7th International Erlang/OTP User Conference

Stockholm, September 27, 2001

Proceedings

EUC'2001 http://www.erlang.se/euc/01/
Ericsson Utvecklings AB
P.O. Box 1505
SE-125 25 Älvsjö Stockholm
Sweden

ERICSSON

ERLANG

Erlang/OTP User Conference 2001

Conference Programme

08.30 *Registration*

Applications I

09.00 **The Best SSL Appliance in the World.**
Claes Wikström, Johan Bevemyr and Tony Rogvall, Alteon WebSystems.

09.45 **Welcome SMS in Erlang - Experiences of Rapid Deployment in a GSM Network.**
Sean Hinde, one2one.

10.10 **Improving Robustness in Distributed Systems.**
Per Bergqvist, Synapse Systems AB.

10.30 *Coffee*

Applications II

11.00 **The migration from Erlang to OTP: A Case Study of a Heavy Duty TCP/IP Client-server System written in Erlang.**
Mickaël Rémond, `erlang-fr.org`, and Francesco Cesarini, Cesarini Consulting Ltd.

11.30 **An Erlang-based Hierarchical Distributed VoD System.**
Miguel Barreiro, José L. Freire, Víctor M. Gulías, Javier Mosquera and Juan J. Sánchez, University of Coruña.

12.00 **Erlang in the Corelatus MTP2 Signalling Gateway.**
Matthias Läng, Corelatus.

12.30 *Lunch*

Technology I

14.00 **Tools for Designing Web Based Interfaces for Erlang/OTP.**
Martin Gustafsson, OTP Unit, Ericsson.

14.20 **3D Graphics with Erlang - The OpenGL Interface.**
Jakob Cederlund, GNO Data.

14.40 **Development of a Verified Erlang Program for Resource Locking.**
Thomas Arts and Clara Benac Earle, CSLab, Ericsson.

15.00 **Erlang Specification Method - A Tool for the Graphical Specification of Distributed Systems.**
Frank Huch, Christian-Albrechts-Universität.

15.30 *Coffee*

Technology II

16.00 **HiPE Version 1.0.**
Kostis Sagonas, Uppsala University.

16.30 **Cross-Module Optimization of Erlang.**
Thomas Lindgren.

16.50 **The EC Erlang Compiler.**
Maurice Castro, Software Engineering Research Centre.

17.15 **Preview of the Erlang/OTP R8 Release.**
Kenneth Lundin, OTP Unit, Ericsson.

17.30 *Close (and pub evening)*

ISD–SSL
The best SSL appliance in the world

By
Claes Wikstrom, Tony Rogvall and
Johan Bevemyr
{klacke,tony,jb}@bluetail.com

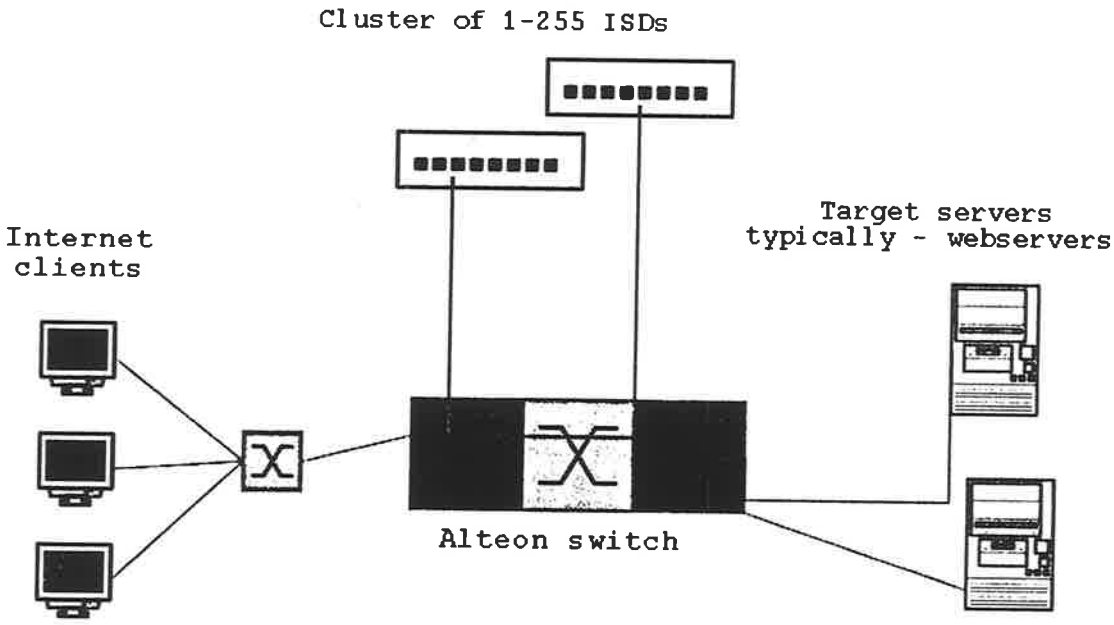
What have we built

- The ISD–SSL – An SSL terminating appliance (built in Stockholm)
- ISD–Core Platform – A general purpose platform intended to build Internet appliances with. (Stockholm)
- ISD–Tungsten – A superfast Checkpoint based firewall (San Jose)

ISD–Core platform

- A standard stackable 1U Dell PC with regular PCI slots, IDE disc and NICs
- Linux – stripped and rewritten /etc/rc structure
- Erlang, Mnesia
- The registry – up/down gradability of config data, semantic checks etc
- SSI – Single system image
- CLI – general purpose CLI engine
- Software managemnet, multiple Oses/applications, alarms logs events system management etc
- C api

Typical setup

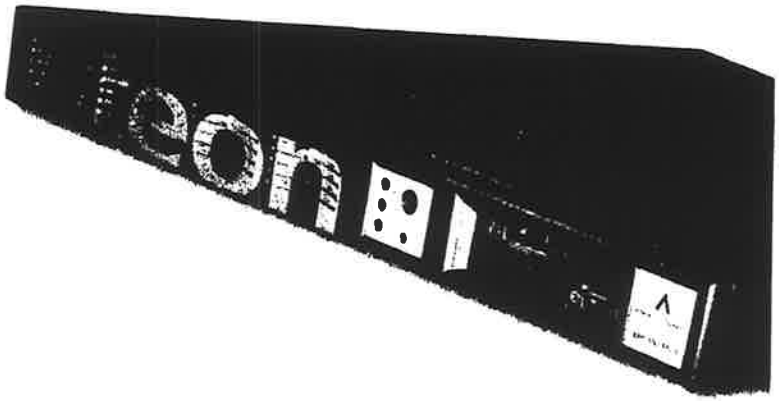


What is that Alteon switch anyway

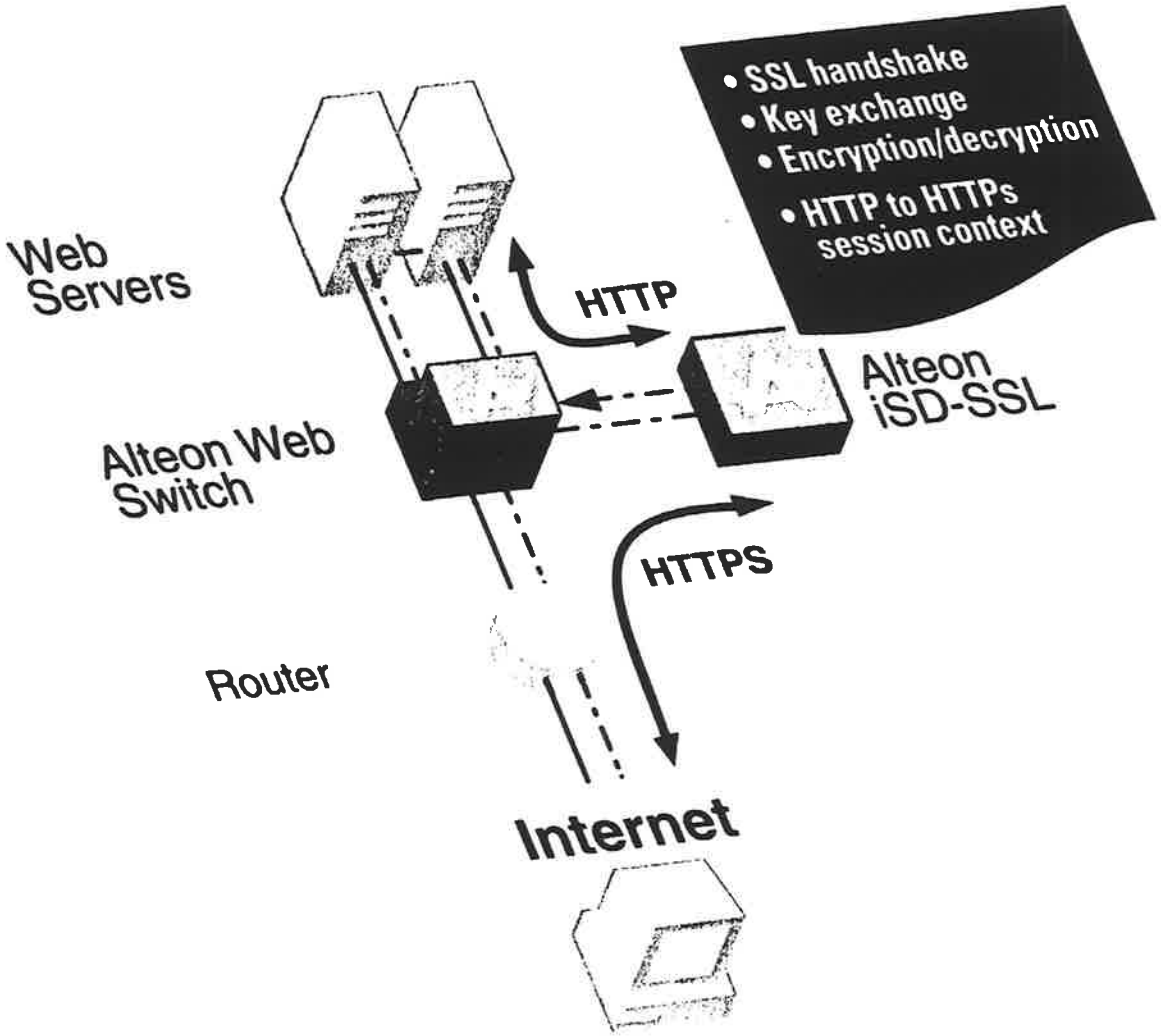
- An L7 high speed switch
- HTTP, WAP, RTSP, ... aware switching
- A load balancer
- Hardware assisted high level switching and loadbalancing
- Typically sits in front of large web farms
- Pretty expensive piece of hardware

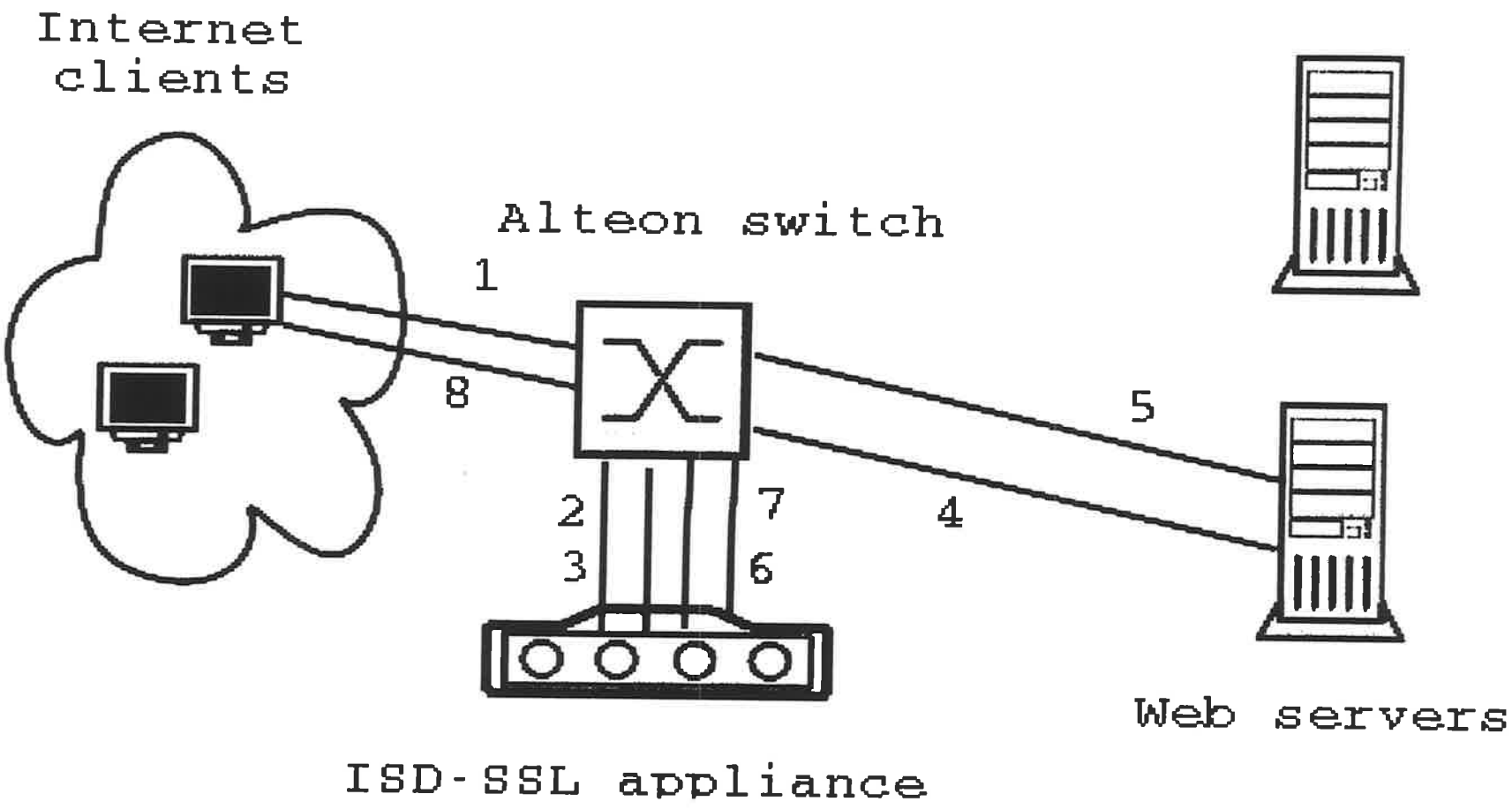
ISD–Core platform gives

- Scalability
- Fault tolerance
- CLI
- OS
- Hardware box
- Configuration



ISD–SSL


• SSL handshake
• Key exchange
• Encryption/decryption
• HTTP to HTTPs session context
Web Servers
HTTP
Alteon iSD-SSL
Alteon Web Switch
HTTPS
Router
Internet



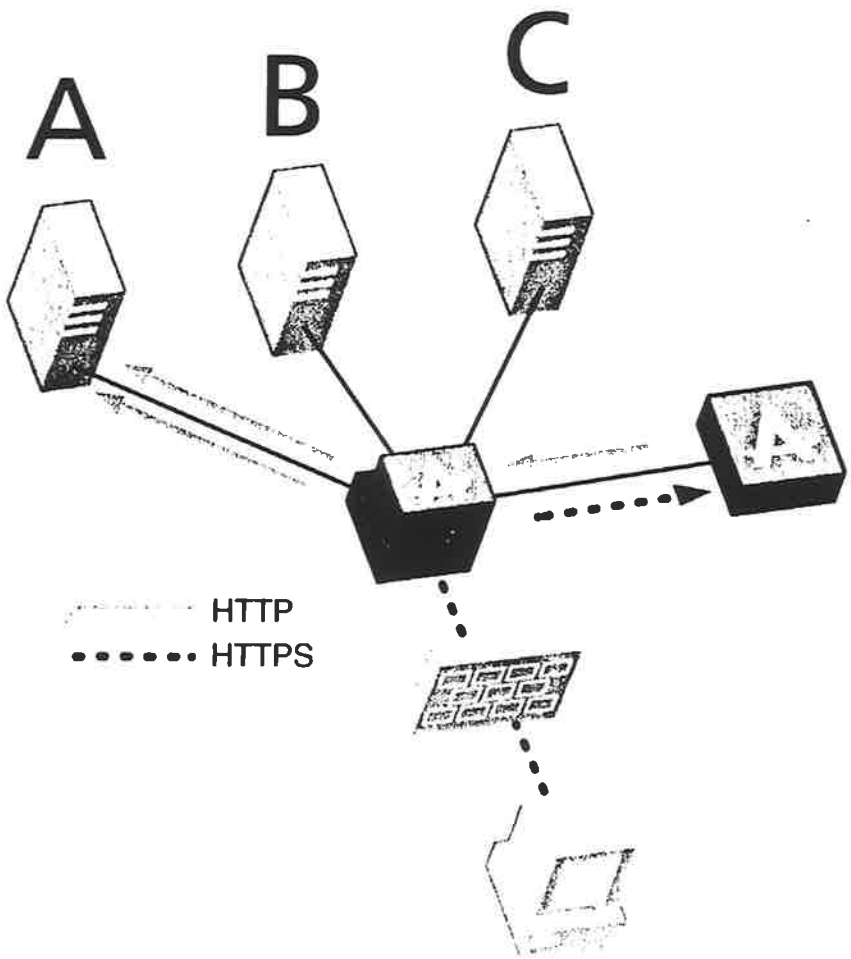
1. Switch redirects and loadbalances SSL traffic to group of ISDSSL boxes
2. ISD terminates SSL session.
3. ISD sets up a new connection to VIP:80
4. Switch load balances clear text traffic to group of webservers
5,6. Outbound traffic from webservers is once again redirected to group of ISDs
7,8 ISD proxies traffic. Encrypted SSL traffic on the client side, and clear text traffic on the servers ide

ISD–SSL feature set

- Easy certificate and key management
- Secure storage of private keys
- Scalable solution – buy more boxes and get more performance
- Hardware HA solution.
- SSL to the client, yet the switch sees clear text traffic and can do it's funky L7 stuff with the traffic
- Offload webserver from RSA and bulk encryption
- Transparent – webservers see real client IP as source IP
- Client authentication, revokation lists
- Based on openssl + bignum hw cards

Shopping Cart Persistence – (uhh)

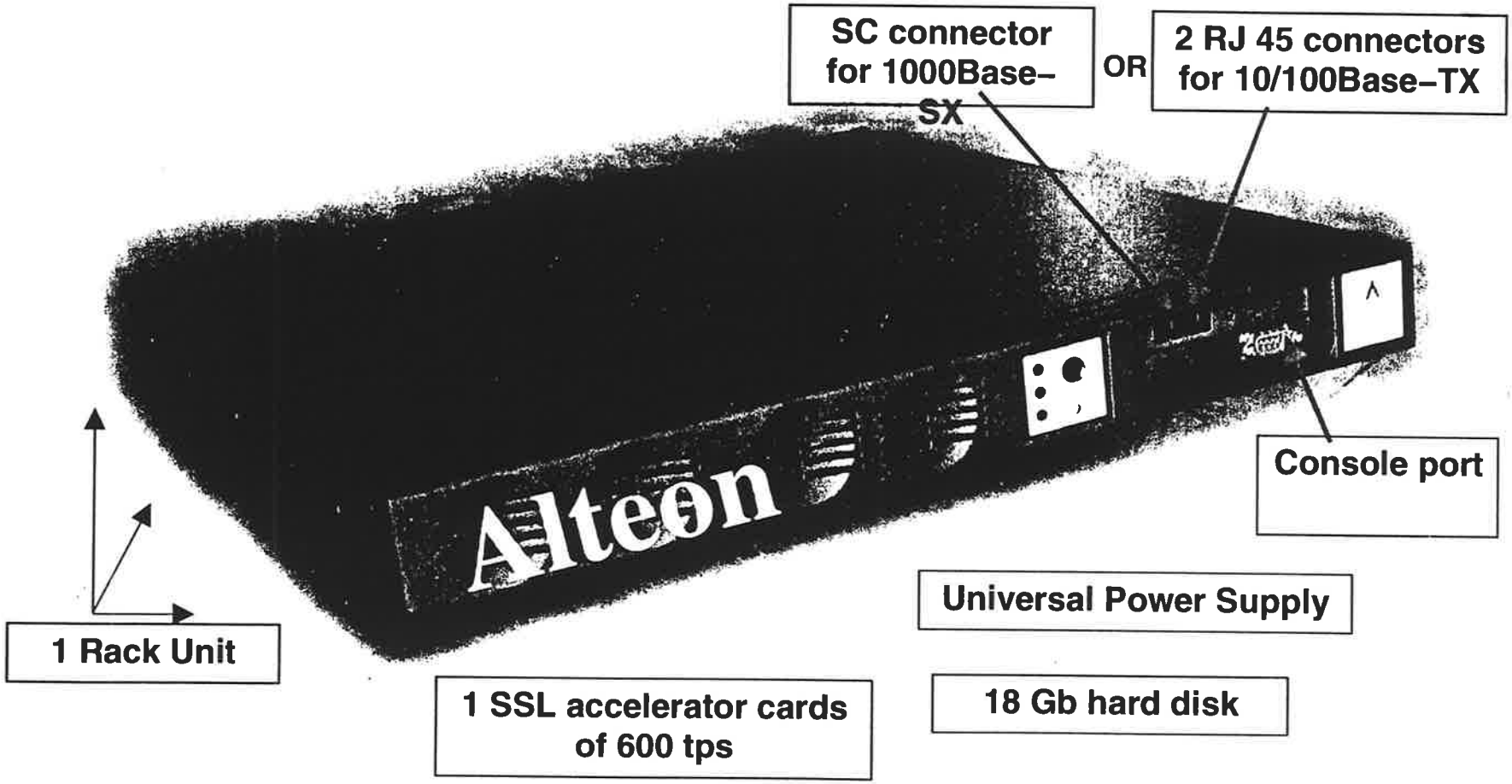
Without Manual Server Configurations



- iSD–SSL makes encrypted cookie visible to Web switch
- Web switch can associate shopping cart (HTTP) and payment (HTTPS) sessions from same user (cookie)
- User cookie is the most accurate persistence indicator
  - Due to mega– and micro–proxies

The BOX

A purpose–built appliance for processing SSL sessions



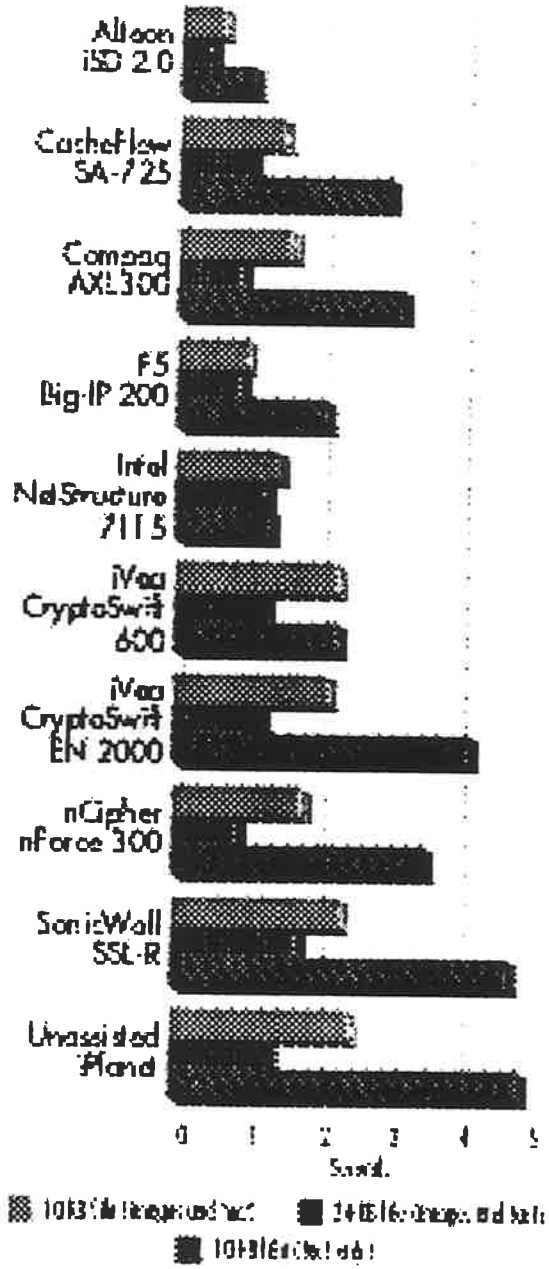
We are best, says
Network Computing magazine

REPORT CARD • External SSL Accelerators

	Weight	Alteon WebSystems iSD 2.0 SSL Accelerator	F5 Networks Big-IP 200 e-Commerce Controller 3.3.1	CacheFlow Server Accelerator 725	Intel NetStructure 7115 e-Commerce Accelerator	SonicWall SSL Accelerator—Rack Mount
SCALABILITY	25%	5	5	4	4	4
SECURITY	25%	4	4	4	4	4
PERFORMANCE	20%	5	4.5	4	3	4
CRYPTOGRAPHIC SUPPORT	10%	5	5	3	5	3
MANAGEMENT/CONFIGURATION	10%	4.5	4	4	3	3.5
PRICING	10%	4	3.5	3	4	3
TOTAL SCORE		**4.60**	**4.40**	**3.80**	**3.80**	**3.75**
		A	**A–**	**B**	**B**	**B**

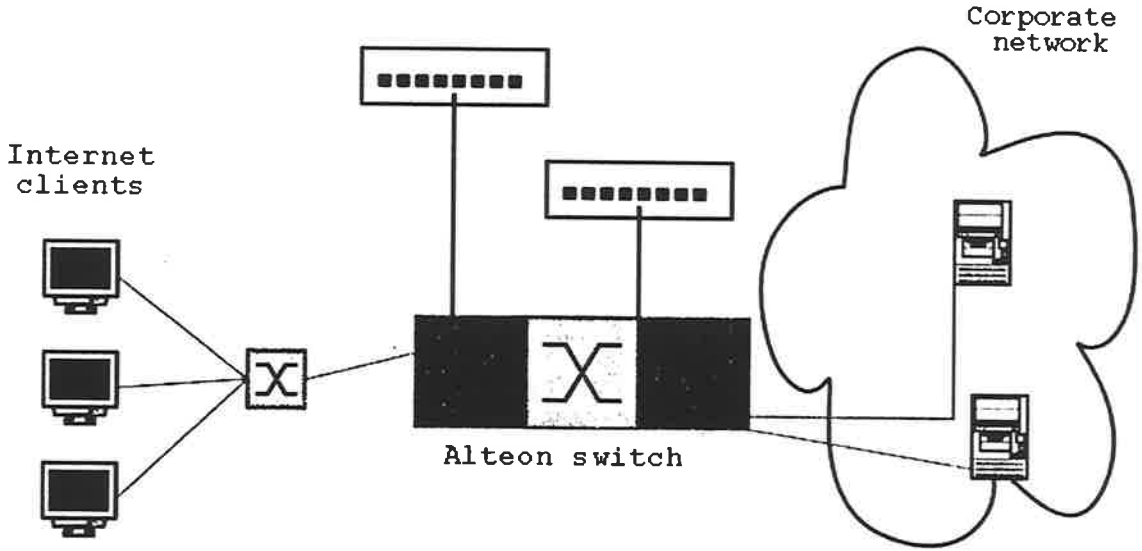
A≥4.3, B≥3.5, C≥2.5, D≥1.5, F<1.5 A-C GRADES INCLUDE + OR – IN THEIR RANGES. TOTAL SCORES AND WEIGHTED SCORES ARE BASED ON A SCALE OF 0-5. CUSTOMIZE THE RESULTS OF THIS REPORT CARD TO YOUR ENVIRONMENT USING THE INTERACTIVE REPORT CARD™, A JAVA APPLET ON NETWORK COMPUTING ONLINE, AT *WWW.NETWORKCOMPUTING.COM*.

CONNECT TIMES



Tungsten Firewall

Cluster of 1-255 Tungsten ISDs
each one running ISD-Core + Checkpoint firewall



- SYN packet arrives at the switch (FIN or RST as well)
- Switch redirects packet to one of the ISDs
- Checkpoint software at the ISD decides what to do with that particular TCP flow
- ISD instructs switch that {src ip, src port, dest ip, dst port} shall be cut–through inside the switch
- FIN or RST arrives at the switch – switch redirects to the same ISD that got the SYN
- ISD terminates that TCP particular TCP flow

Customers

- Nokia
- Speedy tomato
- VMdata
- IRS (US tax department)
- Reuters
- Pentagon
- Zurich Insurance
- Chrysler ... and lots lots more

Welcome SMS

Welcome SMS in Erlang.
Experiences of Rapid Deployment
in a GSM network

one 2 one

T Mobile International

The Requirement

- Send out SMS messages to One 2 One customers when they arrive in a foreign network.
- Don't send them any more messages for at least two weeks.
- Different messages depending on destination Country.
- "Marketing" usable GUI configuration.
- Option to send messages to roaming customers when they arrive in the UK.
- Ultra reliability not a hard requirement
  - Except NOT sending multiple messages

one 2 one

T Mobile International

The Timescale

- 8 (yes, 8) weeks from start of design to launch of service.

one 2 one

T Mobile International

Access to Customer Behaviour?

- Monitor all international c7 signalling links.
- Extract contents of MAP messages:
  - Where is the customer? (VLR Address)
  - Whose customer is it (IMSI)
  - Who should receive the SMS? (MSISDN)
- VLR address and IMSI are in Location Update, MSISDN is in Insert Subscriber Data. These messages can even take different physical routes...

one 2 one

T Mobile International

Business Logic

- Decide which message to send
  - Different depending on class of customer (prepay, postpay, wholesale)
  - Different depending on destination Country
  - Different for inbound roamers
- Have we sent them one already in this country?
- Have they elected not to receive these messages

one 2 one

T Mobile International

Operational Requirements

- Integrated with existing alarm management system (SNMP Traps)
- Generate statistic reports of system activity into existing stats database.
- Load balancing between SMS service centres
- Ability to on the fly reconfigure which SMS service centres are active and the load balancing between them.

one 2 one

T Mobile International

Design Options

- Commercial c7 probes to do automatic correlation of all messages for a particular activity.
  - Best supplier delivery time longer than entire project duration.
  - Time taken to integrate.. Who knows.
- Option to exercise Synergy across T-Mobile International group.
  - Several useful design ideas shared. Again, timescales not lined up
- Standard commercial systems - expensive, don't meet all requirements, integration time huge

one 2 one

T Mobile International

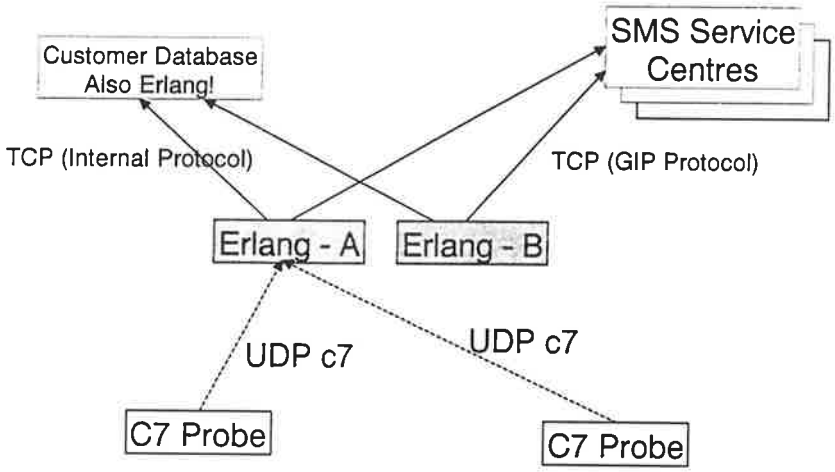
Design Options 2

- Use Erlang?
  - But what about c7 probes?
- Possibility to use standard test set (MPA 7400) to extract all c7 messages as raw data and send out as UDP
- Two spare (but old) 4 CPU Sun machines about to be de-commissioned
- Lead times of all elements OK. Now to work...

one 2 one

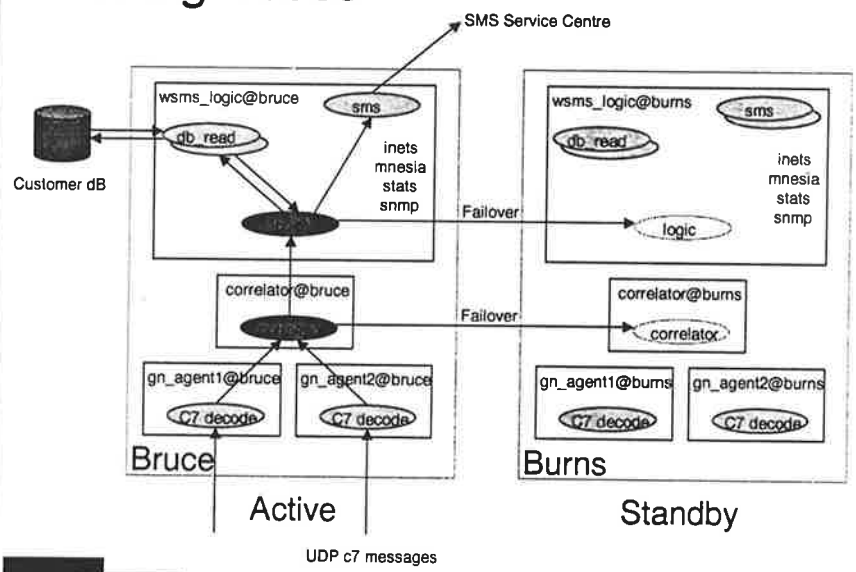
T Mobile International

Design Overview



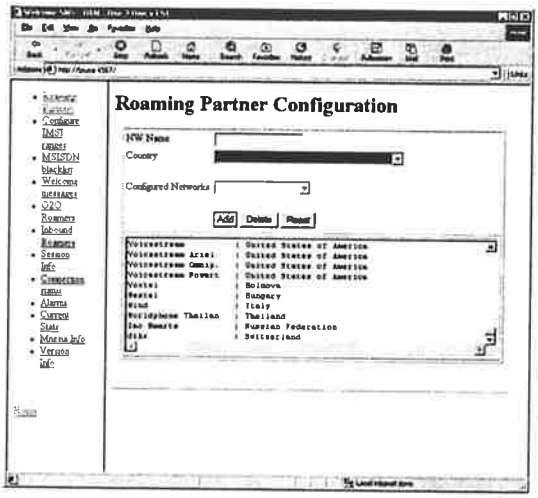
one 2 one

Erlang Nodes



one 2 one

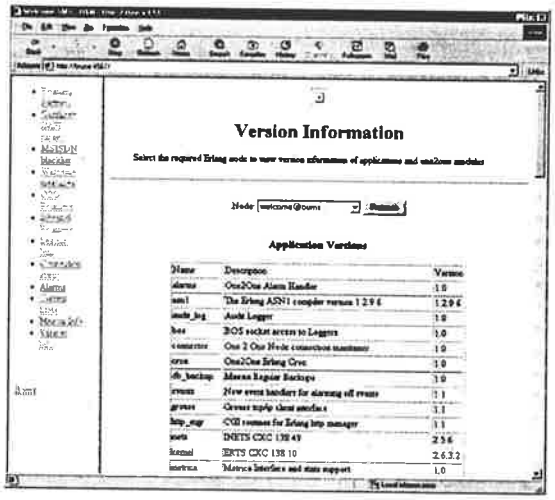
Some screenshots



one 2 one

T Mobile International

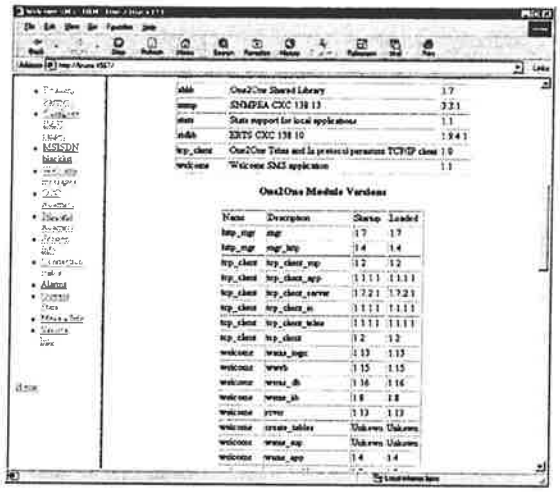
Some screenshots



one 2 one

T Mobile International

Some screenshots



one 2 one

T Mobile International

Problems encountered

- Bug with restart of one process starting up too many sms sending processes.
- LAN Latency - both for UDP and dB reads.
  - Fixed by tuning concurrency and timeouts.
- One module not permanently loaded in the customer Database.
- Scanning an Mnesia table to delete some rows is pretty heavy..

one 2 one

T Mobile International

What else have we been up to?

- Oracle Call Interface Binding to Erlang
  - Multithreaded Driver
  - Pools of Sessions
  - Decent Performance (> 1000 selects per second)
  - 1500 lines of C (so far!)
  - Garbage collection of stale handles

one 2 one

T Mobile International

What else have we been up to?

- Oracle Call Interface Binding to Erlang

```
N = "447956123456",
F=fun() ->
    case oci:exec("select name, balance from cust where telno = :e", [N]) of
      [{Name, Amount}] ->
          A1 = Amount - 10.0,
          oci:exec("update cust set balance = :e where telno = :e",[A1, N])
      [] ->
          oci:abort("Customer not found")
    end,
oci:transaction(Connection, F).
```

one 2 one

T Mobile International

What else have we been up to?

- Generic "behaviour" for tcp/ip clients
  - link management (using heartbeats)
  - simple callbacks for protocol implementation
  - model is all transactions multiplexed down 1 socket
  - time taken to implement new protocol down to a few days
- Call Data Record - record and playback

one 2 one

T Mobile International

What else have we been up to?

- The Original Project used to justify the use of Erlang never got into service. All hardware has been re-deployed for other Erlang applications!

one 2 one

T Mobile International

Improving Robustness in Distributed Systems

Per Bergqvist
per@synapse.se

Erlang User Conference 2001

Design base

- Cluster of cooperating hosts
- Erlang and C
- COTS hardware based
- Unix based (i.e. Solaris or Linux)
- 10/100/1000 base-T backplane ("system area network")

Cluster

- Shared, distributed, system configuration
- Each host have ONE cluster controller
- Dispatch and supervise worker tasks
- Master cluster controller: holds configuration database (persistent replica)
- Slave cluster controller: gets configuration from master cluster controllers
- Cluster is DOWN when all master cluster controllers are inaccessible

Typical system

Key Benefits

- Single system view
- Enforces decoupling of parts of O&M from actual traffic processing

Implementing a cluster

- Cluster->Host->Node->NodeData
- Cluster global parameters
- Subscription mechanisms for conf. changes
- Mnesia as configuration database on master cluster controllers
- Homebrewn configuration distribution to slave controllers (NOT using mnesia)
- (Worker) node supervision

Mnesia gotchas

- Disallow writes when all replicas not accessible
- Use timeout on table load and force load

... BUT ...

- TCP based distribution
- Network partitioning

Network parameters

- Align TCP retransmission intervals w/ Erlang heartbeats
- Aling TCP and IP rerouting parameters

Typical system II: Dual backplane

Erlang multi-homing problem

Multi-home Erlang w/ TCP

- Add an alias interface to loopback i/f
- Patch tcp distribution to bind to alias

- Publish alias interface on (all wanted) via real hw i/f's
  - Method 1: Static routes and gratious/proxy arp
  - Method 2: Use new (routing) protocol

ARP method

- Implement a utility to:
  - broadcast unsolicited ARP responses
  - respond to ARP requests
  for the alias i/f address
- Add static routes on all far end systems
- NOTE: all real i/f needs to be on same IP subnet

New routing protocol

- Broadcast (ethernet frames) what you have, including interface priority
- Let the far end select path based on what/when they receive
- Far end dynamically sets up host routes
- Use short retransmission intervals

Erlang multi-homing resolved ?

Summing up

- Erlang can support multihoming with some additional work
- By using loopback alias i/f, link failure becomes a routing problem (peer-peer association is kept intact)
- Solaris TCP/IP stack parameters are:
  - hard find (only in out-of-date app. notes)
  - hard to set "right"
  - host global
- A distribution mechanism with built-in support for multi-homing preferred

Erlang Distribution over SCTP

Per Bergqvist
per@synapse.se

Erlang User Conference 2002

The migration from Erlang to OTP: A case study of a heavy duty TCP/IP Client Server application.

Francesco Cesarini[1]
Mickaël Rémond[2]

September 27, 2001

Proceedings for the Seventh International Erlang User Conference, Stockholm, Sweden.

Abstract

A team of software engineers at IDEALX set out to build a proxy for a heavy duty TCP-IP application. They had Erlang experience from previous in house projects, but had never come in contact with the Open Telecom Platform's design principles. When an external Erlang/OTP consultant was taken in to review the code, he discovered a system written in pure Erlang. This paper describes reasons why development on OTP design principles were developed in the first place, and explains why they were bypassed in the prototype phase of this project. It concludes by describing how the migration of the Erlang prototype to an OTP product was achieved, looking at the advantages gained through such a migration. The intended readers are companies and individuals considering using Erlang in product development, but do not have access to in house Erlang expertise.

Introduction

IDEALX's business idea is to run in-house software development projects on behalf of customers using open source tools and components. One of these projects consisted in developing an instant messaging server where different protocols were interfaced towards the Jabber protocol, used by the ISP who commissioned the product. Due to the concurrent nature of the system, time scales, fault tolerance, high availability, and scalability, Erlang was chosen to develop the core of the system. The team had previous Erlang knowledge from in-house projects, and unanimously agreed that it was the best route to take. Inspired by the Sendmail[3] presentation at the Sixth Erlang User Conference, the project decided to take in an Erlang/OTP consultant. His task was to review the code, review the system architecture, and provide OTP training. When he started reviewing the code, he discovered a working prototype developed in pure Erlang. It did not use OTP design rules, principles and applications.

The team had been looking for something similar to OTP, but as they did not exactly know what they were looking for, they did not know where to look. Only when they took in external Erlang/OTP help were a lot of their questions answered. When the

[1] francesco@erlang-consulting.com, www.erlang-consulting.com
[2] mickael.remond@erlang-fr.org, www.erlang-fr.org
[3] Christenson et. Al. Sendmail Meets Erlang: Experiences using Erlang for email applications. Proceedings to the Sixth International Erlang User Conference, October 3, 2000.

migration of the system from Erlang to OTP took place, the size of the code was drastically reduced and extra functionality included in OTP was gained. We conclude the paper by sharing some of our ideas and needs to encourage other projects without experienced consultants or mentors to start using at least a subset of the OTP design principles from the start.

Product Requirements

A French internet service provider commissioned an Instant Messaging Proxy server based on the Jabber protocol[4]. Jabber is an XML-based open source initiative aiming at producing an open protocol alongside a gateway to existing competing instant messaging systems such as AOL, Microsoft, and Yahoo! At the time, Jabber could only handle fifty to one hundred simultaneous connections. The requirements from the ISP stated that the proxy needed to handle at least ten thousand simultaneous users. Scalability during run time was another requirement, as was the possibility of flexible software which at a later date could easily allow the addition of services such as advertisement messages to subsets of users.

Why OTP?

The first major product developed in Erlang was the Mobility Server. It was based on extensive prototypes started in early 1991. When working on the first prototype, the software engineers realised that similar problems had been solved differently in the various subsystems. Many models of client / server solutions had been implemented. Processes had different start and restart strategies, and error handling was not standardised.

A few people realised that a common approach, originally limited to start and recovery, was needed. In late 1993, alongside the Mobility Server project, development on BOS, the Basic Operating System, commenced. It introduced behaviours, restart strategies, release handling and code upgrade principles, alongside a set of design rules and guidelines that made processes exhibit a uniform behaviour towards the Erlang virtual machine. In 1995, the development of BOS and Erlang was combined, giving birth to the Open Telecom Platform. Prior to the first release of OTP, BOS was an obvious choice in all Erlang products developed within Ericsson.

Companies using Erlang/OTP have achieved many commercial successes. These successes are in part due to the in house knowledge and experience that has been built through many years of using the technology. Ericsson has been running Erlang/OTP projects since 1991, and has a department specialised in Erlang/OTP training and consulting. They have a direct communication and support channel to the Erlang/OTP maintainers. Nortel Networks employs many of the software engineers who originally invented Erlang, alongside members of the team that implemented the first release of OTP. Other companies, especially in Sweden, have people who have previous experience from Erlang/OTP based projects. They know that OTP should be used from the start. What happens when people come across Erlang through the web or through word of mouth and learn it from the documentation available on line and through the web, but are not made aware of the opportunities of OTP? What if they

---

[4] www.jabber.org and www.jabber.com

then go ahead and use it in a project without having the necessary mentorship? That is what happened at IDEALX.

IDEALX's prototype had many similarities to the early Erlang prototypes developed in the early 1990s. No coherent supervision structure existed. Some subsystems had restart strategies that were able to handle process crashes, others did not. Message passing was used in an uncontrolled fashion, making debugging very hard. In addition, the team had made design decisions based on incorrect assumptions over how the virtual machine and the kernel functioned. These decisions resulted in bottlenecks, inefficient constructs, and bad use of concurrency.

Why OTP was bypassed?

In December 1998, when Erlang was released as an Open Source development environment, Mickaël Rémond, the project leader and co-author of this paper, discovered the language. He found it very appealing because it offered many interesting features that were new to him.

He had not developed many concurrent applications because it meant having to add a magnitude of complexity to the code, drastically changing the programming paradigm he was used to. The languages he had used prior to Erlang were not made to handle concurrency, making the implementation of multithreaded systems appear unnatural and awkward.

What he found interesting about Erlang was that using the language changed and improved his programming style. Concurrency became straightforward and natural. He further realized that mapping processes to truly concurrent activities in the real world meant simplifying and improving the application design. It became easier to think in term of concurrent process than, for example, in terms of objects.

But, as he came from an object-oriented background, there was still something that he missed. Object orientation is not only used to design the system. It is also a way to organise and reuse large part of the source code. How was it possible to implement an efficient and well-structured program without object orientation?

OTP was certainly not the answer that came to mind. When looking at other functional languages, he discovered that some of them offer a mixed of orthogonal approaches. For example, Objective CAML is a functional object oriented language. Erlang had no such obvious approach.

This lead Mickaël Rémond to ask the classical newbie question in response to a mail sent to the Erlang mailing list. *"How do you apply object oriented programming techniques in Erlang?"* It seemed the natural question to ask, and only later did he realise it was the wrong one. Fortunately, Joe Armstrong came to the rescue. He gave him a very important clue in explaining that you can apply a typical Object Oriented Approach in Erlang with a common pattern. An example can be found in Chris Rathman's Shapes OO example[5]. Joe Armstrong however added that object orientation was not needed because Erlang has something more powerful called behaviours.

[5] www.angelfire.com/tx4/cus/shapes/erlang.html

"Something more powerful than object orientation?" It sounded too good to be true. It took some time before Mickaël Rémond really understood what Joe Armstrong meant. Object Orientation is not necessarily the best way to design and structure a piece of code. It might appear easier to learn and use, but when applied to larger system, the theory does not work. Users are forced to add "technical" objects that often result in complicated hierarchies. This makes the system architecture complex and sometimes affects the performance of the application.

It was however difficult for him to understand how to use the behaviour alternatives. The design guidelines are not very well promoted, and you have to search deep into the documentation to find examples. Such an approach is obvious when working with proper Erlang mentorship, but it certainly was not obvious for someone who had downloaded Erlang/OTP soon after it was released as open source.

OTP Deterrents

Behaviours are part of the design principles in the Open Telecom Platform. Saying that, you are facing your first deterrent to dig deeper into behaviours. Unless its name implies it, the Open Telecom Platform was not only made to develop telecom applications. It is a much more general tool. In fact, OTP is useful to organise any big enough Erlang application.

The second problem is that behaviour documentation is not where you would expect it would be, and is more a reference document than a step-by-step tutorial. To find the behaviour documentation, you have to look into "system principles" and "design principles", and in the correct section of the STDLIB documentation. The new organisation of the documentation in R8 is clearer, but still is not sufficient.

We also need to emphasise how behaviours relate to each other and how they can be used to make your application more robust. Maybe the tutorial should illustrate the way to migrate a pure Erlang program into an Erlang/OTP application. A step- by-step explanation, applied to a real application design. A widely available advanced Erlang programming book, focused on practical OTP aspects would also be welcome.

In addition, more Open Source code for applications designed using behaviours are needed. When searching on the net, most examples are written in pure Erlang. It was hard to find good examples of applications designed in OTP. In some cases, not even OTP applications in the Open Source release follow their own design principles.

IDEALX's Erlang team did not really know how they could build a project using OTP's design principles, and thus, it was not the obvious choice. They decided to work on a prototype in pure Erlang, and only later realised their mistake and asked an Erlang/OTP consultant to review their work.

The Migration

Francesco Cesarini was brought in to provide technical consulting and OTP training. His time on location with the team was limited to one week, and that had to include all OTP training. That was a solution he tried to persuade IDEALX not to go ahead

with. Often, when trying to cut training costs, the results are catastrophic. OTP design principles are complex, so people not experienced in Erlang often have a hard time grasping the subject. The group, however, consisted of a set of experienced programmers with a computer science background who had encountered many different programming paradigms. They had programmed extensively in Erlang, both on in house projects and on the Instant Messaging proxy server. By stripping the course material to the bare essentials, they were able to cover design patterns, programming rules and the new bit syntax in two and a half days, exercises included[6].

As soon as the team had a good understanding of how call-back functions worked and felt comfortable with the OTP design principles, it was possible to review their code in regards to style and efficiency. The review allowed the team to explain the architecture of their prototype, and their ideas and misconceptions on which they based their design decisions. A misconception on how the virtual machine worked resulted in a huge bottlenecks in the encoding and decoding processes. Believing that each process saved a private copy of the byte code, they wanted to reduce memory usage at the expense of message passing.

This was followed by putting together a new system architecture with emphasis on the process and application structures. This task only took a few hours because of the preparatory work during the review. The result was a simpler architecture, as the migration helped reduce the complexity while keeping the overall modularity of the system. Prior to the rewrite, the code was hard to follow. The use of the standard design patterns, namely generic servers, finite state machines and supervisors helped improve readability and structure.

The original prototype did not support distribution and load balancing. The OTP migration, however, facilitated designing those aspects in the system. The restart strategy came as a bonus, as it was achieved through the use of supervisors. Due to the time scale of the project, the team did not have time to introduce release handling and hot code upgrade. They did not feel confident with the upgrade, and for political reasons preferred stopping and restarting the system instead of risking a crash due to some error.

The overall performance[7] of the system was greatly improved after the rewrite. The unnecessary concurrency had been removed resulting in a drastic reduction in the overall exchange of data among processes. The code review took into consideration efficient constructs, so the use of binaries and pattern matching on the bit syntax, also played a big role. It was an easy task for the team to migrate from a pure Erlang application to one using OTP design principles. The rewrite of the prototype into a real product took 20 days, and resulted in the reduction of the code size by approximately 50%. The original prototype had taken 25 days to develop.

---

[6] It has to be pointed out that the training session was successful only because the participant's background. As the consultant had to be on location for a whole week, he agreed to the intensive course only because he knew there would have been more time for questions and exercises should the allocated time not have been enough.

[7] No performance measurements were made prior to the rewrite.

Conclusion

The team that worked on the project would today never consider writing an Erlang application without using OTP from the start. It took them only a couple of days to become productive, and they immediately saw both the need and the advantages. But in order to understand behaviours well, they had to first grasp how to design good Erlang applications. Attempting to learn OTP the week after having attended the Erlang course does not work.

Many advantages where achieved in the product after the migration. There where no more message management errors in that all message passing was handled through functional interfaces. Process supervision and restart strategies where standardised. Processes exhibited the same uniform behaviour and had basic event tracing and logging functionality. It was possible to generically control the whole system, distributing it across several nodes and processors. Even if not used in this release of the product, they had the existing tools for release handling, could easily integrate other OTP applications, and had a platform easily allowing the implementation of software upgrade during run time.

By creating BOS and later OTP, a gap that existed with pure Erlang was filled. It is a gap dealing with methodology, structuring of programs and re-usage of code. This was the gap that Mickaël Rémond was trying to fill when he came across Erlang in 1998 and asked about OO methodology.

Unfortunately, the Erlang book came out before OTP. The documentation is well hidden, and few examples of well-structured OTP applications are available. While waiting for new books, new documentation, and more open source examples, we hope that this paper will lead to a better understanding of the power of OTP to companies and people who do not have access to Erlang/OTP expertise but are considering Erlang.

An Erlang-based hierarchical distributed VoD System *

Juan J. Sánchez, José L. Freire, Miguel Barreiro
Víctor M. Gulías, Javier Mosquera

University of Coruña
Computer Science Department, LFCIA Lab
Campus de Elviña – 15071 A Coruña (SPAIN)
e-mail: {juanjo,freire,enano,gulias,mosky}@lfcia.org

Abstract

Video on Demand (VoD) is a service that enables users to request any multimedia content at any time, without being constrained by any pre-established scheduling. Current commercial solutions tend to have two main problems: lack of flexibility, and high cost for a large scale deployment.

The VoD server described in this paper is based in a hierarchical architecture, implemented using a functional programming language (Erlang) and built over a cluster-based architecture (Beowulf).

The described system can be adapted with great flexibility to the underlying network topology, and scaled in order to support a large and growing number of concurrent users, all with a low cost solution, and using commodity hardware whenever desired.

After an initial system design work, where the hierarchical structure was static and composed of three specialized levels (streaming, cache and massive storage), and after the first implementations and testing, the architecture has been evolved to an arbitrary number of levels, made of modules with a known interface, thus achieving high levels of adaptability and versatility.

Keywords: Functional Programming, Distributed Programming, Cluster, Linux, Erlang, Video on Demand.

1 Introduction

A Video on Demand server (VoD), is a system that provides video services, allowing users to request VOs (Video Objects) at any time, without pre-established time restrictions.

*Partially supported by FEDER TIC-1FD97-1759, Xunta de Galicia PGIDT99COM10502 and UDC 2000-5050252026

Services like movie-on-demand (MoD), distance learning tools, or interactive multimedia information services, offering personalized news based on user profiles, are just some multimedia applications examples that may use this kind of servers.

VoD systems must satisfy several key requirements, including:

- Huge storage capacity: in most applications, the system should be able to offer users a big number of multimedia objects, that should be stored in some way at the server. Even though storage capacity grows steadily for a given cost, so do user expectations.

- Support for a great amount of concurrent users: the system should be able to manage a great amount of VO requests. A single user may potentially perform several concurrent requests (for instance, multi-camera movie or conference archival). VOs may also be requested at high rate by automated tools to perform processing.

- High bandwidth: the high bandwidth usage characteristic of multimedia contents service, together with the need of serving objects to a large amount of users, makes of this one of the main requirements: the server must be able to provide a high total throughput.

- Predictable response time: when a user asks for a video object, the system should be able to give – performing statistical estimations that take the system state as input – an approximation of the waiting time. Besides, the server should try to minimize the waiting time for any user.

- Fault tolerance: with 24x7 expected uptime, there's a need for some kind of mechanisms, both hardware and software, to keep working at least in a degraded mode when some kind of fail happens. Users will expect the same kind of uptime as conventional TV (versus: they are used to some downtime on common internet-based services).

In addition to the traditional requirements for this kind of servers, we added three more, that put stricter conditions on the design decisions:

- Upwards and also downwards scalability: the system must be able to service a reduced number of users in a simple, scaled-down environment, and also capable of growing by increasing the used resources, in order to support a very large amount of concurrent users. It should be usable on very economical, commodity hardware as well as on high-end server farms and large SMP systems.

- Adaptability: a system able of adapting to the underlying topology, making an efficient use of the available network bandwidth. The topology of current DOCSIS cable and ADSL networks results in each section of the customer - provider link offering vastly different characteristics, thus there is a pressing need to optimize network resource usage to avoid saturating the slowest paths.

- Low cost, taking into account the total cost of deploying, managing and operating the system.

2 State of the Art

There are currently several video streaming server solutions available from RealNetworks, Microsoft, Apple, Cisco, Philips, IBM, Oracle, Kasenna and others.

While most of them are well suited for low volume video streaming on the Internet and some for corporate video services on a LAN, few try to address the difficulties in serving high volumes of multimedia contents on a large, heterogeneous network.

In general, the detailed analysis of this products, gives the conclusion that most of them represent very expensive, closed, non-scalable and non-adaptable solutions. They however tend to be turnkey solutions, ready to plug and easy to run on a small deployment.

A more detailed study of some of these solutions can be found at [10].

3 VoDKA: a hierarchical distributed functional VoD server

In order to cope with the previous requirements, an innovative solution, based in a distributed and hierarchical storage system [3], built over Linux clusters composed by commodity hardware [2], is proposed: VoDKA *(Video on Demand Kernel Architecture)*.

The hierarchical architecture allows the division of the system functionality among the different levels of the hierarchy, giving a bigger specialization, which makes viable the satisfaction of the "a priori" incompatible requirements. For example, the complex relation between high capacity, low cost, and a very short response time, finds a solution in the layering that is described in detail in section 3.1.

The conclusions obtained after the system implementation with this first proposal, have produced an architectural design refinement, resulting in a more general solution, whose main features are detailed in section 4.

In the analysis and design stages of the system development, design patterns [7] with their adaptations to the programming language are used, and *behaviours* [6] (which represent the equivalent to the design patterns but at a lower abstraction level).

The programming language in which most development has been made is Erlang/OTP [1], which has ideal features for the implementation of monitoring, control and scheduling subsystems. In some of the I/O modules, whose performance is critical for an optimal performance of the system, C has been used instead, after an initial phase of Erlang/OTP prototypes.

In every moment, emphasis has been made in reuse, by using OpenSource tools (Linux, Erlang/OTP), or through the adoption of certain subsystems from open solutions for special functions inside the system, like protocol adaptation or working with special file formats (Quicktime), and subsystems independization (OpenMonet [8], Inets `mod_xsl` [9]). Additionally, the use of OpenSource tools has allowed for easy operating system and hardware neutrality.

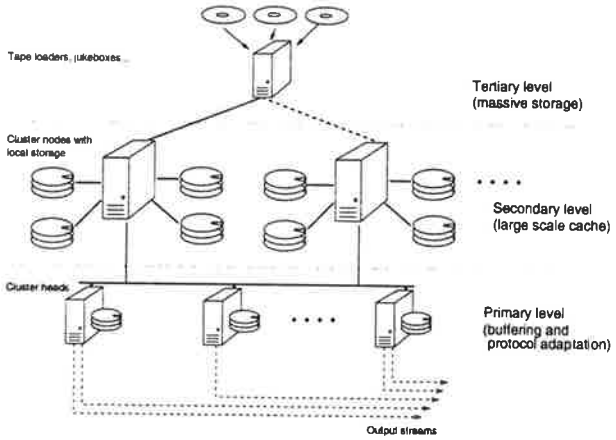


Figure 1: Initial hierarchical structure

3.1 Initial design proposal

The hierarchy in the initial design proposal (figure 1) was composed of three specialized levels, described bottom-up in the following paragraphs:

- Tertiary Level (massive storage level): massive storage level has different requirements in terms of response time, compared with the ones in the higher levels. Its main goal is to store all the available video objects in the VoD server with tape chargers, disk arrays, or any other massive storage system.

  The server global performance depends on this level in terms of load time, latency, throughput, etc. Even though it's interesting to optimize these quantitative parameters, the higher level can alleviate their weights in the global performance measurements, because the secondary level acts as a cache for the tertiary one.

- Secondary Level (cache level): it's composed of a set of nodes, each of them with enough capacity for storing at least a complete video object. The object, read from the tertiary level, is stored temporally in some node before being striped in the primary one. An appropriate scheduling policy should decide which are the videos that should be maintained in the cache, and for how long, in order to satisfy the future user requests and limit accesses to the massive storage level, if possible.

- Primary Level (streaming level): it's composed of a set of nodes in charge of protocol adaptation and sending the final video stream in the right format to the client. The interest in putting the sufficient number of nodes in this level is due to performance as well as to the need of quick reaction after a fail in any of the nodes, sending the tasks dynamically to another node, limiting in this way the losses in the quality of service. This level has important requirements both in bandwidth and in response time.

3.2 Key technologies

Besides the innovative architecture presented, two of the most important and differentiating features of the proposed solution are the use of a functional language and the adaptation to a cluster-based architecture.

3.2.1 Erlang/OTP

The used language, Erlang, has been designed and used at Ericsson for programming distributed control systems. The combination of the functional paradigm and parallel computing gives a declarative language, without side effects, and with a high level of expressiveness, abstraction and ease of prototyping.

Erlang is specially suitable for distributed, fault tolerant, soft real-time systems. It is a language based in asynchronous message passing, transparent transference of values, and higher order communications, that has the capacity of supporting a high number of concurrent processes.

The language is suited for the development of distributed systems, permits the transparent location of processes in different nodes. It also includes primitives for the support of fault tolerance and provides facilities for the replacement of code without having to stop the system.

The proposed solution also uses extensively the libraries and distributed design patterns of the Open Telecom Platform (OTP), including generic servers, supervision mechanisms, a distributed database (Mnesia) with location transparency, fragmentation, replication, and integration with the language, and a lot of useful integration libraries: SNMP, ASN.1, C Interface, Corba, Java, HTTP.

SNMP, the Inets HTTP server, the SASL support libraries, the EVA and MESH alarm and measurement handling applications, and Mnesia are used by the monitoring subsystem. The C interface, the TCP and UDP libraries and others are extensively used in the I/O and streaming layer.

There are also additional modules in development (LDAP administrative interface, user application gateway) that make extensive use of ASN.1, and the Java interface, among others.

Erlang/OTP not only provides many useful libraries and applications; there is also a rather homogeneous philosophy underlying all of OTP, so application design and interfaces is naturally coherent. A high degree of reusability and high programmer efficiency are also encouraged and made possible.

All the cited features makes of Erlang a very well suited language for the development of the proposed system.

3.2.2 The Linux Cluster

The use of Beowulf[1] clusters in the proposed solution proposed in this paper (Linux-based low cost distributed system) has been another of the keys that make proposed server an innovative architecture. The distributed memory architecture complements itself perfectly with the message passing philosophy of the chosen language. While the VoD system can run on other architectures as well (SPARC/Solaris, PowerPC/AIX), a linux cluster provides an ideal environment.

[1] Note the somewhat liberal use of the term *Beowulf*; in strict sense, the way we use Linux clusters does not make them Beowulf-class

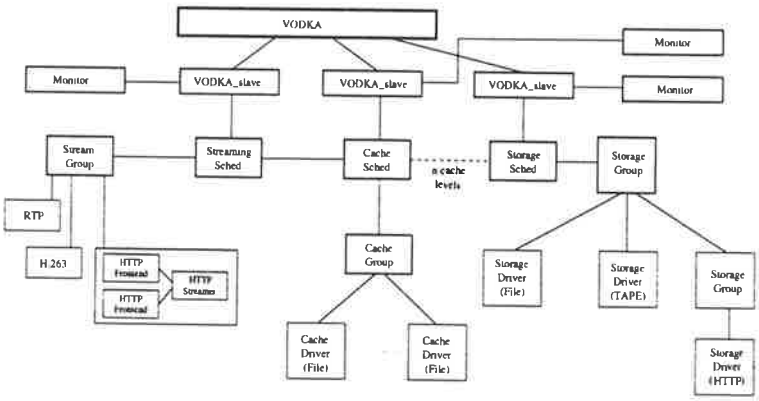


Figure 2: VoDKA Server with n levels

Some of the advantages derived of the utilization of this technology are:

- The wide experience in the OpenSource community and former work with high speed networks, distributed systems and clustering over Linux.
- Source code availability: modification of any part of the software for adapting, correcting, locating problems or instrumentation.
- Homogeneous license (GPL): easy legal treatment (versus, e.g., current MPEG4 situation, where each component has a different license, and their interaction is sometimes annoying and even contradictory).
- Compatibility: code developed with Linux can be ported without problems to Solaris, AIX, Tru64, IRIX, etc. Respect of standards (POSIX 1003.*, SVID, 4.xBSD). Progressive commitment of commercial Unix vendors to further interoperability (SGI and IBM embracing Linux, AIX 5/L providing Linux interfaces).
- Good performance
- Wide availability of development tools
- Support for different hardware platforms (x86, Alpha, SPARC, ARM, S/390 (zSeries), IA64, SH3, MIPS, Power...)

4 Design refinement

The initial design ideas had to be modified in order to satisfy the needs of a production system. In particular, a design generalization was needed, because the three level fixed hierarchical architecture (streaming, cache and massive storage) can be too complex for very small installations, and too rigid for complex network topologies like the ring-based backbone that interconnects some metropolitan cable networks.

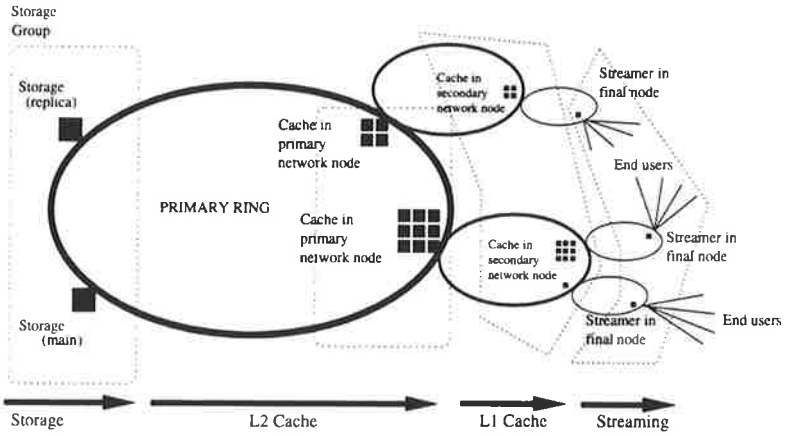


Figure 3: Configuration of the server over a complex network topology

This re-definition of the hierarchical architecture gives a new one, divided in a variable number of specialized levels, all of them sharing the same standard interface (figure 2).

This way, for example, the cache level can be absent in a given installation, or can be augmented producing a complex multilayer cache adapted to the underlying topology (figure 3) or configuring different massive storage levels physically distributed. The usual setup for a DOCSIS based cable network provides a restricted, if sufficient under normal circumstances, bandwidth from the customer to the cable headend, but plenty of available bandwidth through high speed SDH fiber rings among cable headends and the provider central switching offices. In some cases there are additional switching levels to the customer. So, in order to accomodate this situation a series of distributed storage and cache servers are deployed throughout the provider network, optimizing link characteristics.

Even, a server could be used as a storage subsystem of a different server, giving a VoD meta-server.

Besides the flexibility related to the physical distribution of the system, its necessary the system to be able to, inside each of the levels and among different ones, interact using heterogeneous storage and transference protocols. This flexibilization is based in identifying the communication patterns between the different levels, factoring them out as functional abstractions (*pipes*, for transference among the nodes) parametrized by functional closures which encapsulate the particular protocols and are established by schedulers that collaborate doing the tasks related to some work. This generalization in each level controllers ensures the utilization of generic monitorization mechanisms and reflective programming for discovering the features of each level.

It is very important to maintain the system independence of individual digital multimedia content distribution formats, currently in constant evolution, to the front-end of the VoD system, where the content distribution is actually done (streamer), providing different procotols adaptation: HTTP, RTP, etc.

The first logic layering in the VoD server is the one differentiating the video

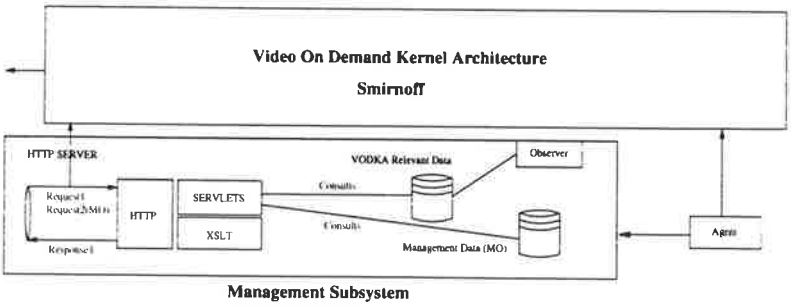


Figure 4: Separation between the VoD server and the management web application

server kernel implemented mainly with distributed functional technology and built over Beowulf clusters, and the different application servers that, using the video distribution capacities offered by the VoD server, give to the end user final services (figure 4).

The applications, developed using conventional technologies, interact with the video server redirecting the VO requests with the adequate parameters; besides, applications use the information obtained by the monitoring subsystem about the state of the system at any moment. Typical applications are cinema and television on demand, distance learning, e-commerce, interactive news, etc.

Figure 5 shows an example of how the media object flow among storage levels works until the actual streaming of the object in a simplified system configuration where the cache has been removed. In this case, the client request is received by an HTTP front-end, which defines the adaptation protocol required for a distribution of type *progressive download* over HTTP of a multimedia object. The front-end interacts with its scheduler (*Streaming Sched*) through the group in which is integrated (*Sched Group*), and decides the way in which the video stream is actually going to be distributed (`DD1`, in this example, instantiated to an HTTP adaptation in a port negotiated with the client). The streaming level scheduler propagates the media object request of the to its successor in the responsability chain [7], incorporating the protocol that the storage should use for the transference (`DD2`, in this example a TCP/IP communication). The successor, the storage level scheduler (*Storage Sched*), propagates the request towards an storage multiplexor (*Storage Group*), connected to different storage systems: a mounted file system (*File Storage Driver*), a tape robot (*Tape Storage Driver*). The scheduler mission is to decide which source is going to be used for obtaining the media object (in the example the *File Storage Driver*, building a *pipe* that connects that data source with the transference protocol suggested by the streaming scheduler, that creates a new *pipe* for taking the storage transference and sending it using the destination suggested by the HTTP adapter.

In addition to the improvements described, the second prototype also introduced reflective generic servers and *Vodka Explorer*, a GUI tool for navigating the logical server topology and reading or changing its state.

During the I/O layer rewrite, it was noted that VoDKA performance bottlenecks were often not CPU, but purely I/O bound, thus it was decided that

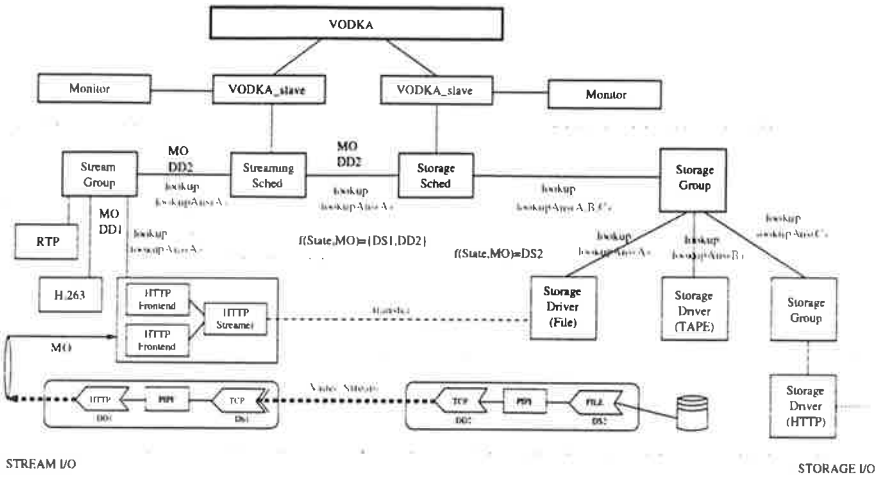


Figure 5: Example of transference in a system configuration without cache

rewriting these modules in C was not worth the effort: Erlang was fast enough, given proper care was taken.

4.1 Example VoD deployment

As an example of the VoDka server configuration flexibility, figure 6 shows a schematic of the responsabilities distribution among nodes in our cluster *Borg*.

- The massive storage is in the node `borg25`, that also has an associated scheduler with two storage controllers (CD unit and tape robot with 0.5 TB of capacity).
- The storage scheduler is the successor of the cache scheduler, that is running in the node `borg24` in the responsability chain. The cache scheduler uses as cache the aggregated bandwidth of the local cache controllers of the nodes `borg1`...`borg23`.
- The node `borg24` itself hosts an streaming scheduler whose successor is the cache scheduler, supporting a progressive download HTTP adapter, that gives video service to the lab using the department switched 10Mbps network.
- The server `covas` hosts a cache scheduler, whose successor is the `borg24` cache scheduler (two cache levels), and a local cache controller. Besides, the server contains an streaming scheduler fed by the cache scheduler, and supporting an progressive download HTTP adapter, using the ATM adapter that is directly connected to the university backbone.
- `borg0`, the *Borg* cluster front-end, is used for the system monitorization.
- One of the nodes is a SPARCstation running Linux, while the rest of the nodes are x86 machines also running Linux and covas is a Sun UltraEnterprise 3000 running Solaris, demonstrating system portability.

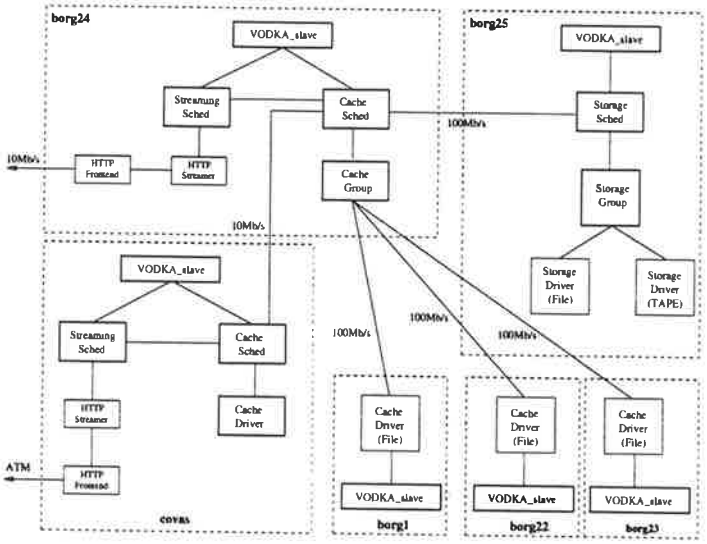


Figure 6: *Borg* configuration in the University Campus

5 Conclusions

A video on demand server that meets the basic requirements of such a system has been developed: great storage capacity, high bandwidth, predictable response time, large number of concurrent users and fault tolerance.

Evolving this system, in a second stage, the solution meets now other important features: scalability in both directions, adaptability to different network complexities and topologies, and low cost. This has been possible in part by using key technologies like the Erlang programming language, design patterns knowledge, and Beowulf clusters.

Currently, the system is being refined, paying special attention to the scheduling subsystem, and the addition of new modules for the support of different formats and storage protocols and distribution of multimedia objects. An important amount of work is also being done in the implementation of external applications, that are going to interact with the end user and the server.

References

[1] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang.* Second Edition, Prentice-Hall. 1996.

[2] M. Barreiro, V. M. Gulías, Cluster setup and its administration. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, Vol. I. Prentice Hall, 1999.

[3] S. G. Chan and F. Tobagi Hierarchical storage systems for interactive Video-on-demand Technical Report, Stanford University, Computer Systems Laboratory, Number CSL-TR-97-723, p. 84. 1997

[4] Cisco Systems, Inc. *A Distributed Video Server Architecture for Flexible Enterprise-Wide Video Delivery* en http://www.cisco.com/warp/public/cc/pd/mxsv/iptv3400/tech/dvsa_wp.htm, White Paper, 2000.

[5] D. Du, J. Hsieh, J. Liu, *Building Video-on-Demand servers Using Shared-Memory Multiprocessors.* Distributed Multimedia Research Center and Computer Science Department, University of Minnesota, and Ronald J. Vetter, Computer Science Department, North Dakota State University. 1996.

[6] U. Ekström, *Design Patterns for simulation in ERLANG/OTP.* Master Thesis, Uppsala University, Sweden, 2000

[7] E. Gamma, R. Helm, R. Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[8] LFCIA, OpenMonet Project. http://sourceforge.net/projects/openmonet

[9] LFCIA, Erlatron Project. http://sourceforge.net/projects/modxsl

[10] J.J. Sánchez, V.M. Gulías, A. Valderruten, J. Mosquera. *State of the Art and Design of VOD Systems.* Proceedings of the *International Conference on Information Systems Analysis, SCI'00-ISAS'00.* ISBN 980-07-6694-4, Orlando, USA, July 2000.

Erlang in the Corelatus MTP2 Signalling Gateway

Matthias Läng: Corelatus AB
matthias@corelatus.se

September 2001

1 Introduction

Corelatus is a startup founded by Thomas Lange, Matthias Läng and Ulf Svarte Bagge approximately one year ago. We build hardware for use in telecommunications networks.

This paper looks at our experiences using Erlang in a stand-alone system which terminates SS7 MTP2 and passes packets to a general-purpose server for further processing. Erlang was used throughout the project: from when it was just an idea to when it became a certified, tested, approved and mass-produced product suitable for placement in the core of an operator's SS7 network.

2 The Concept

Our concept was to build a stand-alone box to handle the parts of a telecommunications system which are difficult or expensive to implement using general-purpose computers. This would enable software developers to build complete systems by combining our hardware with ordinary server hardware. The most important features are:

- Carrier-grade hardware: a rack mountable chassis, dual 48V DC power inputs and no moving parts.
- Approved for use in telecommunication centers. This means passing lightning tests, surge tests, electrical and radio interference tests and so on.
- Support for aspects of media and signalling which are too timing-sensitive to be easily handled by a general-purpose server.
- Narrow interfaces. Electrically, this means *not* having a shared bus with other hardware. Physically, it means being in a separate box. Logically, it means the programming interface runs over TCP (on Ethernet). The narrow interfaces give us complete control of, and responsibility for, everything inside the box. If the box crashes, there is no doubt where the problem lies.

Our customers use our hardware, called the GTH, inside operators' core SS7 networks. Today's SS7 networks are mostly tied together with 2Mbit/s E1 or T1 links carrying SS7 signalling.

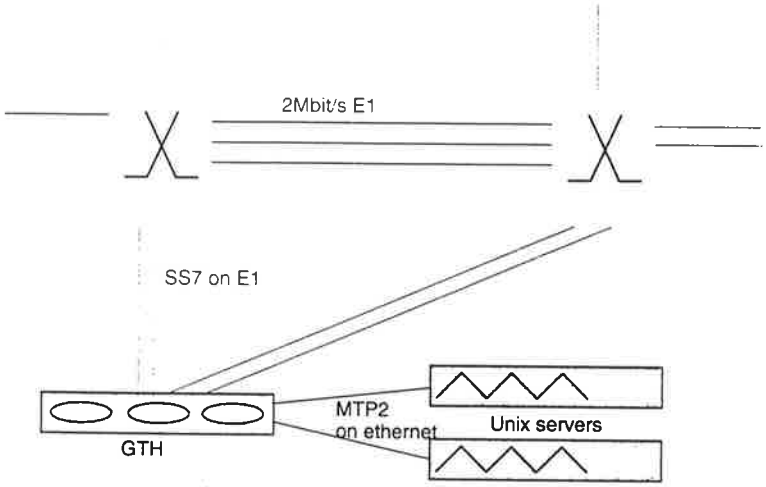


Figure 1: Unix servers connected to the telephony network via GTH

The parts of SS7 which have stringent timing constraints are MTP1 and MTP2. For this application, our hardware provides MTP1 and MTP2 on the E1 links and transfers packets over IP to Unix (or NT) servers for higher-layer SS7 processing. In IETF nomenclature, the Corelatus GTH acts as a *signalling gateway*.

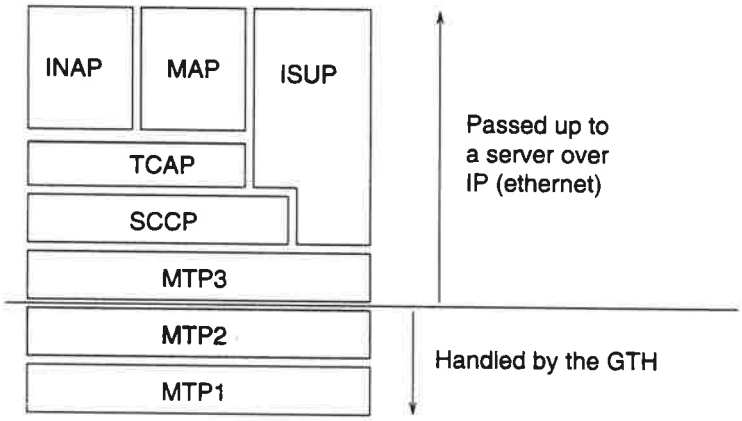


Figure 2: SS7 stack

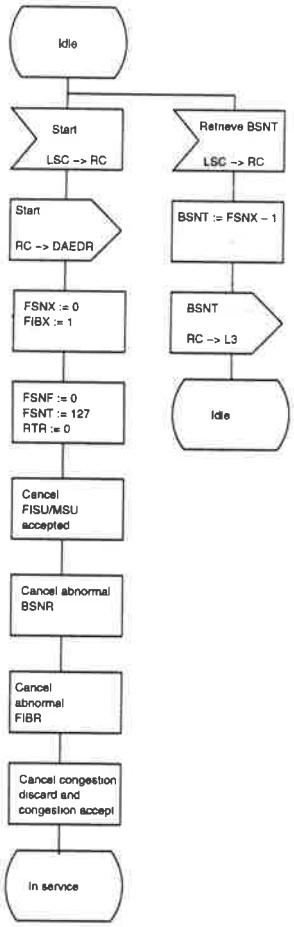
3 MTP2

MTP2 provides three services: it moves packets from point-to point, it removes incorrect packets and it handles retransmission. The official definition of MTP2 is the ITU standard Q.703. When we started, I had never looked at MTP2 before. I decided to implement MTP2 in Erlang as a pleasant way of gaining experience.

3.1 MTP2 in Erlang

The ITU specification is 89 pages long, of which 58 are SDL[4] diagrams. Our Erlang implementation is 2004 lines long, or about 33 pages.

We wrote the Erlang code by directly translating the state machines in the SDL part of the standard to Erlang `gen_fsm` state machines. SDL and Erlang are a good fit for each other: most of the SDL in MTP2 consists of ten concurrently executing state machines sending each other messages (*signals* in SDL-parlance). Using an automatic gen_fsm graphing tool [5] produces diagrams which are strikingly similar to the standard. The resulting code is also easy to compare to the standard:



Q.703, p.59

```
idle(start, StateData) ->
 ok = gen_fsm:send_event(
  StateData#state.daedr, start),
 {next_state, in_service, StateData#state{
   fsnx = 0,
   fibx = 1,
   fsnf = 0,
   fsnt = 127,
   rtr = false,
   fisu_msu_accepted = false,
   abnormal_bsnr = false,
   abnormal_fibr = false,
   congestion_discard = false,
   congestion_accept = false
  }
 };

idle(retrieve_bsnt, StateData) ->
 BSNT = sevenbit_dec(StateData#state.fsnx),
 ok = gen_fsm:send_event(StateData#state.l3,
   {bsnt, BSNT}),
 {next_state, idle, StateData};
```

While testing the Erlang implementation against itself, we found two errors in the ITU specification. [1]

[1] The errors are corrected in an ITU errata.

3.2 Bit-stuffing and Performance

Bit stuffing is central to MTP2 (and related protocols like HDLC). It solves the problem of how to delimit messages in a communications channel which is nothing more than a stream of bits. A special sequence of bits delimits messages, and messages are modified before transmission to ensure that they do not contain the delimiter. Sending the message 0000 1011 1111 1000 looks like this at the transmitter:

1. Send the message delimiter: 0111 1110
2. Modify the message in a way which guarantees that the delimiter never appears in the message. We do this by inserting an extra zero-bit whenever we just sent five one-bits in a row. The message is now 0000 1011 1101 1100 0
3. Send the message delimiter.

The receiver does the reverse, including removing the extra zeros. If we could write the transmitter using the Erlang binary syntax, it would look something like:

```
encode_message(Bin) ->
   Flag = 2#01111110,
   <<Flag:8, stuff_message(Bin), Flag:8>>.

stuff_message(<<2#11111:5, Rest/binary>>) ->
   <<2#111110:6, stuff_message(Rest)>>;
stuff_message(<<A:1, Rest/binary>>) ->
   <<A:1, stuff_message(Rest)/binary>>.
```

The above is not legal Erlang; Erlang does not allow binaries sized anything other than a multiple of 8 bits. We solved this by exploding all binaries into lists of bits. Running on real SS7 signalling data, our reference implementation can handle one or two timeslots of MTP2 (one timeslot is 64kbit/s). The performance is not important, the objectives were to learn and to generate test bitstreams to verify our high-performance MTP2 implementation.

4 Disguising Distributed Erlang

Most of the current and future applications for our hardware use the GTH as part of a larger system, usually with one or more Unix servers controlling the GTH. Since our control system is written in Erlang, this would be a perfect application for distributed Erlang, apart from one catch: Erlang is not popular with non-Erlang users. We examined alternatives, including CORBA, ASN.1 and Megaco and also rejected these for one reason or another. Eventually we were inspired by a pair of comments we stumbled across:

XML is little more than a notation for trees and for tree grammars, a verbose variant of Lisp S-expressions coupled with a poor man's BNF (Phil Wadler) [7]

Any Erlang message can be encoded in XML[3]. Better still, the message stays human-readable while simultaneously looking a lot less obviously like Erlang.

if you think TCP guarantees delivery, which most people probably do, then so does Erlang [when passing messages] (Per Hedeland)[6]

If we assume the converse is also true, then XML and TCP can be combined to produce a protocol with the same functionality as message passing in Erlang. The protocol can be described by a machine-readable BNF-like language (an XML DTD). The user will be comfortably oblivious to the presence of Erlang in their system.

4.1 A General XML Encoding

Using `jinterface` as a guide, we can produce a list of what needs to be encoded

Data type	Representation (example)
Atom	<atom name="abcd"/>
Binary	<binary>base64-encoded data</binary>
Numbers	<number value="12"/>
Exit signal	see section 4.3
List	<list><atom name="abcd"/><binary>B63MGUP</binary></list>
Pid	<pid a="0" b="12" c="19"/>
Port	<port id="1234"/>
Ref	<ref id="12ABC67"/>
Tuple	<tuple><atom name="abcd"/><binary>128YTE63MG8</binary></tuple>

Binaries are encoded as base64, references are converted to binaries before being encoded. An example: our system can play lists of audio messages to a subscriber, perhaps "you have 56 dollars and 22 cents left in your account". In Erlang we represent this as

```
Pid = {start, {player, {pcm, 1, 3},
          [youhave, fiftysix, dollars, and,
           twentytwo, cents, leftinyouraccount]}}
```

Using the above encoding to carry the same information:

```
<tuple>
   <atom name="start"/>
```

```
        <tuple>
           <atom name="player"/>
           <tuple><atom name="pcm"/><number value="1"/>
              <number value="3"/>
           </tuple>
           <list><atom name="youhave"/>
              <atom name="fiftysix"/>...</list>
        </tuple>
     </tuple>
```

This works, but it is only slightly prettier than an obfuscated PERL contest. It also fails to meet our goal of not explicitly exposing Erlang concepts in the API.

4.2 A Problem-specific Encoding

Our API doesn't have that many messages, and the messages are not arbitrary terms. They are all tuples which start with an atom. There are only eleven top-level message types. If we defined a separate XML encoding for each message we could cut the size of the messages, remove the references to tuples and lists and write a more precise DTD to specify the interface.

Example: to start a DTMF detector on the GTH, we use the internal message

```
{start, {dmtf_detector, {pcm, 3, 9}, Message_dest_pid}}
```

A single-purpose encoding for this message is

```
<start>
   <dtmf_detector dest="apic13">
      <pcm_source span="3" timeslot="9">
   </dtmf_detector>
</start>
```

Similarly, the example for message playback becomes

```
<start><player>
       <clip name="youhave"/><clip name="fiftysix"/>...
       <pcm_sink span="3" timeslot="9"/>
</player></start>
```

As a final touch, we replaced the words *start* and *stop* with *new* and *delete* to add to the illusion of object orientation. The DTDs[2] on our website define our API's grammar.

4.3 Processes and Linking

The XML encoding takes care of formatting the data we want to send back and forth. But there is more to distributed Erlang than just sending messages. There is also RPC, a global namespace, node monitoring and process linking. Supporting all of these seemed unnecessary, so instead of modeling the external control system as an Erlang node, we modeled it like an Erlang port.

An Erlang port is linked to the process which created it. An Erlang port can send and receive messages. If the port dies, the controlling process is notified.

In our API, the external system is connected via a TCP socket. If the controlling process dies, we close the TCP socket. If the socket closes, we kill the controlling process. If any of the Erlang API processes die, all remote processes are notified. This poor man's emulation of Erlang's linking and message passing allows the system to recover from a client process death:

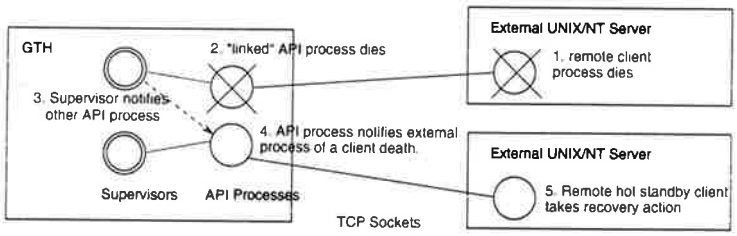


Figure 3: Recovery on a second server due to a first server's death

5 Conclusions

Erlang is a nice protocol specification language. Our reference implementation of MTP2 in Erlang is more compact than the SDL + plain English MTP2 specification. By being able to execute our "specification" we had a ready-made test system for our high-performance implementation and we found two errors in the original ITU specification.

Distributed Erlang is a proven way of building robust distributed applications. By encoding Erlang terms in XML and connecting to the non-Erlang node using TCP, it is possible to create a human-readable, language-neutral protocol. By duplicating a proven approach we know it is possible to build robust systems using this approach.

References

[1] *The GTH API Documentation*
http://www.corelatus.com/gth/api

[2] *The API DTDs*
http://www.corelatus.com/gth/api/gth_in.dtd
http://www.corelatus.com/gth/api/gth_out.dtd

[3] *Extensible Markup Language (XML) 1.0*
W3C Recommendation
http://www.w3.org/TR/2000/REC-xml-20001006

[4] *Specification and Description Language (SDL) Forum*
http://www.sdl-forum.org

[5] *gen_fsm Graphing Tool*
Vance Shipley from Motivity Telecom sent me this tool in private correspondence.

[6] *Post to Erlang-Questions Mailing List* 9. October 1999, archived at www.erlang.org

[7] *The Next 700 Markup Languages*
Philip Wadler. Invited Talk, Second Conference on Domain Specific Languages (DSL'99), Austin, Texas, October 1999.

Tools for Designing Web Based Interfaces for Erlang/OTP

Martin Gustafsson

Program

- Why Web Based User Interfaces.
- Best Desing of Web Based Tools, with Erlang/OTP.
- httpd the Webserver in Erlang/OTP.
- How to use WebTool.

Why Web Based User Interfaces

- Easy for the user.
- Fast development,
  - Easy to Learn.
  - Easy to Use.
- The tool can be used from the network.
- Easy to generate printable reports.

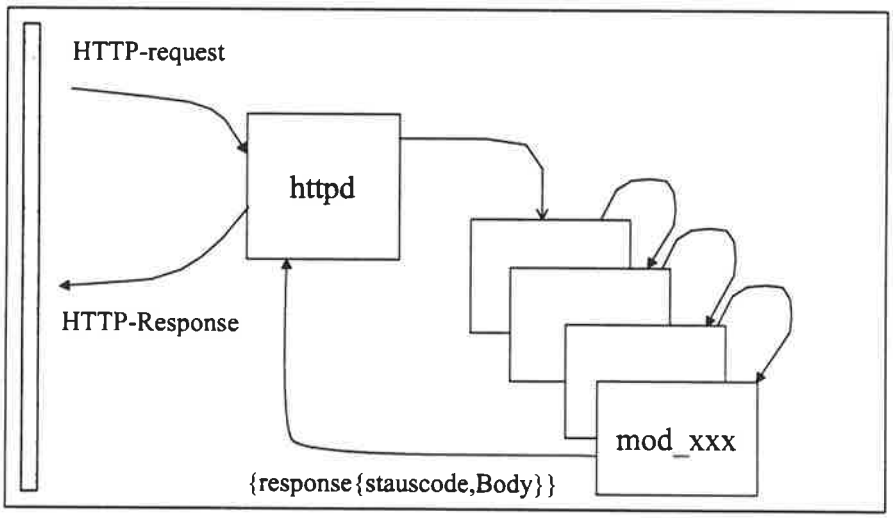
Problems with Web based User Interfaces

- Takes little more time to start than command-line tools.
- When security is a problem,
  - Examples:
    - When the functionality of the program can be used to krasch the Erlang node.
    - When the functionality handles valuable data.

httpd -The Webserver in OTP

- Support for Basic User Authentication.
- Support for SSL.
- Support for creation of Dynamic Web Pages.
- httpd is a Apache styled Webserver.

The functionality of httpd



Creation of dynamic Web Pages

- CGI
- Eval Scheme
  - Ex: http://server:port/EvalSchemAlias/mod/func(anyArg)
- Erl Scheme

  Ex: http://server:port/ErlSchemeAlias/mod/func
  The Webserver will return the result of calling
  mod:func(Env,Input).

WebTool

- WebTool is a framework for web based tools.
- WebTool configures and start the webserver.
- WebTool can start and stop the various web based tools.

Using WebTool

- Add the callback function for each tool to the file webtool-vsn/priv/config.file.
  - Ex: [{webcover,configData,[]},
    {webappmon,configData,[]}]
- Start WebTool.
  - Ex: webtool:start().
    webtool:start("/etc/var/webtool",standard_data).
- Point a browser to http://localhost:8888/

Callback function for WebTool

- WebTool use the callback function to receive configuration data for the tool.
- Configuration data is needed for:
  - Creating links from WebTool to the tool.
  - Configuration of the Webserver.
  - Data about how to start and stop the tool.

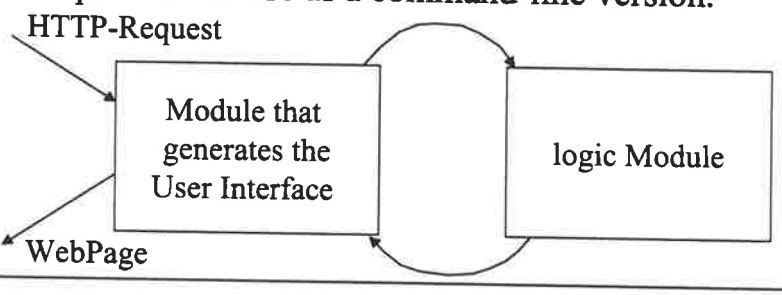
Developing tools to be used via WebTool

- The tools must export a callback function that returns the configuration data needed by WebTool.
  - Example:

```
configData()->
    Application=my_app
    RealPath=code:priv_dir(Application),
    {mytool,[{web_data{"URL","Linktext"}},
    {alias,{erl_alias,"VPath",["Module"]}},
    {alias,{"VirtualPath2",RealPath}},
    {start{{Mod,Func,Arg},{Mod,Func,Arg}}}]}.
```

Design principles for Web based Tools

- A N-layered solution has many advantages
  - Easier to update the code
  - If well designed the logic module might be possible to use as a command-line version.



3D graphics in Erlang

Jakob Cederlund - jakob@gnodata.se
Björn Gustavsson - bjorn@erix.ericsson.se
Dan Gudmundsson - dgud@erix.ericsson.se

3D graphics in erlang

- 3D graphics with hardware acceleration
- OpenGL - an open version of SGI's GL
  - Reasonable API
  - Open standard
  - Cross platform (Wintel, Linux, Sun, Mac)
  - Widespread support from hardware
- SDL - a library supplementing OpenGL with interaction, 2D graphics and more

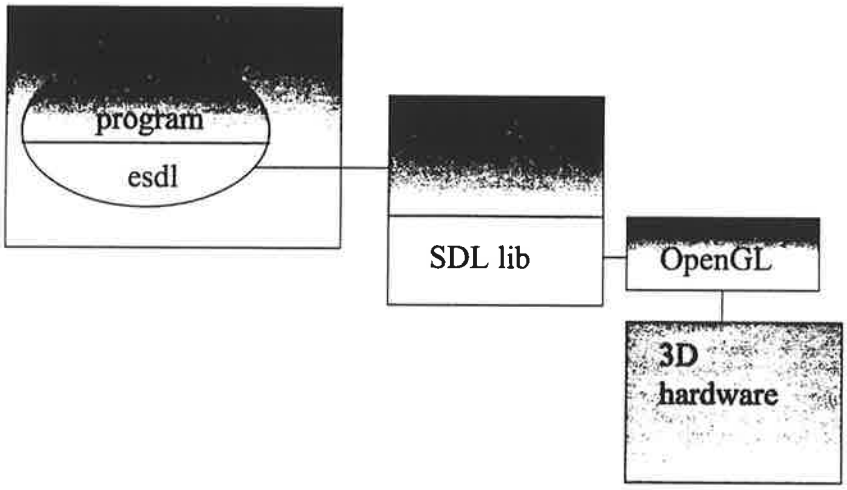
erlang and 3D graphics

- erlang high level language
- 3D graphics needs symbolic language
- Fast development
- Performance critical code mostly in 3D hardware and drivers

ESDL

- An erlang driver that interfaces with SDL
- Supports most OpenGL calls
- Fast
  - Optimized C driver
  - Uses binary syntax
- Available as download from contrib area
- Written by Dan Gudmundsson

ESDL



Wings - a 3D modeller in erlang

- Uses erlang as functional language
- Fast development
- Good performance
- Powerful
- Innovative user interface
- Inspired by Nendo
- Written by Björn Gustavsson
- Some inspiration from Jakob Cederlund

Wings

- Most of the erlang code not performance critical
- Some functions optimized
- Winged-edge structure for 3D objects
- Uses gb_trees and gb_sets (included in OTP R8)

Wings - features

- Powerful modelling
- Every operation applicable to several objects at the same time
- Imports and exports to different formats
  - Wavefront
  - 3D studio
- Easy to do easy things
  - As opposed to other 3D modellers

Wings - development

- Functional
  - Only one process
  - Process dictionary used in for GUI state
- Some things suprisingly easy to do
  - Multiple undo, with shared data
    - Only updates appriopriate parts of the model
  - Continues after crash, e.g. to save work
    - Just a catch, saves dump to file
- Library functions used
  - gb_sets and gb_trees (R8)
  - sofs - Sets OF Sets (R8)
  - digraph and digraph_utils

Wings - development cont.

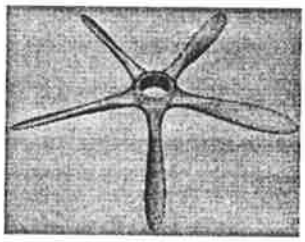
- Funs used a lot
  - special fold operations to iterate over model elements
- Optimization after implementation and testing
- Data structures hidden, used via functions
- Most user commands uses primitive commands
- Example inset
  - Extrude
  - Scale

Wings - examples

- Rendered in Bryce, composited in PhotoShop

Wings - examples cont.

- Modelled in only 5 minutes!
- Rendered afterwards in Bryce



Wings - performance

- Most things fast enough
- Optimized floating point in R8
- Some things take time
  - Smooth
  - Dissolve
    - Used as primitive by many commands
- erlang fast enough, even for interactive tasks

Wings - future

- Long to-do list
- Ongoing development, bugfixing and new features
- Future features
  - Plug-ins, both in erlang and C
  - More export formats, e.g. renderman
  - Cameras, multiple views
  - Materials, textures
- New platforms (MacOSX)

Wings - availability

- Open source
  - Full source on sourceforge
- Compiled versions
  - Currently requires otp, esdl and SDL
  - Stand-alone erlang version with installer will be available shortly
- Documentation and examples
  - Home page http://www.erlang.org/project/wings/

Development of a Verified Erlang Program for Resource Locking

Thomas Arts and Clara Benac Earle

Ericsson, Computer Science Laboratory
Box 1505, 125 25 Älvsjö, Sweden
E-mail: {thomas,clara}@cslab.ericsson.se

Abstract. We have designed a tool to simplify model checking of Erlang programs by translating Erlang into a process algebra with data, called μCRL. As a case-study for this tool we focused on a simplified locker implementation after the locker that is present in the control software of the AXD 301 switch. The translation algorithm has been developed to handle this production-like code. We use the tools accompanying μCRL to generate the transition systems from the specification generated by our tool. With the CÆSAR/ALDÉBARAN tool set, we verified properties for our case-study.

1 Introduction

Within Ericsson the functional programming language Erlang [1] is used for the development of concurrent/distributed safety critical software. Faced with the task of creating support for the development of formally verified Erlang programs, as a subtask we have built a tool to enable the use of model checking for such programs. The tool is aimed to be accessible for Erlang programmers without forcing them to learn an extra language (specific for the model checking tool that is used).

Using model checking for the formal verification of software is by now a well known field of research. Basically there are two branches, either one uses a specification language in combination with a model checker to obtain a correct specification that is used to write an implementation in a programming language, or one takes the program code as a starting point and abstracts from that into a model, which can be checked by a model checker. Either way, the implementation is not proved correct by these approaches, but when an error is encountered, this may indicate an error in the implementation. As such, the use of model checking can be seen as a very accurate debugging method.

For the first approach, one of the most successful of the many examples is the combination of the specification language Promela and model checker SPIN [14]. The attractive merit of Promela is that this language is so close to the implementation language C, that it becomes rather easy to derive the implementation from the specification in a direct, fault free way. In case one uses UML as specification language and Java or C as implementation language, one might need more effort (apart from the fact that model checking UML specifications is still an unsettled topic).

Also with respect to the second approach there are many examples, among which PathFinder [13] and Bandera [6] starting from Java code. There exists even an earlier attempt to use model checking on Erlang code by Huch [15]. Our approach could be added to this list, probably with the difference that we use the knowledge of the occurring design patterns used in the Erlang code to obtain smaller state spaces (cf. [2]). We follow a similar approach to the translation of Java into Promela, checked by SPIN [13]; however, we translate Erlang into μCRL [12] and model check by using CÆSAR/ALDÉBARAN [9]. Compared to Huch's approach we focus much more on the data part and do not abstract *case* statements by non-deterministic choices, but really check the data involved. For that reason we can check mutual exclusion and absence of deadlock for a small locker program that will be the leading example of this paper. If one abstracts from the data in this program in such a way that *case* statements are translated into non-deterministic choices, then mutual exclusion is no longer guaranteed and can hence not be shown.

One of the main goals of our approach is to be able to deal with Erlang code that is written according to the design principles as advocated within Ericsson. Our starting point was a distributed locker algorithm as is running in Ericsson's AXD 301 ATM switch [4]. We started re-designing this locker in such a way that formal verification guides the development. In this paper we illustrate our ideas with one of the first locker prototypes in this development process.

In Section 2 we describe the locker algorithm that we consider in this paper. We show in Section 3 how this locker is implemented in Erlang, using the generic server and supervision tree design principles.

The Erlang modules can automatically be translated into a μCRL specification and in Section 4 we describe our contribution in the form of this translation tool. Verification of the μCRL specification for the classical properties: no deadlock, mutual exclusion and no starvation, is described in Section 5. In the conclusion in Section 6 we discuss the merits and shortcomings of our approach and put it in context with respect to other approaches.

2 Designing the algorithm

The case-study we have at hand in this paper is a classical locker algorithm. Several processes want access to one or more resources from a given, finite set. A locker process is playing arbiter, responding the requests for access to resources in such a way that all clients eventually get their demanded access, but no two clients get access to the same resource at the same time. The client sends one message containing all resources that are requested, waits until access is granted, accesses the resource, gives the resource free and starts asking for other resources again.

Several fault situations are easy to imagine and these should guide us towards solutions for the most rudimentary problems. We describe the analysis of these situations as a pre-study for the actual implementation. However, with the tools we discuss later, one could find these results in an experimenting fashion: implement an idea in Erlang and obtain all possible runs of the program automatically.

Here we discuss the fault situations, using a special notation for scenarios. A scenario is a sequence of states of the locker process. A state of the locker contains a fixed set of resources and for every resource we have three 'fields': the name of the resource, the client that has access to the resource, and the list of clients that want to access the resource. As an example of this notation, we sketch a possible starvation situation. There are two resources, A and B, and three clients, 1, 2 and 3. The algorithm is such that if a demanded set of resources is available for a certain client, then this client gets access to those resources. Here, client 1 requests resource A, client 2 requests resource B, and thereafter client 3 requests both resources. Client 1 releases and requests resource A again, client 2 releases and requests B again. A continuous operation in this way causes client 3 to be waiting for ever to get access, i.e. client 3 is starving.

	A	B
access	1	
pending		
access	1	2
pending		
access	1	2
pending	3	3
access		2
pending	3	3
access	1	2
pending	3	3
access	1	
pending	3	3
access	1	2
pending	3	3

This scenario indicates that in general one has to pay a price for optimal resource usage: viz. a possibility for starvation. Clearly one does not want starvation in the program, but one still may accept it in the algorithm. If one has good evidence to believe that resources are not accessed very frequently, then the above situation might be very unlikely and one might choose to loose performance for client 3 in favor of a better over-all performance.

We, however, assume that the frequency of access to the resources can be rather high and that the different clients may have overlapping demands for resources[1]. Therefore, we need to decide upon a solution to this problem. We choose to use a 'first come, first serve' strategy. A resource is only available if there is no client

[1] If client 1, 2 and 3 would all ask for the same resources, this starvation problem would not occur.

waiting for it, i.e., both access field and list of pending processes for the demanded resource is empty.

	A	B	A	B
access	1			2
pending			3, 1	3
access	1	2	3	3
pending			1	
access	1	2	3	3
pending	3	3	1	2
access		2		
pending	3	3		

Thus, in this solution, a client could have to wait for its resources, even if all demanded resources are unused at the moment. Some optimizations are possible, for example to time-stamp the pending processes and give access to the resource if the first of the pending processes has not yet waited a certain amount of time. This is more involved and we do not consider this or other optimizations in this version.
The action upon a client requesting for a list of resources will be:

- Look whether all demanded resources are available. A resource is available if no other process is accessing it, and there are no processes pending for this resource.
- If all demanded resources are available, then the client is notified and is given access to all resources.
- If any of the demanded resources is unavailable, then for every demanded resource, the client is placed at then end of the list of pending processes.

The client is assumed to release all the previously demanded resources by only one release message; upon a release, the client is removed from all resources and a calculation is performed to see whether one of the other clients can get access to its demanded resources. Similar to the reasoning above, we cannot give access to just any client for which all demanded resources are available. Even for the one resource case it is clear that we need to take a 'first come, first serve' policy. Thus, with only one resource and several clients, we would give the client at the head of the pending list access to the resource. However, one could wonder what happens if there are two resources and both have one or more clients in their pending list.

	A	B
access	1	1
pending	2	3, 2

Here we need an algorithm to decide whether client 2 or 3 gets access to the resource after that client 1 releases. The possibilities we could think of boil down to the construction of one list of the pending processes where the first client in this

list for which all demanded resource are available gets access to these resources (i.e., is notified and removed from the pending lists and put into the access 'field'). Several ways of constructing this combined list are:

1. Merge all list and sort them on the client identifier. This means that the client with lowest identifier has highest priority. Hence, starvation is an obvious problem. When, in the above example, client 2 is given access and client 1 requests both resources again, then by the time client 2 releases, client 1 will be granted access. Repeatedly having 1 and 2 requesting access will cause 3 to starve.
2. Append all lists (and use a small optimization by a unique append, i.e. only appending the clients that are not yet present in the list). Clearly the same starvation problem as above occurs for this solution.
3. Construct a list that contains only those heads of the pending lists that do not occur in one of the tails of a pending list.
   The reason why this can work is that we have the clients always request all the resources at once. Hence, the clients are put in the pending list in a 'sorted' manner. A situation like

	A	B
access		
pending	2, 3	3, 2

   cannot occur in this setting, since either client 2 follows client 3 in all pending lists or vice versa. There might be clients in between, but the order cannot be reversed.
4. Add a time-stamp to any incoming request and save the client information with this time-stamp. The list is now obtained by appending and sorting the time-stamps.
   An equivalent approach is to separately store the list of requesting clients and use the order in which they requested as the priority order for giving access.

We have experimented with both version 3 and 4 and present version 3 here.

3 Locker Implementation in Erlang

The ideas sketched in the previous section are now to be implemented in Erlang. Clients and locker are implemented as Erlang processes that communicate with each other by message passing. The locker is implemented as a server, following one of the generic design patterns given in the Erlang distribution [8]. This *generic server* design pattern prescribes an implementation of the locker as a so called *callback module*. The actual loop that saves the state of the server and receives messages is implemented in a standard module and whenever a message arrives, the appropriate function in the callback module is executed. These callback functions return a new state and a possible reply message, which is by the standard module part send to the caller. In this way, the generic server principle implements synchronous communication on top of Erlang's asynchronous communication primitives. For a detailed operational semantics we refer to [2].

The flow of control between clients and locker should be as follows:

- a client *requests* the locker an exclusive lock on several resources,
- if all requested resources are available, the locker gives an *ok* to the client,
- when the client has performed the necessary operations on the resources, it notifies the locker by a *release* of the locks.

The locker schedules the clients on a first-come first-served basis as explained in the previous section. Note, however, that this scheduling is relative to the resource. A client that requests a resource that is taken, may be served later than the client requesting another, free resource, after it.

The client is programmed as a very simple process, just using the generic server call principle to communicate with the locker. The `gen_server:call` function hides synchronized communication with the server. The second argument of this function contains the message that is sent to the server, which calls the `handle_call` function in the callback module. The client is suspended until `handle_call` returns a reply value, which is passed by the server to be the return value of the `gen_server:call`. For this particular client we are not interested in the actual returned value and just use it for synchronization. The `spawn_link` function is used to create a new process, in this case running the `loop` function with the arguments `Locker` and `Resources`.

```
-module(client).

start(Locker,Resources) ->
    {ok,spawn_link(client,loop,[Locker,Resources])}.

loop(Locker,Resources) ->
    gen_server:call(Locker,{request,Resources}),
    critical_section,
    gen_server:call(Locker,release),
    loop(Locker,Resources).
```

The atom `critical_section` between the two synchronous calls for request and release implements the so called critical section. In a real implementation some critical code should be placed in this critical section, but we abstract from that.

To implement the locks we use a record with the following fields:

- resource: the name of the resource,
- exclusive: the client which is using the resource,
- pending: a list of clients that want to access the resource.

The Erlang program for the locker process is given by a generic server callback module that accepts the messages `{request,Resources}` and `release`.

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

-record(lock,{resource, exclusive = none, pending = []}).

init(Resources) ->
```

```
    {ok,map(fun(Resource) ->
                #lock{resource = Resource}
            end, Resources))}.
```

The `init` function returns for every resource in a given list `Resources` a record of type `lock` where the first field contains the name of the resource and the other two fields are instantiated with the (default value) empty list.

```
handle_call({request,Resources}, Client, Locks) ->
    case check_availables(Resources,Locks) of
       true ->
          {reply, ok,
              map(fun(Lock) ->
                     update_exclusive(Lock,Resources,Client)
                  end, Locks)};
       false ->
          {noreply,
              map(fun(Lock) ->
                     add_pending(Lock,Resources,Client)
                  end, Locks)}
    end;

handle_call(release, Client, Locks) ->
    NewLocks =
      map(fun(Lock) ->
              release_lock(Lock,Client)
          end, Locks),
    Locks_updated =
      send_reply(NewLocks,all_pendings(NewLocks)),
    {reply, ok, Locks_updated}.
```

The generic server automatically supports every message in a `gen_server:call` with the process identifier of the sender and a tag (a kind of time stamp to distinguish different messages from the same client). When obtaining a request, the locker stores the combination of identifier and tag as a pair in the pending list (or exclusive field). When releasing, a new tag is used for the pair (since it is a new message) and removing the pair from the list should be done by only looking at the process identifier. Note that the locker cannot remove the tag already at the moment of receiving the request of a client, since the tag is necessary for a reply, as implemented by `send_reply`. This function checks for every pending client whether its resources are available. If so, the client is notified and the locks are updated.

```
send_reply(Locks,[]) ->
  Locks;
send_reply(Locks,[Pending|Pendings]) ->
  case obtainables(Locks,Pending) of
       true ->
```

```
        gen_server:reply(Pending,ok),
        send_reply(map(fun(Lock) ->
                            promote_pending(Lock,Pending)
                        end, Locks),
                   Pendings);
      false ->
        send_reply(Locks,Pendings)
  end.
```

These are the only functions that contain side effects, viz. the sending and receiving of messages. All other functions are side-effect free and easy to implement.

In addition to client and locker code we also have implemented a so called *supervision tree*, a commonly used design principle to monitor the individual processes [8]. Basically the code for the supervision tree describes a process that is started, which monitors two processes, one is the locker, the other a new supervisor process, which monitors the clients. The code describes what should happen if one of the processes crashes and is instructed to restart clients and locker processes.

All processes together can now be started with only one function call, viz. `supervisor:start`, with in the arguments the number of clients one wants to start it with and the list of resources one considers.

4 A μCRL specification

The Erlang modules described in the previous section are automatically translated into one μCRL specification. The data is directly translated from Erlang to μCRL without any abstraction. The specification is used to generate the transition system, which is used for model checking.

The translation is performed in two steps. First we apply a source-to-source transformation on the level of Erlang, resulting in Erlang code that should be executable in the same way as the original, but is optimized for verification. Second we translate the collection of Erlang modules into one μCRL specification. The advantage of having an intermediate Erlang format is that programmers can easily understand the more severe manipulations of the code and therefore are better able to understand the smaller step to μCRL notation. Moreover, the intermediate code can be input for other verification tools.

4.1 Erlang to Erlang transformation

The source-to-source transformation of the Erlang modules contains many steps and we mention only the more relevant ones, skipping trivial steps like removing the debug statements in the code.

We use the supervision tree structure to obtain a finite set of initial processes. We start the translator with the same arguments as that we would need to build and start the supervision tree. This allows us to bind the number of clients and resources to a certain value. For every different number we need to run a different transformation. The supervisor processes are taken away and the new initialization

function only creates the processes of locker and clients. The handling of a process that crashes is left to be detected in the transition system.

We replace (a predefined set of) higher order functions like `map` by a first-order alternative, since the target specification language does not support higher order functions. Thus, a call `map(fun(X) -> f(X,Y1,...,Yn) end, Xs)` is replaced by a call to a new function `map_f(Xs,Y1,...,Yn)` which is defined and added to the code as

```
map_f([],Y1,...,Yn) ->
  [];
map_f([X|Xs],Y1,...,Yn) ->
  [f(X,Y1,...,Yn)| map_f(Xs,Y1,...,Yn)].
```

In the next phase we determine all functions with side-effects, i.e., those functions that do send or receive a message or call a function doing so. This is a call-graph problem where we keep a list of side-effect free functions in the library modules. The `gen_server:call` function and `handle_call` function are typically added to the functions that contain side-effects.

The most involved operation is now to get rid of the use of return values of functions with side-effects. In μCRL a process may have side-effects, but has no return value; on the other hand, a function in μCRL has a return value, but may not contain a side-effect. In case an Erlang function (in)directly causes a side-effect, its computation part and side-effect part have to be split. For the source-to-source transformation, it suffices to make sure that all return values are matched in a variable and to provide decomposition of the data structure of this return value by means of side-effect free functions. Currently we can deal with basic data types and the compound data types lists, tuples, records and mixtures of these.

4.2 Erlang to μCRL transformation

Given the Erlang modules that are transformed as described above, we generate one μCRL specification from these modules. Erlang is dynamically typed whereas μCRL is strongly typed. Therefore, we construct in μCRL a data type *ErlangTerm* in which all Erlang data types are embedded. All side-effect free functions are added as a term rewriting system with this *ErlangTerm* data type. A standard transformation is used to translate Erlang statements into the term rewriting formalism. In addition we have to define an equivalence relation on data types, which is rather involved. In this particular case with only 14 different atoms and 7 data constructors, 440 equations are reserved for comparing data types, roughly two third of the whole specification.

With respect to the part with side-effects, we benefit from the fact that the Erlang to Erlang transformation was generated for a specific configuration and contains all information on which processes are started. This allows us to define the initial configuration in the μCRL specification. The Erlang processes coincide with the μCRL processes, where a non-terminating Erlang function describes the main loop of the process in the Erlang case. However, when translating this loop, we cannot translate recursive calls to Erlang functions with side-effects in a direct

way to μCRL. In μCRL computation and side-effects cannot be intermingled. The solution is found in the definition of a separate μCRL process implementing a call stack. Communication with this call stack is used to return the values of the computation.

Certain restrictions with respect to the μCRL functions have to be taken into account; there is only one function clause possible, with only sequential composition, non-deterministic choice, and an if-then-else statement for control. We translate case statements and pattern matching by using the if-then-else construct and calls to newly introduced process functions. The `handle_call` and `gen_server:call` are translated into communicating actions in μCRL. The different clauses of the `handle_call` function are combined in one μCRL loop, using the state mentioned in the arguments of `handle_call` as state of the loop. The unique process identifiers used in Erlang are integrated as an argument (`Self`) of all process calls and instantiated by the first call in the initial part.

```
comm
     gen_server_call | handle_call = call
     gen_server_reply | returned = return

proc locker(Self: Term,Locks: Term) =
    sum(Client: Term,
        sum(Resources: Term,
            handle_call(Self,tuple(request,Resources),Client).
            (gen_server_reply(Client,ok,Self).
             locker(Self,
                    map_update_exclusive(Locks,Resources,Client))
                  <| eq(check_availables(Resources,Locks),true) |>
             locker(Self,
                    map_add_pending(Locks,Resources,Client))))) +
    sum(Client: Term,
        handle_call(Self,release,Client).
        send_reply(Self,map_release_lock(Locks,Client),
                   all_pendings(map_release_lock(Locks,Client))).
        sum(Locks2: Term,
            rcallresult(Self,Locks2).
            gen_server_reply(Client,ok,Self).
            locker(Self,Locks2)))

send_reply(Self:Term,Locks:Term,MCRLArg1:Term) =
     (wcallresult(Self,Locks)
       <| eq(equal(MCRLArg1,nil),true) |>
     (gen_server_reply(hd(MCRLArg1),ok,Self).
      send_reply(Self,
                 map_promote_pending(Locks,hd(MCRLArg1)),
                 tl(MCRLArg1))
       <| eq(obtainables(Locks,hd(MCRLArg1)),true) |>
      send_reply(Self,Locks,tl(MCRLArg1))))
```

After this automatic transformation, we can verify a specific configuration, in which the clients repeatedly request all available resources. In order to perform several verifications at once, in particular to verify all situations in which the clients repeatedly request an arbitrary (varying) subset of the resources, we modified the μCRL specification by hand. We used μCRL's possibility to express non-determinism for this. The μCRL specification is used to generate a transition system. The number of states for the generated systems depends on the configuration. We tried several configurations, up to three clients and four resources, the largest resulting in about a million states. Creating such large state spaces takes a few hours on a single processor workstation. Even though this is time consuming, improving this has not highest priority; we plan to focus on small examples in the development phase of the software. Larger examples take more time, but so does testing. The development of on-the-fly model checking and parallelization of the model checker might increase performance dramatically in a later stage.

5 Verifying the model

The three properties we want to verify for this locker are: absence of deadlock, mutual exclusion and no starvation. All are classical properties that are well studied in literature. The first is trivially shown, the second and third need the right formulation and the support of a model checker. Mutual exclusion is a safety property, whereas no starvation is a liveness property. The safety properties are easier to check than the liveness properties, as is explained later and depends on the fact that some infinite traces in the specification are excluded in a real Erlang execution because of the underlying Erlang scheduler.

5.1 Mutual Exclusion

The property for mutual exclusion should express that a resource can only be accessed by one client at the same time. In order to show this, we added two actions to the μCRL specification `use` and `free` with a resource as an argument. As soon as we enter the critical section, the `use` action is applied for all resources that the client requested. Before leaving the critical section, the resources are given free again. We use the macro

$$\textsc{Until}(a_1, a_2) = [\text{-}^*.a_1.(\neg a_2)^*.a_1]\mathit{false}$$

stating that 'on all possible paths, after an a_1 action, any other a_1 action must be preceded by an a_2 action'. The mutual exclusion property depends on the number of resources. In fact we need a different formula for any number of resources. For a system with two resources, r_1 and r_2, the mutual exclusion property is formalized by

$$\begin{aligned}\mathrm{MUTEX}(r_1, r_2) = {} & \textsc{Until}(\mathtt{use}(r_1), \mathtt{free}(r_1)) \wedge \\ & \textsc{Until}(\mathtt{use}(r_2), \mathtt{free}(r_2))\end{aligned}$$

A new version of the model checking tool within the CÆSAR/ALDÉBARAN toolset [9] is under construction and with this new release, we should be able to formulate one property for an arbitrary number of resources.

The mutual exclusion property has been shown for configurations with 2 resources and 2 and 3 clients where the clients repeatedly request an arbitrary (none empty) subset of the resources as well as for the situation with 4 resources and 3 clients. The latter consisted of a model with a million states and it took a few hours to verify the mutual exclusion property. A recently developed parallel model checker has been used to check our largest transition system. The few hours have been reduced to nine minutes on about fifty processors [5]; a promissing development for scaling this approach.

5.2 Starvation

Proving that there is no starvation for the processes turned out to be a problem. This is caused by the fact that there are traces in the transition system that do not correspond to a fair run of the Erlang program. The Erlang processes are scheduled by the use of a certain scheduler and in the model we have (on purpose) abstracted from scheduling and consider all possible sequences of actions, even those in which one single processes gets all execution time.

We want to base our *no starvation* property on the notion of *an action is eventually followed by another action.* In particular, the request of a resource is eventually followed by using that resource. One way of formulating this property is:

$$\text{EVTFOLLOW}(a_1, a_2) = [\text{-}^*.a_1].\mu X.(\langle\text{-}\rangle true \wedge [\neg a_2]X)$$

We used this in a context where we instantiated the actions a_1 and a_2 by the request for a resource and the entering of the critical section, respectively. For the latter, we use the confirmation by the locker, i.e., the returned ok message. The actual property, like in the mutual exclusion case, depends on the number of clients and resources. For three clients and two resources we have:

$$\begin{aligned}&\text{NO\_STARVATION}(c_1, c_2, c_3, i_1, i_2, i_3) = \\ &\quad \text{EVTFOLLOW}(c_1, i_1) \wedge \text{EVTFOLLOW}(c_2, i_2) \wedge \text{EVTFOLLOW}(c_3, i_3)\end{aligned} \tag{1}$$

Unfortunately, this property does not hold, even for simple scenario's where definitely no starvation occurs. As an example consider the following simple scenario with three clients and two resources. The clients repeatedly request only one resource, where client 1 and 2 request A, and client 3 requests B. In such a scenario there is no starvation, since both clients may access their resource, release it and request it again. In the μCRL specification we have the possibility of a loop in which client 3 continuously requests and releases resource B. The clients requesting resource A simply do not get any scheduling time in this sequence. However, in the Erlang program this loop is not present, because of the scheduler. Thus, the problem is to disregard unrealistic loops in the transition system. Removing such loops from the transition system, if we at all could find a way to do so, is

incorrect. In a realistic setting, such a loop could be executed a few times before the scheduler enables the other processes. What in a realistic setting is excluded, is the infinite traversal of only this loop.

We would like to weaken the EVTFOLLOW property, such that non-fair paths, which exist in the model, but not in the implementation due to the scheduling by the Erlang run-time system, are ignored. Because of limitations in the model checking tool (evaluator 3.0) we need to express this property in alternation free μ-calculus. External advice was required to come up with the following reformulation of EVTFOLLOW, describing that even if a loop exists before reaching a_2, it is still possible (from every state of the loop) to reach a_2 after a finite number of steps (modality $\langle -^*.a_2 \rangle true$).

$$\text{EVTFOLLOW}(a_1, a_2) = [-^*.a_1.(\neg a_2)^*]\langle -^*.a_2 \rangle true$$

This property is weaker and in combination with Property (1) it holds for the above mentioned scenario's. Unfortunately, it is too weak, i.e., ignores loops that should be considered. Property (1) with this weaker EVTFOLLOW holds for the first scenario mentioned in Section 2 in which we have starvation in the Erlang context. Recall that for that scenario, client 1 and 2 on their turn take priority over client 3. Thus, there is an ignored loop with only actions of client 1 and 2, although it causes client 3 to starve.

We need to be more precise in the kind of actions that we ignore in a loop and which not. Thinking a little longer about this, it turns out that all actions may appear in the loop. Neither a request nor a release of any other client should be ignored. No matter with action one would like to ignore, there is always a plausible scenario possible from which it is clear that one cannot ignore that action. Even a whole loop should in principle be allowed, as long as it does not occur infinitely often if other actions along the path are also enabled. In our opinion this goes beyond the expressiveness of the logic we use.

Currently we investigate several possibilities to work around this problem, viz. adding explicit scheduling to the μCRL specification, having the model checker changed, or using a different logic (and model checker) that enables reasoning with fairness.

One might wonder whether starvation is an important property at all, since even if a theoretical starvation problem occurs, it might happen that in reality the process always gets served. In regular implementations a timer is set after sending a message and the starvation as such shows as a time out on the client site. This time out is normally followed by a retry and as such the process might get served after a few attempts. We experimented with that by adding such time outs and removing the check for the pending list in the function `check_available` (which leads to starvation for the scenario which was discussed in Section 2). Running this program does not show a starvation at first sight. The client does get access to the resource, occasionally. If we implement the clients with an access time of say, 500 ms, in the critical section, then starvation will show up in the form of a time out of some of the clients. The total number of served requests gets lower, in particular for the clients for which we know that they theoretically starve.

Interesting in this context is that we only detected the performance problem after sufficiently increasing the time spent in the critical section. Here one can

argue that testing would not have been sufficient and that the error could show unexpectedly after having the software in use for a long time. Hence, we find starvation an important property to verify.

6 Conclusions

The main contribution of this work lays in the development of an automatic translation of a class of Erlang programs into μCRL. This enables a development of Erlang programs that goes hand in hand with formal verification; leading to formally verified programs. We do not expect smart abstractions or clever tricks performed by the users of this tool, assuming from them a limited knowledge on verification issues. We provide 'push-button verification' that fits in the existing development cycle. As a leading example for developing our tool we used an implementation of a locker algorithm. Verification of this locker algorithm has only partly been successful. Absence of deadlock and mutual exclusion could be proved, but it could not be shown effectively that the algorithm is starvation free. It is subject to further research to find a way around this problem.

The number of states in our models was not much more than a million, such that real performance problems were not encountered. It takes a while for a complete verification, but a few hours is still considered acceptable in this stage. The majority of the work is put in getting the specification right and formulating the right properties. In this case, in particular for 'no starvation', we have spent much time in the formulation of the, still not satisfactory, property.

We use an approach similar to PathFinder or the Bandera project [13, 6]. It would be interesting to see if a Java version of the same case study could easily be handled by using those tools, but we have not found the opportunity to do so. Running the model checking approach of Huch [15] directly on this example is impossible, since that version does not support the generic server design principle. We could change the program by removing this generic server implementation and use a direct implementation in Erlang instead. However, the approach of Huch would translate the choice whether to return an ok message to the client or to store the client in the pending list, to be a non-deterministic choice. By abstracting away the data in that way, mutual exclusion does not hold for the obtained transition system.

Another approach to verification of Erlang programs which differs from model checking is the use of a theorem prover for checking properties. The Swedish Institute of Computer Science have in cooperation with Ericsson developed a kind of theorem prover specially focussed on Erlang programs [3]. Advantage of using this tool compared to the model checking approach are the possibility of using the full μ-calculus (instead of alternation free), the possibility to reason over an unbounded number of clients and resources, and the completeness of the approach, i.e., if a proof is given, it holds for the program and not only for the specification. Since model checking allows an easier automation, we aim on using this technique for prototyping and use the theorem prover approach for the version we are satisfied with.

With this verification of the locker case-study we posted several questions for further research and we solved several practical issues on the way. We continue

with adding features to the locker, such as shared locks and fault-tolerance, therewith increasing the need for an even better translation tools.

Acknowledgements

We would like to thank Radu Mateescu and Hubert Garavel from INRIA Rhone-Alpes, Izak van Langevelde, Jaco van de Pol and Wan Fokkink from CWI, and Lars-Åke Fredlund and Dilian Gurov from SICS for taking part in the discussions on this case study and supporting us with their advises.

References

[1] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall International, 2nd edition, 1996.

[2] T. Arts and T. Noll, Verifying Generic Erlang Client-Server Implementations. In *Proceedings IFL2000*, LNCS 2011, p. 37–53, Springer Verlag, Berlin, 2000.

[3] T. Arts, G. Chugunov, M. Dam, L-Å. Fredlund, D. Gurov, and T. Noll A Tool for Verifying Software Written in Erlang To appear in: *Int. J. Software Tools for Technology Transfer*, 2001.

[4] S. Blau and J. Rooth, AXD 301 – A new Generation ATM Switching System. *Ericsson Review*, no 1, 1998.

[5] B. Bollig, M. Leucker, and M. Weber, Local Parallel Model Checking for the Alternation Free μ–Calculus. tech. rep. AIB-04-2001, RWTH Aachen, March 2001.

[6] J. Corbett, M. Dwyer, L. Hatcliff, Bandera: A Source-level Interface for Model Checking Java Programs. In *Teaching and Research Demos at ICSE'00*, Limerick, Ireland, 4-11 June, 2000.

[7] CWI, `http://www.cwi.nl/~mcrl`. A *Language and Tool Set to Study Communicating Processes with Data*, February 1999.

[8] Open Source Erlang, `http://www.erlang.org`, 1999.

[9] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireau. CADP (CÆSAR/ALDÉBARAN development package): A protocol validation and verification toolbox. In *Proc. of the 8th Conf. on Computer-Aided Verification*, LNCS 1102, p. 437–440, Springer Verlag, Berlin, 1996.

[10] W. Fokkink, *Introduction to Process Algebra*, Texts in Theoretical Computer Science, Springer Verlag, Heidelberg, 2000.

[11] J. F. Groote, W. Fokkink, M. Reiniers, *Modelling Concurrent Systems: Protocol Verification in μCRL.* course lecture notes, April 2000.

[12] J. F. Groote, The syntax and semantics of timed μCRL. tech. rep. SEN-R9709, CWI, June 1997. Available from `http://www.cwi.nl`.

[13] K. Havelund and T. Pressburger, Model checking JAVA programs using JAVA PathFinder. *Int. J. on Software Tools for Technology Transfer*, Vol 2, Nr 4, pp. 366–381, March 2000.

[14] G. Holzmann, *The Design and Validation of Computer Protocols.* Edgewood Cliffs, MA: Pretence Hall, 1991.

[15] F. Huch, Verification of Erlang Programs using Abstract Interpretation and Model Checking. In *Proc. of ICFP'99*, Sept. 1999.

Erlang Specification Method – A Tool for the Graphical Specification of Distributed Systems

Frank Huch
Institut für Informatik, Christian-Albrechts-Universität Kiel,
D-24098 Kiel, Germany
fhu@informatik.uni-kiel.de

ABSTRACT

We present a tool for the specification, documentation and verification of specifications for distributed systems. The used specification formalism is especially designed for the later implementation using the functional programming language Erlang. The main point of the specification formalism is using the same kind of communication and process concept in the specification as provided by Erlang. Processes are graphically specified as labeled transition systems with sequential behaviour, branching, sending or receiving of values, and the creation of new processes as possible actions. As in Erlang receiving values can be expressed with pattern matching against all messages in the mailbox of a process.

The tool provides an editor for a convenient development of specifications, a consistence checker, a simulator, a component for generating Erlang code from the specification and a model checker for proving properties expressed in LTL for abstractions of the specifications.

Keywords: specification, verification, Erlang, distributed system

1 INTRODUCTION

Growing requirements of industry and society impose greater complexity of software development. Consequently understandability, maintenance and reliability can not be warranted.

Things get even harder whenever leaving the sequential territory and developing distributed systems. In addition to the problem of designing large systems we get the problem that the behaviour of concurrent processes is even harder to understand than a sequential behaviour. In distributed systems many programs are executed concurrently and interact via communication. Hence for a simulation of what happens when a program is executed one has to consider many states in many processes, the system can be in. Considering hundred or more concurrent processes it is clear that this is impossible, just knowing the code of the program. The difference to sequential programs is the great number of possible execution paths of concurrent and distributed system in dependency of schedulers or the performance of system components. Different runs of a program can lead to different results, even deadlocks. Hence a program tested and running well on a plattform can deadlock on another one.

The same problems appear during the development of distributed systems. Therefore we need a formal method for the specification of distributed systems which can also be used as a good documentation of its implementation. The way from the specification to the implementation should be straightforward to provide fast development and should also be seconded by automatic generation of code or code skeletons. But for using the specifications as documentation it is also necessary that the implementation does not differ too much from the specification and that the way back from the implementation to the specification can also be taken. To provide this the specification formalism should be tailored especially to the programming language and use the same kind of process model and communication. Messages should have the same format and new processes should be created in the same way.

In this paper we present the Erlang Specification Method (ESM) for the specification of concurrent and distributed systems designed especially for the distributed functional language Erlang. This formalism was used for the first time for the specification of a communication system. This was done by groups of students in a practical course. In the development almost 70% of work had to be done in specification. But this resulted in very good structured systems which work very robustly and have an almost professional character. Time for the development was very short and we were surprised of the quality and complexity of the developed systems. After the practical course we started to develop a tool for the graphical specification of Erlang processes. It supports a comfortable graph editor, consistence checks of the specifications, a simulation component and a model checker for proving properties of the system expressed

in LTL. Because ESM is very expressive, the verification component also provides abstraction.

In Section 2 we present our specification formalism in detail. For the simulation of a specified system a formal semantics is defined in Section 3. Section 4 shows how easy code skelettons can be generated from the specification. The verification component is described in Section 5 and in the conclusions in Section 6 we show the experiences made in the development of a communication system and discuss future work. For readers, who are not familiar with Erlang, we give short introduction in Appendix A.

2 THE SPECIFICATION FORMALISM ESM

In the development of large distributed systems it is necessary to specify the system design. A large part of the specification of a distributed system is the specification of protocols, which processes use for their communication. Other parts of the system specification are the partitioning in processes, their location in a network and the behaviour of the sequential parts. In this paper we will not discuss the specification of the partitioning in processes and their location. We will concentrate on the protocol specification.

In practice Erlang developers use specification formalisms like UML [Alh98] or SDL [ST87] for the specification of protocols. The main problem of the use of these formalisms is that communication between processes differs from communication in Erlang. But in the specification of a protocol the kind of communication is a major point. Implementing UML or SDL specifications in Erlang leads to a program which simulates the corresponding communication of the specification formalism. We lose the expressiveness of accessing the messages of the mailbox with pattern matching. Another point of implementing these specifications in Erlang is that we have to guarantee that our implementation of the UML or SDL communication is correct. This can especially be hard using UML with remote method invocation. Another problem of UML is that it uses an object oriented view which can be difficult to be implemented in Erlang considering inheritance.

On the first sight SDL seams to match communication in Erlang. And Erlang programmers are supposed to use it for the specification of Erlang programs [AWV93]. In SDL processes are also identified by a pid and every process has a mailbox in which incoming messages are stored. But the mailbox is just a queue and a process has to take the messages out, in the same chronological order, as they come in. So processes have to consider many possible interleaving sequences of incoming messages instead of picking them out in a supposed order. Implementing an SDL specification in Erlang, all receive statements are flat and all possible messages have to be considered in every `receive` statement. The program gets larger than it could be using pattern matching on the mailboxes.

We see that the model of communication is a major point for the choice of a specification formalism. Many other formalisms like LOTOS [vEVD89] and Estelle [ISO89] exist for the specification of protocols. But there is no formalism defined yet, which uses the same communication as Erlang does. But for developing distributed systems in Erlang with all the opportunities of Erlang such a formalism has to be defined. The result is the Erlang Specification Method (ESM):

For every kind of process $p \in \mathcal{P}$ a labeled transition system $\mathcal{T}_p = (S_p, I_p, \longrightarrow_p, vars_p)$ is defined which specifies its behaviour. The states $s \in S_p$ are labeled with variables representing the knowledge of the process in this state. *Vars* is the set of all variables and $vars_p : S_p \longrightarrow Vars^*$, yields the sequence[1] of variables the states are labeled with. The initial states $i \in I_p$ are labeled with the variables the process binds its knowledge in, when it is started. Multiple initial states are possible because a process can be started in different ways.

The labels of the transitions will contain Erlang terms over variables of the predecessor nodes for the specification of calls to sequential parts. Therefore we assume Σ to be set of function symbols with arity. As an example $+/2 \in \Sigma$. Now we can define Erlang terms over a set S, which will later be instantiated with variables or pids:

$$S \subseteq T_\Sigma(S)$$
$$v_1, \ldots, v_n \in T_\Sigma(S) \text{ and } f/n \in \Sigma \Longrightarrow f(v_1, \ldots, v_n) \in T_\Sigma(S)$$

Let $C = \{[\,]/0, [.|.]/2\} \cup \{\{\ldots\}/n \mid n \in \mathbb{N}\} \cup \{a/0 | a \in Atoms\}$ be the set of constructor functions, where *Atoms* is the set of all allowed Erlang atoms, $[\,]$ and $[.|.]$ are the constructors for building lists and $\{\ldots\}$ is the constructor for tuples of any arity. For the set of constructor terms, which will later be the values we calculate on or the possible patterns we then use $T_C(S)$.

The following transitions are possible:

- Sequential evaluation which is specified with an Erlang expression. Its behaviour is specified with natural language. For access to the result we allow pattern matching. The variables which are bound in pattern matching can be part of the knowledge of the destination state.
  Notation: $(s, pat \text{ <- } e, t) \in \longrightarrow_p$, where $s, t \in S_p$, $e \in T_\Sigma(Vars(s))$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Branching in dependence of an the value of an Erlang expression. Again the behaviour of the expression is specified with natural language, but the result is matched against multiple patterns. All matching patterns can be performed. The difference to the sequential behaviour is that branching is allowed with the same expression.
  Notation: $(s, \textbf{case } e \textbf{ of } pat, t) \in \longrightarrow_p$, where $s, t \in S_p$, $e \in T_\Sigma(Vars(s))$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Sending a value to another process. The pid of the other process must be known in this state. We use the Erlang syntax for sending actions.
  Notation: $(s, X!v, t) \in \longrightarrow_p$, where $s, t \in S_p$, $X \in vars_p(s)$, $v \in T_C(Vars(s))$ and $vars_p(t) \subseteq vars_p(s)$

- Receiving values and binding variables to (parts of) these values with pattern matching. The bound variables can be knowledge of the destination state.

[1] In most cases we will handle this sequence as a set of variables. The order is only relevant in the creation of new processes, when values are passed.

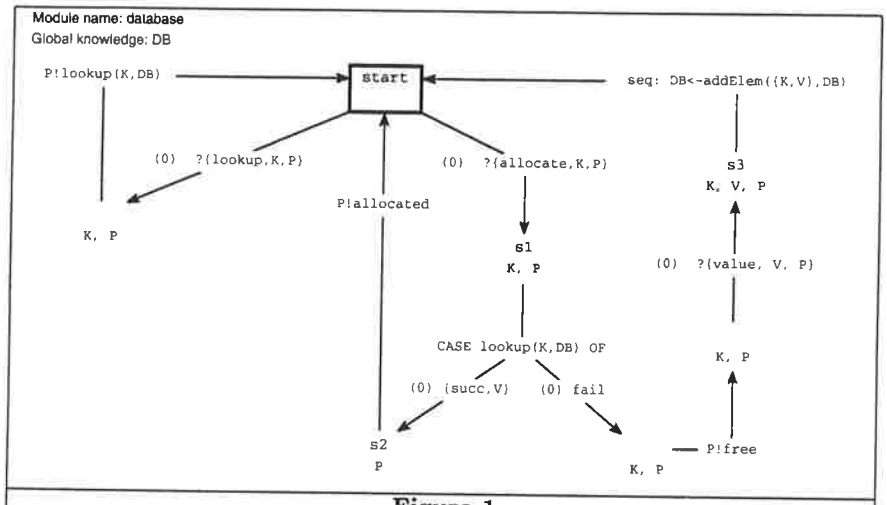


Figure 1
The specification of a database process

Notation: $(s, ?pat, t) \in \longrightarrow_p$, where $s, t \in S_p$, $pat \in T_C(Vars)$ and $vars_p(t) \subseteq vars_p(s) \cup vars(pat)$

- Creation of new processes with a list of arguments. For the creation of new processes we also use the Erlang syntax of `spawn` and add X `<- spawn(...)` if the pid of the spawned process is needed in future states and has to be knowledge of the destination state.
  Notation: $(s, X\text{<-spawn}(p', i, e), t) \in \longrightarrow_p$ or $(s, \text{spawn}(p', i, e), t) \in \longrightarrow_p$, where $s, t \in S_p$, $X \in Vars$, $e \in T_\Sigma(Vars(s))$, $i \in I_{p'}$ an initial state of the process specification $p' \in \mathcal{P}$ and $vars_p(t) \subseteq vars_p(s) \cup \{X\}$

To prevent too much nondeterministic behaviour of a process there may be only two kinds of nodes with more than one outgoing arc. First, a branch can be specified with case arcs, for example a lookup in a database succeeds or fails. All expressions of a branch have to be identical. The user should define non-overlapping patterns, but non-determinism in the protocol specification is possible as well. In the tool, this formal definition is implemented as a special branching pattern, which allows the user to define only one expression. Furthermore branching is allowed, if every outgoing arc of a state is labeled with a receive operation. Again the user should define non-overlapping patterns, to prevent non-determinism, but in an early part of the specification non-determinism can be helpful.

In Figure 1 we present the specification of a database or name server, in which clients can store values with unique keys and look them up with a key. All presented specifications are created with our tool.

In our tool the initial states are marked with a border to distinguish them from the other states. For every state the user can optionally define a name, which is displayed in its first line. Only the initial states must have a name, because with these names they can be spawned from other processes. The variables of the state follow in the next lines. Furthermore it is possible to declare some variables as global knowledge. This is an abbreviation for the occurrence of these variables in every state of the process specification. In the formal semantics we have omitted this. But for clearer specifications we have added the declaration of global knowledge in the tool.

The specification of the database has only one initial state, named `start`. In this state the database accepts only messages matching the patterns `{allocate,K,P}` and `{lookup,K,P}`. The variable `K` is bound to the requested key and `P` to the pid of the requesting process in both cases. In the `lookup`-case the database just answers with the lookup-value of the key in the database, where `lookup` is a sequential function. Its specification can colloquially be defined by: "`lookup` yields the corresponding value of a key stored in the database". This specification is stored in the tool and can be accessed or modified by clicking on the arc. When an `allocate`-message is received, we have to consider two cases. First, the key is already allocated and the database only notifies the requesting process and succeeds in its initial state. Second, the key is free. In the formal syntax this branch was modeled by two arcs labeled with case, the call of the lookup-function and the respective patterns. The numbers in front of the patterns in the specification in Figure 1 are an additional feature of the tool not formalized above. Here the user can defined priorities in which the patterns should be matched, if they overlap. This can sometimes be useful to make a specification more detailed. In our specification the patterns do not overlap and all priorities have no effect and can be by default set to zero. After a notification of the client, the database waits until the corresponding value is received and adds this new values to the database.

The main point in the specification of the database process is, that we guarantee mutual exclusion for multiple clients extending the database. Therefore we use two different states in which we wait for different messages. Furthermore, we check if the `allocate`-message and the `value`-message include the same pids in their third component. The variable `P` in the pattern `{value,V,P}` is already bound to the pid of the requesting process and is no introduction of a fresh unbound variable. Hence the second receive statement suspends until a value with the same pid is sent. We will later prove this property and need to refer to the states `s1`, `s2` and `s3`. Therefore we named them in the specification.

A specification of this database process in SDL would be more complicated. In the second receive statement we would have to list all possible other messages, which could be sent to the database from other clients. But the process may not

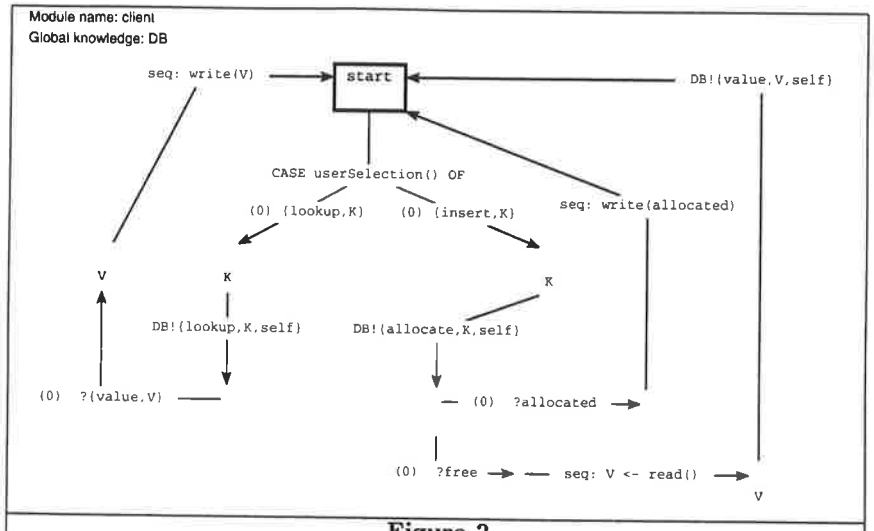


Figure 2
The specification of a client process

ignore these other messages and has to save them in a data structure, which has to be processed later. An alternative would be sending other messages, which do not match the pattern `{value,V,P}`, to the database process itself. But this means busy waiting, which should be impeded.

In Figure 2 we also present the specification of a client process. A client can enroll new entries in the database or lookup keys, in dependence of a users choice. The here presented example is very easy and small and it is no problem to program this directly in Erlang. But programming larger systems with multiple different kinds of processes it is difficult to have an overview of all processes taking part in a protocol. The system helps the developer, because she is restriced to only the communication parts of the protocol and can define the protocol stepwise. She starts with the definition of the sending action starting a special protocol, for example the userSelection in Figure 2 and the resulting `lookup`–message sent to the database. Then she switches to the database and specifies how it reacts. It sends back the corresponding value and succeeds in the initial state. Again the developer switches to the process which will receive the message, the client. After finishing this part of the specification she backtracks and succeeds with the non–finished states. This track through the specification is seconded by a stack, where unfinished states are put and can later be retract and finished.

3 SEMANTICS OF ESM

In the operational semantics multiple processes will be created and every new process will be identified by its own pid. To model these pids we define $Pid := \{@n \mid n \in \mathbb{N}\}$. A state of the operational semantics of an ESM specification is a finite set of processes: $State := \mathcal{P}_{fin}(Proc)$, where a process consist of a pid, a state, the knowledge bound to the variables, and a mailbox: $Proc := Pid \times \bigcup_{p \in \mathcal{P}} S_p \times Know \times MBox$. Mailboxes $MBox := T_C(Pid)^*$ are words over ground terms of an execution. The knowledge of a state is a binding of its variables $Know : Vars \longrightarrow T_C(Pid)$. This knowledge will be updated when a process executes some actions. Therefore we use the function

$$\uplus : Know \times Know \longrightarrow Know$$
$$(\rho_1 \uplus \rho_2)(X) = \begin{cases} \rho_1(X), & \text{if } \rho_1(X) \text{ is defined} \\ \rho_2(X), & \text{otherwise} \end{cases}$$

Finite knowledges, to which we can restrict in the semantics will also be represented as the graph notation of a substitution $([X_1/v_1, \ldots, X_n/v_n])$.

The operational semantics is denoted by the relation $\Longrightarrow \subset State \times Label \times State$, where $Label$ is the set of actions relevant for concurrency:

$$Label := \{!v \mid v \in T_C(Pid)\} \cup \{?v \mid v \in T_C(Pid)\} \cup \{\mathtt{spawn}(f) \mid f/n \in FS(p)\} \cup \{\varepsilon\}$$

We write $s \overset{a}{\Longrightarrow} s'$ for $(s, a, s') \in \Longrightarrow$ and $s \Longrightarrow s'$ for $s \overset{\varepsilon}{\Longrightarrow} s'$.

The semantics of the sequential parts of the specification is defined by an interpretation $\mathcal{A} = (T_C(Pid), \iota)$, where $f_{\mathcal{A}} = \iota(f/n) : T_C(Pid)^n \longrightarrow T_C(Pid)$ interprets the used functions. The semantics of an Erlang term with respect to a local knowledge can then be defined algebraically:

$$\mathcal{E} :: T_\Sigma(Vars) \times Pid \times Know \longrightarrow T_C(Pid)$$
$$\mathcal{E}[\![X]\!]^p \rho = \rho(X)$$
$$\mathcal{E}[\![\mathtt{self}]\!]^p \rho = p$$
$$\mathcal{E}[\![\phi(e_1, \ldots, e_n)]\!]^p \rho = \phi_{\mathcal{A}}(\mathcal{E}[\![e_1]\!]^p \rho, \ldots, \mathcal{E}[\![e_n]\!]^p \rho)$$

The definition of the operational semantics $\Longrightarrow$ is presented in Figure 3.To keep the rules for assignment, case and receiving easily, we use functions match and mailmatch to model Erlang's mechanism of pattern matching. In the rule for assignment the expression is evaluated and matched against the pattern. If this is successful, the knowledge is updated with the corresponding substitution. Compared to other functional languages pattern matching in Erlang is more complicated, because of non–linear patterns with multiple occurrences of the same variable:

$$\text{match}(X,t) = [X/t]$$
$$\text{match}(c(pat_1, \ldots, pat_n), c(v_1, \ldots, v_n)) = \text{match}(pat_1, v_1) \oplus \ldots \oplus \text{match}(pat_n, v_n)$$
$$\text{match}(\_,\_) = \text{Fail, otherwise}$$

[2] $[v_1, \ldots, v_n]$ is an abbreviation for $[v_1 \mid [.. \mid [v_n \mid []]..]]$

$$\frac{(s, pat\texttt{<-}e, t) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad match(pat, v) = \rho'}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t, \rho' \uplus \rho, q)}$$

$$\frac{(s, \texttt{case}\ e\ \texttt{of}\ pat_1, t_1) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad \mathrm{match}(\rho(pat), v) = \rho'}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t_i, \rho' \uplus \rho, q)}$$

$$\frac{\begin{array}{c}\text{let } (s, ?pat_1, t_1), \ldots, (s, ?pat_m, t_m) \in \longrightarrow \text{ be all possible actions in } s \\ (i, j, \rho') \in \mathrm{mailboxmatch}((\rho(pat_1), \ldots, \rho(pat_m)), (v, \ldots, v_n))\end{array}}{\Pi, (p, s, \rho, (v_1, \ldots, v_j, \ldots, v_n)) \Longrightarrow \Pi, (p, t_i, \rho' \uplus \rho, (v_1, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n))}$$

$$\frac{(s, X\texttt{!}e, t) \in \longrightarrow \quad \mathcal{E}[\![e]\!]^p\rho = v \quad \rho(X) = p'}{\Pi, (p, s, \rho, q), (p', u, \rho', q') \Longrightarrow \Pi, (p, t, \rho, q), (p', u, \rho', q' : v)}$$

$$\frac{\begin{array}{c}(s, X\texttt{=spawn}(m\texttt{,}i\texttt{,}e), t) \in \longrightarrow \quad \mathcal{T}_m = (S_m, I_m, \longrightarrow_m) \\ i \in I_m \quad \mathcal{E}[\![e]\!]^p\rho = [v_1, \ldots, v_n]^2 \quad vars(i) = (X_1, \ldots, X_n)\end{array}}{\Pi, (p, s, \rho, q) \Longrightarrow \Pi, (p, t, \rho[X/p'], q), (p', i, [X_1/v_1, \ldots, X_n/v_n], q')}$$

Figure 3: *Operational Semantics*

Two derived substitutions can only be unioned if the overlapping parts are identical. This is tested in the union function $\oplus$, which yields Fail otherwise.

A branch can be chosen, if a pattern matches the semantics of the expression. In contrast to the assignment, the substitution in this rule is also applied to the patterns. Hence a comparison to local values can also be expressed with pattern matching.

The rule for receive actions is a little bit more difficult, but this cannot be impeded, because of the Erlang semantics. We want to allow convenient access to all elements of the mailbox with pattern matching. The elements of the mailbox are successively matched against the patterns. If a value matches one of the patterns it is removed from the mailbox. The sequencing messages do not have to be considered. But our specification can have overlapping patterns. It is possible that multiple patterns match this value. This is formalized by the function mailmatch, which yields a set of triples. Every element consists of the matching position in the patterns, the position of the matching value in the mailbox and the corresponding variable bindings. The position in the mailbox is identical for all yield triples.

$$\begin{aligned}&\mathrm{mailmatch}((pat_1, \ldots, pat_m), (v_1, \ldots, v_u)) \\ &\quad = \{(i, j, \rho) \mid \mathrm{match}(pat_i, v_j) = \rho \neq \mathrm{Fail} \text{ and} \\ &\qquad\qquad \mathrm{match}(pat_k, v_l) = \mathrm{Fail}\ \forall k < i, l < j\}\end{aligned}$$

A value can be sent to another process, if the variable for the destination is bound to a pid. The semantics of the expression, to be sent, is added to the corresponding mailbox. Finally, new processes can be spawned. Here we only present the rule, where the pid of the new process is stored in a variable. the anonymous process generation is defined analogously. In the here presented formal semantics runtime errors are missing. They can occur when a value should be sent to a non-pid, for example a natural number, or a matching fails. For shortness we do not formalize these rules here.

In the tool this semantics is used for the simulation component. The user can execute paths of the operational semantics. Therefore he has to decide two different kinds of non-determinism. First, which process performs a step in the interleaving semantics. Second, which action does a process perform if its behaviour is non-deterministic. The interpretation of the sequential functions cannot be completely given, because this would be their implementation. Hence, the user is asked for their semantics on the actual values, if this is needed. The chosen path is displayed as a message sequent chart and the actual state by coloring the actual states of the processes in their specifications and the corresponding bindings for the variables. The simulation can be used for debugging the protocol. This is better, than a debugging the implementation, because here also unfair paths can be tested. In the implementation the path which can be tested is influenced by the scheduler. A system working correct on one architecture can for example deadlock on another one. To support the tester, it is also possible to backtrack paths and chose other branches or perform random traces. Furthermore the specification is much smaller than its implementation. Finding protocol errors (e.g. deadlocks) is much easier on this abstraction, than in the code. In the specification only the relevant states for the concurrent behavior are considered.

For future work we want to combine this part with the Erlang interpreter, which is able to calculate the expressions, if the occurring functions are already programmed in Erlang. This would lead to a comfortable specification and programming environment. If a function is not defined yet, the user is asked for its interpretation for the concrete values. This

behavior is stored in an Erlang module. If the same calculation is needed once again, the function is already defined for these values. Furthermore, this function also yields a (finite) prototype definition for the real implementation of the function. For testing the behavior of this prototype definition can be compared with the implementation of the function.

4 CODE GENERATION FROM THE SPECIFICATION

In [DG86] Dowling and Gallier present a continuation semantics for flow graphs. This continuation semantics defines a system of equations for the semantics of a flow graph which can be interpreted as a tail recursive functional program. Consequently we can give an easy translation from a flow graph into a tail recursive functional program with the same semantics as the flow graph. We can use this translation for the generation of Erlang code (or a skeleton for an Erlang program) from the process specification. The states of the system are represented by functions. The parameters of the functions are determined by the knowledge of the corresponding states. Note that the global knowledge has to be included, because in functional languages global variables do not exist. In Erlang a global registration of processes is possible, but the use of global knowledge could yield conflicts and should be prevented. As a convention we use the first arguments of the functions for the global knowledge.

The body of a function implements the actions of the outgoing arcs and then the function of the destination state is called. If the process graph contains a deterministic sequence this is translate into a sequence of actions instead of defining a function for every state. A state with multiple outgoing receive actions is implemented as one receive statement. The code for the right hand-side of a pattern is the implementation of the destination state of the corresponding receive action or a call to the function of this implementation. Only states which are reached by multiple arcs have to be represented by their own function.

The result is a tail recursive Erlang skeleton in which the sequential parts or parts that have not yet been specified have to be added. Since Erlang optimizes tail recursion, the execution is efficient and there is no stack overflow after some recursive calls as it would happen in imperative languages.

As an example we present the Erlang code for the database. Only the initial state (multiple arcs lead to the initial state) gets a special function to prevent a blow up of the program size.

```
-module(database).
-export([start/1]).

start(DB) ->
   receive
     {allocate,K,P} ->
       case lookup(K,DB) of
         {succ,V} -> P!allocated,
                     start(DB);
         fail -> P!free,
                 receive
                   {value,V,P} ->
                     start(addElem({K,V},DB))
                 end
       end;
     {lookup,K,P} -> P!lookup(K,DB),
                     start(DB)
end.
```

The code can automatically be generated by the tool and be executed by an Erlang interpreter after adding the called functions for sequential calculations.

5 VERIFICATION

In [Huc99] we presented an approach for the formal verification of Erlang programs using abstract interpretation and model checking. In general it is undecidable if a given program satisfies a property expressed in a temporal logic like linear time logic (LTL) [LP85, Var96], because for example termination can be expressed and an automatic verification contradicts the halting problem.

Although we present a specification formalism in this paper and not a programming language, the same problem appears. As other specification formalisms like SDL and UML our approach is very expressive and model checking for temporal properties in general is undecidable. We have data structures and numbers and can calculate on them.

In [Huc99] we defined a framework for abstract interpretation of Erlang programs, which is based on the standard operational semantics with respect to an abstract interpretation of values, predefined functions and constructor functions. The abstraction is defined in such a manner, that the abstract operational semantics includes all paths of the standard operational semantics, but is under some assumption finite state. A property expressed as an LTL-formula is fulfilled, if all paths of a system fulfill the property. Hence, if the abstract operational semantics fulfills such a property, the property is also proven for the real operational semantics, in other words for the system. Because the abstract operational semantics of the system is finite state under some assumptions, we can use model checking as an automatic proof technique.

We transfered this verification technique to ESM and added it as a component to our tool. In this component the user can define a finite domain abstract interpretation and the operational semantics is interpreted with respect to this abstract interpretation. To understand the abstraction, the user can run the simulation on this abstract domain and she can use it to prove properties of the specified system. If a given specification creates only a finite set of processes at runtime[3] the abstract operational semantics generates a finite transition system and LTL properties can automatically be proven with model checking. If the abstract operational semantics fulfills the property also the system fulfills it. Otherwise a counter example is generated. This path, can either be a counter example for the specified system or an additional path yield because of the additional nondeterminism in the abstraction. The user can inspect this with the simulation and try to refine the abstraction for more determinism.

[3] Also the mailboxes have to be restricted in size. See [Huc99] for details.

A basic abstract interpretation can automatically be generated from a given specification in the tool. Its domain is given by values occurring as patterns in the program. All values that match a pattern are identified by the same value. The parts that differ are represented by ?. For the communication the different pids are essential. Hence pids are also allowed in values of the abstract domain. Values with different pids are distinguished. Additionally an ordering $\preceq$ on this abstract domain is predefined which expresses that an abstract value is more precise than another one. The abstract domain is downwards closed with respect to $\preceq$.

As an example the pattern `{allocate,K,P}` occurs in the specification of the database. In the predefined abstract interpretation the value `{allocate,?,@4}` represents the value `{allocate,[],@4}` as well as `{allocate,4,@4}`. A less precise representation of these values is `{allocate,?,?}` and the worst representation is ? (? $\preceq$ `{allocate,?,?}` $\preceq$ `{allocate,?,@4}`). But during the computation of the operational semantics over the abstract domain, we may lose information of the concrete values, for example when it is stored in a data-structure and took out again. We have to calculate on worth representations.

The functions specifying sequential behaviour yield in this basic abstract interpretation ? for any argument. No information of there behaviour is considered. This basic abstract interpretation can then stepwise be refined. Therefore the user can define the result for relevant arguments.

The logic LTL is defined for unlabeled transition systems with state propositions. The operational semantics defines a labeled transition system. For the verification the user can add propositions to the states, which hold if one of the processes is in this state at runtime. For the propositions we also use Erlang terms, because it is easy to use the known expressions and we can also integrate pids into the propositions.

As an example we can add the following propositions to the specification of the database process:

STATE	PROPOSITION
s1	`{allocate,P}`
s2	`allocated`
s3	`{set,P}`

Then we can prove that a system consisting of a number of clients and a database process guarantees mutual exclusion for writing accesses to the database. This property can be expressed with the following extended LTL formula:

$$\bigwedge_{\substack{p \in Pid \\ p' \neq p}} G\,(\{\texttt{allocate},p\} \rightarrow (\neg\{\texttt{set},p'\}\ U\ (\{\texttt{set},p\} \vee \texttt{allocated})),$$

This formula says, that if a client p allocates a key, no other client sets a value in state s3 until p sets it or the key is allocated. For a finite number of processes only a finite number of pids exist. Hence we can translate this extended LTL formula into an LTL formula. With the basic built in abstract interpretation and the LTL model checker this property can automatically be proven. Because the property holds for the abstraction, it also holds for the specified system and also for the implementation, that can be automatically generated. Hence mutual exclusion is also proven for the implemented system.

6 EXPERIENCES AND FUTURE WORK

We presented a specification formalism especially tailored for Erlang. Processes are specified with labeled transition systems. The states are labeled with variables representing the knowledge a process has in these states. The transitions are labeled with actions similar to the concurrency constructs of Erlang or function calls representing sequential behaviour. A main advantage of ESM is that the developed systems are clearly structured. The protocol/communication part is specified with ESM and the purely sequential parts are programmed in a pure functional language. We showed how ESM can be used in practice and how Erlang code can automatically be created. We developed a tool which provides the following components:

- A graphical editor for the development of specifications. The defined specifications are automatically checked for consistency, for example "Is a variable sent to another process known in the previous state?"
- A simulator for testing the behaviour of a specification. There are often some paths in the specification you do not consider in development. The simulation offers all possible steps and the user can find wrong paths, for example a process has terminated, but another one still sends messages to it.
- A code generator, which produces executable Erlang code for the specification. Only the sequential parts of the program have to be programmed, with respect to their textual specification.
- A verification component. The user can define abstract interpretations, which yield finite state transition systems. She can also specify properties in the expressive logic LTL and verify them automatically for the resulting transition system with model checking. If the abstraction fulfills the property also the system and its automatically generated implementation does.

We have not used the tool for the development of larger systems yet. But in two practical course, we saw that the use of ESM helps structuring the developed distributed systems. Two Groups of four students had to develop a communication system where a server provides services like talk, chat, mail and two graphical distributed real-time game. They specified the systems on a blackboard with ESM and did the simulation in a group, where every student behaved as a process. In this simulation many mistakes occurred, because every student tried to damage the behaviour of the whole system. The systems, the students designed and implemented, had an almost professional character although the time they needed for the development was very short. We also developed two distributed games:

- Atomic Bomberman: Up to four players run through a maze and use bombs to get through or fight against the other players.
- Swinx: An arbitrary number of players is connected in a ring. Each player has four seesaws. From the roof different weighted balls can be thrown down with a wagon.

In dependence of the different weight of the two sides of the seesaws, the balls fly over the board (hopefully to the neighbors). If a column of balls reaches up to the roof, this player has lost. He is eliminated from the ring. To avoid this the balls have different colors. More than three balls of the same color in one row resolve.

To guarantee realtime–requirements, very complex protocols are needed in the implementation of the games. Again we got a clear structure using ESM.

In this practical course we also saw that the ESM specifications are also a good documentation of the systems. The students had to extend older protocolls when new features were added to the communication system. Especially the graphical notation helped to understand the behaviour of the processes and extensions could easily be made.

In the actual implementation the verification component is not efficient enough to be used in the context of larger systems. But it can be used for the verification of smaller subprocesses like the database process. For the verification of larger systems we have to implement partial order reduction [Pel94], as done in SPIN [Hol97]. Another problem with the model checker is that it excepts only LTL formulas. An extension to extend formulas would be interesting. A first approach would be a stupid translation, which yields a combinatorial blow up of the formula. Hence, we want to estimate if the model checking algorithm can profit of the compact structure and verify the property expressed in extended LTL more efficiently.

Another point missing in the tool is the specification of distribution. The user can already specify sending to remote nodes and spawning processes on remote nodes, but Erlang's mechanisms for programming robust applications, like process linking, error handling or monitoring are not provided yet. But for the development of real applications this is needed.

Finally, it would also be interesting to extend our tool to a real development platform for Erlang systems, where parts of the sequential behaviour are only specified and the rest is already implemented. In the simulation this should automatically be considered and the user is only asked for the semantics of a function call, if the corresponding code does not exist. In this environment it would also be possible to change the protocols of a developed system and leave the sequential parts unchanged. Hence, systems could be maintained or extended very easily.

At the moment we are also investigating, if it is possible to generate ESM–specifications from existing Erlang code. This would help to understand the communication structure of existing software. The process specifications are much more compact, than the corresponding source code.

REFERENCES

[Alh98] Sinan Si Alhir. *UML in a Nutshell.* O'Reilly, Sebastapol, CA, 1998.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[DG86] W. F. Dowling and J. H. Gallier. Continuation semantics for flowgraph equations. *Theoretical Computer Science*, 44(3):307–331, 1986.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[Huc99] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[ISO89] ISO. ESTELLE: A formal description technique based on an extended state transition model. Technical Report 9074, ISO, 1989. 1989.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT-SIGPLAN, ACM Press.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–390, 1994.

[ST87] R. Saracco and P. A. J. Tilanus. CCITT SDL: Overview of the language and its applications. *Computer Networks and ISDN Systems*, 13:65–74, 1987.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238ff., 1996.

[vEVD89] Peter H. J. van Eijk, Chris A. Vissers, and Michel Diaz. *The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project.* North-Holland, New York, 1989.

A INTRODUCTION TO ERLANG

Erlang [AWV93] is a functional programming language developed by Ericsson. The reduction strategy is call–by–value and Erlang is untyped. Erlang has additional features for concurrent and distributed programming. During the execution of an Erlang program many processes run concurrently. Processes can communicate with each other via asynchronous message passing. Therefore every process is identified by a unique process identifier (*pid*), to which messages can be sent: $pid!v$. Every process has a mailbox, in which all incoming messages are stored. Erlang provides a convenient access for a process to the elements of its mailbox with pattern matching. With the statement

$$\texttt{receive}\ pat_1\texttt{->}e_1;\ldots;pat_n\texttt{->}e_n\ \texttt{end}$$

the elements of the mailbox of a process are stepwise matched against all the patterns $pat_1, \ldots, pat_n$. The first element of the mailbox that matches a pattern pat_i with a substitution ρ is removed from the mailbox. The whole receive statement reduces to $\rho(e_i)$.

This access to the elements of the mailbox with pattern matching makes concurrent programming very easy in Erlang. You can pick out the messages of the mailbox you are interested in. All other messages are stored in the mailbox and can be handled later on. Hence programmers in Erlang do not have to consider all interleaving sequences of incoming messages. Programming gets easier and programs shorter.

New processes can be created with the statement

$$\texttt{spawn}(m, f, [v_1, \ldots, v_n])$$

and the reduction of the newly created process starts with the application $m : f(v_1, \ldots, v_n)$, where f is an exported function of the module m and the arguments $v_1, \ldots, v_n$ are reduced values. The functional result of the **spawn**–statement is the pid of the created process. This pid can be stored in datastructures or sent to other processes. Every process can also access his own pid, with the function `self()`.

For distributed programming Erlang uses the same process concept as for concurrent programming. In a distributed Erlang system multiple nodes exist on maybe multiple computers in a network. On an Erlang node processes can be executed concurrently. To distinguish nodes located on the same computer, every node gets a name when it is started. Communication between processes executed on different nodes stays the same as before. Therefore the pid of a process also contains the node on which it is executed.

The only difference in distributed programming are the creation of processes on other nodes and the communication between two independently started Erlang processes. Erlang has an extended **spawn**–statement

$$\texttt{spawn}(n, m, f, [v_1, \ldots, v_n])$$

with the same behaviour as **spawn**, except that the new process is started on the node n, which is the name of the node followed by the symbol @ and the hostname of the computer the node is located on. With this extension it is very easy to distribute processes developed in a concurrent environment, for example to optimize speed of an application or to guarantee fault tolerance to the failure of one node.

Creating processes on different nodes is an expressive mechanism for distributed programming, but some applications are inherent distributed and a hierarchical distribution is not possible. Examples for this are telephony or talk, name–servers or cash dispensers. Here processes are started independently and communication between these processes has to be initiated at runtime. In Erlang this can be realized with global registration of processes on a node. With the statement `register(`*name*`,`*pid*`)` the *pid* is globally registered as *name* on the node where the registration is executed. For the communication between two independently started processes on different nodes we can use the extended send operation `{`*name*`,`*node*`}!`v. The message v is sent to the process, which is registered on *node* as *name*. Usually this kind of communication is only chosen for the first contact. Then the pids are exchanged and all further messages are sent to pids, as in concurrent programming.

CROSS-MODULE OPTIMIZATION

Thomas Lindgren
e-mail: ftl@acm.org

September 14, 2001

Abstract

This paper shows that automatic program optimizations that work over module boundaries yield good to excellent speedups on realistic Erlang applications. The optimizer works as a translation from Erlang to Erlang.

The preliminary results are excellent: as an example, the beam compiler is nearly four times faster than without cross-module optimization, while gen_tcp is more than two times faster. Lesser but still significant speedups are shown for other applications.

INTRODUCTION

In this paper, I will describe a new Erlang optimizer, OM. OM profiles the application to detect the 'hot core' of the program, that is, the functions and modules that are most executed. OM then heuristically optimizes the hot core. Subsequent to this, conventional compiler technology can be applied.

OM (for 'optimization manager') is an Erlang-to-Erlang optimizer, itself written in Erlang, and operates entirely without extending the Erlang/OTP R7B2 runtime system. OM consists of roughly 20,000 lines of code, and was developed for flexibility and ease of debugging rather than raw efficiency. While OM successfully optimizes realistic code, it should still be considered a prototype for these reasons.

OM works in two steps: first, the application to be optimized is profiled in several ways to tell where to put in the optimization effort. Second, the optimizations are applied using the found profile information. After this point, the application is run normally.

OPTIMIZATIONS

The fine print of OM:s optimizations are beyond the scope of this paper. Instead, we will show what is done by example.

Profiling

Profiling information is collected by generating a profiling version of the application. In the profiling version, code is deposited that records information into an ets table when executed. For example, when a clause body is entered,

its associated counter is incremented; when a remote call is done, a counter associated with the pair of calling and called module is incremented.

The information recorded is shown below.

1. how often is each (function/case/if/receive/fun) clause executed?
2. how often is a call site visited?
3. when an apply is done, what functions are applied?
4. which modules call each other, and how often?

Higher-order function removal

Higher-order functions are removed heuristically as follows: OM detects calls to common functions defined in the 'lists' module, and emits an equivalent local function along with a call to the new function. This is done only when a fun is supplied in the right position, and permits OM to simplify the code considerably.

The execution profile can be used so that seldom-executed calls to 'lists' remain unchanged, which saves some code. Currently, OM transforms all suitable calls, regardless of how often they are executed.

As an example of higher-order removal, consider the following code from beam_asm.erl:

```
...
encode_arg({field_flags, Flags0}, Dict) ->
    Flags = lists:foldl(fun (F, S) -> S bor flag_to_bit(F) end, 0, Flags0),
    ...;
```

Higher-order removal converts this into a call to a new function, and simplifies the new function:

```
...
encode_arg({field_flags, Flags0}, Dict) ->
    Flags = lists_foldl_1(0, Flags0),
    ...;

%% specialized lists:foldl
lists_foldl_1(Acc0,[X|Xs]) ->
   Acc1 = Acc0 bor flag_to_bit(X),
   lists_foldl_1(Acc1,Xs);
lists_foldl_1(Acc,[]) ->
   Acc.
```

(While the fun in example above has no free variables, OM in general passes free variables as extra parameters to the new function. When there are few or no free variables, the result is a gain.)

Higher-order removal trades the cost of more code for the gain of removing the use of fun:s and apply. However, the gains seem to dominate in practice: the beam compiler uses the lists module heavily, and runs dramatically faster after higher-order removal (see the evaluation section below).

Apply open-coding

Apply open-coding converts a call apply(M,F,As) into a case statement. For example, asn1rt has the following central function for encoding an ASN.1 value. Note that, to make things difficult, the function name to be called is constructed dynamically – this means an ordinary compiler or analyzer won't know what to do.

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch apply(Module,Call,[Term,[]]) of     %% apply
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
{Bytes,Len} ->
    {ok,Bytes};
X ->
    {ok,X}
    end;
encode(ber_bin,Module,Type,Term) -> ...;
encode(per,Module,Type,Term) -> ... .
```

After profiling, OM knows that Call is almost always 'enc_LDAPMessage' and can rewrite the function as:

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch
         (case {Module,Call} of
      {'LDAPv2',enc_LDAPMessage} ->
          'LDAPv2':enc_LDAPMessage(Term,[]);  %% speculative call
      _ ->
                    apply(Module,Call,[Term,[]])) of
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
{Bytes,Len} ->                      %% <- dominant case
    {ok,Bytes};
X ->
    {ok,X}
    end;
encode(ber_bin,Module,Type,Term) -> ...;
```

```
encode(per,Module,Type,Term) -> ... .
```

Outlining

After apply open-coding, OM tries to get rid of cold code inside functions by outlining clauses. Outlining works by turning clauses that seldom are executed into new functions, so as to separate them from the hot code. There are two advantages:

1. Outlined clauses reduces function size, and small functions can be inlined more often.

2. Pattern matching improves when outlined patterns are moved out of a switch.

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch
         (case {Module,Call} of
      {'LDAPv2',enc_LDAPMessage} ->
          'LDAPv2':enc_LDAPMessage(Term,[]);  %% speculative call
      _ ->
                  apply(Module,Call,[Term,[]])) of
        {Bytes,Len} when Bytes =/= 'EXIT' ->  %% note new guard
            {ok,Bytes};
        _X ->
            'OUTLINED-1'(_X,Module,Call)
    end;
encode(Method,Module,Type,Term) ->
    'OUTLINED-4'(Method,Module,Type,Term).

'OUTLINED-1'(_X,Module,Call) ->
    case _X of
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
X ->
    {ok,X}
    end.

'OUTLINED-4'(Method,Module,Type,Term) -> ...
```

Note that OM has reordered the clauses so that the common case is first in this switch. In this particular case, OM can then move all the cold cases into a single common function and pattern matching can be simplified.

Module aggregation

After outlining, each module is split, keeping the hot code in the original module and moving the cold code to a new module. The hot module is optimized (later on, perhaps native code compiled), while the cold module can be kept as compact byte code. OM does not optimize cold modules once they have been generated.

Module splitting prepares for the next step, which aggregates several modules into one. The idea is to turn frequently executed remote calls into local calls, which then can be inlined and optimized.

Erlang has hot code loading, so just putting the modules together will change how the program behaves. While an application developer can merge modules (by hand or with the help of OM), we avoid that approach here. Instead, we note that a remote call m:f(X) can be logically viewed as follows:

1. lookup the module named m;
2. lookup the function named f in m;
3. call the function

The important observation is this: since code change is quite rare, steps 1 and 2 are almost always constant. As long as M is unchanged, the call will invoke the same function. Current systems (eg, BEAM, HIPE) capitalize on this by dynamic linking to the function found in step 3 when the module is linked. But link-time is too late if we want to do compile-time optimizations. We will instead show a compile-time approach below.

Assume that we have merged with module m. We then rewrite a call m:f(X) as follows:

```
m:f(X) -->

(case latest(m) of
    true ->  f_local(X);    %% local/merged version of f
    false -> m:f(X)
end)
```

That is, if m has not been changed since we merged the current module with m, we can invoke the known version of f/1. But if m has changed, we invoke the most recent version of m via an ordinary remote call.

The effect is that, at the cost of an extra test, we have turned the remote call m:f(X) into a local call f_local(X) for as long as m is not changed. In a previous paper [5], I showed how to implement the latest(m) test as part of a linker, meaning it has a negligible runtime cost; there, I also discuss the issues of code change more fully.

Given a profile of which modules invoke each other, the aggregator decides which modules to merge. The basic concept is similar to water drops on a plate: initially, each module is in a drop of its own. Module merging then pushes the drops together to form larger drops. Aggregates are merged in order

of how often they call each other; when an aggregate becomes too large, this process stops. (Thus, outlining and splitting enables OM to aggregate more hot modules, since the hot modules shrink in size.)

Some modules are widely shared (eg, lists), but in the model above, they can merge with only one aggregate; this is inadequate, since many aggregates would benefit from merging, yet merging them all into a super-aggregate would yield unacceptable code size.

The aggregator instead *engulfs* widely shared modules, copying the module into each aggregate that needs it. The amount of copying is restricted by aggregate size, and is done in order of call frequency. (Again, code change needs to be adjusted to change multiple engulfed versions of a module at a time; I will not go into that issue here.)

As an example, consider the somewhat artificial example of aggregating modules m1 and m2, which implement a test for odd/even.

```
%% module m1
-module(m1).
-export([f/1]).

even(X) when X >= 2 ->
    m2:odd(X-1);
even(1) ->
    false;
even(0) ->
    true.

%% module m2
-module(m2).
-export([g/1]).

odd(X) when X >= 2 ->
    m1:even(X-1);
odd(1) ->
    true;
odd(0) ->
    false.
```

We merge modules m1 and m2 into a single aggregate. This consists of two trampoline modules, named m1 and m2, and a core aggregate 'A0'. The trampolines are used as interfaces to the outside world, and simply invoke the core aggregate.

```
%% trampoline module m1
-module(m1).
-export([f/1]).

even(X) -> 'A0':even(X).
```

```
%% trampoline module m2
-module(m2).
-export([odd/1]).

odd(X) -> 'A0':odd(X).

%% core aggregate
-module('A0').
-export([f/1,g/1]).

even(X) when X >= 2 ->
    case latest(m2) of
        true ->
    odd(X-1);
false ->
            m2:odd(X-1)
    end;
even(0) ->
    true.

odd(X) when X >= 2 ->
    case latest(m1) of
        true ->
    even(X-1);
false ->
    m1:even(X-1)
    end;
odd(1) ->
    true;
odd(0) ->
    false.
```

The net effect is to turn the calls to even/1 and odd/1 in the core aggregate 'A0' into local calls, which can be inlined. The latest-tests ensure that the core aggregate is still safe for hot code loading.

Inlining

Finally, profiling has shown which call sites are frequently visited, and inlines them throughout the aggregate in order of descending frequency. Remote calls cannot be inlined, since that would not respect the semantics of hot code loading, but aggregation will have turned the important remote calls into local ones.

The inliner keeps a code size budget, and stops inlining when the budget would be exceeded. The budget is consumed by each inlined call, using the size of the inlined function body.

Profile-driven inlining is often superior to static inlining, since it is applied

only at frequent calls. A static inliner must be careful not to cause a code explosion, which means it usually avoids inlining large functions; at the same time, a static inliner may zoom in on code that is seldom or never executed, merely because some functions happen to be small. In short, the benefits of static inlining are erratic, since a static inliner may work on the wrong code, or not work hard enough on the right code.

Simplification

After inlining, there often are redundancies in the inlined code. A final pass of code simplification heuristically detects and removes such redundancies. These simplifications currently consist of four categories:

1. Functions that are no longer reachable are deleted.
2. Constant and copy propagation (e.g., Y=3 and X=Y mean X can be replaced by the value 3).
3. Constant folding (e.g., some BIFs with constant arguments, guard tests known to succeed or fail).
4. Control simplification (case-of-case, pattern match simplification, nested catch, applied and known fun).

Functions become dead due to inlining and, particularly, from module engulfing: often, only part of an engulfed module is used and the rest of the functions can be deleted.

Constants and copies were not propagated or folded in the measurements discussed below, since the beam compiler already does some of this work.

Redundant control normally arises due to macro expansion and inlining. As an example, consider a function from decode1.erl after macro expansion:

```
decode_action(Flag) ->
    case if
             Flag band 16 == 16 ->
                 true;
             true ->
                 false
         end of
        true ->
    Body1;
        false ->
    Body2
    end.
```

When this function is executed, the inner if-expression returns an atom which is matched by the enclosing case-expression. Simplification short-circuits this into a single test:

```
decode_action(Flag) ->
    if
        Flag band 16 == 16 ->
   Body1;
        true ->
   Body2
    end.
```

RESULTS

The results in this section are preliminary.

We applied OM to a set of realistic benchmarks. (The benchmark programs are freely available. Please contact the author for a copy.)

decode1 (1 module): an example 'inner-loop' doing packet processing;

beam (31 modules): the beam compiler applied to the 'lists' module;

gen_tcp (7 modules): short messages over a local socket;

ldapv2 (5 modules): ASN.1 encoding and decoding using the LDAPv2 specification;

mnesia (29 modules): the mnesia system simulating a simple home location register database.

The applications above are, apart from decode1, popular OTP subsystems. This brings the advantages that we are optimizing "real programs", which still can be made freely available, and that optimizations improving them are relevant to most Erlang developers.

(The benchmarks were modified as follows: ldapv2 uses tuples as records, which seems to be a glitch in the ASN.1 compiler and which was removed; beam was turned into a sequential program to simplify measurements, by calling the compiler directly rather than spawning a process to do it; finally, a guard test in a list comprehension in beam was changed into an if-statement.)

The benchmarks were run on an IBM Thinkpad 600E with 128 MB memory and a 300 MHz processor, running Erlang/OTP R7B2 on RedHat Linux 7.1. Each benchmark was run 30-40 times to get some statistical accuracy, whereafter the results were analyzed. I compared Erlang with Erlang: beam-compiled programs with OM-optimized, beam-compiled programs. The latest-tests introduced by OM were simulated by a simple, "always-true" test, so as to not modify the R7B2 runtime system.

Speedup (runtime)		
	beam/om	
decode1	1.12	
beam	3.96	(2.94 h-o remove * 1.34 other)
gen_tcp	2.54	(2.15 if outliers kept)
ldapv2	1.01	
mnesia	1.17	(1.28 if outliers kept)

Two benchmarks, gen_tcp and mnesia, suffered in that the baseline results showed a very broad standard deviation. This meant it was difficult to compute speedups with any confidence. I thus removed outliers to shrink the standard deviation, which increased speedup for gen_tcp, and reduced it for mnesia. It is unclear why results varied so widely in the first place; my current hypothesis is that since both the offending programs are concurrent, process scheduling plays a role.

OM yields excellent results on gen_tcp and beam, while the others have significant but less spectacular outcomes.

Decode1 is improved simply by outlining improving pattern matching. Beam uses higher-order functions heavily and is improved mainly by higher-order removal. For gen_tcp, OM aggregates and inlines across four modules and an apply to arrive at a result close to the optimum. Ldapv2 yields nearly no improvement, which is somewhat surprising, since the application appears well-suited to take advantage of module aggregation and apply open-coding. The mnesia benchmark complains during profiling (only) that mnesia is overloaded; this means the profile may be inaccurate compared to normal execution. The mnesia result has not been analyzed further. From the viewpoint of OM, the mnesia result shows both that the profiling step is sometimes too expensive as implemented, and that an inaccurate profile still yields speedups.

Random examination of the generated code shows there are still numerous inefficiences to be removed. Inlining and outlining should be tuned and extended. Aggregation seems to work reasonably well for the benchmarks in question. The simplification step should be extended to employ a greater range of techniques, and should be opportunistic by working more aggressively with hotter code.

RELATED WORK

Profile-driven optimization has been successfully used throughout the 1990's for optimizing "integer programs", that is, programs that use linked data structures and complex control flow. Object-oriented languages, such as Smalltalk and SELF, were probably the spearhead in this regard. Today, compilers such as Sun's HotSpot Java compiler and Hewlett-Packard's line of workstation compilers extensively use profile information during compilation. Researchers at HP have shown that large engineering applications can be successfully optimized using profile-driven whole-application methods [2].

Most of the optimizations deployed in OM have their counterparts in other compilers. A generalization of higher-order removal was proposed by Olin Shivers in the context of Scheme. SELF and Smalltalk predict what methods are called similarly to apply open-coding [3]. Ad-hoc partial inlining – an unstructured version of outlining – has been proposed by several research groups and apparently exist in a research IA-64 compiler from Intel Microcomputer Research Labs (Dulong et al, 2000). Module aggregation (splitting, merging and engulfing) is novel. While techniques for optimization across module boundaries have been proposed, they do not deal with hot code loading and are con-

sequently more straightforward. Profile-driven inlining and optimization has been proposed and applied by, e.g., the IMPACT reseach group at University of Illinois. The simplification phase is similar in spirit to what has been used in a long line of functional compilers, ranging from Steele's Rabbit, up to today's Haskell (Santos and Peyton Jones) and SML compilers [1] [7].

To my knowledge, profile-driven optimizations have so far not been applied to functional languages. The results in this paper indicate a substantial opportunity for compiler writers.

Profiles were used in an embryonic form while testing HIPE on AXD301: only the functions that were executed during benchmarking were native-code compiled; this is basically just-in-time compilation as applied to Erlang [4].

The name 'outlining' was taken from a SIGCOMM paper by Mosberger et al [6], where it described an optimization that moved seldom-executed protocol code out-of-line.

FUTURE WORK

There are several paths worth pursuing.

First, OM is currently a prototype, and could be industrialized (a) by making it more efficient, both while compiling and while profiling, (b) by tuning and extending the optimizations, and (c) by automating its use by improving its heuristics and adding ways to re-compile the program (locally or remotely) if the profile changes radically during execution. Extensions to the Erlang runtime system are probably required to do this well.

Low-level profiling support. ets-tables are unsuitable, and the profile can be recorded as a low-level data structure rather than a term.

Better handling of modules. If a module can export several interfaces, the trampoline modules are not needed. Furthermore, the current limit on two versions of a module in the runtime system precludes transparent profile-driven recompilation.

Better handling of records. Records should have unique names throughout the system by default, so as to avoid name clashes when merging modules. The confusion of tuples with records aggravates this problem, since OM cannot rename tuple tags.

Better handling of closures (fun objects). For apply open-coding, OM needs to do function application 'by hand'. Two things would help here: some way to extract free variables from a closure, and some way to locate the code (syntax tree) for each closure at runtime.

Second, further evaluation is needed. One way to test our hypothesis about scalable compiler performance would be to apply HIPE to the aggregates generated by OM, since OM extends the scope of HIPE by enlarging functions. An

orthogonal approach would be to examine the applications for more Erlang-to-Erlang optimization opportunities. Finally, more and larger applications should be studied.

Third, this paper does not explore the possibilities of improved type information. Static type analysis of other dynamically typed languages have yielded immense profits by finding and removing redundant type tests and dead code. However, my previous experiences with type analysis module-at-a-time for Erlang have been very disappointing: modules normally export many functions and call many other modules, which leads to a nearly complete lack of useful information.

Module aggregation may improve type analysis results considerably, simply because it merges modules and so tends to put producers and consumers of values in the same module. It is also possible to exploit profiles to further sharpen the results of type analysis within an aggregate. (An example of this is making private copies of frequently executed functions so that analysis results for the copies are more precise, hopefully yielding more optimizations in frequent code.) Sven-Olof Nyström has written a sophisticated type analyzer for Erlang, but compiler support is needed to make use of the type information.

Fourth, profile-driven optimization can be applied more aggressively than in this paper, in two ways. Conventional optimizations can be done more aggressively if we know that the code is executed often. New, profile-driven optimizations are also possible. We are currently exploring ways to improve module aggregation further with profiles.

CONCLUSION

Real applications are structured as a collection of modules, and running code frequently switches between them. In order to optimize programs well, a compiler must look across module boundaries. In Erlang, a great hindrance to this is that modules can be replaced at runtime.

We have described and evaluated OM, a profile-driven optimizer that aggressively and safely splits, merges and engulfs modules to capture the hot core of an application. The hot core is then optimized by higher-order elimination, inlining and simplification of the resulting code. The results are very encouraging, ranging from no change to nearly 4 times faster than the baseline on realistic code.

OM helps per-function native code compilers indirectly, since functions increase in size from inlining. Furthermore, hot and cold code is separated, meaning the compiler can spend its time budget optimizing the hot code.

OM also assists per-module compilers by aggregating modules into larger units. While we are currently unaware of any such Erlang compiler, we expect per-module compilation to have sufficiently compelling performance advantages that per-module compilers will appear.

ACKNOWLEDGEMENTS

The AXD301 project provided a helpful environment for the initial evaluation of HIPE on real applications, which was done while I was at SARC, an Ericsson research laboratory. My thanks in particular to Mats Cronqvist, Kurt Johansson, Thomas Lindquist, Peter Lundell, Jan Roth, and Ulf Wiger.

Discussions with the HIPE group were very helpful. Mikael Pettersson suggested that "partial inlining" should inline only some clauses, rather than an entire function body, a seed which germinated with outlining. Richard Carlsson wrote the first implementation of a module merger. Sven-Olof Nyström has been an excellent sounding board for far-flung ideas.

The support of HIPE by Bjarne Däcker, Bengt Jonsson, Håkan Millroth, LM Ericsson AB and NUTEK through ASTEC during and after my stay at Uppsala University was crucial as a foundation to this work and is highly appreciated.

STOCKHOLM, SEPTEMBER 2001

REFERENCES

References

[1] A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] A. Ayers, S. de Jong, J. Peyton, R. Schooler. Scalable cross-module optimization. In Proc. PLDI'98.

[3] C. Chambers, D. Ungar, E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language. In Proc. OOPSLA'89.

[4] E. Johansson, S.-O. Nyström, C. Jonsson, T. Lindgren. *Evaluation of HIPE, an Erlang native code compiler.* ASTEC report 99/03, Uppsala University 1999.

[5] T. Lindgren. *Module merging: aggressive optimization and code replacement in highly-available systems.* Technical report 154, Computing Science Department, Uppsala University, 1998.

[6] D. Mosberger, L. Peterson, P. Bridges, S. O'Malley. Analysis of techniques to improve protocol latency. In Proc. SIGCOMM'96.

[7] D. Tarditi. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* Ph.D. thesis, report CMU-CS-97-108, School for Computer Science, Carnegie Mellon, 1997.

The EC Erlang Compiler

Maurice Castro
maurice@serc.rmit.edu.au

Software Engineering Research Centre
Level 3, 110 Victoria St
Carlton, VIC, 3053, Australia

Abstract

The EC compiler generates native machine code for Erlang programs. In addition to its role as a core compiler technology in the Magnus massively scalable computing platform, it allows stand-alone executables to be generated, permitting easy distribution of Erlang programs on conventional computing platforms.

This paper provides the motivations behind the development of EC and describes the structure of the compiler.

We intend releasing EC under a 'Berkeley' style licence and encouraging user community participation in its development (http://www.serc.rmit.edu.au/~ec/).

This paper is a condensed version of SERC Technical Report SERC-0128[8].

1 Introduction

The Magnus massively scalable computing platform is a high performance message based supercomputer targeted at the business transaction market. Erlang[7] is a natural choice as the application programming language for message passing architectures. A survey of the currently available Erlang compilers and platforms – Erlang/OTP[3] (JAM and BEAM), Stand Alone Erlang[6] (SAE), HiPE[15], ETOS[9] and Gerl[18] – showed that they were unsuitable for use in the Magnus architecture and the EC project was commenced to provide a retargetable optimising native code compiler for Erlang.

In addition to its core role in the Magnus platform, EC generates stand-alone native programs from Erlang that can be easily distributed by software developers.

EC has been built with conventional compiler generation tools to encourage the participation of the user community in its further development. It is intended to be released under a 'Berkeley' style open source licence.

This paper will give a brief overview of Magnus architecture (section 2, and survey existing Erlang compilers (section 3). The properties and difficulties we found with using them in the Magnus architecture are identified.

The decision to create an Open Source compiler project has influenced the design philosophy. The resulting iterative, small, short term goal approach is covered in section 4. The role of testing and the directory layout is also covered.

The internals of the compiler (section 5), the runtime library (section 6) and the tool set used (section 7 are discussed in detail.

Both detailed future plans (section 8) and the immediate deficiencies (section 9) of EC are commented on.

2 Magnus

Magnus[2] is a combination of three technologies:

- A programming style that separates parallel activities.

  The key to scaling systems is identifying parallel activities. Although we have no magic bullet for identifying these activities, we have

found that a message passing environment with fast transfer speeds and low cost processes allows the easy development of scalable programs.

- A language that inherently supports message passing semantics.

  Erlang's low cost processes and easy messaging semantics make it ideal for writing the parallel activities that are required to allow massive scaling. Erlang is the core language of the system.

- An interconnect that allows messages directed to different locations to not block each other.

  The interconnect is the passive core of a WDM switch - the Wavelength Division Multiplexer (see figure 1).

Furthermore, by accepting the possibility for lost messages at the application program level significant performance and system stability improvements can be made.

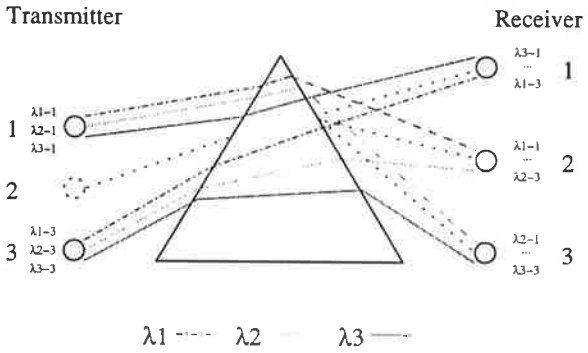


Figure 1: Schematic representation of a 3 Wavelength WDM switch

The remainder of this section will discuss the messaging environment, hardware and developing software for the Magnus platform.

2.1 Messages

A key feature of the Magnus asynchronous messaging system is that we do not consider messages must be delivered at all costs. Many network protocols incorporate complex mechanisms designed to ensure that messages that are lost through congestion or hardware errors are automatically resent. Unlike these protocols Magnus only guarantees that a message will either be correct or not delivered at all. Application programmers determine which messages require reliability and can use application level libraries to implement reliable messaging where required. This philosophy allows the hardware to be greatly simplified by eliminating buffering. The inherent high reliability of the optical components is fully exploited as the need for the latency adding protocol level acknowledge (*ack*) messages and additional buffering is removed.

A large class of messages do not benefit from a protocol based retransmission strategy when a congestion based failure occurs. These types of messages include:

Informational messages - Some messages convey information that is not required for the continuing safe or correct operation of a process. These messages may be discarded.

Timely messages - Some information has a short useful lifetime. Losing this data may be less harmful than getting delayed data.

Results of a functional conversion of data - This data can be regenerated at any time by supplying the same set of input data. Responsibility for recovery of this data can often be deferred to the initiator of the operation.

Overall system reliability is improved as the Magnus system can be efficiently programmed in a way that does not introduce deadlocks. When a message buffer overflows messages are lost. There is no failure or blocking of the sender, receiver, or node. Thus the potential for recovery – at the application program level – by the process that has either been inundated by messages or is having its messages ignored exists.

2.2 Hardware

The hardware of the platform (see figure 2) consists of a number of shared memory multiprocessors (SMP) linked by a message passing interface. The non-blocking optical switch is used to pass messages from interface to interface. Information is passed between machines at the machines' bus speed eliminating the need for buffering on the interface cards.

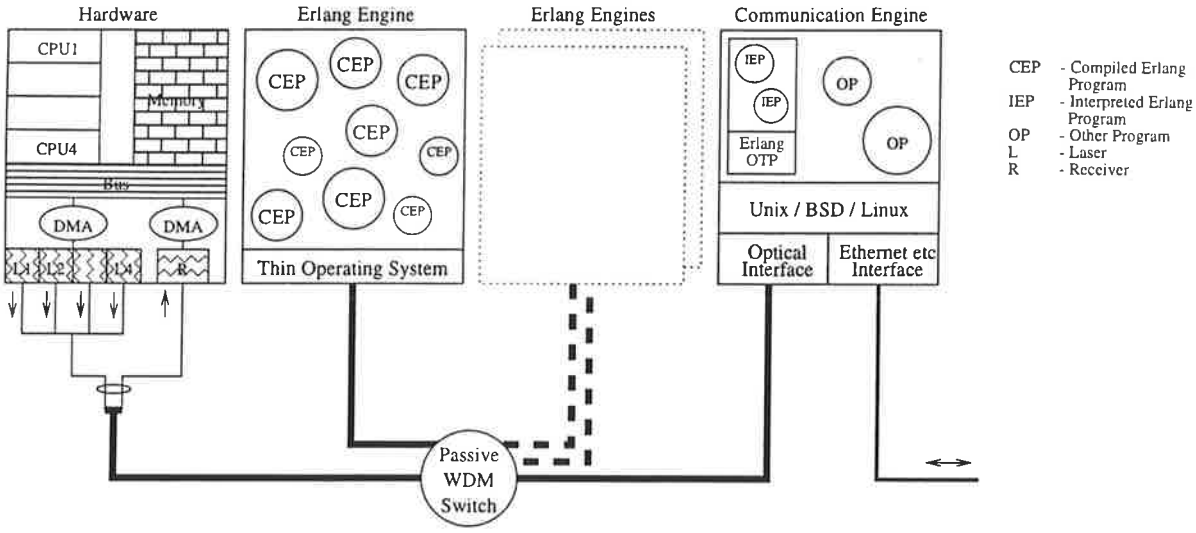


Figure 2: The Magnus Architecture

The message passing architecture and non-blocking switch allow considerable scaling as unlike bus architectures parallel unrelated tasks need not interfere with each other when communicating at the same time.

The interface cards act as Direct Memory Access (DMA) processors, freeing the main processors of the need to perform communication tasks. The interface cards can only receive a single message at a time, hence collisions occur when 2 ports try to talk to the same port at the same time. A collision results in the loss of the colliding messages.

As each machine is equipped with a bank of lasers on its interface card, a machine can broadcast to all or some of the machines connected to the non-blocking switch. This facility is particularly useful for starting a group of parallel worker processes on a task.

The majority of the SMP machines are configured as Erlang Engines (see figure 2). These systems have a minimal operating system and run compiled Erlang programs. The remainder of the systems are configured as Communication Engines. These systems have a fully fledged operating system and run interpreted Erlang and any programs required in other programming languages. The Communication Engines perform protocol intense or operating system service intense tasks, while the Erlang Engines run Erlang programs as quickly as possible.

2.3 Software

All software to be run on the Erlang Engines are initially developed under the Erlang/OTP environment to make use of the advantages of it provides such as a fully fledged development and debugging environment. When the programs are developed they are recompiled with an optimising compiler and deployed on the Erlang Engines. On the Erlang Engines the programs can gain performance from the purpose built support environment.

This approach aims to capture the full advantages we have found in developing in Erlang while supporting high performance environment. To achieve this aim it is necessary for the dialect of Erlang accepted by the optimising compiler to be a subset of the interpreted language. Furthermore, it is extremely desirable to implement as many of the features of the interpreted language as possible to ensure that programs do not have to be majorly rewritten for them to compile for the Erlang Engine.

For the Erlang Engine environment the compiler:

- may be slow - as the programmer typically interacts with the Erlang/OTP compiler
- must produce code that runs quickly

- must support a very large subset of the Erlang/OTP compiler's language
- must produce code that does not require the Erlang/OTP runtime to operate

3 Survey

Four projects with the potential to produce an Erlang compiler suitable for use in Magnus were identified: OSE, HiPE, ETOS and Gerl. Each of these takes a different approach to generating executable Erlang programs. Their features and suitability are discussed below.

3.1 OSE

The current implementation of Open Source Erlang[3] (OSE) is based on the BEAM machine a threaded interpreter. Earlier releases incorporated the JAM machine. JAM was based on the Warren Abstract Machine. JAM is no longer supported. This release of Erlang is robust and provides an excellent development environment. It does not produce standalone code.

Stand Alone Erlang[6] (SAE) is a package based on OSE that groups the interpretive environment from OSE with compiled BEAM files allowing a single executable to be constructed. Although a single executable is produced this code depends on a large number of operating system facilities and is still interpreted.

3.2 HiPE

High-Performance Erlang[15] (HiPE) uses native code to improve the speed of the existing BEAM compiler. It produces code that is seamlessly called from the interpreted environment. This compiler does not appear to produce code that operates independently from the interpreted environment.

The current release v0.92 only supports the UltraSPARC architecture although an x86 implementation has been promised for some time.

3.3 ETOS

The ETOS Compiler[9] converts Erlang into Scheme, the Scheme is compiled with the Gambit Scheme Compiler to generate C code which is then compiled to a native executable. This process produces native machine code.

The compiler is supplied as either a pre-built binary or as source code. We were forced to select the source release, even though it was significantly older than the binary release, as it was likely that we would need to make changes to the compiler to support the Magnus environment.

The source release (Version 1.4) of ETOS[10] does not implement some significant parts of the language of particular note are the operators *and*, *or*, *band*, *bor* and '−−'. Furthermore we had significant difficulty with performing all the steps required to compile even the supplied test programs under Solaris.

3.4 Gerl

Geoff's Erlang[18, 19] (Gerl) converts a large subset of the Erlang programming language into C code which can then be compiled into native machine code.

This system was developed by a past staff member of our research group and is released under the GNU Public Licence.

3.5 Conclusions

Gerl would seem to be the obvious candidate for a native code generating compiler to be used in the Magnus project as it generates native code and has source code available. The age of the implementation and the subsequent changes in the Erlang language were a severe disincentive to adopting and extending Gerl. The final reason for not choosing it was too much knowledge about how it was implemented. Having watched a very talented programmer cajole C++ into interacting correctly with Flex and Bison we felt that making changes to Gerl could be quite difficult.

The complicated compilation process, and our initial poor out-of-the-box experience combined with the requirement for a commercial licence for a compiler that would require significant work persuaded us not to pursue ETOS.

As neither of the other alternatives appeared to generate native code, we were forced to commence another Erlang compiler project: EC.

4 Development Philosophy

The development of EC will be / has been an iterative process with priority given to implementing the correct and complete semantics of the language, and then producing faster code. Although the initial phases of the project have been accomplished by a single programmer it is intended that many developers should be able to work on several aspects of the compiler at the same time. As a result of this approach the compiler has been structured as group of independent programs with well defined interfaces and phases which perform optimisations that are clearly separated from the basic compilation of code. Furthermore, the project will be guided by a number of goals which are intended to be achievable by individuals working part time on the project. These goals will be as independent as possible to avoid sub-projects blocking each other.

A rigorous testing regime have greatly speeded the development of the compiler. Each phase of the compiler and library has its own test cases stored with it in addition to to a set of end-to-end tests. For end-to-end testing the output of each phase is compared with a sample held in the associated phase directory. This allows the impact of changes to be found rapidly.

Test targets are included in each makefile to ensure that the standard tests are both easy to run and complete.

In general, any test case that reveals a fault in the compiler is added into the collection either in the phase or library in which the fault was localised or, if more widespread, in the end-to-end cases.

5 An Overview of EC

The top level design of EC conforms to the classic model used for implementing an optimising compiler[4, 5]. Figure 3 shows the phases of compilation. EC links each phase through a textual representation of the data structures produced by a proceeding phase and then read by a subsequent phase.

Retargeting the compiler is achieved by creating new machine descriptions in the phase of compilation that generates assembler code.

The remainder of this section describes some of the major design decisions, the phases of compilation, and the representations used.

5.1 Design Decisions

5.1.1 Separate Pre-processor

The pre-processor and records were introduced into Erlang after version 4.4 and there has been some debate about their appropriateness [1]:

> Macros are at the lexical level, like C's macros, right? Why not at the syntactic level, like Prolog's term_expansion and Lisp's defmacro?
>
> TorbjörnTörnvkist : Yes, that was probably a mistake. At the Erlang user's Conf. last year (1998) Richard O'Keefe actually presented a proposal called Abstract Patterns that would make it possible to get rid of the preprocessor.
>
> Robert(RobertVirding): It won't be able to get rid of all of the pre-processor as it can't handle including files and conditional compilation. Actually I wrote epp (the Erlang pre-processor) as a joke, but some people missed the point.:-) I also missed adding macros with the same name but different number of arguments.

A separate pre-processor that implements existing record and macro functionality can be written. Alternative pre-processors with less contentious behaviours may be easily substituted.

The pre-processor is not a core part of the EC project and is not discussed further.

5.1.2 Separate Processes for each Phase

Separating the phases of compilation into separate processes ensures that loose pointers cause faults to become apparent in the phase where the fault exists, simplifying debugging task. Furthermore, memory management – particularly the deallocation of complex data structures – can be avoided in the early stages of development. Typically the majority of structures used in a phase are required from their instantiation until the end of the phase,

[1] Erlang Flaws: http://www.bluetail.com/wiki/showPage?node=ErlangFlaws

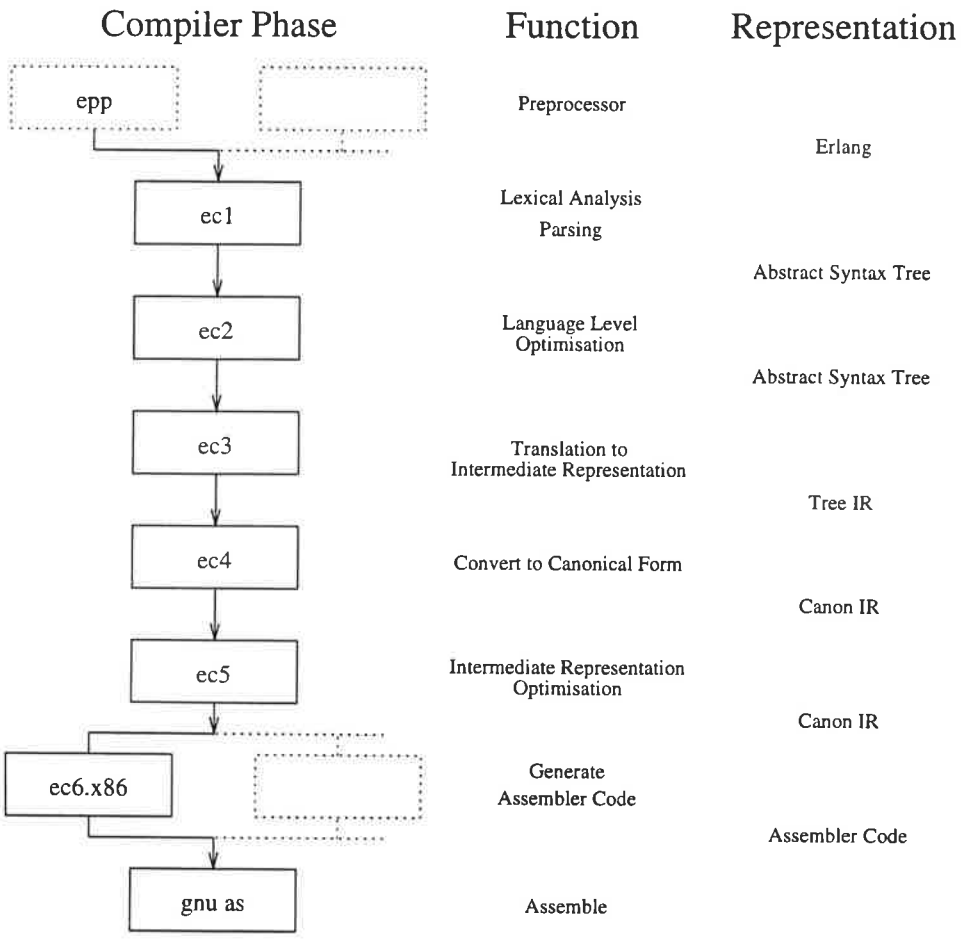


Figure 3: The Phases of Compilation in EC

so there is little advanatage in deallocating structures before the completion of a phase.

If the phases are integrated into a single process a zoned memory allocator similar to the one used in lcc[11] can be used to deallocate all the memory used by a phase on its completion.

5.1.3 Linking Passes

The use of textual representations to pass data between phases of compilation is extremely unusual. The flattening of a data structure into a disk file and then reconstructing the structure is clearly wasteful. In most modern compiler developments a pretty printer is used to output a textual representation of the data structures being passed in memory between phases of the compiler.

We chose to both read and write textual representations for passing data between phases for the dual reasons of:

- ease of debugging the output of a pass, and;
- ensuring that the textual output exactly matches the contents of the data structure

The pervasive presence of the textual form ensures that a convenient human readable document is always available when a fault is detected in the compiler. The pretty printer approach is less desirable as errors and omissions in the pretty printer can be confused with a fault in the compiler.

The interfaces between phases are contained in

libraries that are linked to by the communicating phases.

In a production environment, the compiler could be sped up by integrating the passes into a single executable and eliminating the reading and writing operations.

5.1.4 Retargeting

To retarget the compiler a new machine description is generated. Machine descriptions consist of a modified burg grammar[13] (see section 7.4) and associated C code.

EC is unusual in that it defers all decisions relating to the targeting of the compiler to the phase just before assembling code. Other retargetable compilers[5, 11] would use some information about the targeting of the compiler in earlier phases. In most cases such information would be used when activation records are generated that is when the Abstract Syntax Tree is translated into the Intermediate Representation.

EC defers targeting by generating `T_fun` elements which are interpreted by the sixth stage of the compiler to generate an activation record.

The benefit of retaining common code and behaviour in all earlier stages of compilation is at the cost of some additional complexity in the generation of assembler code and reduced opportunity to optimise to a machines characteristics at earlier stages.

5.2 Phases of Compilation

5.2.1 ec1

The ec1 phase performs lexical analysis and parses the Erlang input file to produce an Abstract Syntax Tree (AST). The grammar is closely related to the Erlang grammar found in Erlang 4.7.3 Reference Manual[17]. Some of the grammar is drawn from Concurrent Programming in Erlang[7] where the grammar provided by the reference manual was not felt to fit adequately.

The grammar used employs the changes suggested in section 2.6.3 of the Reference Manual[17]. These changes make the grammar LALR(1). The modified grammar is more permissive than it should be so additional checks are required in the code to ensure that patterns contain only the elements permitted in patterns and that the *UniversalPattern* (_) does not appear in an expression.

The implemented grammar recognises the presence of records, macros and Mnesia queries and produces an error when these are encountered. Records and macros are intended to be handled by the pre-processor. Mnesia queries may be incorporated when the support libraries are written.

We considered expressing guards as a logical expression using *logical-or* to replace disjunction. No real savings were encountered as it risked the potential for rapid short-circuit evaluation and required special treatment of the `A_or` operation for guards. We now implement guards as an `A_Expseq` so that we can choose the implementation later in the compilation process.

5.2.2 ec2

The AST generated by ec1 is transformed and optimised by ec2. A new AST is produced.

The ec2 phase performs the following transformations:

Character to Integer - Characters are converted into integers. That is `A_CharVal` elements are converted to `A_IntegerVal` elements.

String to List - Strings are converted into lists of integers. That is `A_StringVal` elements are converted to `A_List` elements containing `A_IntegerVal` elements.

Strings and characters are not fundamental types in Erlang. The 2 transformations convert strings and characters into fundamental types.

The ec2 phase performs the following optimisations:

Constant folding - constants linked by operators are merged by evaluating the operators.

The transforms and optimisations are repeated until no further changes are made to the AST (a fixed point is achieved).

5.2.3 ec3

The ec3 phase transforms an AST into an Intermediate Representation (IR). ASTs are closely related to input languages. IRs are closely related to

compiler output languages. This phase recognises sections of an AST and generates a sequence of IR elements. These IR elements are typically simple operations (arithmetic, tests and data movements) that can be easily translated into machine instructions.

Interesting features of ec3 are:

- the number of temporaries – temporaries become registers after register allocation – is notionally infinite.
- lists of import, export and defined functions are made and included in the IR representation
- `T_Fun` elements mark the beginning of functions. These elements allow function preambles to be inserted at the assembler language generation phase.
- we emit a jump to the label `ToEpilogue` when the result of a clause is calculated. The epilogue contains code used to clear the current frame off the stack and return to the caller.
- the label `Epilogue` is used to mark the end of a function. The label epilogue is followed by a jump to `ToEpilogue` at the end of each function. The jump is used to simplify the trace scheduler.
- a nesting counter is used to identify the contexts in which a variable is defined. Between each clause of a conditional language element (`if`, `case`, or `receive`), all the variables are marked as *undefined*, or if they are already *undefined* then the *unsafe* bit in the flags is set. At the completion of a conditional language element all the variables in inner contexts are marked as *inner*. They can then be used freely. The the first use of an *inner* variable results in a warning message been generated. Subsequent uses do not generate warnings. A warning is also generated if an *unsafe* variable is used (OSE halts compilation if an unsafe variable is used). We choose to allow compilation as the code could be correct, and merely unsafe. Note that before inner variables can be used they are checked to see if they have a null value. If they are successfully used in the outermost context of a function they are then marked as *normal defined* variables.
- *funs* are implemented through a data structure that contains the address to be jumped to, the name of the module, the name of the function and the arity of the function. Calls to local *funs* require only the checking of the arity and the use of the pointer. Calls to anonymous *funs* are made by checking the arity and calling the pointer. Funs in other modules require the looking up of the modules and function name.

5.2.4 ec4

The ec4 phase transforms the tree-form of the IR into the canon-form. Canonicalisation removes the `T_seq` and `T_eseq` elements and replaces them with `T_stmseq` elements. This process is known as linearisation. After linearising the code it is assembled into basic blocks which are then scheduled to reduce the number of jumps between blocks. Initialised data is separated out at the start of the phase and merged in at the completion of the phase. The order of initialised data must be preserved to ensure that data structures are not reordered.

The method closely follows that used by Appel[5].

5.2.5 ec5

The canon IR generated by ec4 is optimised by ec5. A new canon IR is produced.

At present the only optimisation performed is the folding of temporaries. A partial graph analysis is performed to identify which temporaries are aliases for each other. Duplicated temporaries are replaced with a single temporary.

5.2.6 ec6

The ec6 phase generates assembler code for each of the architectures supported. Much of the code is common between the architectures and only the code in the machine description file should need to be changed to support a new architecture.

The machine description file consists of a collection of C functions and a modified burg grammar. Figure 4 shows some fragments of the description used for the x86 processor. This compact description provides all the rules for outputing x86 assembler code. Retargeting is accomplished by rewriting these rules for a new processor type.

... Preamble

```
%start stmt
%term MOVE=1 JUMP=2 CJUMP=3 INTEGER=4 FLOAT=10 LABEL=5 CALL=6
...
%%
stmt:   CALL(label);     "call #0\n";     1
stmt:   CALL(temp);      "call #0\n";     1
stmt:   CALL(INDIR(temp));       "call (#0)\n";  1
stmt:   JUMP(label);     "jmp #0\n";      1
stmt:   CJUMP(cond, label);      "#0 #1\n";       2
stmt:   MOVE(const, dest);       "movl #0, #1\n";         1
...
stmt:   MOVE(catch, dest);       "#0\n\tmovl #r, #1\n";  1
stmt:   ACTARG(const);  "pushl #0\n";    1
...
stmt:   AND(const, dest);        "andl #0, #1\n";         1
...
catch:  CATCH;  "call CatchHead\n\tpushl %eax\n\tcall setjmp\n\taddl $4, %esp"; 1
dest:   TEMP;   "#t"
dest:   RESULT; "#r"
const:  INTEGER;         "$#i"
const:  FLOAT;  "$#f"
const:  ADD(integer, label);     "$#1+#0"
temp:   TEMP;   "#t"
formal: FRMARG; "#a"
label:  LABEL;  "#l"
result: RESULT; "#r"
integer:         INTEGER;         "#i"
cond:   EQ(temp, temp); "cmpl #1, #0\n\tje"
...
%%
```

C code ...

Figure 4: Parts of the x86 machine description file

The tree grammar describes rules for covering a binary tree. The cover is not unique and the cover with the lowest cost should be selected. Once a cover is chosen the template strings corresponding to the cover are emitted.

The burg grammar rule consists of up to 4 elements:

1. Term - a non-terminal
2. Pattern - defines a node in a tree. A pattern consists of a terminal and up to two parenthesised non-terminals
3. Template - defines a string in the `template` table corresponding to the current rule
4. Cost - defines the cost of using the rule

The elements are separated by semicolons - optional tabs may be used to improve readability. Rules containing 3 elements are assumed to have a default cost. Rules containing 2 elements are considered to have an empty template and a default cost.

When template strings are emitted the symbols in table 1 are replaced with their values.

At present a very primative register scheduling algorithm is used, a more sophisticated algorithm such as graph coloring may yield major improvements. The current mechanism allocates registers from a pool of unused registers and dumps the contents of registers when either the pool is empty or

Symbol	Value
#n	value of child n (where $0 \leq n \leq 9$)
#l	label
#r	return register
#i	integer
#f	float
#t	temporary (allocated register)
#a	formal arg

Table 1: Template substitutions

the end of a basic block is reached. The return register is allocated last. Allocating the return register last reduces the likelihood of needing to dump the register before using it.

The ec6 phase also generates the tables that are used for accessing remote functions.

An interesting feature of ec6 is that arguments are generated in the order required to push them onto a stack and numbered in C call order. This simplifies the development of both stack and register passing models. It also allows the calling of C functions easily.

5.3 Representations

5.4 AST

A single representation is used for all Abstract Syntax Trees in EC. This representation consists of 2 sequencing elements, a clause elements, and an expression element. For convenience each of these elements contains a position identifier, used to generate errors which contain the line number of the source code being compiled.

The sequencing elements are Clause Sequences (`A_ClsSeq`) and Expression Sequences (`A_ExpSeq`).

Clause (`A_Cls`) elements consist of 3 sequences of expressions. The sequences represent patterns, guards and expressions in the Erlang source.

The expression element represents all other aspects of the language.

5.5 IR

EC supports two intermediate representations known as Tree IR and Canon IR respectively. The majority of the components are common to both representations. The difference lies in the sequencing elements used:

- Tree IR uses the 2 sequence elements statement sequence (`T_seq`) and expresion sequence (`T_eseq`).
- Cannon IR uses the sequence element sequence of statements (`T_stmseq`).

The IR closely resembles Appel's[5]. The major differences lie in the implementation of conditional jumps, initialised memory locations and the implementation of functions. EC has adopted the CJUMP definition

CJUMP(cond, true_dest, false_dest)

This definition explicitly identifies the condition to the C type system. Explicit elements have been introduced to declare initialised blocks of memory, this allows later phases of the compiler to handle these elements separately, simplifying analysis. Unlike Appel's Tiger compiler, EC is not monolithic, thus EC must identify functions (`T_fun`) and arguments (`T_frmarg`) in the IR so that other phases of the compiler can generate preambles and epilogues for them. The Tiger compiler was able to avoid this as the IR needed to handle only one function at a time.

The common elements of the IR are Expressions (`T_exp`), Statements (`T_stm`) and Conditions (`T_cnd`)

6 Runtime Library

Runtime support is required for the code generated by the compiler. At present this support is targeted towards Unix to allow easy testing under the development environment. Each target platform (including Magnus) will require its own runtime support library.

The tasks of the runtime library include:

- allocating and deallocating memory.
- creating basic elements of the languages (eg. integers and tuples).
- implementing high level language operations which are not implemented directly in the compiler (eg compare and list subtract).

- implementing BIFs.
- implementing IO functions.

The majority of these operations are self explanatory. However, IO operations require further explanation. The implementation of runtime types and lists are also discussed.

6.1 IO Functions

The compiler accesses messages through a set of IO functions defined in the runtime library. The following functions must be defined to send and receive messages:

`send(pid,payload)` send the message *payload* to the process *pid*.

`MessageFirst()` get the first message in the message queue and return it. If there are no messages return `0`.

`MessageNext()` get the next message in the message queue and return it. If there are no remaining messages return `0`.

6.2 Runtime Types

Since the JAM machine, a number of bits in pointers to Erlang data items have been reserved to identify the type of an item. This mechanism can save considerable amounts of time in pattern matching code. EC uses the 3 least significant bits as type tags. The meanings of the bit patterns are summarised in table 2.

Bit Pattern	Meaning
000_2	Atom
001_2	Cons cell
010_2	Tuple
011_2	Integer
100_2	Float
101_2	Function
110_2	Binary
111_2	Pid, Port, Reference

Table 2: Type Tags

Pids, ports and references are distinguished by a secondary tag field in the data item itself.

6.3 Lists

During the development of Gerl[18] it was found that appending to lists was an expensive operation under JAM and BEAM. This operation is O(n) on the length of the first list. Adding a tail pointer to lists in Gerl made a significant difference to list performance. As a result EC has adopted a list implementation which retains a tail pointer.

Lists are structured as chains of cons cells (see figure 5) each cons cell contains a pointer to its payload, a pointer to the next cell and a pointer to the tail. When lists are appended, the tail pointer of the head cell is updated to point to the tail of the second list. This can greatly improve append performance.

Tail pointers inside the list are guaranteed to point to cons cells further down the list, but they are not guaranteed to point to the end of the list. Thus O(1) list append performance is assured only when the head of the first and second lists are provided to append.

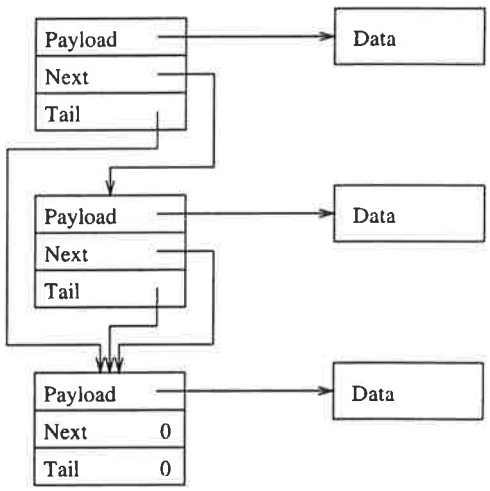


Figure 5: Chains of cons cells

Using cons cells has the additional advantages of easily implementing the semantics of ill-formed lists – lists which do not end in an empty list – and requiring only the allocation and deallocation of a single cons cell on altering a list.

7 Tools

The tool-set chosen to implement EC was determined by the desire to allow as many of the developer community to participate as possible. This meant that widely available tools had to be used. We selected the following development environment:

- Unix
- Lex and Yacc
- C
- IBURG
- GNU as

7.1 Unix

We chose to begin development of EC under the Unix environment because of its prevalence in research environments and the low cost availability of the Unix like operating systems FreeBSD and Linux.

EC uses many tools that are shipped with or commonly available under Unix including: awk, grep, make and sed. Where multiple versions of the tools exist, with differing dialects, the intersection of the dialect has been chosen.

The selection of Unix tools does not prevent development on other platforms. There is a long history of supporting projects that have started in the Unix environment in other environments.

7.2 Lex & Yacc

We chose to use well known tools from traditional compiler development including: Lex and Yacc. These tools are commonly used in University compiler courses and are often included in Unix distributions.

Use of these tools simplifies the development of the front-end components of the compiler significantly. As the Erlang language is still in development, using a hand crafted parser and lexer would make tracking changes in the language more difficult.

7.3 C

We considered using the existing OSE compiler and its Yecc grammar and generating assembler code directly. However, this approach would have greatly reduced the pool of potential programmers available, as we would immediately limit the project to Erlang programmers.

C++ was considered as an implementation language, as it offers convenient encapsulation of the many compiler data structures. Observation of some of the difficulties in the Gerl[18] project in using C++ with Yacc was not trivial. We decided to forego the benefits of C++ for the simplicity of using C with Yacc.

A large part of the common compiler literature is directly applicable to C based projects. Choosing C allowed potential programmers easy access to this literature.

7.4 IBURG

At the beginning of the project it was hoped that we could use either an existing code generator or C-- [1]. Neither of these options proved fruitful as code generators are extremely closely tied to the compilers that they serve, and the MLRISC[14] project which attempted to develop a portable, reusable code generator seems to have stalled, with current code being incomplete and a subset of SMLNJ[16] distribution, furthermore there was no documentation for the x86 back-end. Development of C-- appears to be still in its infancy with the specification subject to rapid change and only prototype implementations available at the time of writing. At the time of writing *Quick C--* had announced its intention to support garbage collection and exceptions.

The only available alternative appeared to be to implement a code generator from scratch.

Much of the work of a code generator can be modelled as tree parsing. IBURG[12] is a fast tree parser based on Bottom-Up Rewrite System (BURS) techniques. We use IBURG to generate a tree parser using an annotated burg grammar to generate the perform instruction selection. This results in more compact machine descriptions which should permit easier porting to other architectures.

Key factors in the selection of IBURG were the free availability of the tool - critical in open source

projects - and the generation of unencumbered code. The tool itself is licensed in such a way that it may not be modified and redistributed without permission.

The licence of the code forced the development of a script to take our modified burg grammar and transform it into a form acceptable to IBURG.

7.5 GNU as

GNU as was developed as the back-end assembler for GCC. The assembler is commonly available and its syntax resembles that used in assemblers shipped on Unix systems derived from AT&T Unix.

Earlier experience by the author with the use of this assembler was positive and its only disadvantage is the unfamiliarity of its syntax and operand ordering to developers of Intel code on Microsoft platforms. The high availability on Unix platforms – our primary platform – and the similarity of its syntax across a wide range of processor families outweighed this disadvantage.

8 Future Plans

There are a number of medium term plans for improving EC. These improvements include:

- Pre-Processors
  - A separate project to develop a preprocessor to handle record and macro transformations into EC's subset of Erlang.
- ec1
  - Implement the binary syntax. This was postponed at the beginning of the project to allow the syntax to stabilise. It is now clear that there are a considerable number of users of the current syntax and only a limited range of changes are likely.
- ec2
  - The existing operator evaluation of the constant folder should be extended to perform BIF evaluation.
  - A constant propagator should be added to allow variables which are bound to constants to be replaced with the constants.
  - Perform profiling to determine if it would be beneficial for strings and characters to become genuine types of the language for use in later phases of compilation.
  - Implement a limited form of partial evaluation. Where functions can be evaluated to constants within a module the constants should be substituted for the function call.
- ec4
  - Dead code removal should be added to this phase. Dead traces can be trivially removed by disposing of any trace which is not jumped to by a trace which can be reached from a trace beginning at a `T_fun` by following the labels at the ends of each trace recursively.
- ec5
  - Duplicate sub-expression removal code can be added to ec5.
- ec6
  - Supporting more processor types. The SPARC processor is an obvious next target.
  - Improved register allocation.
  - Identification of common parts of the machine description and moving them out of the machine description into the common code.
  - Removing null code, in particular moves from a register into the same register.

9 Deficiencies

At the time of writing the a number of critical tasks remain uncompleted. These tasks will completed in the next few months.

Implementations of:

Funs - At present only the `fun Fun/n` form of *fun* is implemented. Work is proceeding on other types of *anonymous funs* and implementing `apply`.

Last Call Optimisation - Should be implementable within the assembler language generation stage of compilation.

Dynamic code loading / code replacement - A Unix implementation using the `dlopen` family of functions is planned

Garbage collection - The work on garbage collection in Gerl[19] should be adaptable to EC.

10 Conclusion

EC implements a large proportion of the Erlang language and generates native assembler code that can easily call C functions. This paper has described EC, the motivation for its development, and the final target environment.

It is our hope that by using common compiler development tools, and employing a 'Berkeley' style licence other developers will be able to start using and developing the compiler to its full potential.

The project home page is at http://www.serc.rmit.edu.au/~ec/

References

[1] C-- home. http://www.cminusminus.org/ (viewed July 2001).

[2] A scalable data processing system, 12 2000. Australian Provisional Patent Application No. PR2365/00.

[3] Ericsson Utvecklings AB. Erlang. http://www.erlang.org (viewed July 2001).

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, Reading, Mass, 1986.

[5] Andrew W Appel and Maia Ginsburg. *Modern compiler implementation in C.* Cambridge University Press, Cambridge, 1998.

[6] Joe Armstrong. Stand Alone Erlang. http://www.bluetail.com/~joe/sae_r7b/sae.html (viewed July 2001).

[7] Joe Armstrong, Robert Virding, Mike Williams, and Claes Wikström. *Concurrent Programming in Erlang.* Prentice-Hall, 2nd edition, 1996. ISBN 0-13-285792-8.

[8] Maurice Castro. EC: An Erlang Compiler. Technical Report SERC-0128, Software Engineering Research Centre, RMIT University, 7 2001.

[9] Marc Feeley. ETOS Compiler. http://www.iro.umontreal.ca/~etos/ (viewed July 2001).

[10] Marc Feeley. ETOS version 1.4. http://www.iro.umontreal.ca/~etos/cgi/etos14.tar.gz.cgi (viewed July 2001).

[11] Christopher W Fraser and David R Hanson. *A retargetable C compiler: design and implementation.* Benjamin/Cummings, Redwood City, CA, 1995.

[12] Christopher W Fraser, David R Hanson, and Todd A Proebsting. Engineering a Simple, Efficient Code Generator Generator. In *ACM Letters on Programming Languages and Systems*, number 3, pages 213–226, Sep 1992.

[13] Christopher W Fraser, Robert R Henry, and Todd A Proebsting. Burg - Fast Optimal Instruction Selection and Tree Passing. In *Sigplan Notices*, number 4, pages 68–76, April 1992.

[14] Lal George and Allen Leung. MLISC: A Framework for retargetable and optimizing compiler backends. http://www.cs.nyu.edu/leunga/www/MLRISC/Doc/latex/mlrisc.ps (viewed July 2001).

[15] Erik Johanson. HiPE and the Ix Compiler: Technical Reference BETA Version 0.9.2, 2 2001. http://www.csd.uu.se/~happi/hipe/hipe_manual.ps.

[16] Lucent Technologies Bell Laboratories. Standard ML of New Jersey. http://cm.bell-labs.com/cm/cs/what/smlnj/index.html (viewed July 2001).

[17] Robert Virding and Jonas Barklund. Erlang 4.7.3 Reference Manual Draft 0.7, 6 1999.

[18] Geoff Wong. Gerl - A free implemenation of Erlang. http://goanna.cs.rmit.edu.au/~geoff/erlang/ (viewed July 2001).

[19] Geoff Wong. Compiling Erlang via C. Technical Report SERC-0079, Software Engineering Research Centre, RMIT University, 12 1998.

Highlights
Erlang 5.1/OTP R8B

This document describes the major new features and changes that will be introduced in the next Erlang/OTP version R8B which will be released the 17:th of October 2001.

The major focus in the development of this version has been:

- Improved characteristics for huge systems and systems with high demands regarding soft real time characteristics. The work here is mainly concentrated on multithreaded I/O, memory handling and improvements in disk and ram based tables and Mnesia.
- Introduce the important new application Megaco/H.248 as part of OTP.
- Introduction of a permuted index in the html version of the documentation as well as a reduction of levels of html pages for each application.

Below follows a brief description of the most significant changes in each application.

Appmon, graphical view of applications and processes

- A web based user interface is added in parallell with the old graphical interface based on GS.

Asn1, compiler and runtime functions for ASN.1

- New option `per_bin` which makes the encode/decode functions use binaries and the bit syntax instead of lists. The use of binaries will be the default solution in forthcoming versions.
- Now supporting the most common use of Information Objects according to X.681 for both BER and PER (aligned).
- The new option `ber_bin` which uses binaries in the encode/decode functions is now officially supported. It is recommended to use ber_bin instead of ber, the use of binaries will be the default in forthcoming versions.
- In combination with the BER encoding rules now the DER (Distinguished Encoding Rules) is supported. The DER encoding rules is defined as a number of restrictions to the BER encoding rules and is used mainly by the X.500 directory standards.

For more details, see Release Notes.

Comet, the COM client for Erlang

Various improvements and some new example. One example shows how to access various data sources with ActiveX Data Objects (ADO). For more details see the Release Notes.

Compiler

- Various code optimizations, especially records are handled better.
- Floating point arithmetic is faster.
- A more powerful inliner is added.
- New syntax for setting all remaining fields in a record to the same value. This is useful when creating a wildcard record which e.g is used as input to `mnesia:match`. The following statement sets field a to Val and all other fields to '_':
  `#rec{a=Val,_='_'}`
  This is very practical for records with many fields.
- Improved handling of list comprehensions resulting in faster execution.
- New logical operators `andalso` and `orelse`.

CosEvent, (a CORBA service)

This version is a completely new version of the cosEvent application; older versions were **not** compliant with the OMG specification. The "look and feel" has been changed to be more uniform with the other COS-services.

cosFileTransfer *NEW* (a CORBA service)

An Erlang implementation of the OMG CORBA FileTransfer service.

Note: The OMG CosFileTransfer specification have not been finalized yet. Hence, the API may be changed in the future.

CosNotification (a CORBA service)

Minor changes, see Release Notes.

cosProperty *NEW* (a CORBA service)

An Erlang implementation of the OMG CORBA Property service.

CosTime (a CORBA service)

Minor changes, see Release Notes.

CosTransactions (a CORBA service)

Minor changes, see Release Notes.

Crypto, MD5, SHA and CBC-DES encryption/decryption

No changes since R7B01.

Debugger

Internal changes.

Erlang emulator (ERTS)

- Improved multithreaded I/O
- Improved inet_driver (IP)
- Named ports
- Improved memory management on Unix (using mmap) gives less memory fragmentation.
- Improved diagnostic BIF's retrieving memory information
- Improved floating point arithmetic
- Support for scatter/gather I/O in the file driver to improve performance.
- Improvements of the file drivers real-time characteristics for the single threaded case (default) when reading/writing very large blocks.

Erl_Interface, a C library for integration of Erlang and C programs

A lot of improvements, the library "ei" now can be used without support from the old library "erl_interface". The "ei" library is reentrant and makes it possible to have i.e. multiple ei-nodes on VxWorks (which means multiple ic generated nodes too). The old "erl_interface" API is still there but only for backward compatibility reasons. The "Erl_interface" API and lib is now obsolete and should not be used in new development.

EVA, event and alarm handling

No changes since R7B01.

GS, the Graphics System

- Some internal modules and processes in GS were prefixed with `gtk`. This prefix has been changed to `gstk` not to clash with modules that interface to the graphical toolkit Gtk.

Other minor internal changes, see GS Release Notes.

IC, the IDL compiler and runtime functions

- Added support for erlang::binary in compiler and C backends.

- Multiple backend option for faster compilation. Makes it possible to generate code with several different backends in one compiler run.
- Various bugfixes and optimizations.

Inets, HTTP server and FTP client.

Minor changes, see Release Notes.

Jinterface, a library for integration of Erlang and Java programs

No changes since R7B01.

Kernel, mandatory basic applications

- Erlang nodes can be hidden in which case they don't participate in the global name space and dont appear in the list of nodes returned by the nodes/0 Bif. New Bif's are added to retrieve information about hidden nodes. The distribution mechanisms are also generally improved to meet demands for larger networks of distributed erlang nodes.
- The `inet`, `inet_tcp`, `inet_udp` components are further improved and optimized
- The global name service is faster.

Megaco, a Megaco/H.248 protocol stack *NEW*

A framework for building applications on top of the Megaco/H.248 protocol. Megaco/H.248 is a protocol for control of elements in a physically decomposed multimedia gateway, enabling separation of call control from media conversion. A Media Gateway Controller (MGC) controls one or more Media Gateways (MG).

The semantics of the protocol has jointly been defined by two standardization bodies:

- IETF - which calls the protocol Megaco
- ITU - which calls the protocol H.248

Mesh

This application has been removed.

Mnemosyne, database queries

No changes since R7B01.

Mnesia, a DBMS

Already in R7 (Mnesia-3.10)

The implementation for tables with property `disc_copies` is significantly improved and does not use `dets` anymore. This will improve over all system performance.

R8 (Mnesia-4.X and later)

The improvements in `dets` will yield an improvement on the `disc_only_copies` storage type.

`select/[23]` is implemented which should decrease the need to use Mnemosyne and `match_object`. By use of `select` the search performance is improved. The data returned from a search with `select` can be taylored by the user. See ERTS User's Guide for syntax of match expressions.

Synced transactions have been implemented, these kind of transactions do not increase the performance but have a better behaviour from a system perspective. A node B cannot be overloaded by running fast transactions on node A and you cannot overload `disk_log` as you can with the normal `mnesia:transaction/[123]` call.

Old dependencies, workarounds and support for old Erlang/OTP release will be removed.

Mnesia_Session, interface to Mnesia

Minor internal changes, see Mnesia_Session Release Notes.

ODBC, interface to SQL databases

The application has been rewritten. The main change is that allocation and deallocation om memory is now handled by the application.

ODBC also need handlers for communication with the database. These handlers are now allocated by application.

New interface functions have been added which replace the old ones. The old interface functions are retained for backwards compatibility but may be removed in the future. For more information, see the ODBC Reference Manual.

Orber, the Erlang ORB

The overhead for Inter-ORB communication has been reduced significantly compared with the first R7B version (3.1.8). As a part of this work the memory usage has also been reduced.

OS_Mon, monitoring of disk-usage and OS resources

- The `memsup` and `disksup` "supervisors" are improved to give faster responses and generate less overhead.
- Now works on Linux and FreeBSD

Parsetools, a parser generator

No changes since R7B01.

Pman, graphical tool for inspection and tracing of processes

Minor internal changes, see Pman Release Notes.

Runtime_Tools, trace functionality etc.

Tracing is generally improved and bugfixed, see the dbg and erlang manual sections

SASL, release handling, upgrade etc.

- The report browser (`rb`) no longer ignores faulty events - it prints the raw terms instead. It also presents user-generated `error_report` and `info_report` events more nicely.
- `systools_make` uses Xref instead of Exref for cross reference checks. The format of the warnings for undefined functions has been changed.
  (*** POTENTIAL INCOMPATIBILITY ***)

SNMP, Simple Network Management Protocol

Minor changes, see Release Notes.

SSL, Secure Socket Layer

Minor changes, see Release Notes.

STDLIB, mandatory library functions

ETS/DETS

- Ets and Dets now includes more powerful alternatives to the `(d)ets:match/2`, `(d)ets:match_object/2` and `(d)ets:match_delete/2` calls. The new matching alternatives are called `(d)ets:select/2` and `(d)ets:select_delete/2`.
- The new calls `(d)ets:match/1`, `(d)ets:match/3` and corresponding `match_object/select` calls makes it possible to fetch smaller chunks of matching objects for processing instead of always retrieving all matching objects at the same time
- The predicate `(d)ets:member/2` makes it possible to look for a key in an ets table without retrieving the whole object
- `(d)ets:insert/2` now also accepts a list of objects to be inserted in one call
- Furthermore the following (d)ets calls are added: delete_object, delete_all_objects, from_(d)ets, to_(d)ets, init_table and test_ms (only ets). See the documentation for details
- Dets is much improved in terms of speed, resource consumption and works better together with multithreaded I/O.

Disk_log

Improved performance, especially when using multithreaded I/O.

Toolbar, graphical launcer

Minor internal changes, see Toolbar Release Notes.

Tools, various useful development tools

- Coast was much too slow and inefficient, especially for large modules. Also, it was discovered that Coast in some cases returned erroneous values.

  Consequently, Coast has been discontinued and is replaced with a new tool **Cover** with similar interface and functionality. Refer to the Tools User's Guide and Reference Manual for more information about Cover. (*** POTENTIAL INCOMPATIBILITY ***)
  Added a web based user interface to Cover.

- `xref` represents unresolved calls with calls where the module, the function or the number of arguments are atoms and integers that are unlikely to occur in an Erlang system. Calls to `erlang:apply/2` and the like are included among the external calls. This is in contrast to R7 where calls to `apply` and `spawn` were used for representing unresolved calls.
  (*** POTENTIAL INCOMPATIBILITY ***)

- A new profiling tool `fprof` which measure how time is used in your Erlang programs. Uses trace to file to minimize performance impact, and displays time for calling and called functions.

Other minor changes, see Tools Release Notes.

TV, grapical viewer for ets and mnesia tables

Minor internal changes, see TV Release Notes.

WebTool *NEW*

A framework for web based tools.

Erlang/OTP User Conference 2001 - Participants

Chairman and speakers			
Thomas Arts **IFL**	CSLab, UAB	Stockholm, Sweden	thomas@erix.ericsson.se
Miguel Barreiro Paz **IFL**	University of Coruña	La Coruña, Spain	enano@lfcia.org
Per Bergqvist	Synapse Systems AB	Stockholm, Sweden	per@synapse.se
Johan Bevemyr	Alteon WebSystems	Stockholm, Sweden	jb@bluetail.com
Maurice Castro	SERC	Melbourne, Australia	maurice@serc.rmit.edu.au
Jakob Cederlund	GNO Data	Stockholm, Sweden	jakob@gnodata.se
Francesco Cesarini	Cesarini Consulting Ltd	London, UK	francesco@erlang-consulting.com
José Luis Freire Nistal **IFL**	University of Coruña	La Coruña, Spain	freire@dc.fi.udc.es
Dan Gudmundsson	OTP Unit, UAB	Stockholm, Sweden	Dan.Gudmundsson@uab.ericsson.se
Martin Gustafsson	OTP Unit, UAB	Stockholm, Sweden	marting@erix.ericsson.se
Björn Gustavsson	OTP Unit, UAB	Stockholm, Sweden	Bjorn.Gustavsson@uab.ericsson.se
Sean Hinde	one2one	Borehamwood, Herts, UK	Sean.Hinde@one2one.co.uk
Frank Huch **IFL**	Christian-Albrechts-Universität	Kiel, Germany	fhu@informatik.uni-kiel.de
Thomas Lindgren		Stockholm, Sweden	ftl@acm.org
Björn Lisper **IFL**	Mälardalen University	Västerås, Sweden	lisper@it.kth.se
Kenneth Lundin	OTP Unit, UAB	Stockholm, Sweden	kenneth.lundin@uab.ericsson.se
Matthias Läng	Corelatus	Stockholm, Sweden	matthias@corelatus.com
Jan Nyström	University of Uppsala	Uppsala, Sweden	jann@DoCS.UU.SE
Mickaël Rémond	IdealX	Paris, France	mickael.remond@IDEALX.com
Tony Rogvall	Alteon WebSystems	Stockholm, Sweden	tony@bluetail.com
Kostis Sagonas	Uppsala University	Uppsala, Sweden	kostis@csd.uu.se

Juan J. Sánchez **IFL**	University of Coruña	La Coruña, Spain	juanjo@lfcia.org
Claes Wikström	Alteon WebSystems	Stockholm, Sweden	klacke@bluetail.com
Participants			
Tomas Abrahamsson	Ericsson Radio Systems AB	Linköping, Sweden	epktoab@lmera.ericsson.se
Ingemar Ahlberg	Ericsson Internet Applications	Stockholm, Sweden	Ingemar.Ahlberg@era-t.ericsson.se
Kristoffer Andersson	MobileArts	Stockholm, Sweden	
Ola Andersson	CellPoint Systems AB	Stockholm, Sweden	ola.andersson@cellpoint.com
Peter Andersson	Ericsson Utvecklings AB	Stockholm, Sweden	Peter.Andersson@uab.ericsson.se
Ingela Anderton	Ericsson Utvecklings AB	Stockholm, Sweden	Ingela.Anderton@uab.ericsson.se
Marcus Arendt	Marcus Arendt AB	Stockholm, Sweden	marcus@arendt.se
Joe Armstrong		Stockholm, Sweden	joe.armstrong@telia.com
Gösta Ask	Ericsson Telecom AB	Stockholm, Sweden	Gosta.Ask@era.ericsson.se
Karl Avedal	Ironflare AB	Stockholm, Sweden	karl@orionserver.com
Vahagn Avedian	Ericsson Telecom AB	Stockholm, Sweden	vahagn.avedian@etx.ericsson.se
Björn Axelsson	HiQ	Stockholm, Sweden	Bjorn.Axelsson@hiq.se
Per Bengtsson	Telia Promotor AB	Uppsala, Sweden	Per.X.Bengtsson@telia.se
Mia Berg	Sjöland & Thyselius Telecom AB	Stockholm, Sweden	
Martin Björklund	Alteon WebSystems	Stockholm, Sweden	
Hans Bolinder	OTP Unit, UAB	Stockholm, Sweden	Hans.Bolinder@uab.ericsson.se
Kent Boortz	OTP Unit, UAB	Stockholm, Sweden	kent@erix.ericsson.se
Pascal Brisset **IFL**	Cellicium	Cachan, France	pascal.brisset@cellicium.com
Mikael Bylund	Telia Promotor AB	Uppsala, Sweden	Mikael.M.Bylund@telia.se
Göran Båge	MobileArts	Stockholm, Sweden	goran.bage@home.se
Lars Carlsson	Ericsson Telecom AB	Stockholm, Sweden	lars.carlsson@ericsson.com
Richard Carlsson	Uppsala University	Uppsala, Sweden	Richard.Carlsson@csd.uu.se

Mats Cronqvist	Ericsson Telecom AB	Stockholm, Sweden	etxmacr@etxb.ericsson.se
Mats Cullberg	Lemon Planet AB	Stockholm, Sweden	mats.cullberg@lemonplanet.com
Bjarne Däcker	CSLab, UAB	Stockholm, Sweden	bjarne@erix.ericsson.se
Marko van Eekelen **IFL**	University of Nijmegen	Nijmegen, The Netherlands	marko@cs.kun.nl
Niclas Eklund	OTP Unit, UAB	Stockholm, Sweden	Niclas.Eklund@uab.ericsson.se
Karl-Filip Faxén **IFL**	Royal Institute of Technology	Stockholm, Sweden	kff@it.kth.se
Gerd Flaig	Bei Schlund + Partner AG	Karlsruhe, Germany	gerd@schlund.de
Christopher Foley	Ericsson Systems Expertise Ltd	Athlone, Ireland	Christopher.Foley@eei.ericsson.se
Lars-Åke Fredlund	SICS	Stockholm, Sweden	fred@sics.se
Magnus Fröberg	Alteon WebSystems	Stockholm, Sweden	magnus@bluetail.com
Martin Gasbichler **IFL**	University of Tübingen	Tübingen, Germany	gasbichl@informatik.uni-tuebingen.de
Luke Gorrie	Alteon WebSystems	Stockholm, Sweden	luke@bluetail.com
Catrin Granbom	Ericsson Utvecklings AB	Stockholm, Sweden	Catrin.Hansson-Granbom@uab.ericsson.se
Joacim Grebenö	Alteon WebSystems	Stockholm, Sweden	jocke@bluetail.com
Richard Green	OTP Unit, UAB	Stockholm, Sweden	Rickard.Green@uab.ericsson.se
Clemens Grelck **IFL**	University of Lubeck	Lubeck, Germany	grelck@isp.mu-luebeck.de
Siri Hansen	Ericsson AS	Grimstad, Norway	Siri.Hansen@eto.ericsson.se
Bogumil Hausman	CellPoint Systems AB	Stockholm, Sweden	bogumil.hausman@cellpoint.com
Per Hedeland	Alteon WebSystems	Stockholm, Sweden	per@bluetail.com
Pekka Hedqvist	PH IT Konsult	Stockholm, Sweden	pekka@home.se
Joakim Hirsch	Ericsson Utvecklings AB	Stockholm, Sweden	Joakim.Hirsch@uab.ericsson.se
John Hughes	Chalmers University of Technology	Göteborg, Sweden	rjmh@cs.chalmers.se
Gunilla Hugosson	OTP Unit, UAB	Stockholm, Sweden	gunilla@erix.ericsson.se
Erik Johansson	Uppsala University	Uppsala, Sweden	happi@csd.uu.se
Rikard Johansson	MobileArts	Stockholm, Sweden	

Torbjörn Johnson		Stockholm, Sweden	torbjorn.k.johnson@swipnet.se
Henrik Jonasson	Ericsson Telecom AB	Stockholm, Sweden	Henrik.Jonasson@etx.ericsson.se
Micael Karlberg	OTP Unit, UAB	Stockholm, Sweden	bmk@erix.ericsson.se
Bertil Karlsson	OTP Unit, UAB	Stockholm, Sweden	Bertil.Karlsson@uab.ericsson.se
Håkan Karlsson	CSLab, UAB	Stockholm, Sweden	hk@erix.ericsson.se
Mikael Karlsson	Alteon WebSystems	Stockholm, Sweden	
Brian Kelly	Ericsson Telecom AB	Stockholm, Sweden	Brian.Kelly@etx.ericsson.se
Bengt Kleberg	Ericsson Telecom AB	Stockholm, Sweden	eleberg@etxb.ericsson.se
Pieter Koopman **IFL**	University of Nijmegen	Nijmegen, The Netherlands	pieter@cs.kun.nl
Håkan Larsson	CSLab, UAB	Stockholm, Sweden	hl@erix.ericsson.se
Tony Larsson	Ericsson Utvecklings AB	Stockholm, Sweden	Tony.Larsson@uab.ericsson.se
Tord Larsson	Alteon WebSystems	Stockholm, Sweden	
John Launchbury	Galois Connections, Inc.	Beaverton, Oregon, USA	John@galconn.com
Fredrik Linder	BlueLabs AB	Solna, Sweden	fredrik.linder@home.se
Olof Lindgren	SSF	Stockholm, Sweden	ol@stratresearch.se
Leif Lorentzen	Cyberode AB	Stockholm, Sweden	leif@cyberode.se
Martin Lundmark	Ericsson Radio Systems AB	Linköping, Sweden	Martin.A.Lundmark@era.ericsson.se
Ann-Marie Löf	Sjöland & Thyselius Telecom AB	Stockholm, Sweden	
Anna Löfgren	Sjöland & Thyselius Telecom AB	Stockholm, Sweden	
Ricardo Massa Ferreira Lima **IFL**	Chalmers University of Technology	Göteborg, Sweden	ricardo@cs.chalmers.se
Thomas Mattisson	MobileArts	Stockholm, Sweden	thomas.mattisson@mobilearts.se
Håkan Mattsson	CSLab, UAB	Stockholm, Sweden	hakan@erix.ericsson.se
Ingvar Meyer	OTP Unit, UAB	Stockholm, Sweden	Ingvar.Meyer@uab.ericsson.se
Chandrashekhar Mullaparthi	one2one	Borehamwood, Herts, UK	Chandrashekhar.Mullaparthi@one2one.co.uk
Ray Murphy	Ericsson Telecom AB	Stockholm, Sweden	Ray.Murphy@etx.ericsson.se

Peter Mustel	Ericsson Telecom AB	Stockholm, Sweden	etxmuel@etxb.ericsson.se
Anna Mårtensson	Ericsson Telecom AB	Stockholm, Sweden	Anna.Martensson@etx.ericsson.se
Hans Nahringbauer	Telia Promotor AB	Uppsala, Sweden	Hans.H.Nahringbauer@telia.se
Daniel Néri	Sigicom AB	Tullinge, Sweden	daniel.neri@sigicom.com
Bengt Nilsson	OTP Unit, UAB	Stockholm, Sweden	
Hans Nilsson	CSLab, UAB	Älvsjö, Sweden	hans@erix.ericsson.se
Raimo Niskanen	OTP Unit, UAB	Stockholm, Sweden	raimo@erix.ericsson.se
Patrik Nyblom	OTP Unit, UAB	Stockholm, Sweden	Patrik.Nyblom@uab.ericsson.se
Sven-Olof Nyström	Uppsala University	Uppsala, Sweden	svenolof@csd.uu.se
John O'Donnell **IFL**	University of Glasgow	Glasgow, UK	jtod@dcs.gla.ac.uk
Janine O'Keeffe	Ericsson Telecom AB	Stockholm, Sweden	janine.okeeffe@etx.ericsson.se
Åke Olsson	CellPoint Systems AB	Stockholm, Sweden	ake.olsson@cellpoint.com
Rex Page	University of Oklahoma	Oklahoma, USA	page@ou.edu
Ricardo Pena **IFL**	Universidad Complutense de Madrid	Madrid, Spain	ricardo@sip.ucm.es
Oswaldo Perdomo	Ericsson Utvecklings AB	Stockholm, Sweden	Oswaldo.Perdomo@uab.ericsson.se
Mikael Pettersson	Uppsala University	Uppsala, Sweden	Mikael.Pettersson@csd.uu.se
Susanne Petersson	Ericsson Telecom AB	Stockholm, Sweden	etxsepn@etxb.ericsson.se
Rinus Plasmeijer **IFL**	University of Nijmegen	Nijmegen, The Netherlands	rinus@cs.kun.nl
Anders Ramsell	Telia Promotor AB	Uppsala, Sweden	Anders.A.Ramsell@telia.se
Claus Reinke **IFL**	University of Kent	Canterbury, UK	C.Reinke@ukc.ac.uk
Jay Riddle **IFL**	MicronPC	Nampa, Idaho, USA	jcriddle@micronpc.com
Ola Samuelsson	Cyberode AB	Stockholm, Sweden	ola@cyberode.se
Peter Schneider-Kamp	RWTH	Aachen, Germany	peter@lundata.se
Sven-Bodo Scholz **IFL**	Christian-Albrechts-Universität	Kiel, Germany	sbs@informatik.uni-kiel.de
Clara Segura **IFL**	Universidad Complutense de Madrid	Madrid, Spain	csegura@sip.ucm.es

Håkan Stenholm	Ericsson Telecom AB	Stockholm, Sweden	hokan.stenholm@telia.com
Volker Stolz **IFL**	RWTH	Aachen, Germany	stolz@i2.informatik.rwth-aachen.de
Per Sternås	Ericsson Enterprise IT	Stockholm, Sweden	Per.Sternas@ebc.ericsson.se
Per Einar Strömme	Ericsson Telecom AB	Stockholm, Sweden	etxnep@etxb.ericsson.se
Leonid A. Timochouk **IFL**	University of Kent	Canterbury, UK	lat2@ukc.ac.uk
Lars Thorsén	CSLab, UAB	Stockholm, Sweden	lars@erix.ericsson.se
Robert Tjärnström	Ericsson Radio Systems AB	Stockholm, Sweden	erandig@al.etx.ericsson.se
Henrik Torelm	CellPoint Systems AB	Stockholm, Sweden	henrik.torelm@cellpoint.com
Torbjörn Törnkvist	Alteon WebSystems	Stockholm, Sweden	tobbe@bluetail.com
Alberto de la Encina Vara **IFL**	Universidad Complutense de Madrid	Madrid, Spain	albertoe@sip.ucm.es
Robert Virding		Stockholm, Sweden	rv@bluetail.com
Vladimir Vlassov	Royal Institute of Technology	Stockholm, Sweden	vlad@it.kth.se
Jane Walerud		Stockholm, Sweden	jane@walerud.com
Carl Wilhelm Welin	CSLab, UAB	Älvsjö, Sweden	calle@erix.ericsson.se
Ulf Wiger	Ericsson Telecom AB	Stockholm, Sweden	etxuwig@etxb.ericsson.se
Daniel Wiik	Ericsson Radio Systems AB	Linköping, Sweden	daniel.wiik@era.ericsson.se
Jesper Wilhelmsson	Uppsala University	Uppsala, Sweden	
Stefan Willehadson	Ericsson Internet Applications	Stockholm, Sweden	stefan.willehadson@ericsson.com
Lon Willett	SSE Ltd	Dublin, Ireland	Lon.Willett@sse.ie
Christopher Williams	Ericsson Systems Expertise Ltd	Athlone, Ireland	chris.williams@ericsson.com
Lian Wu	Ericsson AS	Grimstad, Norway	Lian.Wu@eto.ericsson.se
Jan Zaar	CellPoint Systems AB	Stockholm, Sweden	jan.zaar@cellpoint.com
Vincent Zweije **IFL**	University of Nijmegen	Nijmegen, The Netherlands	zweije@cs.kun.nl
Lennart Öhman	Sjöland & Thyselius Telecom AB	Stockholm, Sweden	lennart.ohman@st.se

OTP = Open Telecom Platform
UAB = Ericsson Utvecklings AB
CSLab = Computer Science Laboratory

IFL = Attends also the *13th International Workshop on the Implementation of Functional Languages* (IFL 2001).