beam files as input, works fast and eliminates the problem that the old tool Exref had with include files.
  Xref is intended to replace Exref in forthcoming versions of Erlang/OTP.
- Eprof has been optimised to minimise the measuring overhead. Typically, the measured program runs at about 50% of its ordinary speed.



Comet - an Erlang-to-COM Port

- Comet is a port and a gen_server module that enables Erlang programs to deploy COM components
- Comet is under development, an early version is part of OTP release 7B



COM

- Common Object Model
- A standard for component development from Microsoft
- Windows-only (although a third-party version exist on Solaris)
- Rival to CORBA on the Windows platform
- All Microsoft programs use COM
- Support for distribution, DCOM and COM+

COM Model

- Classes presents interfaces
- Interfaces are a bunch of related functions or methods
- No data are exposed, only interfaces
- Properties of objects accessible through access-functions

COM Model continued

- IDL describes classes and interfaces
- IDL compiles into a type-library that can be browsed with a tool
- Two ways to use a class
  - Dispatch - a special interface for interpreted languages
  - Virtual Interface - faster, for compiled languages
- Comet can use both (although dispatch is safer)

COM Memory Handling

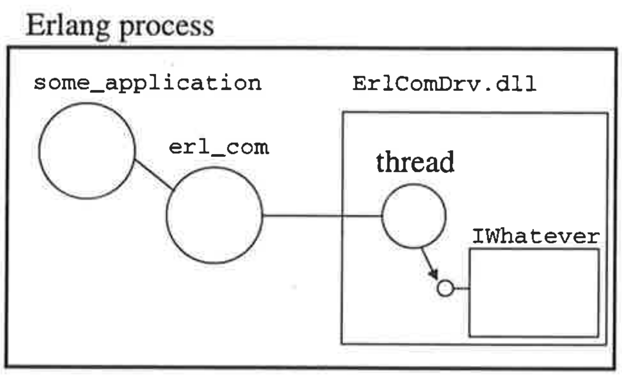
- Reference-counting
- Language support in Visual Basic, “Java” (and C#)
- Erlang programs must (currently) explicitly free interfaces

Erlang Ports

- A way to call external code from Erlang
- Implemented as a linked in driver (DLL) or a port program
- Comet offers both
  - port driver is considerably faster
  - port program is safer, if the COM server crashes, it won’t bring the emulator down
- A gen_server module interfaces to the port program or driver
- The Comet port driver and program are multithreaded

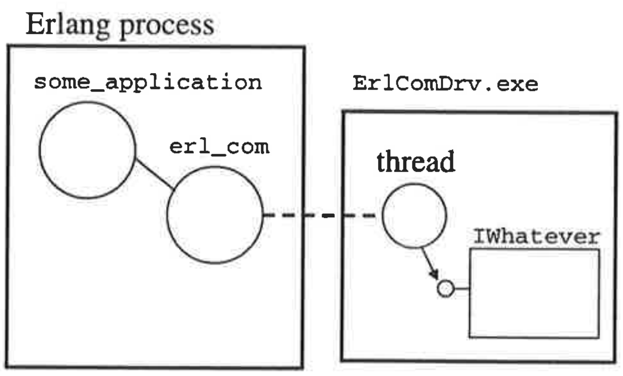
Comet as a Port Driver

- An application calling a COM Object
  - Comet as a driver



Comet as a Port Program

- An application calling a COM Object
  - Comet as a port program



Calling COM from Erlang

- All calls through the gen_server module "erl_com"
- erl_com provides methods for calling
- erl_com has functions for:
  - creating objects
  - fetching interfaces
  - releasing interfaces and objects
  - retrieving type information of objects and types
  - creating threads for calling COM objects asynchronously

A Simple Example

- An interface that implements some utilities

```
interface ISomeUtilities : IDispatch
{
    [id(1)]
    HRESULT DaysBetween([in] DATE date1, [in] DATE date2,
                        [out, retval] double* daysBetween);
    [id(2)]
    HRESULT ReplaceAll([in] BSTR inStr, [in] BSTR keyStr, [in] BSTR newStr,
                       [out, retval] BSTR* outStr);
...
};
```

- Calling it from Erlang

```
...
S= "It was a dark and stormy night...",
I= erl_com:create_dispatch(T, "(class id for SomeUtilities)"),
S2= erl_com:invoke(I, "ReplaceAll", [S, "stormy", "still"]),
D= erl_com:invoke(I, "DaysBetween", [{vt_date, {{2000, 1, 1}, {0, 0, 0}}},
                                     {vt_date, erlang:now()}])
erl_com:release(I),
...
```

Mapping COM Types to Erlang

- COM uses a small set of types
- Comet mapps Erlang types to COM types through the use of tuples
  - Basic types are converted properly: integers, floats, strings and booleans
  - Other types are prefixed in a tuple, e.g. `{vt_date {1999, 12, 12}, {}}`
- Constants in COM are enumerations
- Strings currently 8-bits in Comet
- Complex types as structures, are currently not supported

Invoke (dispatch interface)

- The invoke method in the dispatch interface is used to late-bind to interfaces
- Comet provides the methods `invoke, property_put` and `property_get`

```
...
Obj= erl_com:create_object(T, "{class id}"),
I= erl_com:query_interface(Obj, "(a dispatch interface id)"),
Value= erl_com:invoke(I, "Method", [parameters]),
erl_com:property_put(I, "Property", [parameters], Value2),
Value3= erl_com:property_get(I, "AnotherProperty"),
...
```

- Errors returned as `{com_error, Code}`
- Can have named parameters (not support in Comet yet)

Calling a Virtual Interface

- A virtual interface is an array of function pointers
  - Virtual Method Table used for C++ objects
- Called in Comet using assembler glue
  - Every parameter, including return value, must be explicitly typed
  - Address of virtual function must be specified
  - Only practical when code is generated
- Wrong parameters cause Comet to crash

```
...
[Outstr]= erl_com:com_call(I_, 36, [{vt_str, InStr}, {vt_str, out}]),
...
```

Browser Example

- The Internet Explorer browser presents COM interfaces
- Example: creating an Internet Explorer and navigating to a URL
  1 opens a Comet process and a thread in it
  2 creates an object, retrieves its default interface
  3 invokes the methods “navigate” and the “visible”

```
 {ok, Pid}= erl_com:start_process(),          ①
    T= erl_com:new_thread(Pid),
    Obj= erl_com:create_dispatch(T, "InternetExplorer.Application",
?CLSCTX_LOCAL_SERVER),                                              ②
    erl_com:invoke(Obj, "Navigate", ["www.erlang.org"]),
    erl_com:property_put(Obj, "Visible", true),          ③
    Obj.
```

Excel Example

- Excel is also accessible through COM
- Easiest way is to start with a Visual Basic-program
  - The Excel macro recorder can generate these
- Example: adding a graph

Visual Basic:

```
Charts.Add
ActiveChart.ChartType = xlPieExploded
```

Erlang:

```
Charts = erl_com:package_interface(E, erl_com:property_get(E, "Charts")),
erl_com:invoke(Charts, "Add"),
C= erl_com:package_interface(E, erl_com:property_get(E, "ActiveChart")),
erl_com:property_put(C, "ChartType", ?XlPieExploded),
```

Generating Glue Code

- Can be used for both virtual- and dispatch-interfaces
- Type libraries, compiled from COM IDL, describes COM classes and interfaces
- Comet reads information from Type Libraries
- Erlang modules are generated with glue code
- Each interface generates a module
- Each enum (set of constants) generates a module and a header-file with macros

Excel Example with Generated Code

- (Code is generated from the Excel type-library)

Visual Basic:

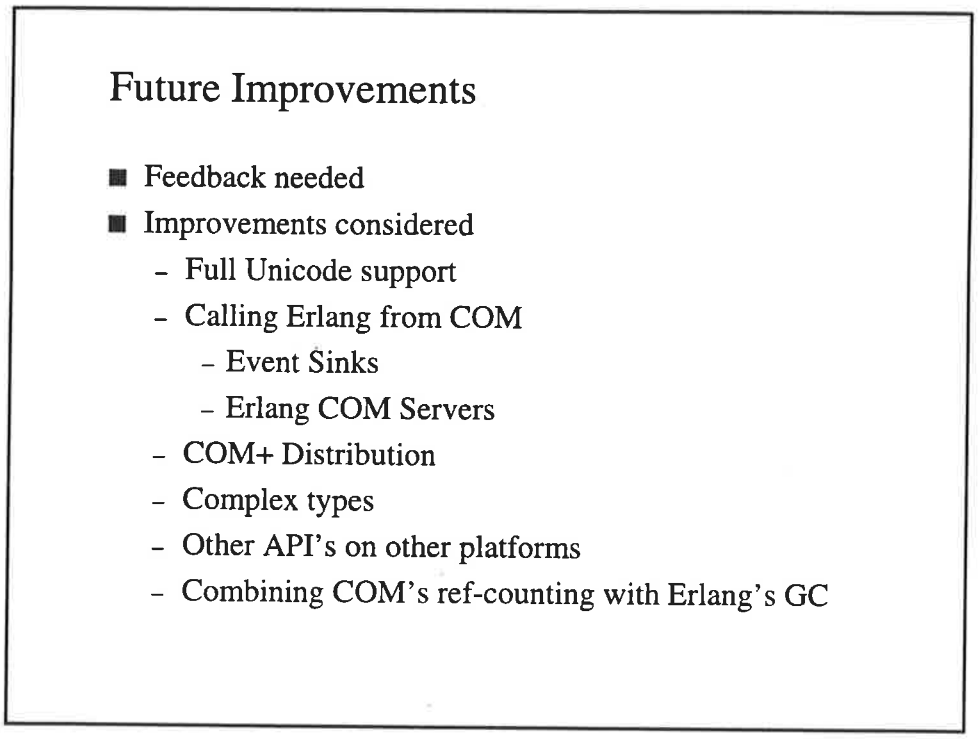
```
Charts.Add
ActiveChart.ChartType = xlPieExploded
ActiveChart.SetSourceData _
     Source:=Sheets("Sheet1").Range("B2:C4"), _
     PlotBy:=xlColumns
ActiveChart.Location Where:=xlLocationAsObject, Name:="Sheet1"
```

Erlang:

```
charts:add(xc_Application:charts(E)),
ActiveChart= xc_Application:activeChart(E),
chart:chartType(ActiveChart, ?XlPieExploded),
R= sheets:range(xc_Application.sheets(E, "Sheet1"), "B2:C4"),
chart:setSourceData(ActiveChart, R, ?xlColumns),
chart:location(ActiveChart, ?xlLocationAsObject, "Sheet1"),
```

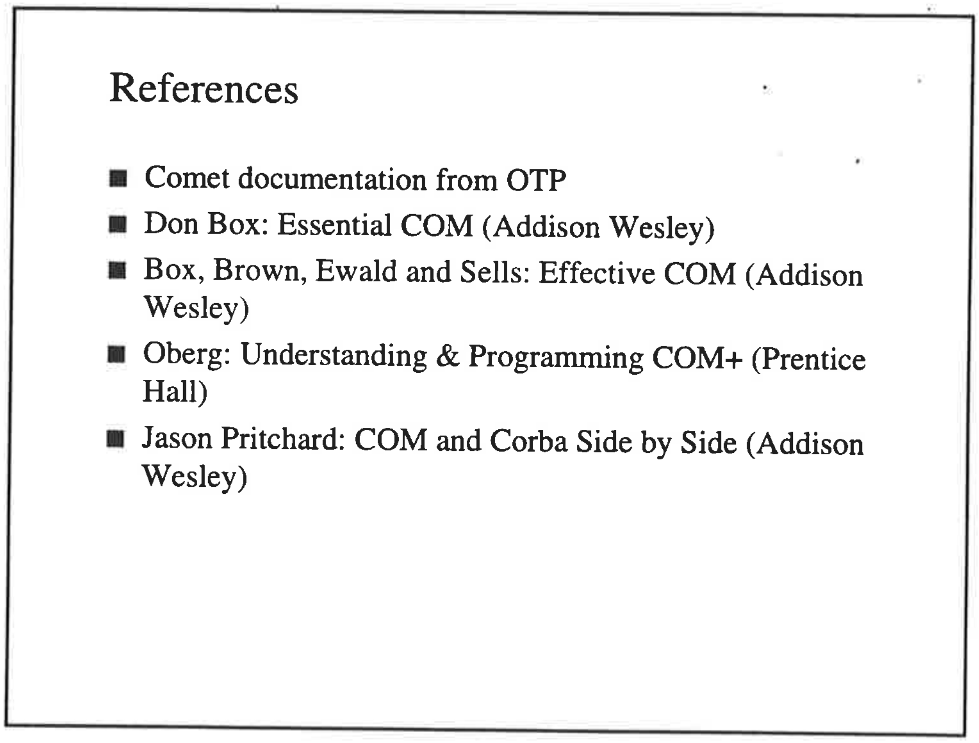
Problems

- Combining an object-oriented approach with Erlang's semi-functional
- Handling state
- Memory management
- Type conversions between Erlang and other system
- Asynchronous operations
- Performance considerations
- Robustness



Future Improvements

- Feedback needed
- Improvements considered
  - Full Unicode support
  - Calling Erlang from COM
    - Event Sinks
    - Erlang COM Servers
  - COM+ Distribution
  - Complex types
  - Other API's on other platforms
  - Combining COM's ref-counting with Erlang's GC



References

- Comet documentation from OTP
- Don Box: Essential COM (Addison Wesley)
- Box, Brown, Ewald and Sells: Effective COM (Addison Wesley)
- Oberg: Understanding & Programming COM+ (Prentice Hall)
- Jason Pritchard: COM and Corba Side by Side (Addison Wesley)

6 The bit syntax

This section describes the "bit syntax" which was added to the Erlang language in release 5.0 (R7A). Compared to the original bit syntax prototype by Claes Wikström and Tony Rogvall (presented on the Erlang User's Conference 1999), this implementation differs primarily in the following respects,

1. the character pairs '<<' and '>>' are used to delimit a binary patterns and constructor (not '<' and '>' as in the prototype),
2. the tail syntax ('|Variable') has been eliminated,
3. all size expressions must be bound,
4. a type `unit:U` has been added,
5. lists and tuples cannot be generated
6. there are no paddings whatsoever.

6.1 Introduction

In Erlang a Bin is used for constructing binaries and matching binary patterns. A Bin is written with the following syntax:

```
<<E1, E2, ... En>>
```

A Bin is a low-level sequence of bytes. The purpose of a Bin is to be able to, from a high level, **construct** a binary,

```
Bin = <<E1, E2, ... En>>
```

in which case all elements must be bound, or to **match** a binary,

```
<<E1, E2, ... En>> = Bin
```

where `Bin` is bound, and where the elements are bound or unbound, as in any match.

Each element specifies a certain **segment** of the binary. A segment is is a set of contiguous bits of the binary (not neccessarily on a byte boundary). The first element specifies the initial segment, the second element specifies the following segment etc.

The following examples illustrate how binaries are constructed or matched, and how elements and tails are specified.

6.1.1 Examples

Example 1: A binary can be constructed from a set of constants or a string literal:

```
Bin11 = <<1, 17, 42>>,
     Bin12 = <<"abc">>
```

yields binaries of size 3; `binary_to_list(Bin11)` evaluates to `[1, 17, 42]`, and `binary_to_list(Bin12)` evaluates to `[97, 98, 99]`.

Example 2: Similarly, a binary can be constructed from a set of bound variables:

```
A = 1, B = 17, C = 42,
     Bin2 = <<A, B, C:16>>
```

yields a binary of size 4, and `binary_to_list(Bin2)` evaluates to `[1, 17, 00, 42]` too. Here we used a **size expression** for the variable `C` in order to specify a 16-bits segment of `Bin2`.

Example 3: A Bin can also be used for matching: if `D`, `E`, and `F` are unbound variables, and `Bin2` is bound as in the former example,

```
<<D:16, E, F/binary>> = Bin2
```

yields `D = 273`, `E = 00`, and F binds to a binary of size 1: `binary_to_list(F) = [42]`.

Example 4: The following is a more elaborate example of matching, where `Dgram` is bound to the consecutive bytes of an IP datagram of IP protocol version 4, and where we want to extract the header and the data of the datagram:

```
-define(IP_VERSION, 4).
     -define(IP_MIN_HDR_LEN, 5).

     DgramSize = size(Dgram),
     case Dgram of
     <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
     ID:16, Flgs:3, FragOff:13,
     TTL:8, Proto:8, HdrChkSum:16,
     SrcIP:32,
     DestIP:32, RestDgram/binary>> when HLen >= 5, 4*HLen =< DgramSize -
     OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
     <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
     ...
     end.
```

Here the segment corresponding to the `Opts` variable has a **type modifier** specifying that `Opts` should bind to a binary. All other variables have the default type equal to unsigned integer.

An IP datagram header is of variable length, and its length - measured in the number of 32-bit words - is given in the segment corresponding to `HLen`, the minimum value of which is 5. It is the segment corresponding to `Opts` that is variable: if `HLen` is equal to 5, `Opts` will be an empty binary.

The tail variables `RestDgram` and `Data` bind to binaries, as all tail variables do. Both may bind to empty binaries.

If the first 4-bits segment of `Dgram` is not equal to 4, or if `HLen` is less than 5, or if the size of `Dgram` is less than `4*HLen`, the match of `Dgram` fails.

6.2 A Lexical Note

Note that "`B=<<1>>`" will be interpreted as "`B =< ;<1>>`", which is a syntax error. The correct way to write the expression is "`B = <<1>>`".

6.3 Segments

Each segment has the following general syntax:

```
Value:Size/TypeSpecifierList
```

Both the `Size` and the `TypeSpecifier` or both may be omitted; thus the following variations are allowed:

```
Value

Value:Size

Value/TypeSpecifierList
```

Default values will be used for missing specifications. The default values are described in the section "Defaults" below.

Used in binary construction, the `Value` part is any expression. Used in binary matching, the `Value` part must be a literal or variable. You can read more about the `Value` part in the sections about constructing binaries and matching binaries.

The `Size` part of the segment multiplied by the unit in the `TypeSpecifierList` (described below) gives the number of bits for the segment. In construction, `Size` is any expresssion that evaluates to an integer. In matching, `Size` must be a constant expression or a variable.

The `TypeSpecifierList` is a list of type specifiers separated by hyphens.

Type
  The type can be `integer`, `float`, or `binary`.
Signedness
  The signedness specification can be either `signed` or `unsigned`. Note that signedness only matters for matching.
Endianness
  The endianness specification can be either `big` or `little`.
Unit
  The unit size is given as `unit:IntegerLiteral`. The allowed range is 1-256. It will be multiplied by the `Size` specifier to give the effective size of the segment.

Example:

```
X:4/little-signed-integer-unit:8
```

This element has a total size of 4*8 = 32 bits, and it contains a signed integer in little-endian order.

6.4 Defaults

The default type for a segment is `integer`. The default type does **not** depend on the value, even if the value is a literal. For instance, the default type in '`<<3.14>>`' is `integer`, not `float`.

The default `Size` depends on the type. For `integer` it is 8. For `float` it is 64. For `binary` it is all of the binary. In matching, this default value is only valid for the very last element. All other binary elements in matching must have a size specification.

The default unit depends on the the type. For `integer` and `float` it is 1. For `binary` it is 8.

The default signedness is `unsigned`.

The default endianness is `big`.

6.5 Constructing binaries

This section describes the rules for constructing binaries using the bit syntax. Unlike when constructing lists or tuples, the construction of a binary can fail with a `badarg` exception.

There can be zero or more segments in a binary to be constructed. The expression '`<<>>`' constructs a zero length binary.

Each segment in a binary can consist of zero or more bits. There are no alignment rules for individual segments, but the total number of bits in all segments must be evenly divisible by 8, or in other words, the resulting binary must consist of a whole number of bytes. An `badarg` exception will be thrown if the resulting binary is not byte-aligned. Example:

```
<<X:1,Y:6>>
```

The total number of bits is 7, which is not evenly divisible by 8; thus, there will be `badarg` exception (and a compiler warning as well). The following example

```
<<X:1,Y:6,Z:1>>
```

will successfully construct a binary of 8 bits, or one byte. (Provided that all of X, Y and Z are integers.)

As noted earlier, segments have the following general syntax:

```
Value:Size/TypeSpecifierList
```

When constructing binaries, `Value` and `Size` can be any Erlang expression. However, for syntactial reasons, both `Value` and `Size` must be enclosed in parenthesis if the expression consists of anything more than a single literal or variable. The following gives a compiler syntax error:

```
<<X+1:8>>
```

This expression must be rewritten to

```
<<(X+1):8>>
```

in order to be accepted by the compiler.

6.5.1 Including literal strings

As syntatic sugar, an literal string may be written instead of a element.

```
<<"hello">>
```

which is syntactic sugar for

```
<<$h,$e,$l,$l,$o>>
```

6.6 Matching binaries

This section describes the rules for matching binaries using the bit syntax.

There can be zero or more segments in a binary binary pattern. A binary pattern can occur in every place patterns are allowed, also inside other patterns. Binary patterns cannot be nested.

The pattern '<<>>' matches a zero length binary.

Each segment in a binary can consist of zero or more bits.

A segment of type `binary` must have a size evenly divisible by 8.

This means that the following head will never match:

```
foo(<<X:7/binary,Y:1/binary>>) ->
```

As noted earlier, segments have the following general syntax:

```
Value:Size/TypeSpecifierList
```

When matching `Value` value must be either a variable or an integer or floating point literal. Expressions are not allowed.

`Size` must be an integer literal, or a previously bound variable. Note that the following is not allowed:

```
foo(N, <<X:N,T/binary>>) ->
   {X,T}.
```

The two occurences of `N` are not related. The compiler will complain that the `N` in the size field is unbound.

The correct way to write this example is like this:

```
foo(N, Bin) ->
   <<X:N,T/binary>> = Bin,
   {X,T}.
```

6.6.1 Getting the rest of the binary

To match out the rest of binary, specify a binary field without size:

```
foo(<<A:8,Rest/binary>>)
```

As always, the size of the tail must evenly divisible by 8.

6.7 Traps and pitfalls

A Tool for Verifying Software Written in Erlang

Thomas Arts[1], Gennady Chugunov[2], Mads Dam[2], Lars-åke Fredlund[2], Dilian Gurov[2], and Thomas Noll[3]

[1] Ericsson Computer Science Laboratory, Ericsson Utvecklings AB,
`thomas@cslab.ericsson.se`
[2] Swedish Institute of Computer Science,
`gena@sics.se,mfd@sics.se,fred@sics.se,dilian@sics.se`
[3] Department of Teleinformatics, Royal Institute of Technology, Sweden,
`noll@it.kth.se`

Abstract. The present paper presents an overview of the main results of the ASTEC project Verification of Erlang Programs, focusing in particular on the Erlang verification tool. This is a theorem-proving tool which assists in obtaining proofs that Erlang applications satisfy their correctness requirements formulated in a specification logic. We give a summary of the verification framework as supported by the tool, discuss reasoning principles essential for successful verification such as inductive and compositional reasoning, and an efficient treatment of side-effect-free code. The experiences of applying the verification tool in an industrial case study are summarised, and an approach for supporting verification in the presence of program libraries is outlined.
The verification tool is essentially a classical proof assistant, or theorem-proving tool, requiring users to intervene in the proof process at crucial steps such as stating program invariants. However, the tool offers considerable support for automatic proof discovery through higher-level tactics tailored to the particular task of verification of Erlang programs. In addition, a graphical interface permits easy navigation through proof tableaux, proof reuse, and meaningful feedback about the current proof state, to assist users in taking informed proof decisions.

1 Introduction

Software written for telecommunication applications has to meet high quality demands. Such software is often highly concurrent, and *testing* is not by itself sufficient to ensure reliability requirements are met. At the Ericsson corporation the functional language Erlang [1] has been developed for programming telecommunication applications. Today many commercially available products offered by Ericsson are partly programmed in Erlang. The software of such products is typically organised into many, relatively small source modules, and at runtime consists of a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing.

Our approach to verification of software programmed in Erlang is code verification: we prove that Erlang code satisfies a set of properties formalized in a

2 Foundation

In this section we briefly highlight the foundations of the tool: the Erlang programming language, a specification logic for capturing correctness requirements of Erlang programs, and a proof system for proof derivation.

2.1 The Erlang Programming Language

Erlang/OTP is a *programming platform* providing the necessary functionality for programming of open distributed (telecom) systems: a language Erlang with support for concurrency, and middleware OTP (Open Telecom Platform) providing ready-to-use components (libraries) and services such as e.g. a distributed data base manager, support for "hot code replacement", and design guidelines for using the components.

In the following we consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as integers, lists, tuples, or process identifiers (pid's), using asynchronous, call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as modules, distribution of processes (onto nodes), and support for interfacing with non–Erlang code written in, e.g., C or Java.

Besides Erlang *expressions* e we operate with the syntactical categories of *matches* m, *patterns* p, *guards* g, and *values* v. The abstract syntax of Core Erlang expressions is summarised as follows:

$$\begin{aligned}
e ::= {} & V \\
\mid {} & op(e_1, \ldots, e_n) \\
\mid {} & \texttt{begin}\ e_1, \ldots, e_n\ \texttt{end} \\
\mid {} & e(e_1, \ldots, e_n) \\
\mid {} & \texttt{case}\ e\ \texttt{of}\ m\ \texttt{end} \\
\mid {} & \texttt{catch}\ e \\
\mid {} & \texttt{receive}\ m\ \texttt{end} \\
\mid {} & e_1 ! e_2
\end{aligned}$$

$$\begin{aligned}
v &::= op(v_1, \ldots, v_n) \\
p &::= op(p_1, \ldots, p_n) \mid V \\
m &::= p_1\ [\texttt{when}\ g_1]\ \texttt{->}\ e_{11}, \ldots, e_{1k}; \\
&\quad \vdots \\
&\quad\ \ p_n\ [\texttt{when}\ g_n]\ \texttt{->}\ e_{n1}, \ldots, e_{nk} \\
g &::= e_1, \ldots, e_n
\end{aligned}$$

Here V ranges over Erlang variables, and op ranges over a set of primitive constants and operations including the integers ranged over by i, tupling $\{e_1, e_2\}$, the empty list $[\,]$, list prefix $[e_1|e_2]$, pid constants ranged over by pid, and atom constants ranged over by a and f.

Note above that Erlang variables are always upper-case (`F` and `Pid`) while atoms are lower-case (`server`, `exec` and `spawn`). The `server` function is continuously prepared to receive tuples containing the name of a function `F` and a process identifier `Pid`. It then spawns off a new process evaluating the `exec` function, which simply invokes the received function `F` and sends any result to the process addressed by the process identifier `Pid`.

Since Erlang is an untyped language a possible outcome of sending a wrongly typed message to the server process is that the newly spawned process will terminate due to a runtime error.

A Semantics for Erlang The formal semantics of Erlang is given as an operational semantics in the form of a set of rules for deriving labelled transitions between structured states [26]. Our semantics for Erlang is a small-step operational one [11], which is motivated by the free intermixing of functional and side-effect concerns found in Erlang.

A natural approach to handling the different conceptual layers of entities in the language (i.e., functional expressions and concurrent processes) supporting modular (i.e., compositional) reasoning, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. Thus first the Erlang expressions are provided with a semantics that does not require any notion of processes. The actions here are a computation step τ, an output $pid!v$, $read(q,v)$ which represents the reading of a value v from the queue of the process in whose context the expression executes, and $f(v_1,\dots,v_n)$ which represents the calling of a builtin function (like `spawn` for process spawning) with side effects on the process level state.

Then the transition behaviour of Erlang systems (concurrent processes executing expressions in the context of a unique process identifier and a mailbox of incoming messages) is captured through a set of transition rules separated into two cases: (i) a single process constraining the behaviours of an Erlang expression and (ii) the (parallel) composition of two Erlang systems into a single one expressed by the parallel composition construct "$\|$". The system actions are computation steps τ, output $pid!v$ and input $pid?v$.

Example 2 (Erlang Semantics).

We will illustrate the operational semantics by considering the case of a built-in function with side-effects, like for instance `spawn`. On the level of Erlang expression the evaluation of a built-in function like `spawn` is covered by the transition rule:

$$\frac{\mathit{isProcFun}(\mathrm{f})}{f(v_1,\dots,v_n) \xrightarrow{f(v_1,\dots,v_n)} v}$$

where $\mathit{isProcFun}(f)$ tests whether the function f names a built-in function, and v represents any Erlang value (akin to an input parameter). As seen in the above rule the operational semantics is infinitely branching, due to occurrence of the number of negations, to ensure that the corresponding fixed points exists (due to monotonicity).

The syntax of the logic can then be summarised:

$$\begin{array}{rl}\phi ::= & t_1 = t_2 \mid \texttt{true} \mid \texttt{false} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ \mid & \exists T : T_t.\phi \mid \forall T : T_t.\phi \mid \lambda T : T_t.\phi \mid \phi\ t \text{ (application)} \\ \mid & \langle\alpha\rangle\,\phi \mid [\alpha]\,\phi \\ \mid & \nu X.\phi \mid \mu X.\phi \mid X \text{ (fixed point reference)}\end{array}$$

This powerful logic is capable of expressing a wide range of important system properties, ranging from type–like assertions to complex reactivity properties of the interaction behaviour of a telecommunication system. As a syntactic convention fixed point formulas can be named, e.g., $name \Leftarrow \phi$ abbreviates the least fixed point $\mu X.\phi[X/name]$ and $name \Rightarrow \phi$ abbreviates the greatest fixed point $\nu X.\phi[X/name]$ (X is assumed fresh in ϕ).

The semantics of a formula in the logic is defined in the usual (denotational) fashion, as the set of Erlang systems that satisfy the formula (see [7] for details).

Example 3. The type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined recursively as a least fixed–point, assuming the term constructors 0 and +1:

$$nat \Leftarrow \lambda N.\,(N = 0 \;\vee\; \exists V.\,(nat\ V \wedge N = V + 1))$$

2.3 The Proof System

Reasoning about open distributed systems written in Erlang requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen–style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of assertions. A judgment is deemed *valid* if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $\psi\ x \vdash \phi\ P(x)$ states that object P has property ϕ provided the parameter x of P satisfies property ψ.

The proof rules of the proof system are mostly standard from accounts of first-order logic in Gentzen–style proof systems, with rules like $\forall_R$ and $\forall_L$ shown below:

$$(\forall_L)\ \frac{\Gamma, \phi\{v/V\} \vdash \Delta}{\Gamma, \forall V : T_t.\phi \vdash \Delta}\, v \in T_t$$

$$(\forall_R)\ \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall V : T_t.\phi, \Delta}\, V \textit{ fresh}$$

defined. The introduction of subtyping in the underlying theory, can, as usual, introduce typing proof obligations during parsing of terms and formulas.

For types considered to be freely-generated (intuitively the types where "semantic equality" coincides with the syntactic notion of equality) such as e.g. the natural numbers, recursive predicates can be automatically generated that permit structural induction style arguments about elements of the type.

Sequents $\Gamma \vdash \Delta$ are pairs of ordered sequences of formulas (assertions) $\Gamma = \phi_1, \ldots, \phi_n$ and $\Delta = \psi_1, \ldots, \psi_k$. These formulas may contain free variables, which are of two kinds: *parameters* which are generated by rules such as $\forall_R$ above, and *meta-variables*, the result of postponing the choice of a witness in a proof rule such as $\forall_L$. To ensure that assignments to meta-variables are sound a simple scheme based on associating indices to variables, from [27], is used. Bound variables are represented using de Bruijn indices, to permit checking equality of formulas quickly up to α-conversion, which is important for obtaining efficient implementations of the discharge rule.

From a user's point of view, proving a property of an Erlang program using the verification tool involves "backward" (i.e., goal-directed) construction of a proof graph (tableau). A proof graph is, here, an acyclic directed graph of proof nodes containing sequents and rooted in an initial proof node. Each proof node in the graph is either a leaf node, meaning that it either represents an open goal, or that the sequent was solved by the application of an axiom proof rule without premises, or it is a parent node that has been reduced by applying the proof rule and such that its children nodes correspond to the premises of the rule. An application of the discharge rule is represented in the proof graph by a directed arc from the discharged node to its companion node. Arcs in the proof tree are labelled by the proof rule that caused the arc to appear, to permit flexible display of proofs and portable proofs (to allow for, as an example, proof carrying code schemes [21]).

Open proof goals may also be (copy)discharged (or subsumed in more standard terminology) when instances of the goal can be found elsewhere in the proof graph. In practice application of the copydischarge rule is absolutely essential to, for example, combat the state explosion caused by the interleaving semantics of Erlang. However, there are two restrictions to its use. First, no open proof goal can be copydischarged against an ancestor proof node. Second an acyclicity condition is enforced to prevent cyclic copydischarges, for example such that node N is copydischarged against node M which is in turn copydischarged against node N. A finished proof graph is a proof graph that contains no open goals.

The application of a proof rule can be canceled (undone), resulting potentially in non-local cancellation effects on the proof tree when e.g. the companion node of a copydischarge node is canceled, naturally also causing the copydischarge to fail. Another such problematic case is when a meta-variable is assigned, and canceled in a proof branch, but where the meta-variable is also present in another proof branch. In such a situation both the assignment, and cancellation, may also affect the proof steps in the second proof branch. To permit a sound cancellation scheme in spite of these difficulties a global ordering of proof sequents with tactic t1, otherwise tactic t2 is chosen. Finally t_fix can be used to write recursive tactics, the first argument being an arbitrary initialization value, the second a function of an arbitrary parameter and a "continuation", and returning a tactic.

As an example of the usage of such tactical combinators a derived tactic forall_all_r is given below, that tries to apply the rule $\forall_R$ to all right-hand assertions. We assume that the number of assertions to the right is returned by the function length_assertions_right:

```
val forall_all_r =
  t_fix 1
  (fn position => fn continuation =>
    t_bool
      (fn seq =>
       position <= length_assertions_right seq)
      (t_orelse
        (t_compose
          (forall_r position)
          [continuation (position+1)])
        (continuation (position+1)))
      t_skip) : tactic;
```

3.3 User Interfaces and Commands

The standard user interface to the proof assistant is the conventional command line interface of Standard ML (of New Jersey) to which a number of commands to interact with the proof assistant has been added. Conceptually the user interface defines notions such as "which is the current proof graph" and "which is the current proof node". The commands of the proof assistant operate on proof graphs, possibly with side effects. For instance, there are commands to start a new proof, to define a lemma, to navigate proof graphs (to modify the notion of the current proof node), to navigate the hierarchy of proof graphs, to grow (or complete) a proof graph by applying a tactic to its current sequent resulting possibly in new proof branches, and to cancel a previous proof step. As another example the discharge and copydischarge proof rules are implemented as commands rather than tactics, since they cause global effects to the graph structure.

A clear alternative to combining tactics using tacticals is to directly use the Standard ML programming language facilities to define functions executing proof commands. This works reasonably well, but has the disadvantage that all intermediate proof nodes are kept. In contrast, using tactical combinators, no intermediate proof nodes are ever kept.

A second, graphical, user interface is under development. This user interface consists of two parts: the first is programmed in Java and provides additional user assistance through the implementation of modern theorem prover features [4] such as "proof-by-pointing" (to suggest, based on the proof context, the next proof rule to apply), a more structured database of lemmata, proof recording selecting one progressing ordinal variable per discharge node, a total ordering $<_l$ on these ordinal variables can be found such that if $k <_l k'$ then at the discharge node where k' progresses, k is preserved. This corresponds to a lexicographical ordering on the vector of related ordinal variables.

3.5 Embedding of Erlang

The Erlang program constructs are encoded as terms of the many-sorted first-order logic. The current tool generation contains a definition of the transition relations (on the expression and system levels) as recursive predicates in the underlying logic. In addition, and to improve the speed with which new transitions are computed, a set of low-level rules was implemented directly, for inferring transitions $e \xrightarrow{\alpha} e'$ that trigger on the syntactic shape of the Erlang construct e. An example of such a rule is shown below, for the case of input under parallel composition to the left in a sequent (S is assumed fresh in Γ, Δ):

$$\| ?_L \frac{\begin{array}{c} \Gamma, s_1 \xrightarrow{pid?v} S, s' = S \parallel s_2 \vdash \Delta \\ \Gamma, s_2 \xrightarrow{pid?v} S, s' = s_1 \parallel S \vdash \Delta \end{array}}{\Gamma, s_1 \parallel s_2 \xrightarrow{pid?v} s' \vdash \Delta}$$

In general the handling of operational semantics in EVT is split into two parts: one language dependent part where tactics corresponding to the operational semantics of the language in question are introduced and a second, largely language-independent part, for deriving valid transitions from such sets of operational semantics tactics.

3.6 Tactics for Deriving Transitions

The present tool implements four high-level tactics, **diasem_l**, **diasem_r**, **boxsem_l** and **boxsem_r**, for reasoning about combinations of program terms and modalities. For example, the **diasem_r** and **boxsem_r** tactics tries to achieve the result of the rules $\langle\,\rangle_r$ and $[\,]_r$ below, with intuitive definitions (assuming $t \xrightarrow{\alpha} t_1, \ldots, t \xrightarrow{\alpha} t_n$ and no other such continuation state t_x exists):

$$\langle\,\rangle_r \frac{\Gamma \vdash t_1 : \phi, \ldots, t_n : \phi, \Delta}{\Gamma \vdash s : \langle\alpha\rangle\phi, \Delta}$$

$$[\,]_r \frac{\Gamma \vdash t_1 : \phi, \Delta \quad \ldots \quad \Gamma \vdash t_n : \phi, \Delta}{\Gamma \vdash s : [\alpha]\phi, \Delta}$$

Above, the syntax $t : \phi$ represents the statement t has the property ϕ, a synonyme of $\phi\ t$.

The means of realising tactics achieving the effect of these rules is by repeatedly applying language specific operational semantic tactics such as, e.g., $\| ?_L$ shown before, together with simple general simplification steps like splitting conjunctions and reasoning about term equalities. In addition language dependent tactics for handling data are applied.

levels simultaneously, in manners which, when properly formalized, may be exceedingly complicated. Our proof theoretic approach, using loop detection, or discharge, allows very substantial parts of this formalisation to be almost completely hidden from the user. In effect the discharge mechanism, as described in the previous section, attempts to cast the proof as so far constructed as a proof by simultaneous induction, by seeking an ordering that makes the dependency relation between induction and co–induction variables a well–founded one. Maintaining the constraints on this dependency ordering is done by the proof editor. Thus there is no need for users to specify the sequence, nesting, or mutual dependencies of simultaneous inductive arguments, or even to state that induction is being used. All this is managed by the tool. However, the user will need to have a basic understanding of the general principles of simultaneous induction for the operation of the discharge rule to be understandable. And, most importantly, the tool has *no* builtin support for finding inductive assertions. Such support can be programmed (as tactics), or must alternatively be provided explicitly.

4.2 Compositional Reasoning

The essence of compositional verification is the reduction of an argument about the behaviour of a compound system to arguments about the behaviour of its components. A system P containing component Q can be represented through term substitution as $P[Q/X]$, where X is a variable ranging over entities of the type of Q. We can relativise an assertion $P[Q/X] : \phi$ about the compound object $P[Q/X]$ to a certain property ψ of its component Q by considering Q as a parameter for which property ψ is assumed, provided we can show that Q indeed satisfies the assumed property ψ. Technically, we achieve this through a *term–cut* proof rule of the shape:

$$(\mathsf{TermCut})\ \frac{\Gamma \vdash Q : \psi, \Delta \qquad \Gamma, X : \psi \vdash P : \phi, \Delta}{\Gamma \vdash P[Q/X] : \phi, \Delta}$$

Very often, constructors occurring within the scope of recursion give rise to unbounded state spaces. An example is a process spawning statement, giving rise to the formation of an unbounded process set. In such cases we have to combine (co–)inductive with compositional reasoning. For example, after a new process P has been spawned off by a recursive process Q one can apply the above term–cut rule to relativise the proof on the specification of Q rather than on its implementation, thus avoiding new processes from being generated by Q explicitly in the process term, and thus allowing the (co–)induction through loop detection and discharge to go through.

The above term–cut rule provides the basic low–level facility for compositional reasoning. Applying the rule requires a suitable choice of the cut–property ψ. It should capture the essence of the behaviour of Q needed for completing the proof. In some special cases we can give a concrete structure to the formation of ψ, as illustrated in the next subsection, and give (and support through tactics) more high–level decomposition principles exploiting this additional structure.

For example, a list sorting function **sort** can be specified as a satisfaction pair $\texttt{sort}(L) : \mathit{prepost}(L : \mathit{list}, \theta_{sort}\, L)$ where type *list* is defined by:

$$\begin{aligned}
&L : \mathit{list} \Leftarrow \\
&\quad L = [] \\
&\quad \vee \exists P, R : \texttt{ErlangValue}. \\
&\qquad L = [P|R] \wedge R : \mathit{list}
\end{aligned}$$

and type $\theta_{sort}\, L$ is defined by:

$$\begin{aligned}
&V : \theta_{sort}\, L = \\
&\quad \mathit{isSorted}\, V \\
&\quad \wedge\, \mathit{isPermutation}\ V\ L
\end{aligned}$$

A more detailed account of how to deal with side-effect-free Erlang code can be found in [15].

5 Case Study: A Distributed Database Lookup Manager

Erlang is used extensively for writing robust distributed telecommunication applications. Central in many of these applications is a distributed database, Mnesia [28], also written in Erlang. The Mnesia system is crucial to the robustness of many Erlang based products developed at Ericsson. It is, for instance, responsible for error recovery, the prompt and safe handling of which is essential in telecommunication applications. These features make the Mnesia system a rewarding object of study when trying out new verification techniques.

The case study at hand concerns only a small part of the Mnesia system, a protocol for the evaluation of a query which is distributed over several computers in a network. The starting point for this case study was the Erlang code implementing the distributed database. We extracted, from the real implementation, the code for the distributed query evaluation protocol and added some code to provide a very simple simulated interface to parts of the system that were irrelevant for the problem at hand. The result was an Erlang program that could be seen as a very precise, and in some sense formal, description of the underlying algorithm. Isolation of the code responsible for the lookup mechanism and analysing the intended behaviour of the code resulted, as a side effect, in a clear and patentable picture of the underlying protocol [22].

As input the protocol receives a database query divided into subqueries. These subqueries are distributed over the network in the form of processes on those computers where the specific data for a subquery is stored. Whenever a subquery process receives a message, it extracts the corresponding data from the database tables and sends it along the network.

One process is responsible for initialising the lookup process ring, and for collecting the resulting data. To avoid excessive delays and storage consumption, query answers are collected in segments, managed by the lookup manager.

subqueries. The first one manages the original input list. The second one receives a list of client names and returns a list of pairs of the form {`Name, Number`}, enumerating every phone number that is owned by the respective client. And finally, for each of the numbers, the third subquery collects the target number and the fee of every phone call that was originating from it, returning a list of quadruples of the form {`Name, Number, Target, Fee`}.

After spawning the ring, the initial process P_1 is ready to receive a message of the form {`user_request, UserPid, NrSolutions`} where the triple represents an atom `user_request` to identify the message type, the pid of the requesting process and the maximum number of solutions that the latter process wants to receive.Whenever this message arrives, a message is sent to the consecutive process in the ring P_n, which is the first process able to perform a subquery lookup. The process P_1 subsequently calls the function `counting`, which collects all answers that the subqueries of the ring produce. The idea is that for all solutions that a process in the ring receives, it computes all new solutions using its subquery lookup function. This might result in an increase or decrease of the number of solutions. These new solutions are passed to the next process and so on, until P_1 receives the answers and can present them to the user.

However, in order not to overload the network, the processes in the ring are not sending all the answers they find, but just a fixed number given by *PacketSize*, which is dynamically determined by P_1, and depends on the number of requested solutions and the network load. Thus, the number *PacketSize* is sent along in the message from P_1 to the next process P_n in the ring. The latter process computes all answers it can find according to its subquery and sends at most *PacketSize* of these answers to the next process, whereas the remaining answers are kept in the store. All consecutive processes in the ring perform the same actions and eventually P_1 receives at most *PacketSize* answers. The process P_1 may now add these answers to its store and as long as the store contains less than the demanded number of answers (`NrSolutions`) a message will be sent to the process P_n requesting to produce new answers.

Except for the initial processes, all other processes in the ring, i.e., $P_2, \ldots, P_n$, are evaluating the function `process_in_ring` having three arguments, the process identifier (pid) of the next process in the ring, the function representing the subquery, and the empty list representing a local store for the process. These processes wait for a message containing at most *PacketSize* answers of the previous process and the value *PacketSize* itself. The number of stored answers is compared to the number *PacketSize* of demanded answers and if enough answers are already in the store, these are sent along to the next process and new answers are computed. In case not enough answers are stored, first all new answers are computed, whereafter at most *PacketSize* answers are sent to the next process and all other answers are stored for the next round.

The property that we want to verify is informally described as:

Is the retrieval of the information terminating?

In other words, given an arbitrary query and an arbitrary positive integer, whenever we build a ring corresponding to this query and send a message of the form and then, by continuing state exploration, to a subgoal of the shape

$$\begin{array}{l} \text{some assumptions} \vdash \\ \quad \mathit{proc}(\mathtt{mk\_ring}(\cdots),\cdots) \parallel \\ \quad \mathit{proc}(\mathtt{process\_in\_ring}(\cdots),\cdots) : \mathit{rootspec}(\cdots) \end{array} \tag{4}$$

The idea is to prove two lemmas, one stating the correctness of `process_in_ring`,

$$\begin{array}{l} \text{some assumptions} \vdash \\ \quad \mathit{proc}(\mathtt{process\_in\_ring}(\cdots)) : \mathit{proc\_wait\_for\_input}(\cdots) \end{array} \tag{5}$$

and one concerning the composability of *rootspec* with *proc_wait_for_input*,

$$\begin{array}{l} C1 : \mathit{rootspec}(\cdots), \\ C2 : \mathit{proc\_wait\_for\_input}(\cdots) \vdash \\ \quad C1 \parallel C2 : \mathit{rootspec}(\cdots) \end{array} \tag{6}$$

Subgoal (6) states a compositional property of root and ring processes: putting together a (possibly aggregate) process (P_1) acting as a root with a (possibly aggregate) process acting as a ring element results in an aggregate process which again acts as a root. Obviously the correctness of this statement is crucially dependent on input and outputs being properly connected, which are matters we will not be concerned with here.

By themselves, (5) and (6) are not sufficient to conclude (4). However, using (5) and (6) it is possible to reduce to a goal which is actually an instance of the goal (3), and the remarkable fact is that, in principle, an inductive argument can be set up such that at this point the proof can be completed (c.f. [7]).

Properties of the Separate Processes We are thus left with two main subgoals, one of the shape (5), and one of the shape (6). We do not comment further on (6) other than observe that the ring process property *proc_wait_for_input* we are looking for must be strong enough to permit (6) to be proved. Instead we turn to *proc_wait_for_input*.

We start by observing the role of a special token that is initially sent by the first process (P_1) in the ring and implies termination as soon as it is also received by this process, i.e., when the token has gone through the entire ring. This special token (`{[],PacketSize}`), which we call the *end_token* for convenience, is repeatedly sent by P_1 to P_n after initially sending `{[[]],PacketSize}` once. In case the number of demanded solutions is larger than the number of solutions present in the database, the process P_1 can only respond to the user when this *end_token* is received from the process P_2.

The first process in the ring P_1 plays a special role and the abstract states we distinguish for this process are

1. the process is waiting for a user_request,

A compositional reasoning framework will turn out to be useful especially in connection with standard program libraries. Since these are developed to be used frequently, it is worth spending much effort in analysing and describing their properties since many applications will potentially benefit from this knowledge.

In our approach we try to capture the behaviour of library functions by specifying their operational semantics on an abstract level, regardless of their concrete implementation. To this aim we provide rules in the style of Section 2.1 which describe the possible transitions that any Erlang process evaluating the respective function can take, restricted by the shape of the environment if necessary. Adding these rules to the general proof system of Section 2.3 enables us to argue about any program that uses the library module without having to consider the module's source code. In this way we support a compositional style of reasoning which is relativised by the assumption that the concrete implementation of a library follows its specification.

From a pragmatical point of view we can argue that such assumptions are justified since software libraries are usually well–tested, and since their frequent use uncovers unexpected behaviour very soon. From a conceptual point of view however, the consistency between the library–specific transition rules and the concrete implementation with respect to the general proof system is an issue: do the specific rules fully reflect the behaviour of the library functions, or are they too abstract in the sense that certain details of the implementation are ignored although they have an impact on the verification problem? Or, in other words: is the (low–level) implementation of the library module correct with respect to the (high–level) specification?

We now concretely demonstrate our ideas using a specific class of programs which plays an important rôle in open distributed applications. The essential characteristics of this class are described in the following subsection.

6.1 Generic Client–Server Implementations

To support the software development process, the Erlang OTP Team has devised a wide range of design principles which describe how to structure a concrete Erlang software architecture. In particular several kinds of *behaviour* modules are offered as templates to build concrete systems. Among these one finds the **gen_server** behaviour which is widely used to implement client–server applications in a standardized way.

The **gen_server** module offers a number of interface functions which provide synchronous communication, debugging support, error handling, and other administrative tasks. The actual, application–specific implementation of the server has to be provided by the user in a separate module, called the *callback module*. Whenever the generic part of a server receives a request, the corresponding callback function is being invoked.

For example, **gen_server** provides the `call` function which can be invoked in the user process to send a request to a server:

```
Answer = gen_server:call(Server, Req)
```

If the handle_call function yields an answer, it is immediately returned to the waiting user process, and the server changes into the idle state again:

$$\begin{aligned} &\langle r[\mathtt{wait}(pid')], pid, q\rangle \parallel \\ &\langle \{\mathtt{reply}, answer, newstate\}, pid', q'\rangle \\ \longrightarrow\ &\langle r[answer], pid, q\rangle \parallel \langle \mathtt{loop}(newstate), pid', q'\rangle \end{aligned}$$

The remaining cases are handled in a similar fashion.

We are currently in the process of extending the proof system by appropriate transition rules and applying it to simple examples, starting with systems which consist of a finite number of clients and servers. Later, for more elaborated case studies, we will try to identify tactics and tacticals which automatically take (most of) the decisions described in Section 3, and we will try to extend the method to programs which involve dynamic process creation. The whole approach is also easily adaptable to several other libraries in the Erlang distribution, like systems of finite–state machines implemented by the generic gen_fsm module.

7 Conclusion

The present paper gives an overview of the main results obtained in the ASTEC project Verification of Erlang Programs, focusing in particular on the Erlang verification tool, a theorem-proving tool which assists in obtaining proofs that Erlang applications satisfy their correctness requirements formulated in a specification logic. We gave a summary of the verification framework as supported by the tool, discussed reasoning principles essential for successful verification such as inductive and compositional reasoning and reasoning about side-effect-free code, summarized our experience from a larger industrial case study, and suggested a practical approach for supporting verification in the presence of program libraries.

The experience gained in the project shows clearly the potential of the chosen approach to verification of Erlang programs. We were able to verify Erlang systems which are beyond the scope of most other existing verification approaches due to their dynamic nature. The price to pay is undecidability of the general verification problem. The verification task has to be split into automatable and manually assisted parts. Thus, the success of the approach depends crucially on the efficiency of the decision procedures employed and on the support provided for minimizing the need for human intervention in terms of high-level reasoning principles and user interface.

To make the presented verification method practically useful a considerable additional effort is required in several research directions. These include providing automatic support for identifying appropriate induction schemes, providing easy and context-sensitive access to the available proof machinery through the GUI, and designing efficient decision procedures automating the straightforward low-level reasoning and finite state space exploration.

21. George C. Necula. Proof-carrying code. In *Proc. POPL'97*, 1997.
22. H. Nilsson. Patent Application, 1999.
23. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. CAV'96,* Lecture Notes in Computer Science, 1102:411–414, 1996.
24. D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.
25. L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer Verlag (LNCS 828), 1994.
26. G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.
27. D. Sahlin, T. Franzén, and S. Haridi. An intuitionistic predicate logic theorem prover. In *Journal of Logic and Computation*, 2(5):619–656, October 1992.
28. C. Wikström, H. Nilsson, and H. Mattson. Mnesia database management system. In *Open Telecom Platform Users Manual.* Open Systems, Ericsson Utvecklings AB, Stockholm, Sweden, 1997.

A Monitoring and Instrumentation Tool developed in Erlang *

Miguel Barreiro, Victor M. Gulias, Juan J. Sanchez

LFCIA, Department of Computer Science
University of A Coruña, SPAIN
{enano, gulias, juanjo}@lfcia.org

Abstract

MONET is a simple and flexible monitoring tool, suitable for network monitoring, host performance tracking, and for the instrumentation of complex systems, among others. The whole foundation for MONET is the Erlang/OTP platform which was chosen for its robustness features, ease of integration and speed of development.

In order to keep MONET independent from the output device used to present system information, all data is generated as XML documents which are transformed by ERLATRON, a distributed XSLT processor also implemented in Erlang using a C++ library.

Keywords: monitoring, instrumentation, distributed systems, functional programming, Erlang, XML

1 Introduction

MONET is a simple and flexible monitoring tool, suitable for network monitoring, host performance tracking, and for the instrumentation of complex systems, among others. It was originally developed as a debugging and monitoring aid for a video-on-demand server under development at the LFCIA lab, and then evolved and generalized.

The whole foundation for MONET is the Erlang/OTP [1] platform, including MESH, EVA, SASL and Mnesia. Erlang has been chosen for its robustness features, ease of integration with the rest of the system, speed of development, and the functional background of many LFCIA members.

MONET was initially intended as a replacement for simple monitoring tools as MON [5], with additional support for variable tracking and graphing, as well as more complex instrumentation features.

The paper is structured as follows: Firstly, the main ideas of MONET operation are presented, paying attention to the classes and organization tree, the measurement objects (including the adapters for MON monitors), alarm and event handlers, and the alarm destinations. Section 3 is devoted to the external XML interface and how the ERLATRON subsystem can be used to perform XSL transformations to browse such output information. Section 4 presents a small example of MONET output, an alarm report, transformed to fit different output devices. Finally, some conclusions are presented in section 5.

2 Principles of Operation

MONET follows the MESH and EVA design, leveraging the infrastructure they provide and extending it. Figure 1 presents the modular structure of the system. For each measurement an MO (measurement object) is created, supervised by an MRP (measurement responsible process). A master handler (`main_handler`) receives MESH and EVA events and alarms, and may invoke specific action handlers according to complex configured conditions. Event and alarm logging is performed through the standard MESH logs; additionally, custom loggers can be plugged into `main_handler` or directly into EVA. The user interface is implemented through a web adaptation, using ERLATRON (see section 3) and INETS from *erlets*.

*Partially supported by FEDER TIC-1FD97-1759 and Xunta de Galicia PGIDT99COM10502

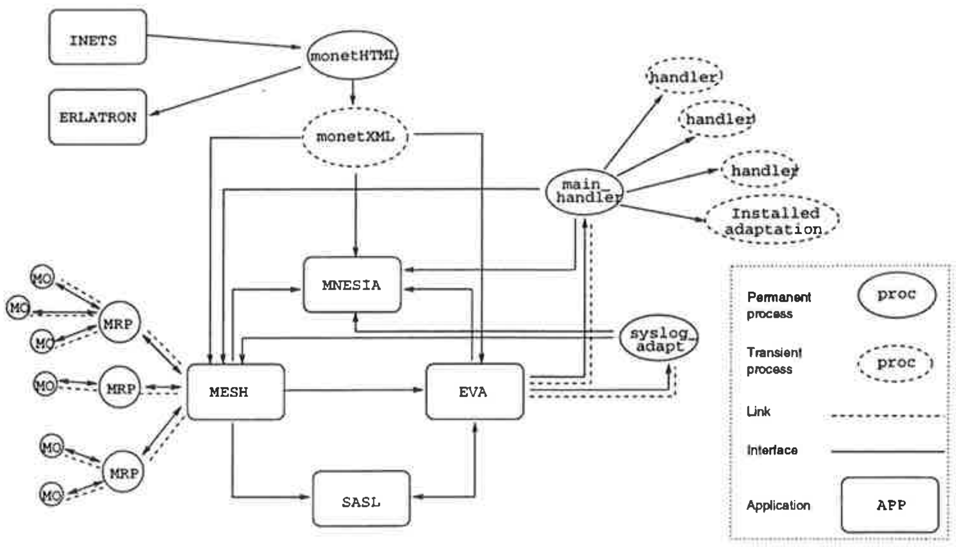


Figure 1: MONET structure

2.1 The Classes Tree

Instead of defining MESH Measurement Objects one by one, MONET creates a tree of monitored object classes, together with the measurement objects that will take objects of that class as monitored resources, like the one shown in figure 2. Each object class — a tree node — can be further specialized as desired.

Leaf nodes usually represent individual hosts or resources, but are not special in any way.

The tree is traversed from the root towards a node to determine the measurement classes suitable for that node.

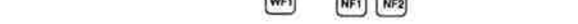


Figure 2: A sample classes tree

2.2 The Organization Tree

Resources to be monitored are defined as a directed graph, reflecting the logical grouping as managers see them. This logical grouping is completely artificial, and can be based on their physical, topological or simply organizational structure. Note that, despite its name, this structure is a directed graph and not strictly a tree, because branches can merge at any point (see figure 3 for an example). It is supposed to make sense to humans and has no other constraints, and does not even require all monitored resources to be present.

Each resource can have an attached list of classes; these classes position the resource in the *classes tree*, thus implicitly declaring the measurement objects that will monitor it.

Resources can also contain additional data in order to store their physical position, network connections, desired graphical representation, etc. This information can then be used by measurement objects to get additional configuration or by the user interfaces when representing the tree.

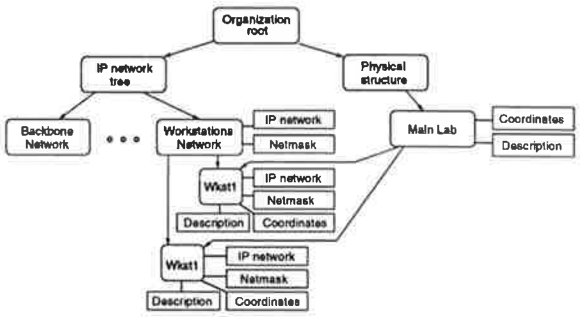


Figure 3: A sample organization tree

2.3 Measurement Objects

Following the MESH conventions, specific monitors are Measurement Objects, supervised by a Measurement Responsible Process. Measurement Objects can run in remote nodes.

In order to ease the development of simple, common monitors, MONET provides two generic measurement types which act as bridges between MESH and Erlang functions or Unix executables, respectively. Thus, existing monitor scripts and executables — such as those from MON— or Erlang code can be used as monitors without additional coding.

2.4 Alarm Handlers

Alarm handlers are called from `main_handler` when the appropriate alarms and conditions are triggered. Again, they can be either external executables or Erlang functions.

2.5 Alarm Destinations

Whenever an event or alarm is received, MONET checks whether it should call a handler. Alarm destinations are currently defined as functional expressions in order to define complex conditions, such as:

```
[
% Call dumphandler whenever an alarm is received.
%  (ie., when stddeps:always/5 returns true)
 { always, dumphandler},

% Call myHandler if the sender is enano2@borg
 { {sender, [enano2@borg]}, myHandler},

% Call thisHandler if extmodule:extfunc/5
%  returns true (called as
%    extmodule:extfunc(Name, Sender,
%     Severity, Class, [LocalArgs]))
 { {extmodule,extfunc,[localargs]},thisHandler},

% Call dumper if either always()
%                    or never() are true:
 { {any, [ {always,[]},{never,[]} ]}, dumper },

% Call selHandler if the two former are true:
 { { all, [{any, [ {always,[]},{never,[]} ]},
          {extmodule,extfunc,[localargs]} ]},
            selHandler}
]
```

This allows for composition of arbitrarily complex conditions as long as they can be expressed in Erlang. The use of higher-order functions to express complex configuration conditions is also employed in stylesheet selection (section 3.3).

2.6 XML Interface

The main MONET user interface is implemented as a set of *erlets* called from INETS. They never produce HTML output directly, but call the appropriate XML interface functions and transform the result instead, using ERLATRON (see section 3).

As long as MONET information can be retrieved from different devices, it is advisable to provide a general mechanism to adapt this content to the specific features of such output devices. Contrary to when style information is hard-coded into the content, separation of style from content allows for the same data to be presented in different ways. This enables:

- *Reuse of fragments of data:*

  Same content should look different in different contexts.

- *Multiple output formats:*

  Different media (paper, online), different sizes (manuals, reports), different classes of output devices (workstations, hand-held devices).

- *Styles tailored to the reader's preference:*

  Print size, color, and so on; easier support for blind or visually impaired users.

- *Standardized styles:*

  Corporate stylesheets can be applied to the content at any time.

- *Freedom from style issues:*

  Technical writers do not need to be concerned with layout issues because the correct style can be applied later.

- *Easy interface with third-party software:*

  Simple, standard interface for data interchange with other applications.

In order to achieve such goals, MONET generates most of its results as XML documents which can be transformed at a later stage according to an XSL stylesheet.

3 The Erlatron Subsystem

To perform the XSL transformation, MONET uses ERLATRON, a distributed XSLT processor implemented in Erlang using a C++ library (SABLOTRON [4]).

3.1 Overview

Figure 4 presents the actors involved in the ERLATRON subsystem. ERLATRON adapts the SABLOTRON library by using a port which performs basic transformations of a couple of Erlang binaries representing the XSL stylesheet and the XML source. SABLOTRON is based on the EXPAT library [3] and has been designed for performing fast and compact transformations. The port is managed by a generic server that offers XSLT services to any client. This *slave server* constitutes the basic processing unit and, considering the CPU cost of performing XSL transformations, low additional overhead is expected when used from Erlang.

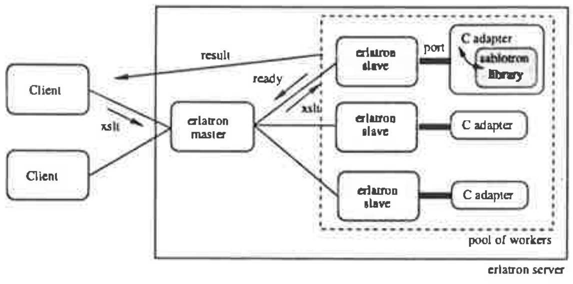


Figure 4: ERLATRON actors

In order to exploit a distributed framework, such as the Beowulf cluster introduced in [2], a simple master/slave architecture is deployed. In this setup, a *master server* is used to distribute requests to different slave servers running on a pool of computer nodes. The state of the master server is a collection of pending transformations as well as the information about idle slaves. The dispatching of requests, carried out by the master scheduler, consists of pairing a pending transformation with an idle slave server. Figure 5 shows the interaction among actors when solving an XSLT service.

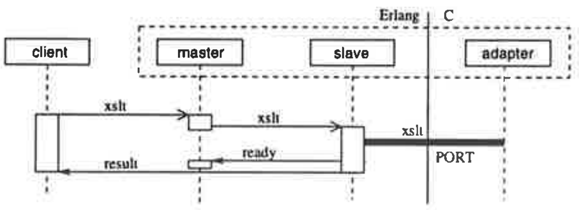


Figure 5: Interaction among actors

Clients interact with the master server through a global registered name, making an ERLATRON server takeover possible in case of failure without disturbing client interaction.

3.2 Benchmark

As the reader can guess, ERLATRON architecture seems to be quite interesting for web sites with a high number of requests which involve many XSL independent trasformations for dynamic content generation. We are going to present some preliminary results when using ERLATRON on a Linux Beowulf cluster with a frontend and up to 22 nodes. The frontend (Dual Pentium II 350MHz 384MB) runs INETS, serving a 64KB HTML generated using a 25KB XML document and a small XSL stylesheet; each node (AMD K6 300-266MHz, 96MB) hosts an ERLATRON slave. All the nodes are linked using a switched 100Mb Fast Ethernet.

Figure 6 shows the requests per second achieved when running Apache Bench (`ab`), a tool for benchmarking an HTTP server, at the frontend. As new XSLT processors are added to the slave pool, the server is able to increase its service rate until the concurrency level (C, number of simultaneous requests, `-c`) matches the pool size. Figure 7 presents the time taken to attend a collection of requests, varying the concurrency level.

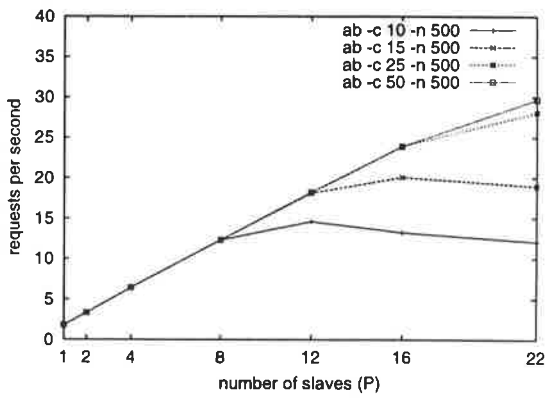


Figure 6: Requests per second

Figure 8 shows how the average time for a request can be kept almost constant while concurrency level is not greater than the number of slaves. It should be pointed out that the overhead for the case *slaves* = 22 and C low is due to the use of nodes 17-22 which are AMD K6 266Mhz, while nodes 1-16 are AMD K6 300Mhz. The master scheduler should take this fact into account

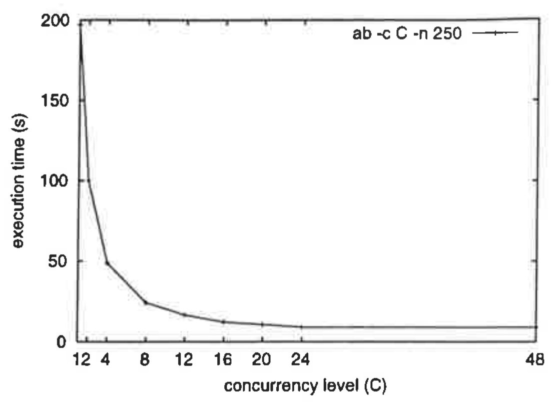


Figure 7: Execution time

and introduce some kind of priority among slaves. This is particularly interesting when some slaves run at the frontend because that will reduce network communications.

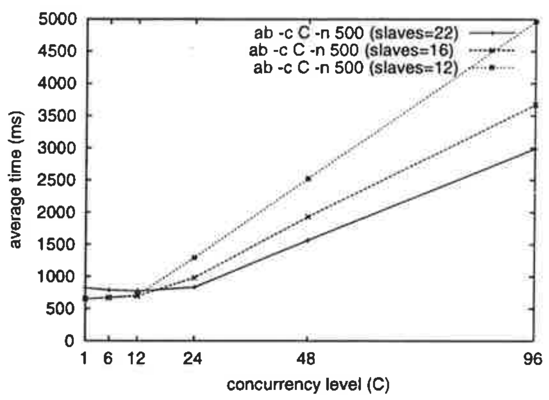


Figure 8: Average service time

3.3 Stylesheet Selection

Given a valid XML document, one of the ERLATRON duties must be the selection of an appropriate XSL stylesheet based on the type of the requested document and the nature of the output device, even though other parameters can be taken into account such as system workload (e.g. a simpler formatting can be chosen under heavy load condition). In order to accomplish this requirement, ERLATRON subsystem incorporates a module with two basic features:

- *Stylesheet registration:*

  A Mnesia database is used to store the binding between a document type and an XSL stylesheet. As the same document type can have different stylesheets, the database also stores a *rule* that decides whether the style can be applied.

- *Stylesheet selection:*

  Using the information stored in the XSL repository, a stylesheet, matching its rule in the given context, is chosen.

A rule for a given stylesheet is defined as a function that takes a context and an extra argument and returns `true` if the stylesheet can be applied to the document in such context. The *context* is defined as a list of pairs {*Key*, *Value*}, tipically the result of parsing the `QUERY_STRING` or the environment of an HTTP query. Some useful functions are provided to test for an specific {*Key*, *Value*} pair (`equal`) or for matching a regexp (`match`). For example, a rule to trigger an stylesheet when using the Lynx browser is `lynx`, defined as:

```
lynx(Context, _Extra) ->
  match({"User-Agent", "Lynx"}, Context).
```

Higher-order functions are used to define complex queries combining simpler rules. For example, the rule `all` takes a list of rules (functions and, optionally, the extra parameters) and tests all the conditions. As an example, the following Erlang term binds the document type `meas` with the XSL `http://my.host/xsl/style.xsl` when using Lynx and a simple style is required.

```
{meas,
    "simple lynx style for meas",
    "http://my.host/xsl/style.xsl",
    {all, [lynx,
          {equal,{"style", "simple"}}]}}.
```

4 Examples

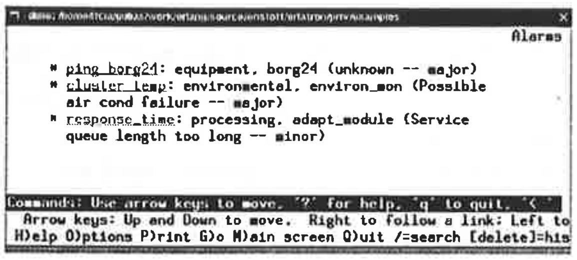
The following small example shows a simple alarm list browsed from Netscape, Lynx, and a WAP emulator, all with different styles.

The initial XML was, in all cases:

```
<ALARMS>
 <ALARM>
   <NAME>ping_borg24</NAME>
   <SENDER>borg24</SENDER>
   <CAUSE>unknown</CAUSE>
   <SEVERITY>major</SEVERITY>
   <CLASS>equipment</CLASS>
 </ALARM>
```

```
  <ALARM>
    <NAME>cluster_temp</NAME>
    <SENDER>environ_mon</SENDER>
    <CAUSE>Possible air cond failure</CAUSE>
    <SEVERITY>major</SEVERITY>
    <CLASS>environmental</CLASS>
  </ALARM>
  <ALARM>
    <NAME>response_time</NAME>
    <SENDER>adapt_module</SENDER>
    <CAUSE>Service queue length too long</CAUSE>
    <SEVERITY>minor</SEVERITY>
    <CLASS>processing</CLASS>
  </ALARM>
</ALARMS>
```

If the User-Agent is Lynx, the style chosen is a very simple one, without tables (figure 9).



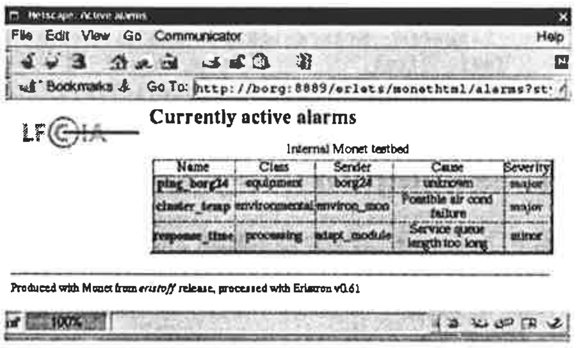
```
                                                                    Alarms

   * ping_borg24: equipment, borg24 (unknown -- major)
   * cluster_temp: environmental, environ_mon (Possible
     air cond failure -- major)
   * response_time: processing, adapt_module (Service
     queue length too long -- minor)

Commands: Use arrow keys to move, '?' for help, 'q' to quit, '<-'
  Arrow keys: Up and Down to move.  Right to follow a link: Left to
 H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=his
```

Figure 9: Alarms browsed from Lynx

In the User-agent is Mozilla (Netscape) instead, a richer XSL is chosen, displaying the alarms as a table (figure 10).



http://borg:8889/erlets/monethtml/alarms?st

Currently active alarms

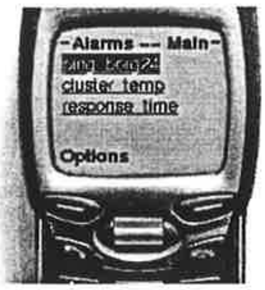
Internal Monet testbed

Name	Class	Sender	Cause	Severity
ping_borg24	equipment	borg24	unknown	major
cluster_temp	environmental	environ_mon	Possible air cond failure	major
response_time	processing	adapt_module	Service queue length too long	minor

Produced with Monet from *erutoff* release, processed with Erlstron v0.61

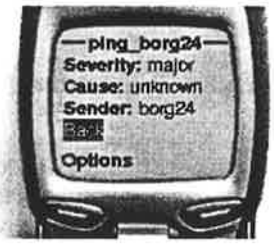
Figure 10: Alarms browsed from Netscape

Figure 11 shows a compact alarm list produced for a WAP terminal. In order to speed download time, specific alarm details are included as WML cards in this WAP style, as shown in figure 12.



Alarms — Main
ping_borg24
cluster_temp
response_time
Options

Figure 11: Alarms browsed from a WAP terminal



ping_borg24
Severity: major
Cause: unknown
Sender: borg24
Back
Options

Figure 12: An alarm card from a WAP terminal

5 Conclusions

MONET is a simple and flexible monitoring and instrumentation tool, developed in Erlang, making use of the OTP facilities and interfacing to external monitors and handlers if desired. All information presented is produced as XML and transformed using XSL according to the user profile, browser or other variables.

During the development, Erlang/OTP is showing an extremely high productivity in absolute and relative terms, after an initial learning curve.

References

[1] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] M. Barreiro and V. Gulias. General Issues on Cluster Administration. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume I. Prentice Hall, 1999.

[3] J. Clark. *Expat - XML Parser Toolkit 1.1.* `http://www.jclark.com/xml/expat.html`.

[4] T. Kaiser. *Sablotron.* Ginger Alliance Ltd. `http://www.gingerall.com`.

[5] J. Trocki. *mon, Service Monitoring Daemon.* `http://www.kernel.org/software/mon/`.

EUC'2000 - Participants

Conference chairman and speakers			
Kostis Sagonas	Uppsala University	Sweden	kostis@csd.uu.se
Miguel Barreiro Paz	Universidade da Coruña	Spain	enano@ceu.fi.udc.es
Per Bergqvist	CellPoint Systems AB	Sweden	per@cellpt.com
Richard Carlsson	Uppsala University	Sweden	richardc@csd.uu.se
Jakob Cederlund	UAB/OTP	Sweden	jakob@erix.ericsson.se
Lars-Åke Fredlund	SICS	Sweden	fred@sics.se
Scott Lystig Fritchie	Sendmail Inc	USA	scott@sendmail.com
Sean Hinde	one2one	UK	Sean.Hinde@one2one.co.uk
Peter Lundell	Ericsson Telecom AB	Sweden	peter.lundell@etx.ericsson.se
Kenneth Lundin	UAB/OTP	Sweden	kenneth@erix.ericsson.se
Patrik Nyblom	UAB/OTP	Sweden	Patrik.Nyblom@uab.ericsson.se
Richard A. O'Keefe	Otago University	New Zealand	ok@atlas.otago.ac.nz
Mikael Pettersson	Uppsala University	Sweden	Mikael.Pettersson@csd.uu.se
Bengt Tillman	Ericsson Radio Systems AB	Sweden	bengt.tillman@era.ericsson.se
Robert Tjärnström	Ericsson Telecom AB	Sweden	erandig@al.etx.ericsson.se
Ulf Wiger	Ericsson Telecom AB	Sweden	etxuwig@etxb.ericsson.se
Program chairman and organizers			
Bjarne Däcker	UAB/CSLab	Sweden	bjarne@cslab.ericsson.se
Tuula Carlsson	UAB	Sweden	Tuula.Carlsson@uab.ericsson.se
Anna Fedoriw	UAB	Sweden	Anna.Fedoriw@uab.ericsson.se
Lotta Larsson Hannu	UAB/ErlSys	Sweden	lotta@erlang.ericsson.se
Eeva Lintunen	UAB	Sweden	Eeva.Lintunen@uab.ericsson.se
Participants			
Tomas Abrahamsson	Ericsson Radio Systems AB	Sweden	epktoab@lmera.ericsson.se
Ola Andersson	WiseOne AB	Sweden	Ola.Andersson@wiseone.se
Peter Andersson	Ericsson Expertise Inc	Ireland	Peter.Andersson@eei.ericsson.se
Ingela Anderton	UAB/ErlSys	Sweden	ingela@erlang.ericsson.se
Marcus Arendt	Marcus Arendt AB	Sweden	marcus@arendt.se
Joe Armstrong	Bluetail AB	Sweden	joe@bluetail.com
Thomas Arts	UAB/CSLab	Sweden	thomas@cslab.ericsson.se

Gösta Ask	Ericsson Radio Systems AB	Sweden	Gosta.Ask@era.ericsson.se
Björn Axelsson	UAB/ErlSys	Sweden	bjorn@erlang.ericsson.se
Knut Bakke	Ericsson AS	Norway	Knut.Bakke@eto.ericsson.se
Didier Begay	France Telecom R&D	France	didier.begay@francetelecom.fr
Clara Benac Earle	UAB/CSLab	Sweden	clara@cslab.ericsson.se
Per Bengtsson	TeliaPromotor AB	Sweden	
Johan Bevemyr	Bluetail AB	Sweden	jb@bluetail.com
Gerald Biederbeck	Ericsson Eurolab Deutschland GmbH	Germany	Gerald.Biederbeck@eed.ericsson.se
Martin Björklund	Bluetail AB	Sweden	mbj@bluetail.com
Hans Bolinder	UAB/OTP	Sweden	hasse@erix.ericsson.se
Kent Boortz	UAB/OTP	Sweden	kent@erix.ericsson.se
Thomas Buksholt	Ericsson AS	Norway	Thomas.Buksholt@eto.ericsson.se
Mikael Bylund	TeliaPromotor AB	Sweden	
Göran Båge	Ericsson Radio Systems AB	Sweden	goran.bage@era-t.ericsson.se
Emil Bäckmark	Ericsson Telecom AB	Sweden	etxelar@etxb.ericsson.se
Markus Cech	Ericsson Eurolab Deutschland GmbH	Germany	Markus.Cech@eed.ericsson.se
Francesco Cesarini	Cesarini Consulting Ltd.	UK	francesco@home.se
Gennady Chugunov	SICS	Sweden	gena@sics.se
Ornella Ciotti	Ericsson Lab Italy	Italy	ornella.ciotti@tei.ericsson.se
Pierre Crégut	France Telecom R&D	France	pierre.cregut@francetelecom.fr
Mats Cronqvist	Ericsson Telecom AB	Sweden	etxmacr@etxb.ericsson.se
Mats Cullberg	Ericsson Telecom AB	Sweden	mats.cullberg@ericsson.com
Mads Dam	SICS	Sweden	mfd@sics.se
Maurizio Di Stefano	Ericsson Lab Italy	Italy	teimads@tei.ericsson.se
Niclas Eklund	UAB/OTP	Sweden	nick@erix.ericsson.se
Dermot O. Flaherty	Ericsson Expertise Inc	Ireland	Dermot.OFlaherty@eei.ericsson.se
Dick Forsberg	Ericsson Telecom AB	Sweden	etxdifo@tb.etx.ericsson.se
Henrik Forsgren	Starthouse	Sweden	henrik@starthouse.se
Magnus Fröberg	Bluetail AB	Sweden	magnus@bluetail.com
Li Ye Ge	Ericsson Communications Software	China	rdcliyg@ecnshsr201.etc.ericsson.se
Luke Gorrie	Bluetail AB	Sweden	luke@bluetail.com

Catrin Granbom	UAB/ErlSys	Sweden	catrin@erix.ericsson.se
Pär Grandin	Ericsson Eurolab Deutschland GmbH	Germany	Paer.Grandin@eed.ericsson.se
Joakim Grebenö	Bluetail AB	Sweden	jocke@bluetail.com
Rickard Green	UAB/OTP	Sweden	rickard@erix.ericsson.se
Dan Gudmundsson	UAB/OTP	Sweden	dgud@erix.ericsson.se
Victor M. Gulias	Universidade da Coruña	Spain	gulias@dc.fi.udc.es
Dilian Gurov	SICS	Sweden	dilian@sics.se
Björn Gustavsson	UAB/OTP	Sweden	bjorn@erix.ericsson.se
Siri Hansen	Ericsson Expertise Inc	Ireland	Siri.Hansen@eei.ericsson.se
Per Hedeland	Bluetail AB	Sweden	per@bluetail.com
John Hughes	Chalmers Technical University	Sweden	rjmh@cs.chalmers.se
Gunilla Hugosson	UAB/OTP	Sweden	gunilla@erix.ericsson.se
Peter Ingels	Ericsson Inc	USA	EUSPEIN@am1.ericsson.se
Jonas Jacobsson	Ericsson Radio Systems AB	Sweden	jonas.k.jacobsson@era.ericsson.se
Patrik Jansson	Chalmers Technical University	Sweden	patrikj@cs.chalmers.se
Erik Johansson	Uppsala University	Sweden	Erik.Johansson@csd.uu.se
Thomas Johnsson	Carlstedt Research & Technology	Sweden	johnsson@crt.se
Per-Johan Josefsson	BlueLabs AB	Sweden	Per-Johan.Josefsson@sth.frontec.se
Bertil Karlsson	UAB/OTP	Sweden	bertil@erix.ericsson.se
Håkan Karlsson	UAB/CSLab	Sweden	hakan.karlsson@ericsson.com
Magnus Karlson	UAB/OTP	Sweden	mk@erlang.ericsson.se
Mathias Karlsson	Ericsson Radio Systems AB	Sweden	mathias.karlsson@era.ericsson.se
Mikael Karlsson	Creado Systems	Sweden	mikael.karlsson@creado.com
Torbjörn Keisu	UAB/SARC	Sweden	Torbjorn.Keisu@ericsson.com
Håkan Larsson	UAB/CSLab	Sweden	hakan.larsson@ericsson.com
Edmond Lew	Ericsson Australia EPA	Australia	edmond.lew@ericsson.com.au
Magnus Lille	WiseOne AB	Sweden	
Fredrik Linder	BlueLabs AB	Sweden	fredrik.linder@bluelabs.se
Thomas Lindgren	Bluetail AB	Sweden	thomasl@bluetail.com
Olof Lindström	Ericsson Inc	USA	EUSOLL@am1.ericsson.se
Rutger Ljungqvist	Consafe Infotech AB	Sweden	rutger.ljungqvist@ericsson.no
Leslaw Lopacki	Ericsson AS	Norway	etolel@ericsson.no
Matthias Läng	UAB/CSLab	Sweden	Matthias.Lang@ericsson.com

Arild Løvendahl	Ericsson AS	Norway	Arild.Sigmund.Lovendahl@ericsson.no
Håkan Mattsson	UAB/CSLab	Sweden	hakan@cslab.ericsson.se
Ingvar Meyer	UAB/OTP	Sweden	ingmey@erix.ericsson.se
Yehiel Milman	Amdocs	Israel	RONITBEN@Amdocs.com
Parastoo Mohagheghi	Ericsson AS	Norway	Parastoo.Mohagheghi@eto.ericsson.se
Hans Nahringbauer	TeliaPromotor AB	Sweden	Hans.H.Nahringbauer@telia.se
Per Nehlin	Ericsson Telecom AB	Sweden	per.nehlin@etx.ericsson.se
Daniel Neri	Sigicom AB	Sweden	daniel.neri@sigicom.com
Anna Neovius	Askus AB	Sweden	Anna.Neovius@etx.ericsson.se
Hans Nilsson	UAB/CSLab	Sweden	hans@cslab.ericsson.se
Raimo Niskanen	UAB/OTP	Sweden	raimo@erix.ericsson.se
Thomas Noll	Royal Institute of Technology	Sweden	noll@it.kth.se
Arto Nummelin	Ericsson Telecom AB	Sweden	arto.nummelin@etx.ericsson.se
Peter Nyström	Aragon Fondkommission AB	Sweden	pnystrom@aragon.se
Sven-Olof Nyström	Uppsala University	Sweden	svenolof@csd.uu.se
Simon Olofsson	Ericsson Telecom AB	Sweden	simon.olofsson@etx.ericsson.se
Rex Page	University of Oklahoma	USA	page@ou.edu
David B. Paul	Ericsson Inc	USA	exudpau@exu.ericsson.se
Tony Pedley	Ericsson Intracom	UK	tonyp@terminus.ericsson.se
Oswaldo Perdomo	UAB/ErlSys	Sweden	Oswaldo.Perdomo@uab.ericsson.se
Liu Qi	Ericsson Communications Software	China	rdcliuq@ecnshsr201.etc.ericsson.se
Anders Ramsell	TeliaPromotor AB	Sweden	
Mickaël Rémond	IDEALX	France	mickael.remond@IDEALX.com
Tony Rogvall	Bluetail AB	Sweden	tony@bluetail.com
Per Romin	Ericsson Business Networks AB	Sweden	Per.Romin@ebc.ericsson.se
Zhou Rong	Ericsson Communications Software	China	rdczhro@ecnshsr201.etc.ericsson.se
Dan Sahlin	UAB/CSLab	Sweden	dan@cslab.ericsson.se
Bo Samuelsson	Ericsson Telecom AB	Sweden	etxsamo@etxb.ericsson.se
Ola Samuelsson	Cyberode IT AB	Sweden	qtxolas@etxb.ericsson.se
Juan J. Sanchez	Universidade da Coruña	Spain	juanjo@dc.fi.udc.es
Staffan Sjödin	UAB/OTP	Sweden	Staffan.Sjodin@ebc.ericsson.se

Hal Snyder	Vail Systems	USA	hal@vailsys.com
Igor Solovyev	Ericsson Telecom AB	Sweden	Igor.Soloviev@etx.ericsson.se
Mikhail Solovyev	Ericsson Telecom AB	Sweden	etxmsol@tn.etx.ericsson.se
David Sonnek	SEB Företagsinvest	Sweden	David.Sonnek@seb.se
Denise Stack	Ericsson Expertise Inc	Ireland	denise.stack@eei.ericsson.se
Milorad Stanic	BlueLabs AB	Sweden	Milorad.Stanic@sth.frontec.se
Håkan Stenholm	Ericsson Telecom AB	Sweden	etxhste@etxb.ericsson.se
Per Sternås	Ericsson Business Networks AB	Sweden	Per.Sternas@ebc.ericsson.se
Kristina Stjärngren	UAB/ErlSys	Sweden	star@erlang.ericsson.se
Sebastian Strollo	Bluetail AB	Sweden	seb@bluetail.com
Per Einar Strömme	Ericsson Telecom AB	Sweden	etxnep@etxb.ericsson.se
Göran Stupalo	UAB/OTP	Sweden	stupalo@erix.ericsson.se
Henrik Swerin	UAB/ErlSys	Sweden	henriks@erlang.ericsson.se
Lars Thorsén	UAB/CSLab	Sweden	lars@erix.ericsson.se
Torbjörn Törnkvist	Bluetail AB	Sweden	tobbe@bluetail.com
Robert Virding	Bluetail AB	Sweden	rv@bluetail.com
Jane Walerud	Bluetail AB	Sweden	jane@bluetail.com
Jörn Wegener	UAB/ErlSys	Sweden	Jorn.Wegener@uab.ericsson.se
Sverker Wiberg	UAB/OTP	Sweden	sverkerw@erix.ericsson.se
Daniel Wiik	Ericsson Radio Systems AB	Sweden	daniel.wiik@era.ericsson.se
Claes Wikström	Bluetail AB	Sweden	klacke@bluetail.com
Christopher Williams	Ericsson Expertise Inc	Ireland	chris.williams@ericsson.com
Patrik Winroth	Bluetail AB	Sweden	patrik@bluetail.com
Gao Yu	Ericsson Communications Software	China	rdcgayu@ecnshsr201.etc.ericsson.se
Lulseged Zerfu	Ericsson Telecom AB	Sweden	etxluze@tb.etx.ericsson.se
Lennart Öhman	Sjöland & Thyselius Telecom AB	Sweden	lennart.ohman@st.se
Göran Östlund	Sveriges Verkstadsindustrier	Sweden	Goran.Ostlund@vi.se

UAB =	Ericsson Utvecklings AB		
CSLab =	Computer Science Laboratory	SARC =	Software Architecture Laboratory
ErlSys =	Erlang Systems	OTP =	OTP Product Unit