12th International Erlang/OTP User Conference

Stockholm, November 9-10, 2006

Proceedings

http://www.erlang.se/euc/06/

corelatus

ERICSSON

Erlang Training and Consulting

tail-f

synapse mobile networks s.a.

ERLANG

Sjöland&Thyselius

Mobile Arts

Erlang/OTP User Conference 2006

Conference Program

08.30 *Registration.*

Session I

09.00 **Betting on Functional Programming and Winning.**
Erik Stenman, Kreditor, Sweden.

09.30 **Horde Leader, a Framework to Build Cluster Aware Erlang Web Administration Console.**
Jérôme Sautret and Mickaël Rémond, Process-one, France.

10.00 **Experiences from Using Erlang for Autonomous Robots.**
Vincenzo Nicosia and Corrado Santoro, University of Catania, Italy.

10.30 *Coffee.*

Session II

11.00 **Vixo.com - A Case Study in Developing a Web/SMS Start-up in Erlang.**
Dale Harvey, vixo.com, Scotland.

11.30 **CEAN, a Comprehensive Erlang Archive Network, or How to Make any Erlang Software Deployment a Child's Play.**
Christophe Romain and Mickaël Rémond, Process-one, France.

12.00 **Testing a Media Proxy with Quviq QuickCheck.**
John Hughes, Chalmers University, and Thomas Arts, IT University, Sweden.

12.30 *Lunch.*

Session III

14.00 **Refactoring Erlang Programs.**
Zoltan Horvath, Huiqing Li and Simon Thompson, University of Kent, England.

14.25 **Comparing C++ and Erlang for Motorola Telecoms Software.**
Henry Nyström, Erlang Training and Consulting, Sweden.

14.50 **Using GNU Autoconf to Configure Erlang Programs.**
Romain Lenglet, Tokyo Institute of Technology, Japan.

15.00 **Configuration Aware Distributed System Design in Erlang.**
Gabor Batori, Zoltan Theisz and Domonkos Asztalos, Ericsson, Hungary.

15.40 *Coffee.*

Session IV

16.00 **Eliminating overlapping of Pattern Matching when Verifying Erlang Programs in μCRL.**
Qiang Guo and John Derrick, University of Sheffield, England.

16.25 **ErlHive - Safe Erlang Reloaded!**
Ulf Wiger, Ericsson, Sweden.

16.40 **Erlang Message Receive Fundamentals.**
Jay Nelson, DuoMark International, USA (presented by Ulf Wiger).

17.10 **Current Erlang/OTP Developments.**
Kenneth Lundin, Ericsson, Sweden.

17.30 *Close followed by bus transport to an ErLounge.*

Demonstrations

Chalmers students demonstrate robot logic in Erlang.

Betting on FP (and winning?)

Erik Stenman

KREDITOR

Introduction

I will talk about KREDITOR, a company that bet it's future on Functional Programming

- Conventional wisdom...
  choose proven technology
- ... the KREDITOR way ...
  choose Erlang
- ... are they the same?

I will tell you what Kreditor does, how we do it, why we do it this way, and whether it worked out or not... at least, so far.

KREDITOR

Kreditor Europe AB

- The business model:
  - Bring trust to Internet shopping.
  - Bring old style billing into the new IT-economy.
- Brief background:
  - Founded in December 2004.
  - With < $100,000 in venture capital.
  - Live system in March 2005.
- The company vision:
  - "Be the coolest company in Sweden."

KREDITOR

The Problem

- Internet shopping is a question of **trust**.
  - The **shop** has to **trust** the **customer** to get paid.
  - The **customer** has to **trust** the **shop** to send the right stuff.
- Many **customers** are **uncomfortable** using credit card over the Internet.
- Many banks are actually **worried about the security** of Internet **shops** handling credit card information.
- Also, doing a *partial return* when using credit card is a **hassle**, both for the **customer** and the **shop**.

KREDITOR

The Solution

- Bring in *a trusted party*, i.e., KREDITOR.
- Send an invoice with the goods to the **customer**.
- The **customer** pays after receiving the goods and takes no risk. The **customer** does **not** have to trust anyone.
- The **shop** is guaranteed (by contract) to get money from KREDITOR. The **shop** only have to trust KREDITOR with whom they have a written contract.

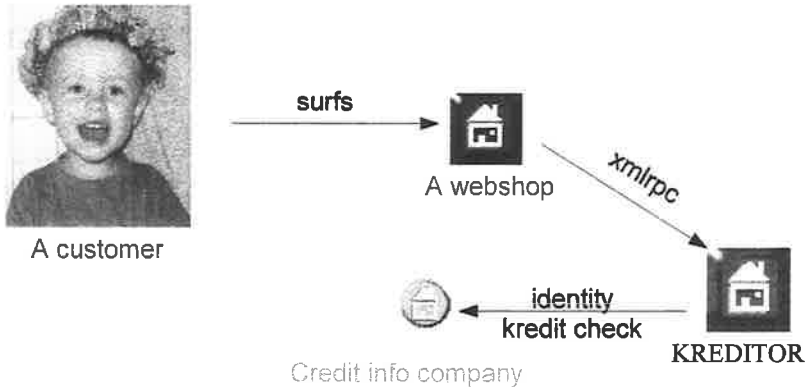
KREDITOR

Added benefits

- The **customer** gets credit.
- The **customer** can pay using familiar methods.
- Returning goods is easy.
- Better fraud detection.
- Advanced credit assessments.
- Easy to add similar features like *pre-pay* and *subscriptions*.

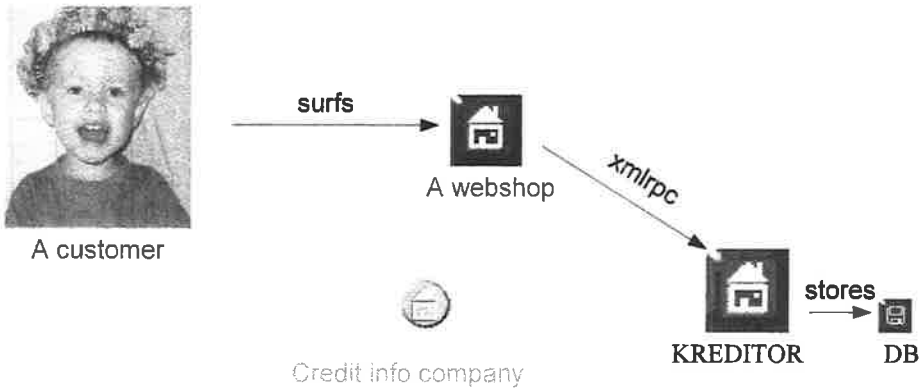
KREDITOR

The Implementation



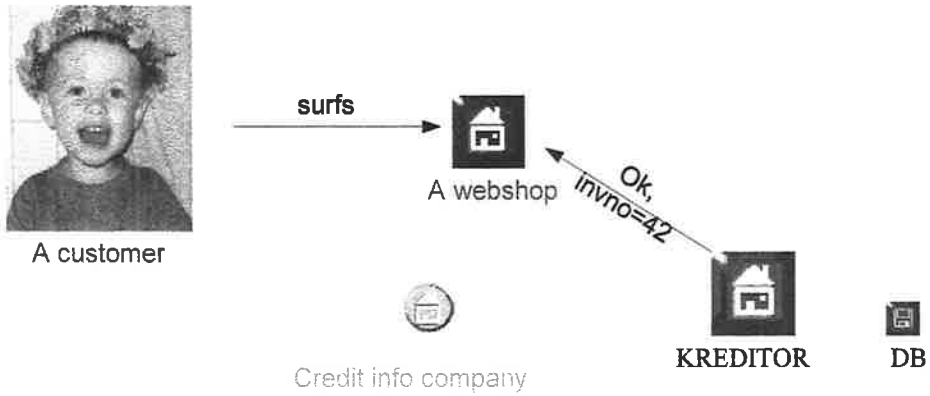
KREDITOR

The Implementation



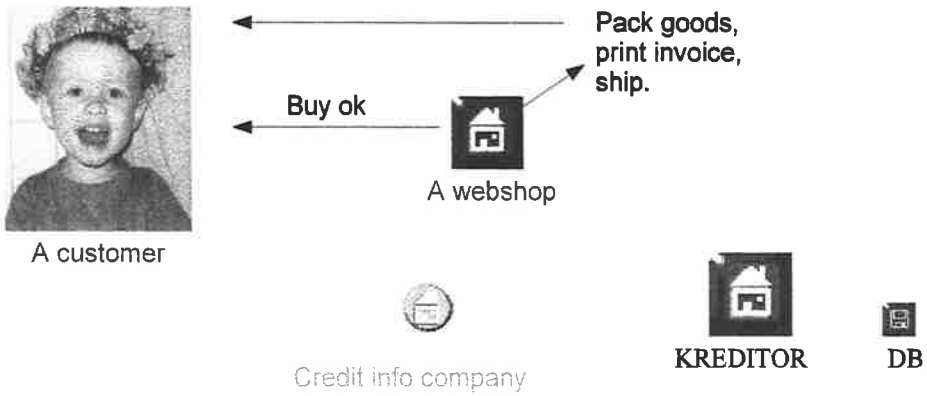
KREDITOR

The Implementation



KREDITOR

The Implementation



KREDITOR

The Implementation

A customer

A webshop

Pay

Pay

Credit info company

KREDITOR

DB

Pay

Bank

KREDITOR

The Implementation

A customer

A webshop

Credit info company

KREDITOR

DB

Reminder

Bank

Print&mail company

KREDITOR

The Implementation

A customer

A webshop

Credit info company

KREDITOR

DB

Bank

Print&mail company

KREDITOR

Some details

- The system is built from scratch using LYME (Linux, Yaws, Mnesia, and Erlang).
- So far we only operate in Sweden and Norway.
- We have a distributed system with multiple servers to provide a fault tolerant, high availability solution.
- We aim for 5 nines availability, in a setting where we introduce new features in the system every week.
- The problem fits Erlang really well.

KREDITOR

Why was Erlang Chosen?

- The three founders didn't really know what Erlang was.
- The developers didn't get the scope of the system. "It will be like the programming contest, we'll do it over the weekend" - Claes Wikström.
- Thus Kreditor and Teknikerl were born.
- Teknikerl built the first version of the system, in Erlang, in < 4 month.(While starting another company called Tail-f, but that's another story.)

KREDITOR

Why was Erlang Chosen?

- Jane Walerud saw a business presentation by three enthusiastic entrepreneurs-to-be at a business "green house".
- The Idea looked promising and she happened to know five former Bluetailers who where looking for something new to do.

KREDITOR

Did it work?

- Erlang has been a great help in providing rapid development with maintained high availability.
- KREDITOR has introduced four new major services since last December, and added over 500 new customers.
- We have expanded into Norway.
- The system has never gone down except for short maintenance stops.

KREDITOR

Did it work?

- Using Erlang has meant low development costs.
- Our main competitor busted this summer after burning more than 90 million SEK (~$12M).
  (I think they had some php/.net solution.)
- We know of some banks that have invested over 200MSEK to try to get systems that does something like our base service in a much more cumbersome way.

KREDITOR

Did it work?

- The business model has been sound.
- Total investment < $100,000
- Turnover:
  - 2004: ~0 SEK
  - 2005: 1.5 million SEK ~ $200,000
  - *2006: 15 million SEK ~ **$2 million,** (500million SEK in invoices)*
- Number of connected stores:
  2004: 0, 2005: ~200, Today: > 700
- Number of employees > 17.
- **Second place on the list of Sweden's most promising entrepreneurs by the Swedish magazine "Internet World".**

KREDITOR

Why not use Erlang?

- The main reasons that I have heard of are:
  1. Politics – Erlang is not C/Java, company policy.
  2. One provider – Concern that Ericsson will stop supporting Erlang.
  3. Lack of programmers – Erlang is still not mainstream how can we ensure we get qualified staff?
- When starting a new company, 1 is not a problem.
- I can't see 2 happening, and it's open source anyway.
- When setting up in Stockholm, 3 is not a problem.

KREDITOR

Conclusion

KREDITOR took a bet on Erlang, and so far seem to be winning.

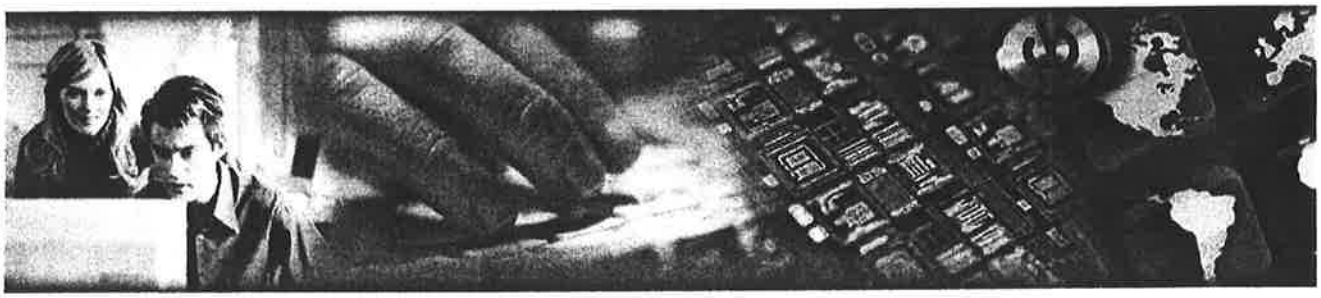
Questions?

KREDITOR

Horde Leader, a Framework to Build Cluster Aware Erlang Web Administration Console

November 9, 2006

Jérôme Sautret



Process One

CONTENTS

Process One

2 (7)

History and Status of the project

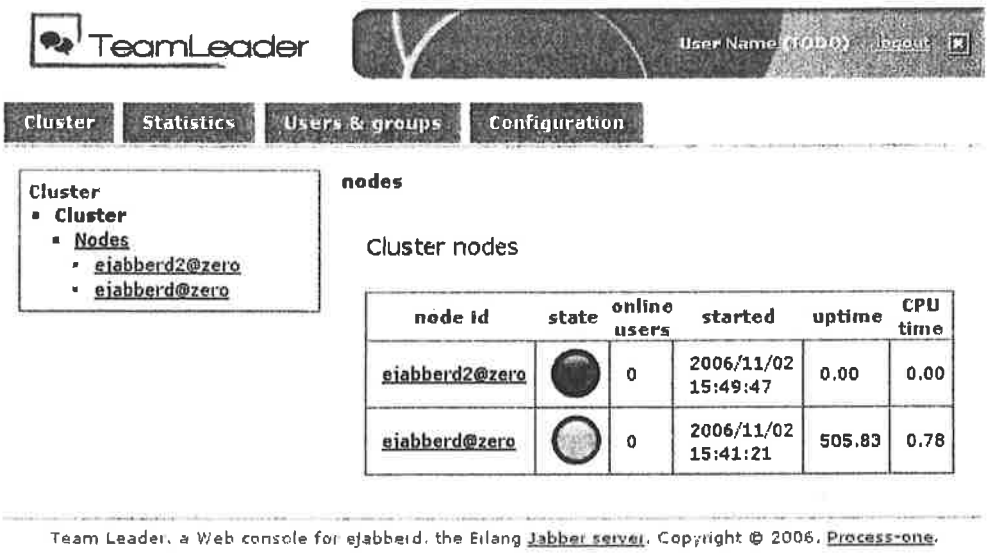
- **Team Leader, a web console for ejabberd**
  - Currently usable
  - Some ejabberd specific code in the core application

- **Horde Leader, a framework to build web consoles**
  - Generic
  - To be release in Open Source
  - Work in progress

Process One

Purpose (1/5)

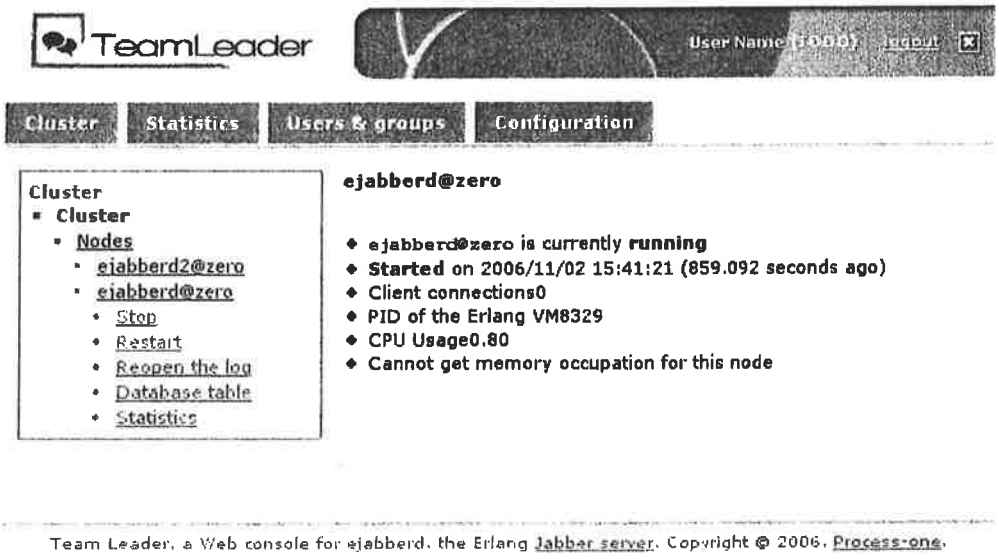
- **A Framework to build Web Consoles**
  - Monitoring
  - Control
  - Configuration
  - Statistics
  - ...



Process One

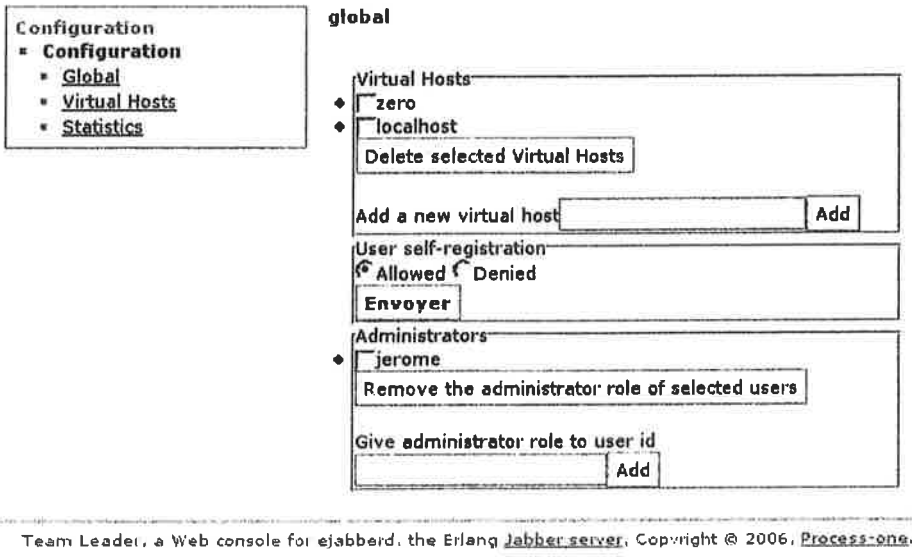
Purpose (2/5)

- **A Framework to build Web Consoles**
  - Monitoring
  - **Control**
  - Configuration
  - Statistics
  - ...



Purpose (3/5)

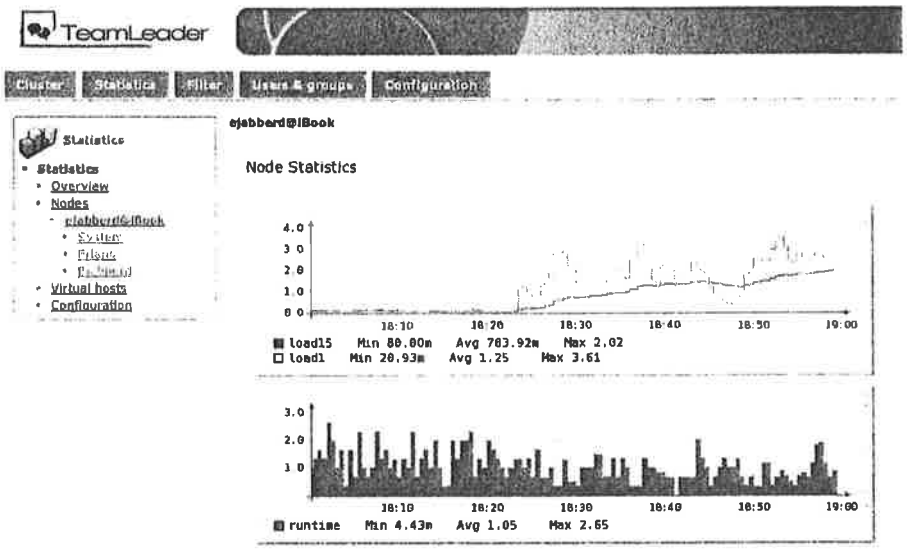
- **A Framework to build Web Consoles**
  - Monitoring
  - Control
  - **Configuration**
  - Statistics
  - ...



4(7)

Purpose (4/5)

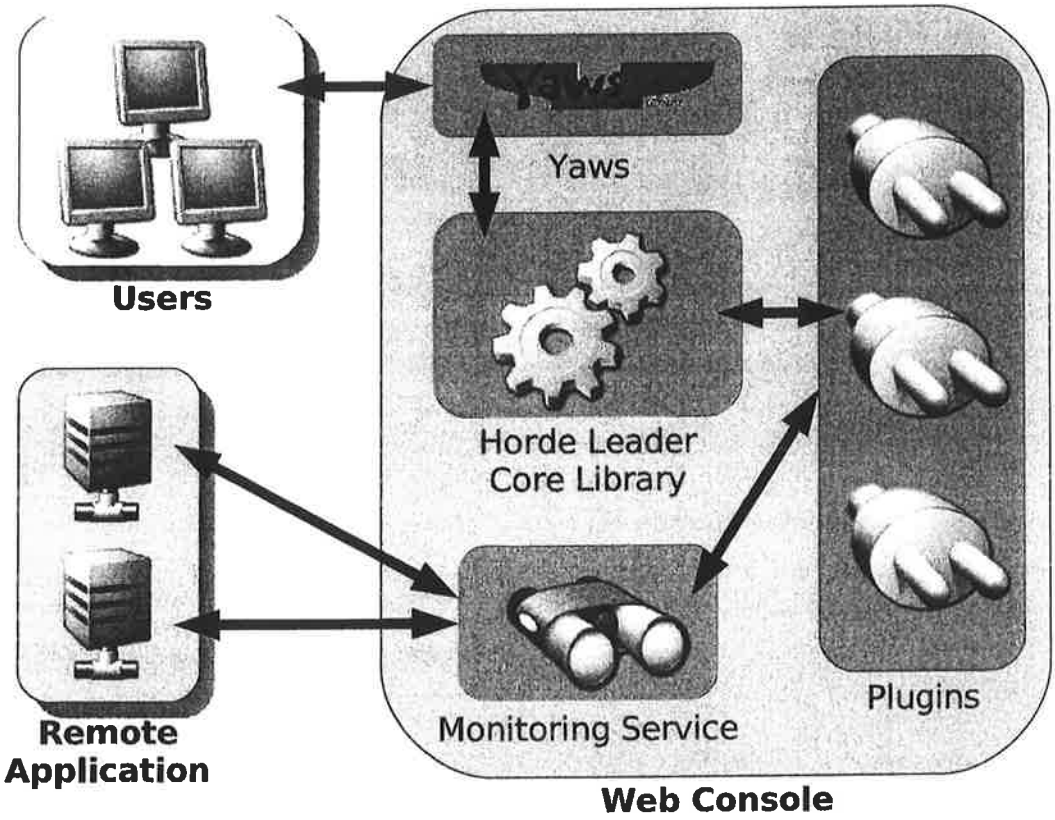
- **A Framework to build Web Consoles**
  - Monitoring
  - Control
  - Configuration
  - **Statistics**
  - ...



Purpose (5/5)

- **A Framework to build Web Consoles**
  - Monitoring
  - Control
  - Configuration
  - Statistics
  - ...
- **Plugin based**
  - All the console logic is provided by plugins
  - Plugin can be added or removed to customize or enhance the console
  - No deployment on remote application node
- **Cluster aware**
  - The console can handle an application running on several nodes

Architecture



Architecture

- **Core Library**
  - Authentication
  - URL -> Plugin function
  - Page display
    - **Navigation Menu**
    - **Widgets**
    - **Tables**
    - **Information/Error messages**
  - Various Helper Functions
- **Monitoring Service**
  - Monitor remote node states
  - Store node states in database
  - Communicate with remote nodes using RPC
  - Send module to remote nodes

6(7)

Architecture

- **Plugins**
  - Handle page content
  - Manage all the console logic
- **What is a Plugin**
  - Erlang module
  - Adds entries in the navigation tree

```
?INIT(_A) ->
        #plugin{
                id=?MODULE_STRING,
                menu=[
                        ?MENU_ITEM([],"users","Users & groups",20,index),
                        ?MENU_ITEM(["users"],"all","All users",10,all),
                        ?MENU_ITEM(["users"],"online",
                                   "Online users",20,online),
                        ?MENU_ITEM(["users"],"create",
                                   "Create",30,create),
                        ?MENU_ITEM(["users"],"delete",
                                   "Delete",40,delete),
                        ?MENU_ITEM(["users"],"search",
                                   "Search",50,search),
                        ?MENU_HIDDEN_ITEM(["users"],"user",user)
                ]}.
```

Process One

Architecture

- **How a Plugin display a page**
  - The Core Library handle the generic parts
  - The plugin returns only the content as Yaws HTML

```
adduser(A) ->
    team_leader:get_common_page(
      A,
      {ehtml,
       {form, [{method, "post"}, {action, ?ROOT_URL(A)++"users/create"},
          {class, "form"}],
       {fieldset, [],
        [?ASK_QUIT_CONFIRMATION,
        {legend, [], ?TXT("Add a new jabber user")},
        {label, [{for, "userid"}, {class, "required"},
         {title, ?TXT("User id, without the @domain")}],
          ?TXT("User id")},
        {input, [{type, "text"}, {name, "userid"}, {class, "field"}], []},
        {label, [{for, "password"}, {class, "required"},
        {title, ?TXT("Password of the new user")}],
          ?TXT("Password")},
        {input, [{type, "text"}, {name, "password"}, {class, "field"}],
         []},
        {input, [{type, "submit"}, {name, "submit"}, {class, "button"}],
         []}
]}}}).
```

Process One

7(7

Non-erlang applications

- **A console for non-erlang applications ?**
  - Console/Application communication uses erlang RPC
  - Remote Module can be send dynamically
- **Prerequisites**
  - Remote module able to communicate with application
    - **Read logs**
    - **Edit configuration files**
    - **System calls**
  - Erlang installed on all remote nodes
  - Erlang VM running on all remote nodes

```
su -c "erl -sname hl_agent >/dev/null 2>&1" - horde_leader <<EoF &
erlang:set_cookie(node(), 'secret').
Ref = make_ref().
receive
  Ref ->
    ok
end.
EoF
```

Process One

Roadmap

- **Team Leader**
- **Remove all ejabberd specific code from the Core Library and the Monitor Service**
- **Write some generic documentation**
- **Publish the code on http://forge.process-one.net**
- **Write more plugins...**

Process One

1(6)

Software Agents for Autonomous Robots: the Eurobot 2006 Experience

Vincenzo Nicosia[1], Concetto Spampinato[1] and Corrado Santoro[2] for the Eurobot DIIT Team
Università di Catania
[1]Facoltà di Ingegneria - Dipartimento di Ingegneria Informatica e delle Telecomunicazioni
[2]Facoltà di Informatica - Dipartimento di Matematica e Informatica
Viale A. Doria, 6 - 95125, Catania, Italy

***Abstract*— Agent-based software architectures have been used and exploited in many application fields. In this paper, we report our experience about using intelligent agents for an unusual task: controlling an autonomous robot playing a kind of "golf" game in an international robotic competition. Driving a real robot is a practical application field for software agents, because different subsystems need to be controlled and synchronised in order to realize a global game strategy: cooperating agents can easily fit the target. Since this application requires a soft real-time platform to guarantee fast and reliable actions, and also a valuable communication system to gain feedback from sensors and to issue commands to actuators, we chose Erlang as programming language. A two-layer multi-agent system was thus designed and realized, composed of a lower layer, hosting agents taking care of the interface with sensors and actuators, and a higher layer, where agents are in charge of "intelligent" activities related to game strategy.**

***Keywords*—** Mobile and Autonomous Robots, Computer Vision, Autonomous Agents, Real-Time Systems, Erlang.

I. Introduction

Software agents are autonomous entities that, living in a virtual world, are in charge of accomplishing the goal they are programmed for. In doing so, agents interact with the environment where they live in, by sensing its state and acting onto it, in order to achieve their goal. For these reasons, they are often called "software robots".

In spite of this similarity between (software) agents and (real) robots, agents, and above all multi-agent systems, are mainly exploited in realizing complex software systems and applications requiring intelligence, flexibility, interoperability, etc., while the area of robotics is often a matter of research on real-time and control systems. However, when a (autonomous) robot needs some intelligence to perform its activities in a more efficient and effective manner, the use of agent technology seems a natural choice [17].

The issue is that, in these cases, agents have to face the problems related to the interface to physical sensors and actuators, which connect the computer system with a physical environment that also changes during time. Therefore, an agent–enabled robot has not only to tackle the problems related to direct use of input/output ports, acquisition and driving boards, serial ports etc., but it should also take in account the fact that the scenario is time-constrained. In fact, as it is known, an information acquired from sensors (e.g. the position of the robot or of its arm) has a deadline after which the data become stale and no more useful, unless a fresh value is obtained. These problems are quite known in the area of real-time systems and their solution is achieved by means of platforms and/or operating systems that regulate program execution—in terms of process/task scheduling, race condition and delay control—in order to guarantee that deadlines are met.

Since such a real-time support is needed also in the case of the use of an agent-based system to control robot activities, the traditional and well-known agent platforms, which are mainly based on Java, cannot be employed at all: at it is known, the main problem of Java is the garbage collector, which introduces unpredictable latencies that prevent any attempt to build a time-constrained system. Indeed, RTSJ specification [6] provides a set of classes and some programming rules that allow the realization of real-time Java systems, but the specification introduces hard constraints in object allocation and reference that require an existing Java program (and thus an agent platform) to be rewritten in order to make it RTSJ-compliant [22], [16], [8].

In the context of agents and real-time systems, a language that features some interesting characteristics is *Erlang* [5], [4], [1]. It is a functional and symbolic programming language that has been proved to be suitable for the implementation of multi-agent and intelligent systems [21], [10], [12], [11], [13], [15], [14], [9]; moreover, since the Erlang runtime system is able to provide *soft real-time*[1] capabilities [18], [3], it seems also quite useful for the realization of an autonomous robot controlled by autonomous agents. In this context, this paper describes the authors' experience in designing and implementing an autonomous robot, for the Eurobot 2006 competition[2,3], by means of a multi-agent system written using the Erlang programming language. A layered multi-agent system has been designed, composed of two layers: a *back-end* (lower layer), comprising agents performing the interface with robot's physical sensors and actuators, and handling low-level control activities; and a *front-end* (upper layer), hosting agents dealing with the game strategy. Thanks to this layered architecture, hardware-level interactions and

[1]A system is called *soft real-time* if it is able to take into account deadlines, but if a deadline is not met, it has no particular consequences [19], [20].

[2]`http://www.eurobot.org`

[3]`http://pciso.diit.unict.it/~eurobot`

intelligent activities are clearly decoupled, making the design and implementation of the software system more easy, and also allowing the programmer to easily reuse some parts and/or to improve or change the functionalities of the system.

The paper is structured as follows. Section II describes the game that robots have to play at Eurobot 2006. Section III illustrates the basic hardware and mechanical structure of the robot developed. Section IV deals with the software architecture of the control system of the robot, describing the agents composing the system, their role and their activities. Section V discusses some implementation issues. Section VI reports our conclusions.

II. The Game at Eurobot 2006

Eurobot is an international robotics competition which involves students and amateurs in challenging and amazing robot games. The main target of the event is to encourage sharing of technical knowledge and creativity among students and young people from Europe and, in the last two editions, from all around the world.

Every year a different robotic game is chosen, so that all teams start from the same initial status and new teams are stimulated to participate. Here we report an overview of the rules for the 2006 edition of Eurobot[4], when the selected game was "Funny Golf", a simplified version of a golf game where robots had to search balls in the play-field and to put them into holes of a predefined colour.

A. Field and Game Concepts

As Figure 1 shows, the play-field is a green rectangle of 210x300 mm, surrounded by a wooden border. Borders on the short sides of the field have a red (resp. blue) central stripe which delimits the starting area for each robot. The field has 28 holes, 14 of them encircled by red rings and the other by blue rings. A total amount of 31 white balls and 10 black balls are available during the game. Fifteen white balls and two black balls are placed into the playing area at predefined positions, while four more black balls are randomly positioned into holes, two for each colour. The remaining balls (sixteen white and four black) could be released by automatic ejection mechanisms positioned at each corner of the field. Finally, four yellow "totems" are positioned into the field and are both obstacles for robots and switches for the ball–ejection mechanisms.

Robots must be absolutely autonomous: any kind of communication with the robot, both wired or wireless, is not allowed during matches. Robots have spatial limits, in terms of height, perimeter and so on, and have to pass a homologation test before being accepted for the competition. Each robot can also use any kind of positioning and obstacle-avoidance system, and supports are provided at the borders of the playing area to place (homologated) beacons, if needed.

[4]This edition took place in Catania, Italy.

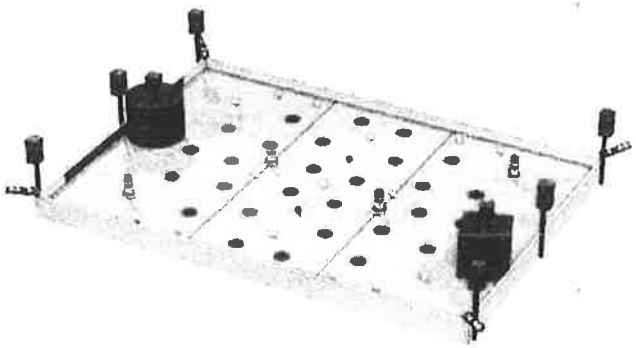


Fig. 1. The playing area

B. Playing Funny Golf

Before starting, each robot is assigned a colour, either red or blue. Robots start from the border opposite to their playing area, i.e. in the opponent's field, and at least one side of the robot must touch the starting area (short border of the play-field). After robots are placed into the field and all setup procedures by team members are over, the referees choose the positions of totems and black balls, by means of a random selection. When all the components in the play-field are set up, one of the referees gives the start signal and robots can play. Each robot has to put as many white balls as possible into its holes in a time of 90 seconds. Robots can also put black balls into opponent's holes, suck them out of their holes, or even suck white balls out of opponent's holes. There is no restriction about strategies or techniques adopted in order to search, catch, release and suck out balls. It is not allowed to hurt the other robot or to obstacle or damage it in any way. It is neither permitted to damage the playing area or playing objects (such as balls, holes, totems or ejecting mechanisms). The ejecting mechanisms can be triggered by touching a totem for a given amount of time: this closes a simple electric circuit and allow balls into the ejector to be released. At the end of the match, each white ball in the right hole is considered as a point, and the robots which has the highest score is the winner.

III. The DIIT Team Robot

Building an autonomous robot to play "Funny Golf" is not a trivial task, since different subsystems are needed to perform ball searching, catching and putting, and many physical constraints are imposed by game rules themselves. The following subsections describe the robot realized by the DIIT Team[5], which participates (for the first time) to the 2006 Eurobot edition.

A. The Core

An embedded VIA 900Mhz CPU is the core of the robot. We used a motherboard produced by AXIOM Inc. which incorporates Ethernet, parallel port, 4 serial ports, USB, IDE

[5]"DIIT" means Dipartimento di Ingegneria Informatica e delle Telecomunicazioni.

controller and other amenities (such as PC/104 bus, not used in our configuration). The operating system used is a Debian GNU/Linux (Etch), with kernel 2.6.12 and glibc 2.3.5. GNU/Linux was selected because of its stability and robustness, that are important features when driving a robot.

B. Locomotion System

In order to guarantee fast movements, we decided to use a locomotion system based on two independent double-wheels, driven by DC motors. Wheels diameter is small enough to allow fast rotation and large enough to avoid holes. DC motors are directly connected to a motor-controller, driven by a RS232 serial line. The controller allows to set different speeds for each wheel, both for forward and backward directions. Each wheel is connected to an optical encoder, driven by a serial mouse circuitry, which feeds back to the software system information about real rotation speed and position of the wheel. This information is then used by the Motion Control agent to adjust the speed and the trajectory.

C. Vision

Searching balls in the playing area requires a kind of vision system to find them. We chose to use a simple USB webcam to capture video frames at a rate of about 4 frames/sec, still enough to guarantee an accurate and fast analysis of objects in the field. The webcam is able to "view" the field from 30 to 160 centimetres in front of the robot, with a visual angle of about 100 degrees in total. Frame grabbed by the webcam are passed to the "Object Detector" agent, which filters them to find balls (both black and white) and holes (both red and blue).

D. Catching and putting balls

Once balls are detected, it is necessary to put them, somehow, into the right hole. We decided to suck balls using a fan, and to choose where to put them using a simple selector, driven by a servo-motor. Balls are saved into a small buffer if they are white and the buffer has enough space, or ejected out if they are black or if the buffer is full. The fan is powerful enough to suck balls at a distance of about 12 centimetres from the front side of the robot, and it is also able to suck balls out of holes when a special small bulkhead on the front side is closed. A simple release mechanism, which uses a servo-motor, allows balls to be dropped down to the final piece of the buffer and to fall into a hole.

E. Sensors and Positioning

Many sensors have been used onto the robot. First of all, a colour sensor for balls is installed into the ball selector, to recognise if a sucked ball is white or black. A complex system of proximity sensors is installed in the bottom side of the robot to recognise holes when the robot walks over them, and to allow a smart and fine positioning during the ball putting phase. A presence sensor (made by a simple LED–photo-resistor couple) is placed in the final part of the buffer, to reveal the presence of a ball ready to be dropped into a hole. The same sensor is used to detect when the ball has been successfully put into a hole.

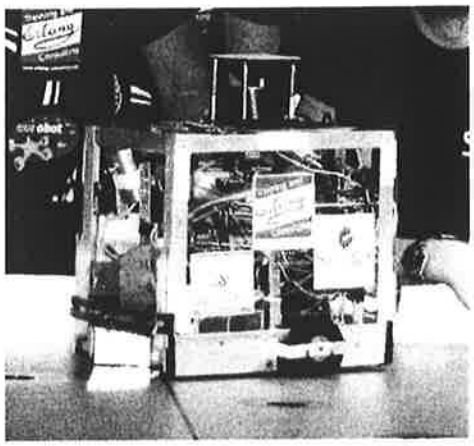


Fig. 2. The Robot in the playing area

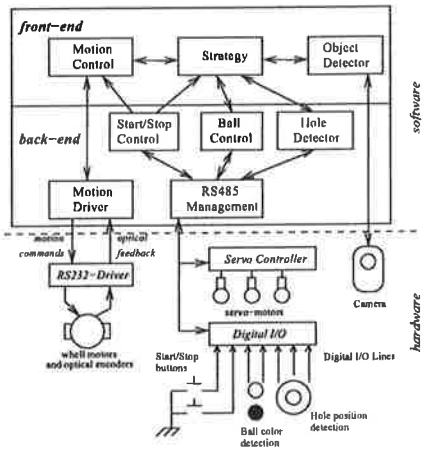


Fig. 3. Hardware/Software Architecture

IV. The Robot's Software Architecture

Given the robot structure illustrated in the previous Section, it is clear that the implementation of the system to control it has to face some problems that are not present in traditional (only software) multi-agent systems: the interface with physical sensors and actuators. For this reason, the basic software architecture of the robot, which is sketched in Figure 3, is composed of *two layers*, (*i*) a lower one, called the *back-end*, including reactive-only agents, responsible for a direct interaction with the hardware, and (*ii*) a higher layer, called the *front-end*, hosting the "robot's intelligence" by means of a set of agents implementing the artificial vision system, the game strategy, the motion control, etc., and interacting with back-end's agents in order to sense and act onto the environment. All of these agents comply with an ad-hoc model which, together with the details on functionality of the overall system, is described in the following Subsections.

A. Agent Model

As reported in Section I, due to real time requirements and other peculiarities of a robotic application, well-know Java-based agent platforms cannot be employed; therefore, according to authors' past research work [21], [10], [12], [11], [13], [15], [14], [9], we decided to use the Erlang language [5], [4], [1] for the development of the robot's software system. In addition to its soft-real time features, Erlang has a concurrent and distributed programming model that perfectly fit the model of multi-agent systems: an Erlang application is in fact composed by a set of *independent processes*, each having a *state*, *sharing nothing* with other processes and communicating only by means of *message passing*. Such processes can be all local (i.e. in the same PC) or spread over a computer network; this is transparent to the application because the language constructs for sending and receiving messages do not change should the interacting processes be local or remote.

Given these features and the requirements for the robot control application, a suited agent model has been developed, which is based on two abstractions called *BasicFSM* and *PeriodicFSM*. The former, *BasicFSM*, is essentially a finite-state machine model, in which transitions are triggered by either the arrival of a message or the elapsing of a given timeout, and a specified per-state activity is executed (one-shot) when a new state is reached. The latter, *PeriodicFSM*, is instead a finite-state machine in which transitions are activated only by the arrival of a message, while the per-state activity is executed, when a state is reached, *periodically*, according to a fixed time period and within a deadline, which is equal to the period itself.

As it will be illustrated in the following, BasicFSM model is used for front-end agents, while the PeriodicFSM model is essentially exploited for those interacting with sensors and actuators and thus running in the back-end.

B. The Back-End

As Figure 3 illustrates, the *back-end* layer is composed by the following agents: *Motion Driver*, *RS485 Management*, *Start/Stop Control*, *Ball Control* and *Hole Detector*. All of these agents use the PeriodicFSM model but only the first two are directly connected with hardware resources.

The *Motion Driver* agent is in charge of driving wheel motors and gathering feedback from optical encoders. It basically handles messages (sent by front-end agents) specifying the *speed* to set for the left and right wheel, forwarding it (after measurement unit conversion) to the motor controller connected through the RS232 line. On the other hand, its periodic activity entails receiving the feedback from optical encoders (i.e. tick count), acquired through another RS232 line, and then computing tick frequency, thus evaluating the real speed of the wheels: the obtained value is used to adjust the value(s) sent to motor controller in order to make each wheel to reach the desired speed[6].

[6]This is obtained by means of a proportional-integrative-derivative software controller.

The *RS485 Management* agent is responsible for driving two external boards connected, to the PC, through the same RS485 serial bus: a controller for servo-motors and a board offering a certain number of I/O digital lines. Since each servo-motor and each I/O line is then used by different agents, the RS485 Management acts as a de-/multiplexer for actions and sensed data. Its periodic activity is the sampling of digital inputs, by means of a request/reply transaction through serial messages exchanged with the I/O board; polled data are thus stored in the agent's state in order to make them available for requests coming from other agents. In addition, the RS485 Management is able to receive messages containing commands to be sent to servo-motor, through the servo-controller; in particular, each command specifies the servo-motor to drive and the rotation angle to be set.

The *Start/Stop Control* agent is a reactive one that periodically queries the RS485 Management in order to check if the "start" or "stop" buttons have been pushed. On this basis, it sends appropriate start/stop messages to the Strategy agent (see below) in order activate (resp. block) its behaviour when a match begins (resp. ends). Since the duration of a match is fixed (90 seconds), this agent embeds also a timer that, armed after a start, automatically sends a stop message when the 90 seconds are due.

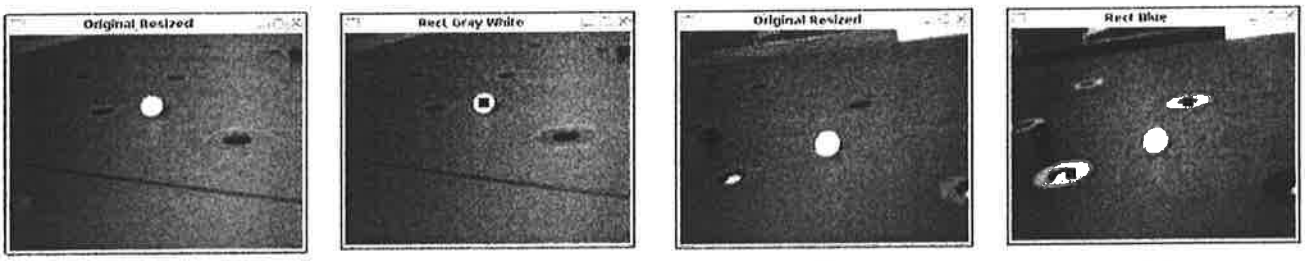
The *Ball Control* agent is responsible for managing the ball sucking system, the buffer and the ball release system. During its periodic activity, it queries the RS485 Management agent in order to check the input lines signalling that a new ball has been sucked: if this event occurs, on the basis of the colour of the ball[7], it drives the sucking system's arm servo-motor in order to put the ball in the buffer—if the ball is white and the buffer is not full—or to throw the ball away—if the ball is black or the buffer is full. This agent also holds the number of balls in the buffer, information that, queried by the Strategy agent, is used by the latter to control robot behaviour. As for ball release, the Ball Control agent, following a proper command message, is able to interact with the RS485 Management agent and thus drive the servo-motor controlling the release of a ball. Finally, by checking the status of another input digital line, the Ball Control agent is able to understand if a released ball has been successfully put into a hole.

The last agent of the back-end, the *Hole Detector*, reads, through a proper interaction with the RS485 Management agent, the data coming from proximity sensors placed under the robot for hole detection and positioning. It is able to understand the position of the robot, with respect to the hole to catch, and can thus forward this information to the Strategy agent, which, in turn, will drive the wheels to centre the hole and put the ball into it.

C. The Front-End

The front-end layer implements the high-level activities that drive the robot to reach its goal, i.e. placing the most quantity

[7]The colour is detected through a sensor connected to another digital input line.



(a) (b) (c) (d)

Fig. 4. Recognition by Object Detector agent

of balls into its holes. This layer is composed of three agents: *Object Detector*, *Motion Control* and *Strategy*.

The *Object Detector* has the task of observing the playing area, by means of a USB camera, detecting the objects needed for the game, i.e. balls and holes, and computing their coordinates with respect to the robot position. Since it uses a computation-intensive image manipulation algorithm, this is the sole agent written in C and not in Erlang[8]. This algorithm, whose execution is triggered by a suited message sent by the Strategy agent, exploits artificial vision techniques and performs a series of transformation (i.e. filtering, threshold, binarisation) on RGB planes of each frame acquired in order to isolate and recognise the required objects. Figure 4 reports some screen-shots of the functioning of the Object Detector. In particular, Figures 4a and 4c show two acquired frames, while Figures 4b and 4d illustrate the filtered images with the objects (respectively a white ball and two blue holes) detected by the agent.

The *Motion Control* agent, which is the only PeriodicFSM type, has the task of controlling the robot's path: it receives, from the Strategy agent, messages containing commands for robot positioning, such as *go to X,Y* or *rotate T*, computes the speed of the wheels needed to reach the target, and sends such speeds to the Motion Driver agents. Moreover, in order to ensure that the target is reached, the Motion Control agent periodically requests to Motion Driver the tick count of optical encoders and calculates the absolute position and orientation of the robot [7]. These values are thus compared with the target, making subsequent speed adjustment, if necessary[9]. Another task of the Motion Control agent is obstacle detection. Since the robot has no sensors to detect if an obstacle (e.g. the opponent's robot, a totem, etc.) is in front of it, the Motion Control agent checks if there is no wheel movement within a certain time window (given that wheel's speeds are greater than zero); if this is the case, an obstacle exiting algorithm is started, which entails to move the robot backwards and then rotate it.

The last agent, *Strategy*, is the "brain" of the robot. Being a BasicFSM agent, it is responsible of collecting and putting

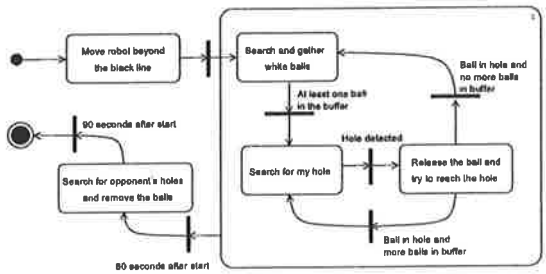


Fig. 5. Strategy Agent Behaviour

together information about environment and robot subsystems to obtain a valuable and effective playing strategy. Even if the field is mostly immutable (except for the position of totems, which are set before each match) and many of the balls involved are still in fixed position, we chose to implements an intelligent and adaptive strategy instead of a simple "fixed–path" one. For this reason the Strategy agent has to adaptively choose the right action to perform at each time, elaborating data coming from other agents. As Figure 5 illustrates, the very first step of the implemented strategy is "move beyond the first black line", since this guarantees the collection of at least one point[10]. This is performed by suitable commands sent to Motion Control agent. When the black line has been passed, the main strategy loop begins. First the robot looks for white balls and suck them into the buffer: if any white ball is seen by the Object Detector, then the Motion Control agent is issued the commands needed to reach the ball; on the other hand, if no ball has been detected, the Strategy agent tries to search elsewhere, by rotating of a random angle in order to look at other zones of the field. When a ball has been sucked and the Ball Control agent reports the presence of at least one white ball into the buffer, the Strategy agent starts to search a right hole to drop it into (i.e. a hole of the colour assigned to the team, either red or blue), looking at messages from Detector and moving toward a hole as soon as it has been found. When the selected hole is no more visible (i.e. outside the camera scope) a ball is released and, by means of messages coming from Hole Detector, a sequence of commands for fine positioning are sent to Motion Control. If the hole is centred

[8]It uses the OpenCV library [2], which provides a set of fast and optimised image manipulation functions. Proper Erlang-to-C library functions allows this agent to interact with Erlang processes.

[9]Also in this case, a proportional-integrative-derivative software controller is employed.

[10]If the robot does not pass the first black line, then it obtains no points at the end of the match.

and the ball goes into it, the Ball Control agent sends a "Ball Successfully Dropped" message, so the Strategy agent decides to search another hole, if more white balls are present into the buffer, or to look for more white balls. If the ball is not dropped into a given amount of time (for example because of errors in fine positioning) the Strategy agent searches another hole and tries to drop the ball into it. Finally, in the last 30 seconds of game, the Strategy agent tries to find opponent's holes to suck white balls out of them.

V. Implementation Issues

As it has been previously said in the paper, with the exception of the Object Detector, the system has been implemented using the Erlang language. However, even if our research group has realized a FIPA-compliant Erlang agent platform (called eXAT [10], [12], [11], [13], [15], [14]), we did not use it in order to avoid overhead introduced by platform's components for inference, behaviour handling, standard FIPA messaging, etc. This is required in order to have a fast and effective support for agents, rather than the possibility of interacting with other external agents (according to Eurobot rules, the robot must be autonomous and not connected to any network). To this aim, each agent of the robot has been encapsulated in an Erlang process and a suitable library has been developed to support the BasicFSM and PeriodicFSM deadline-aware abstractions. Message passing has been realized by means of the native Erlang constructs to perform inter-process communication (which are designed to be very fast): this resulted in an optimised code able to meet to real-time requirements of the target application.

VI. Conclusions

This paper described the architecture of an autonomous mobile robot, developed by the DIIT Team of the University of Catania to participate to the Eurobot competition. A multi-agent system has been employed for this purpose, composed of several agents in charge of both interacting with physical sensors and actuators, and supporting the game strategy for the robot. A layered architecture has been designed to clearly separate the aspects above—physical world interface and intelligence—and to favour design, modularity and reuse. Due to real time constraints, the system has been implemented using the Erlang language by means of a proper library to support the abstraction needed for using agents in a robotic environment. This allowed us to develop a fast code able to effectively support robot's activities.

VII. Acknowledgements

The authors wish to thank so much all the other components of the Eurobot DIIT Team, who gave a terrific and fundamental contribution in the realization of the robot described in this paper and made this experience not only very useful but also very funny.

These people are **Roberto Di Salvo, Andrea Nicotra, Luca Nicotra, Massimiliano Nicotra, Stefano Palmeri, Francesco Pellegrino, Matteo Pietro Russo, Carmelo Sciuto, Danilo Treffiletti** and **Carmelo Zampaglione**.

Moreover, the authors wish to thank also the official sponsors of the Eurobot DIIT Team, which are **Siatel Srl** (from Catania, Italy) and **Erlang Training & Consulting Ltd**[11] (from London, UK), that, with their support, contributed to make our dream real.

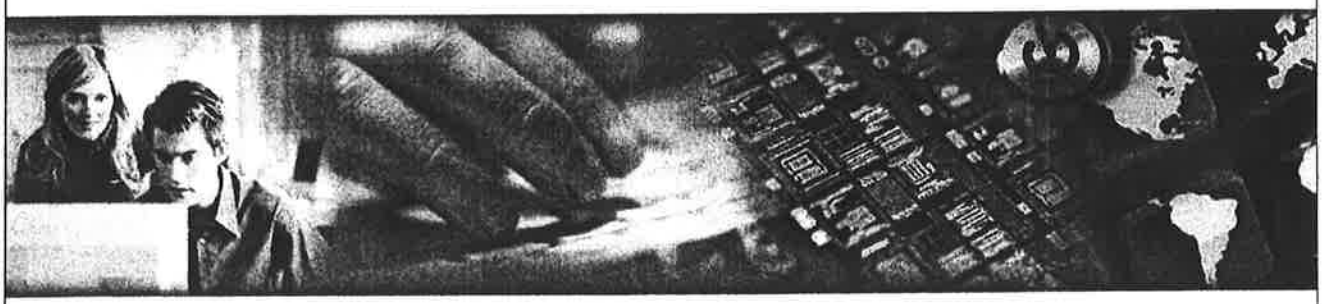
References

[1] "http://www.erlang.org. Erlang Language Home Page," 2004.
[2] "http://opencvlibrary.sourceforge.net/," 2006.
[3] J. Armstrong, B. Dacker, R. Virding, and M. Williams, "Implementing a Functional Language for Highly Parallel Real Time Applications," 1992.
[4] J. L. Armstrong, "The development of Erlang," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, A. Press, Ed., 1997, pp. 196–203.
[5] J. L. Armstrong, M. C. Williams, C. Wikstrom, and S. C. Virding, *Concurrent Programming in Erlang, 2nd Edition*. Prentice-Hall, 1995.
[6] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.
[7] J. Borenstein, H. R. Everett, and L. Feng, *Where am I? — Systems and Methods for Mobile Robot Positioning*. WWW, University of Michigan, USA, http://www-personal.engin.umich.edu/~johannb/position.htm, 1996.
[8] A. Corsaro and C. Santoro, "Design Patterns for RTSJ Application Development," in *Proceedings of 2nd JTRES 2004 Workshop, OTM'04 Federated Conferences*. LNCS 3292, Springer, Oct. 25-29 2004, pp. 394–405.
[9] A. Di Stefano, F. Gangemi, and C. Santoro, "ERESYE: Artificial Intelligence in Erlang Programs," in *Erlang Workshop at 2005 Intl. ACM Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 25 Sept. 2005.
[10] A. Di Stefano and C. Santoro, "eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang," in *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2003)*, Villasimius, CA, Italy, 10–11 Sept. 2003.
[11] ——, "eXAT: A Platform to Develop Erlang Agents," in *Agent Exhibition Workshop at Net.ObjectDays 2004*, Erfurt, Germany, 27–30 Sept. 2004.
[12] ——, "Designing Collaborative Agents with eXAT," in *ACEC 2004 Workshop at WETICE 2004*, Modena, Italy, 14–16 June 2004.
[13] ——, "On the use of Erlang as a Promising Language to Develop Agent Systems," in *AI*IA/TABOO Joint Workshop on Objects and Agents (WOA 2004)*, Torino, Italy, 29–30 Nov. 2004.
[14] ——, "Supporting Agent Development in Erlang through the eXAT Platform," in *Software Agent-Based Applications, Platforms and Development Kits*. Whitestein Technologies, 2005.
[15] ——, "Using the Erlang Language for Multi-Agent Systems Implementation," in *2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'05)*, Compiégne, France, 19–22 Sept. 2005.
[16] P. Dibble, *Real-Time Java Platform Programming*. Prentice Hall PTR, 2002.
[17] I. Infantino, M. Cossentino, and A. Chella, "An agent based multilevel architecture for robotics vision systems." in *Proceedings of the International Conference on Artificial Intelligence, IC-AI '02, June 24 - 27, 2002, Las Vegas, Nevada, USA, Volume 1*, 2002, pp. 386–390.
[18] E. Johansson, M. Pettersson, and K. Sagonas, "A High Performance Erlang System," in *2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, Sept. 20–22 2000.
[19] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
[20] Liu, J. W. S., *Real-Time Systems*. Prentice Hall, 2000.
[21] C. Varela, C. Abalde, L. Castro, and J. Gulias, "On Modelling Agent Systems with Erlang," in *3rd ACM SIGPLAN Erlang Workshop*, Snowbird, Utah, USA, 22 Sept. 2004.
[22] A. Wellings, *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[11] http://www.erlang-consulting.com

Comprehensive Erlang Archive Network

Stockholm, November 9, 2006



Process One

What is CEAN ?

- **An Erlang distribution and packaging system**
  - a package has a description file, and an archive file available on internet
  - a package can contain an Erlang application, an OTP Library, any user contribution code
- **Provides easy Erlang installation without compilation**
  - Basic install is a 3Mb self extractable binary archive running on:
  - linux-alpha, linux-amd64, linux-arm, linux-hppa, linux-ia64, linux-m68k, linux-mips, linux-mipsel, linux-powerpc, linux-s390, linux-sparc, linux-x86, darwin-powerpc, darwin-x86, freebsd-x86, openbsd-x86, sunos-sparc, windows
- **Allows to install/uninstall/upgrade Erlang packages**
  - no need to browse the internet, this is done using erlang shell
  - package updates can be lested at any time using erlang shell
- **Allows to create custom Erlang/OTP installation on production systems**
  - one can deploy the basic CEAN Erlang bootstrap and start a distribution profile script
- **Aims to be a centrale place to find erlang code**
  - CEAN can use tar, zip, cvs, svn to fetch sources to build packages. So any code we can find on the internet can be packaged making binary version of the code be available on CEAN site. Up to 200 packages already available.
- **Brings interresting statistics on Erlang use**
  - Downloaded packages, system and architecture used, Erlang version...

Process One

History

- **REPOS**
  - Version 1.0 in December 2004
  - Repository of Erlang-Projects.Org Software selection
  - CD-ROM image collecting major ready-to-work Erlang software for Linux x86, MacOS X PPC, and Windows
- **ErlRT**
  - Version 1.0 in April 2006
  - Allows minimal Erlang installation and provides package repository for Erlang/OTP and few applications
  - Automatic package generation improvements
- **CEAN**
  - Version 1.0 in November 2006
  - Is a merge of REPOS and ErlRT
  - Includes standalone application generator, more packages (Erlang/OTP, user contribs, jungerl, etc...), improved web site.
  - Contributor script improvements
  - Makes use of BitRock to generate graphical application installer

Process One

CEAN in use

- **The .pub files**
  The way to describe a package
- **The contributor archive**
  To allow easy contribution, making binary archives
- **The CEAN build process**
  How to transform binary archives to CEAN packages
- **The CEAN web site**
  First overview
- **The CEAN library usage**
  The way to use CEAN using Erlang shell
- **Some Examples**
  Let's try....

Process One

The .pub files

- **One .pub file to describe a CEAN packaged library or software**

```
{author, {"Process-One", "contact@process-one.net"}}.
{packager, {"Christophe Romain", "christophe.romain@process-one.net"}}.
{name, "ejabberd"}.
{vsn, "1.1.2"}.
{depends, ["asn1","crypto","mnesia","odbc","ssl","tools"]}.
{keywords, ["jabber","xmpp","server"]}.
{summary, "Erlang jabber/XMPP server"}.
{abstract, "ejabberd is a high-performance instant messaging server. "
 "An instant messaging server allows to transfer presence and status information "
 "between users connected to server and support real-time communications between them.<br>"
 "ejabberd relies on XMPP (<i>eXtensible Messaging and Presence Protocol</i>) protocol."}.
{home, "http://www.process-one.net/en/projects/ejabberd"}.
{sources, {svn, "http://svn.process-one.net/ejabberd/branches/ejabberd-1.1.2"}}.
```

note: abstract can contain HTML code (viewable on CEAN web site)

The contributor archive

- **One archive to make CEAN packages from scratch**
  This archive includes an erlang bootstrap, all .pub files, and an automatic build script
- **Performs the initial build process** *(for each package)*

$$\text{.pub} \xrightarrow{\text{fetch}} \text{sources} \xrightarrow{\text{build}} \text{arch.tar} + \text{common.tar}$$

- **Sends architecture specific binaries to Process-One**

$$\text{sum(arch.tar)} \xrightarrow{\text{email}} \text{contrib@process-one.net}$$

- **Allows people to add new package (adding .pub files)**

$$\text{new .pub} \xrightarrow{\text{email}} \text{contrib@process-one.net}$$

The CEAN build process

- **Build package for each contribution and each available architecture**

  This stage allows to apply some patchs needed to support multi-platform integration. Automatic transformation are done to comply with CEAN packaging standard.

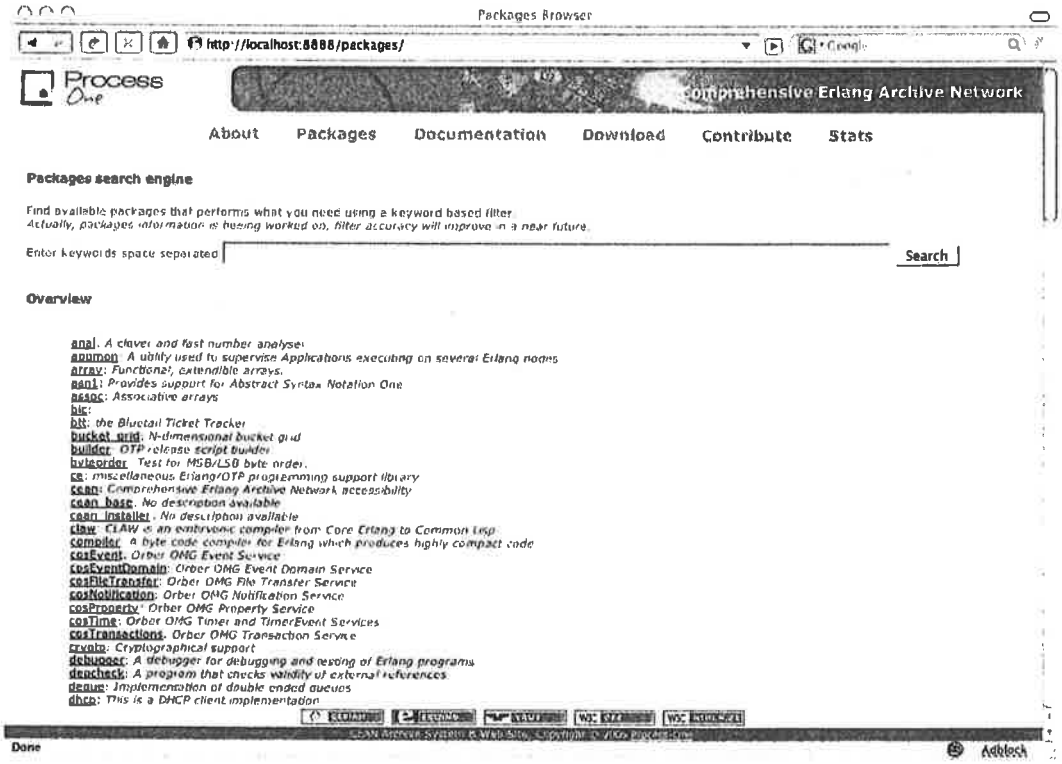
arch.tar+common.tar
CEAN patchs
$\xrightarrow{\text{build process}}$
CEAN repository

- **Generate standalone application archive and installer**

application+dependencies
Erlang bootstrap
From CEAN repository
$\xrightarrow{\text{merge}}$
app-arch-standalone.tgz
$\downarrow$ BitRock installer
app-arch-installer.bin

The CEAN web site

- **The packages browser**



The CEAN web site

Package description

Packages Browser

http://localhost:8888/packages/index.yaws?action=detail&name=yaws

Process One — Comprehensive Erlang Archive Network

About Packages Documentation Download Contribute Stats

Packages search engine

Find available packages that performs what you need using a keyword based filter
Actually, packages information is being worked on, filter accuracy will improve in a near future

Enter keywords space separated [ ] Search

Overview

yaws

Package: yaws-1.65
Author: Claes Wikstrom *klacke at hyber dot org*
Packager: Christophe Romain *christophe dot romain at process-one dot net*
Date
Depends:
Sources: (tar) http://yaws.hyber.org/download/yaws-1.65 tar gz
Home Page: http://yaws.hyber.org
Summary: YAWS is an ERLANG web server
Abstract: YAWS has a wide feature set, it supports:
- HTTP 1.0 and HTTP 1.1
- Static content page delivery
- Dynamic content generation using embedded ERLANG code in the HTML pages
- Common Log Format traffic logs
- Virtual hosting with several servers on the same IP address
- Multiple servers on multiple IP addresses
- HTTP tracing for debugging
- An interactive interpreter environment in the Web server while developing and debugging the web site
- RAM caching of commonly accessed pages
- Full streaming capabilities of both up and down load of dynamically generated pages
- SSL
- Support for WWW-Authenticated pages
- Support API for cookie based sessions
- Application Modules where virtual directory hierarchies can be made
- Embedded mode

Done — Adblock

The CEAN web site

Downloads

CEAN Downloads

http://localhost:8888/download/

Process One — Comprehensive Erlang Archive Network

About Packages Documentation Download Contribute Stats

CEAN Base System

In order to use CEAN, you need to install the base system. It includes minimal Erlang/OTP and CEAN library that will allow you to customize your installation

Select archive type
- *installer* is a self extractable archive (does not work under windows)
- *gzipped tar* is a standard tar.gz archive
- *zip* is a zip archive (for windows users)

Select the Erlang/OTP version included into base system
Select distribution
- *Production* includes the minimum required with stripped binaries
- *Developer* includes non stripped binaries, sources and documentation

Select Operating System and architecture

Archive type [installer] *note: on Windows, installer is a zip file by now*
CEAN Erlang/OTP [R10B] . [Production] distribution, running on [Apple MacOSX PowerPC] Download

CEAN Packaged Applications

Available soon, ready to use applications installer.

Archive type [installer] *note: on Windows, installer is a zip file by now*
Application [Erlang/OTP Full (Standard)] , running on [Apple MacOSX PowerPC] Download

Done — Adblock

The CEAN web site

- **Statistics**

CEAN Usage Statistics

http://localhost:8888/stats/

Process One

Comprehensive Erlang Archive Network

About Packages Documentation Download Contribute Stats

General CEAN statistics

Packages : 182
Platforms : 4
Contributors : 60

Erlang/OTP related version downloads

R10B : 18 (100.0%)

System architecture downloads

Apple MacOS/ PowerPC : 11 (35.5%)
Linux Intel x86 : 10 (32.3%)
REPOS : 0 (0.0%)
Microsoft Windows : 10 (32.3%)

Packages downloads

anal : 0 (0.0%)
appmon : 0 (0.0%)
array : 0 (0.0%)
atom : 0 (0.0%)
assoc : 0 (0.0%)
bif : 0 (0.0%)
btf : 3 (30.0%)
bucket_grid : 0 (0.0%)
builder : 0 (0.0%)
byteorder : 0 (0.0%)
ce : 0 (0.0%)
cean : 0 (0.0%)
cean_base : 0 (0.0%)
cean_installer : 3 (30.0%)
clow : 0 (0.0%)
compiler : 0 (0.0%)

Done

Adblock

The CEAN library usage

- **Lists available packages**
  cean:available(). % returns [List]
- **List installed packages**
  cean:installed(). % returns [List]
- **Check for a package status**
  cean:installed(stdlib). % returns bool()
- **Search for a package**
  cean:search("web server"). % returns [List]
- **Install a package and all its dependencies**
  cean:install(yaws). % returns ok
- **Uninstall a package**
  cean:uninstall(yaws). % returns ok
- **List all new versions of installed packages**
  cean:new(). % returns [List]
- **Upgrade a package**
  cean:upgrade(ejabberd). % returns bool()
- **Upgrade the whole distribution**
  cean:upgrade(). % returns ok
- **Check CEAN version**
  cean:version(). % returns string()

7(8)

Some examples

- **CEAN Installation and usage**

```
chris@iBook:~> sh ./cean_installer.bin
please wait...
Erlang (BEAM) emulator version 5.4.13 [source] [hipe] [threads:0]
Eshell V5.4.13  (abort with ^G)
1> cean:installed().
["cean","ibrowse","kernel","stdlib"]
2> cean:install(mnesia).
+ mnesia md5=<<160,59,224,210,56,36,169,26,180,156,142,150,164,39,8,166>>
ok
3> mnesia:start().
ok
4> mnesia:system_info(tables).
[schema]
5> cean:installed(mnesia).
true
6> cean:uninstall(mnesia).
- mnesia
ok
7> cean:installed().
["cean","ibrowse","kernel","stdlib"]
```


Some examples

- **Package search**

```
1> cean:search("database").
[{"view_backup","Simple program for loading mnesia backup files"},
 {"safedets","A version of dets that never enters the repair mode"},
 {"rdbms","A relational database management layer on top of mnesia"},
 {"mnesia","A heavy duty real-time distributed database"},
 {"gridfile","Adaptable, Symmetric Multikey File Structure"},
 {"dynarray","Expanding array for heap-based storage"}]
2> cean:search("server").
[{"yaws","YAWS is an ERLANG web server"},
 {"shbuf","Erlang server for sharing Emacs buffers & Emacs-Lisp client"},
 {"nfs","NFS server"},
 {"inets","A set of services such as a Web server and a ftp client"},
 {"gen_leader",
  "This application implements a leader election behaviour modeled after gen_server. This
     behaviour intends to make it reasonably straightforward to implement a fully distributed server
     with master-slave semantics"},
 {"enfs","Minimal NFS v2 server in Erlang"},
 {"ejabberd","Erlang jabber/XMPP server"}]
```

Planning

- **CEAN 1.0 beta**
  - Full archives for R10B
  - Partial archives for R11B
  - Ejabberd self installer generated with CEAN
  - Planned for mid November 2006
- **CEAN 1.1**
  - Full archives for R11B
  - Automatic remote installation and deployment
  - CEAN as an 'on demand' code server
  - More self installable applications
  - Planned for Christmass

Process One

Testing a Media Proxy with QuviQ QuickCheck

Thomas Arts
John Hughes
Chalmers/ITU
QuviQ

Joakim Johansson
Ulf Wiger
Ericsson

QuickCheck: Properties not Test Cases

- Write *general properties* of code instead of test cases

```
prop_reverse() ->
  ?FORALL(Xs,list(int()),
  ?FORALL(Ys,list(int()),
    lists:reverse(Xs++Ys)==
    lists:reverse(Xs)++lists:reverse(Ys))).
```

- Test in many, many randomly generated cases

QuickCheck Testing

```
3> eqc:quickcheck(test:prop_reverse()).
...........Failed! After 12 tests.
[-3,2]
[-3,1]
Shrinking..........(10 times)
[0]
[1]
false
```

A random counter-example: Xs and Ys

NEW: Automatic simplification of failing cases

- Simplification is extremely important—separates the *signal* from the *noise!*

Why Did it Fail?

```
prop_reverse() ->
  ?FORALL(Xs,list(int()),
  ?FORALL(Ys,list(int()),
    lists:reverse(Xs++Ys)==
    lists:reverse(Xs)++lists:reverse(Ys))).
```

- QuickCheck says: Xs=[0], Ys=[1]
- reverse([0,1])==reverse([0])++reverse([1])?
- Xs, Ys the wrong way round

QuickCheck in a Nutshell

- Features
  - Properties, not test cases
  - Controlled random generation
  - Automated simplification
- Result
  - Compact, reusable test code
  - Specification *checked against the code*
  - Shortcut from property to bug

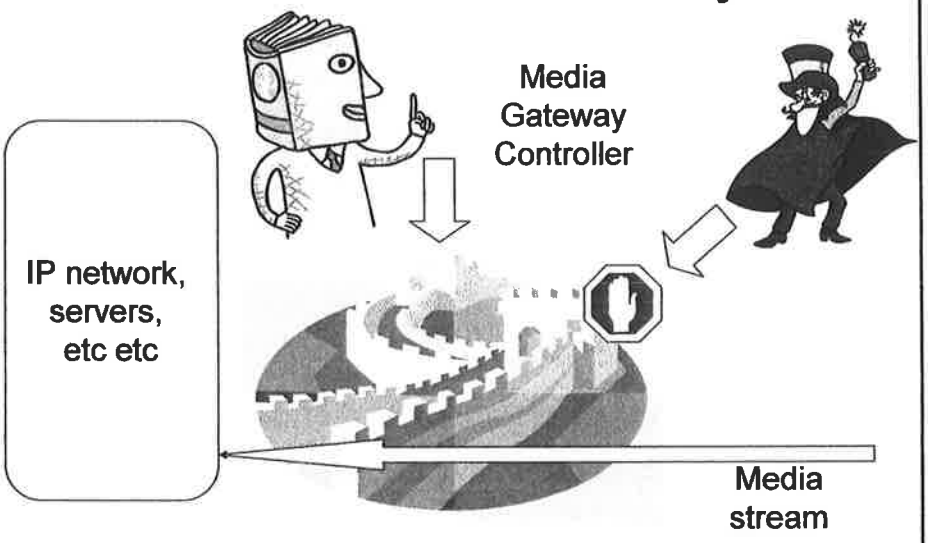
Property First Development?

- Code code code… quickcheck 

A Bug!
- Fix it… quickcheck 

A Bug!
- Fix it… quickcheck… quickcheck quickcheck quickcheck
- Code some more…

Ericsson's Media Proxy



How hard can it be?

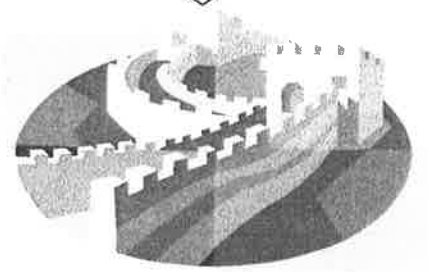
- All we need to do is open and close "media pinholes" when the controller says…
- Megaco H.248 protocol
  - ITU standard… 212 pages!
  - Ericsson Interwork Description… 183 pages!
  - Control software… 150KLOC
    - 20-30K Megaco

Our Strategy

QuickCheck

Sends random command sequences

Results make sense?



Completely random commands are all just rejected—non-sensical! Poor test data

Generating Sensible Messages

Message Generators

- **Example:** A *media descriptor* contains a list of streams
  - ASN.1 from the standard (simplified):

```
MediaDescriptor    ::= SEQUENCE
{
    streams        CHOICE
    {
        oneStream  StreamParms,
        multiStream SEQUENCE OF StreamDescriptor
    }
}
```

QuickCheck Generator

```
mediadescriptor(Streams) when Streams=/=[]->
    {mediaDescriptor,
     #'MediaDescriptor'{
        streams =
          case Streams of
               [{Id,Mode}] ->
                   oneof([{oneStream,streamParms(Mode)},
                          {multiStream,[stream(Id,Mode)]}]);
               _ ->
                   {multiStream,
                       [stream(I,M) || {I,M}<-Streams]}
          end}}.

stream(I,Mode) ->
    #'StreamDescriptor'{
        streamID = I, streamParms = streamParms(Mode)}.
```

Records generated by ASN.1 compiler

Message construction
Logic from the IWD
QuickCheck

Conditions in the IWD

Add Request						
Desc.	Desc.	Desc.	Properties,	Package	M	Comment
Media					M	
	Stream				M	Multiple Stream descriptors can be included.
		Local control			O	LocalControl will be included in all cases except when no media (m-line) is defined in the remote SDP.
			mode		O	The default value of the mode property is "Inactive", the property is not mandatory if the wanted value is "Inactive".

```
StreamParms        ::= SEQUENCE
{
   localControlDescriptor LocalControlDescriptor OPTIONAL,
   localDescriptor        LocalRemoteDescriptor OPTIONAL,
   remoteDescriptor       LocalRemoteDescriptor OPTIONAL,
   ...,
   statisticsDescriptor StatisticsDescriptor OPTIONAL
}
```

Remote SDP in here

Two Cases: With and Without Remote Media

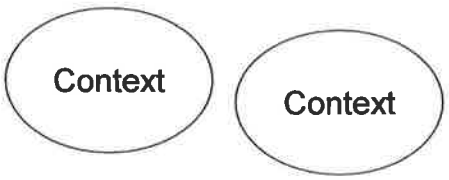
```
streamParms(Mode) ->
    ?LET(RemoteMediaDefined, bool(),
        if RemoteMediaDefined ->
                #'StreamParms'{
                    localControlDescriptor =
                      localControl(Mode),
                    localDescriptor =
                      localDescriptor(RemoteMediaDefined),
                    remoteDescriptor =
                      remoteDescriptor(RemoteMediaDefined)};
           true ->
                ......
        end).
```

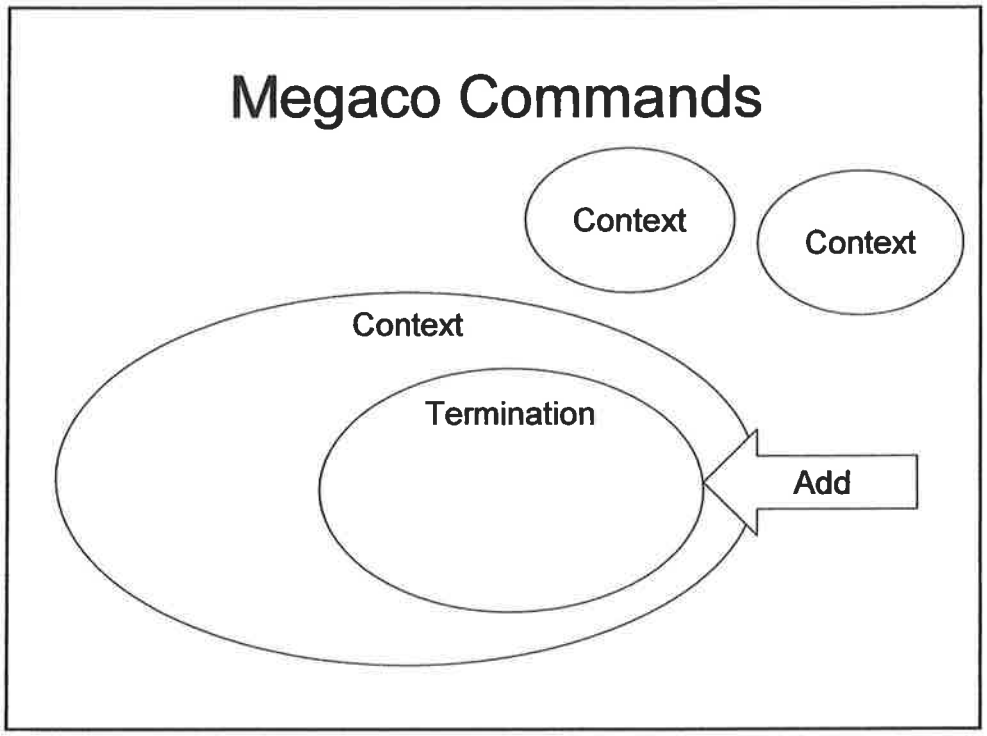
Included in this case

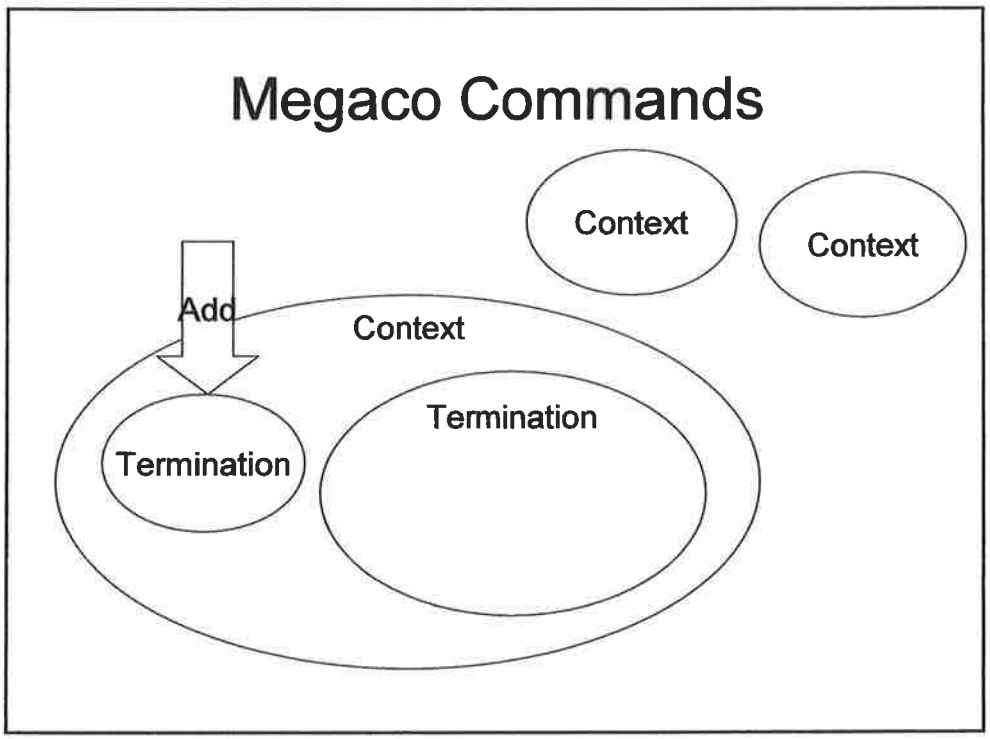
Passed on to ensure an m-line is generated

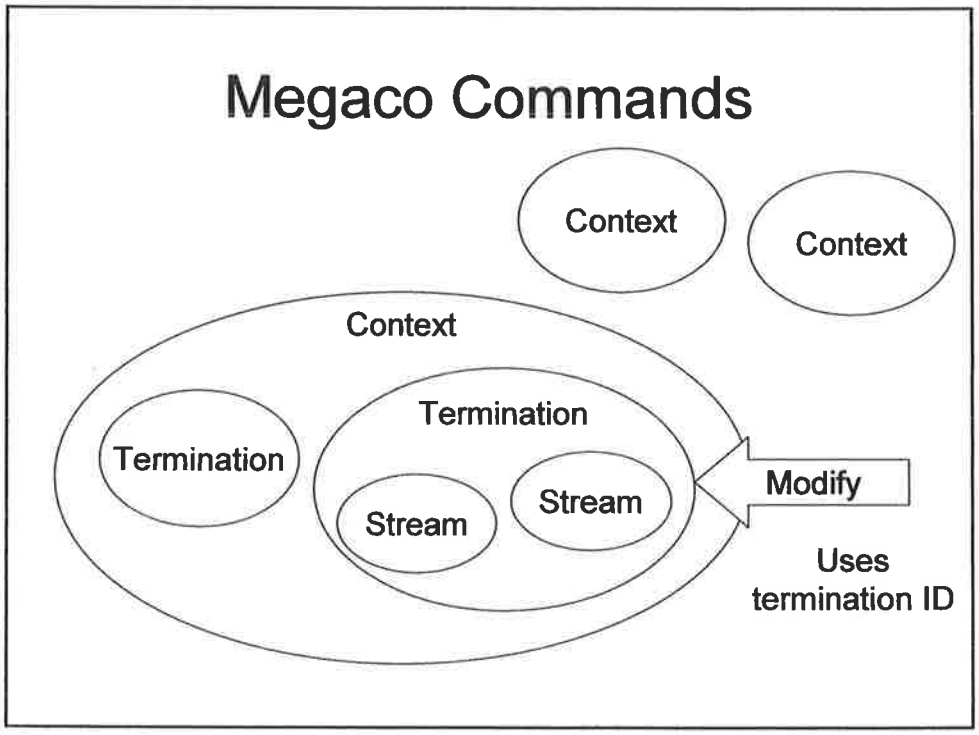
Generating Sensible Commands

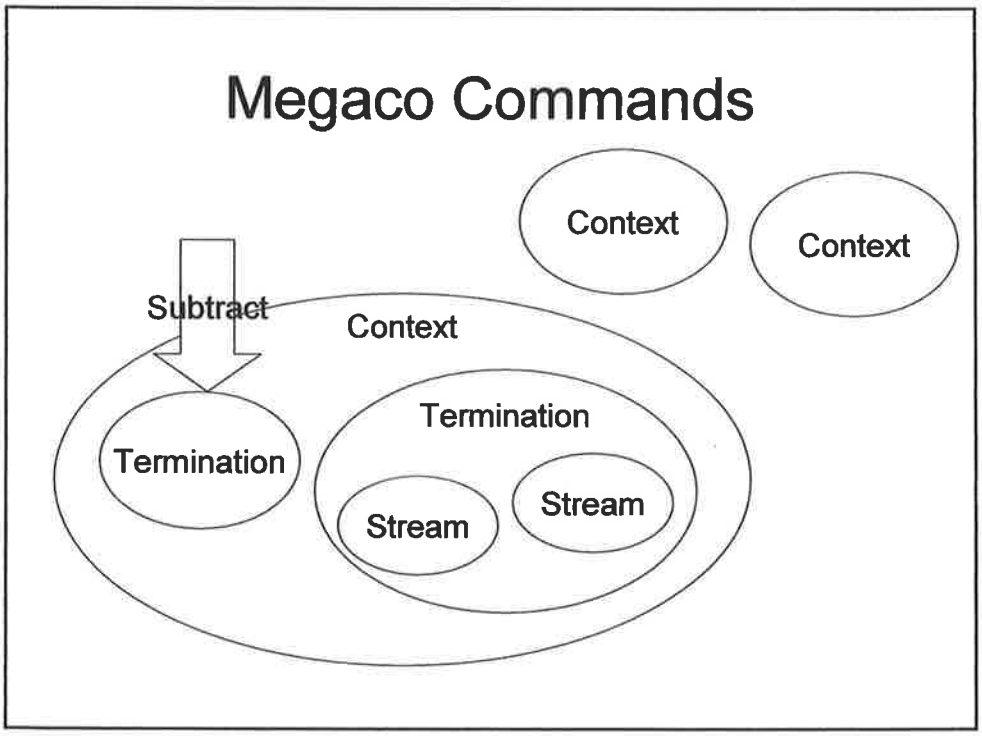
Megaco Commands



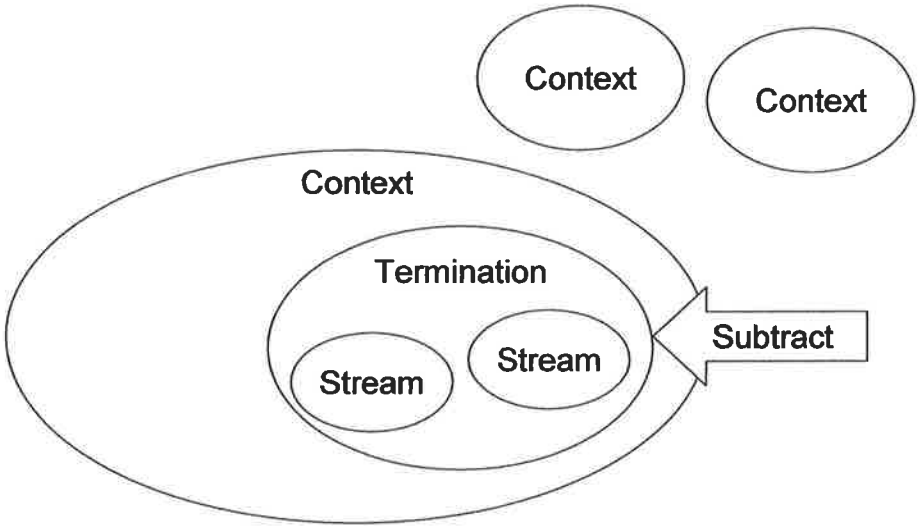

Megaco Commands
Context
Context
Context
Termination
Add


Megaco Commands
Context
Context
Add
Context
Termination
Termination

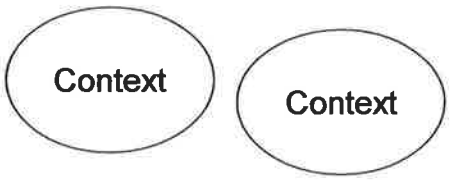

Megaco Commands
Context
Context
Context
Termination
Termination
Stream
Stream
Modify
Uses
termination ID


Megaco Commands
Context
Context
Subtract
Context
Termination
Termination
Stream
Stream

Megaco Commands



Megaco Commands



Sensible Command Sequences?

- Track the *state* of each test case
  - What calls in progress, which terminations in each call...
  - Simple to model in Erlang
- State machine model
  - *Preconditions* say which commands are appropriate
  - *Postconditions* check command results
  - *Next state* function specifies how commands behave
- QuickCheck library
  - generates valid sequences, tests, and shrinks them
  - (developed "just in time")

Faults

- Proxy was already well tested
- 6 days of work, mostly writing generators
- 5 faults found:
  - One in Megaco message encoding/decoding
  - Four command sequences crashed

Error-provoking Sequences

- Add → Mod → Streams must have two ends
- Add → Sub → Here today, gone tomorrow…
- Add → Add → Mod → …with differing numbers of streams

The Best Error!

Shrinking reduced 160 commands to seven!

Add → Add → Sub → Add → Sub → Add → Sub →

Due to data corruption here

No one in their right minds would test this!

Conclusions

- QuickCheck + Erlang
  - Simple declarative models of the SUT
  - Concise, maintainable test code
- Testing: a great application for FP
  - Performance irrelevant
  - No need to commit to Erlang in the product
- **A winning combination?**

Refactoring Erlang Programs *

Huiqing Li, Simon Thompson
University of Kent, UK

László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, Tamás Nagy
Eötvös Loránd University, Hungary

Abstract

We describe refactoring for Erlang programs, and work in progress to build two tools to give machine support for refactoring systems written in Erlang. We comment on some of the peculiarities of refactoring Erlang programs, and describe in some detail a number of refactorings characteristic of Erlang.

1. Introduction

Refactoring [6] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs. Separating general software updates into functionality changes and refactorings has well-known benefits. While it is possible to refactor a program by hand, tool support is invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the refactoring transformation itself, making the process less painful and error-prone.

Refactoring has been applied to a number of languages and paradigms, but most of the work in building tools has concentrated on object-oriented programming. In this paper we report on work in progress at our universities to build tools to support the refactoring of Erlang programs.

The paper begins with a brief introduction to refactoring, which is followed by a discussion of the particular question of refactoring Erlang systems. We then describe the approaches taken by our two teams: in a nutshell, the Kent team work over a enriched abstract syntax tree (AST), whereas the research at Eötvös Loránd University builds the representation in a relational database.

After describing the systems we speculate on what refactorings are the most appropriate to Erlang and are most useful to the working Erlang programmer, before concluding and surveying future work for both teams.

2. Refactoring

Refactorings transform the structure of a program with out changing its functionality. They are characterised by being *diffuse* and *bureaucratic*. They ate diffuse in the sense that a typical refactoring will affect the whole of a module or set of modules, rather than a single definition in a program, which is often the case for a program optimising transformation. They are bureaucratic in that they require attention to detail; for instance, taking into account the binding structure of a program.

Refactorings are not simply syntactic. In order to preserve the functionality of a program, refactorings require awareness of various aspects of the semantics of the program including types and module structure and most importantly the *static semantics* of the program: that is the scope of definitions, the binding structure of the program (the association between the use of an identifier and its definition), the uniqueness of definitions and so forth.

Each refactoring comes with a set of side conditions, which embody when a refactoring can be applied to a program without changing its meaning. Our experience of building refactoring tools so far shows that for most refactorings, the side-condition analysis is more complex than the program transformation part. Taking

* Supported by EPSRC in the UK, GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, Bolyai Research Fellowship and ELTE CNL in Hungary

a concrete example, the general side conditions for renaming an identifier could be as follows.

The existing binding structure should not be affected. No binding for the new name may intervene between the binding of the old name and any of its uses, since the renamed identifier would be captured by the renaming. Conversely, the binding to be renamed must not intervene between bindings and uses of the new name.

These side-conditions apply to most programming languages. However, each programming language may also impose its own particular constraints on this refactoring. For example, in an Erlang program using the OTP library, a user should not rename certain functions exported by a call-back module. For some languages, refactoring conditions can be checked at compile time; the more dynamic nature of Erlang means that some necessary conditions can only decided at run-time; we return to this point below.

2.1 Tool Support for Refactorings

Although it is possible to refactor a program manually, it would be both tedious and error-prone to refactor large programs this way. Interactive tool support for refactoring is therefore necessary, as it allows refactorings to be performed easily and reliably and to be undone equally easily.

A refactoring tool needs to get access to both the syntactic and static semantic information of the program under refactoring. While detailed implementation techniques might be different, most refactoring tools go through the following process: first transform the program source to some internal representation, such as an abstract syntax tree (AST) or database table; then analyse the program to extract the necessary static semantic information, such as the binding structure of the program, type information and so forth.

After that, program analysis is carried out based on the internal representation of the program and the static semantics information to validate the side-conditions of the refactoring. If the side-conditions are not satisfied, the refactoring process stops and the original program is unchanged, otherwise the internal representation of the program is transformed according to the refactoring. Finally, the transformed representation of the program need to be presented to the programmer in program source form, with comments and the original program appearance preserved as much as possible.

```
-module (sample).
-export([printList/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.
```

Figure 1. The initial program

```
-module (sample).
-export([printList/1, broadcast/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.

broadcast([H|T]) ->
    H ! "The message",
    broadcast(T);
broadcast([]) -> true.
```

Figure 2. Adding a new function naïvely

The Kent group are responsible for the project 'Refactoring Functional Programs' [7], which has developed the Haskell Refactorer, HaRe [9], providing support for refactoring Haskell programs. HaRe is a mature tool covering the full Haskell 98 standard, including "notoriously nasty" features such as monads, and is integrated with the two most popular development environments for Haskell programs: Vim and (X)Emacs. HaRe refactorings apply equally well to single- and multiple-module projects. HaRe is itself implemented in Haskell.

Haskell layout style tends to be idiomatic and personal, especially when a standard layout is not enforced by the program editor, and so needs to be preserved as much as possible by refactorings. HaRe does this, and also retains comments, so that users can recognise their source code after a refactoring. The current release of HaRe supports 24 refactorings, and also exposes an API [10] for defining Haskell refactorings and program transformations.

3. Refactoring Erlang Programs

Figures 1 - 5 illustrate how refactoring techniques can be used in the Erlang program development process.

```
-module (sample).
-export([printList/1]).

printList(L) ->
    printList(fun(H) ->
        io:format("~p\n", [H]) end, L).

printList(F,[H|T]) ->
    F(H),
    printList(F, T);
printList(F,[]) -> true.
```

Figure 3. The program after generalisation

```
-module (sample).
-export([printList/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 4. The program after renaming

The example presented here is small-scale, but it is chosen to illustrate aspects of refactoring which can scale to larger programs and multi-module systems.

In Figure 1, the function `printList/1` has been defined to print all elements of a list to the standard output. Next, suppose the user would like to define another function, `broadcast/1`, which broadcasts a message to a list of processes. `broadcast/1` has a very similar structure to `printList/1`, as they both iterate over a list doing something to each element in the list. Naïvely, the new function could be added by copy, paste, and modification as shown in Figure 2. However, a *refactor then modify* strategy, as shown in Figures 3 - 5, would make the resulting code easier to maintain and reuse.

Figure 3 shows the result of generalising the function `printList` on the sub-expression

```
io:format("~p/n",~[H])
```

The expression contains the variable `H`, which is only in scope within the body of `printList`. Instead of generalising over the expression itself, the transformation is achieved by first abstracting over the free variable `H`, and by making the generalised parameter a *function* `F`. In the body of `printList` the expression `io:format("~p/n", H])` has been replaced with `F` applied to the local variable `H`.

```
-module (sample).
-export([printList/1, broadcast/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

broadcast(Pids)->
    forEach(fun(H) ->
        H ! "The message" end, Pids).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 5. The program after adding a function

The arity of the `printList` has thus changed; in order to preserve the interface of the module, we create a new function, `printList/1`, as an application instance of `printList/2` with the first parameter supplied with the function expression:

```
fun(H) -> io:format("~p/n", [H]) end.
```

Note that this transformation gives `printList` a functional argument, thus making it a characteristically 'functional' refactoring.

Figure 4 shows the result of renaming `printList/2` to `forEach/2`. The new function name reflects the functionality of the function more precisely. In Figure 5, function `braodcast/1` is added as another application instance of `forEach/2`.

Refactorings to generalise a function definition and to rename an identifier are typical structural refactorings, implemented in our work on both Haskell and Erlang.

3.1 Language Issues

In working with Erlang we have been able to compare our experience with what we have done in writing refactorings for Haskell. Erlang is a smaller language than Haskell, and in its pure functional part, very straightforward to use. It does however have a number

of irregularities in its static semantics, such as the fact that it is possible

- to have multiple defining occurrences of identifiers, and
- to nest scopes, despite the perception that there is no shadowing of identifiers in Erlang.

Erlang is also substantially complicated by its possibilities of reflection: function names, which are atoms, can be computed dynamically, and then called using the `apply` operator; similar remarks apply to modules. Thus, in principle it is impossible to give a complete analysis of the call structure of an Erlang system statically, and so the framing of side-conditions on refactorings which are both necessary and sufficient is impossible.

Two solutions to this present themselves. It is possible to frame *sufficient* conditions which prevent dynamic function invocation, hot code swap and so forth. Whilst these conditions can guarantee that behaviour is preserved, they will in practice be too stringent for the practical programmer. The other option is to *articulate* the conditions to the programmer, and to pass the responsibility of complying with them to him or her. This has the advantage of making explicit the conditions without over restricting the programmer through statically-checked conditions. It is, of course, possible to insert assertions into the transformed code to signal condition transgressions.

Compared to Haskell users, Erlang users are more willing to stick to the standard layout, on which the Erlang Emacs mode is based. Therefore a pretty-printer which produces code according to the standard layout is more acceptable to Erlang users.

3.2 Infrastructure Issues

A number of tools support our work with Erlang. Notable among these is the `syntax-tools` package which provides a representation of the Erlang AST within Erlang. The extensible nature of the package allows syntax trees to be equipped with additional information as necessary. For example, Erlang Syntax Tools provides functionalities for reading comment lines from Erlang source code, and for inserting comments as attachments on the AST at the correct places; and also the functionality for pretty printing of abstract Erlang syntax trees decorated with comments.

The *Distel* infrastructure helps us to integrate refactorings with Emacs, and thus make them available within the most popular Erlang IDE.

4. Our Approaches

Both University of Kent and Eötvös Loránd University are now in the process of building a refactoring tool for Erlang programs, however different techniques have been used to represent and manipulate the program under refactoring. The Kent approach uses the *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs, and program analysis and transformation manipulate the AASTs directly; whereas the Eötvös Loránd approach uses relational database, MySQL, to store both syntactic and semantic information of the Erlang program under refactoring, therefore program analysis and transformation are carried out by manipulating the information stored in the database.

One thing that is common between the two refactoring tools is the interface. Both refactoring tools are embedded in the Emacs editing environment, and both make use of the functionalities provided by Distel [8], an Emacs-based user interface toolkit for Erlang, to manage the communication between the refactoring tool and Emacs.

In this section, we first illustrate the interface of the refactoring tools, explain how a refactoring can be invoked, then give an overview of the two implementation approaches. A preliminary comparison of the two approaches follows.

4.1 The Interface

While the catalogue of supported refactorings is slightly different at this stage, the interfaces of the two refactoring tools share the same look and feel. In this paper, we take the refactoring tool from University of Kent as an example to illustrate how the tool can be used.

A snapshot of the Erlang refactorer, which is called *Wrangler*, is shown in Figure 6. To perform a refactoring, the source of interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at *any* of its occurrences; an expression is selected by highlighting it with the cursor. Next, the user chooses the refactoring command from the *Refactor* menu, which is a submenu of the *Erlang* menu, and input the parameter(s) in the mini-buffer if prompted.

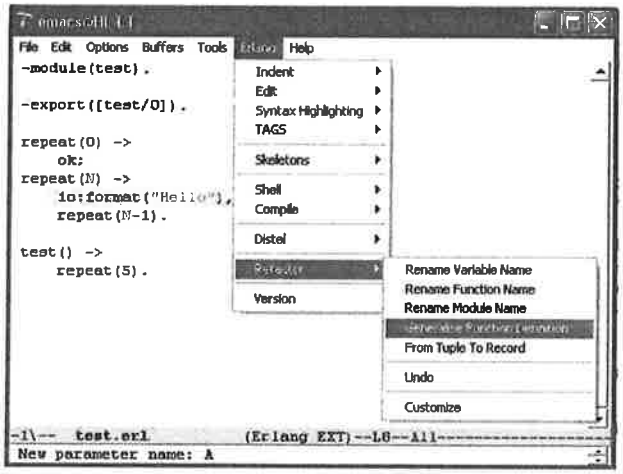


Figure 6. A snapshot of Wrangler

After that, the refactorer will check the selected source is suitable for the refactoring, and the parameters are valid, and the refactoring's side-conditions are satisfied. If all checks are successful, the refactoring will perform the refactoring and update the program with the new result, otherwise it will give an error message and abort the refactoring with the program unchanged.

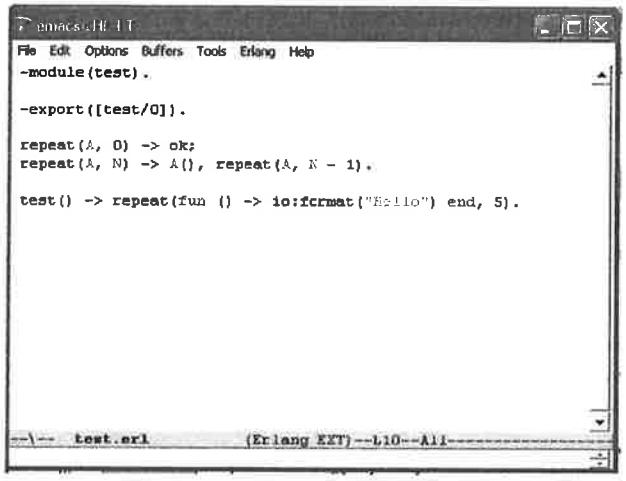


Figure 7. A snapshot of Wrangler showing the result of generalising a definition

Figure 6 shows a particular refactoring scenario. The user has selected the expression `io:format("Hello")` in the definition of `repeat/1`, has chosen the *Generalise Function Definition* command from the *Refactor* menu, and is just entering a new parameter name `A` in the mini-buffer. After this, the user would press the *Enter* key to perform the refactoring. The result of this refactoring is shown in Figure 7: the new parameter `A` has been added to the definition of `repeat/1`, which now becomes `repeat/2`, and the selected expression, wrapped in a fun-expression because of the side-effect problem, is now supplied to the call-site of the generalised function as an actual parameter.

All the implemented refactorings are module-aware. In the case that a refactoring affects more than one module in the program, a message telling which unopened files, if there is any, have been modified by the refactorer will be given after the refactoring has been successfully done. The *customize* command from the *Refactor* menu allows the user to specify the boundary of the program, i.e. the directories that will be searched and analysed by the refactorer.

Undo is supported by the refactorer. Applying *undo* once will revert the program back to the status right before the last refactoring performed.

4.2 The Kent Approach

In this approach, the refactoring engine is built on top of the infrastructure provided by SyntaxTools [1]. It uses the *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs; both program analysis and transformation manipulate the AASTs directly.

4.2.1 The SyntaxTools Package

After having investigated the few available Erlang frontends, we decided to build our refactoring tool on top of the infrastructure provided by SyntaxTools. SyntaxTools is a library from the Erlang/OTP release. This library contains modules for handling Erlang abstract syntax trees (ASTs), in a way that is compatible with the "parse trees" of the standard library module `erl_parse`, together with utilities for reading source files in unusual ways, e.g. bypassing the Erlang preprocessor, and pretty-printing syntax trees. The data types of the abstract syntax is nicely defined so that the nodes in an AST have an uniform structure, and their types are context-independent. We chose to build our refactoring tool on top of SyntaxTools for the following reasons:

- The uniform representation of AST nodes and the type information (more precisely, the syntax category of the syntax phrase represented by the node) stored in each AST node allow us to write generic functions that traverse into subtrees of an AST while

treating most nodes in an uniform way, but those nodes with a specific type in a specific way. This is a great help as both program analysis and transformation involve frequent AST traversals.

- The representation of AST nodes allows users to add their own annotations associated with each AST node. The annotation may be any terms. This facility can be used by the refactorer to attach static semantic information, or any necessary information, to the AST nodes.
- SyntaxTools also contains functionalities for attaching comments to the ASTs representing a program, and a pretty-printer for printing Erlang ASTs attached with comments. This liberates us from the comment-preservation problem, which is also critical for a refactoring tool to be usable in practice. While a pretty-printer could produce a program that has a slightly different appearance with the original one, this does not seem to be a big problem in a community where people tend to accept and use the standard program layout rules.

4.2.2 Adding Static Semantics, Locations, and Type Information

SyntaxTools provides the basic infrastructure for source-to-source Erlang program transformation, even some utility functions for free/bound variable analysis, AST traversals, etc, to ease the analysis of source code structure. However, in order to make program analysis, transformation, as well as the mapping from textual presentation of a syntax phrase to its AST presentation easier, we have annotated the ASTs produced by SyntaxTools with even more information, therefore comes the *Annotated Abstract Syntax Tree* (AAST). What follows summaries the main information we have added, or are trying to add, to the Erlang AST.

- Binding information. The binding information of variables and function names is annotated in the AST in terms of defining and use locations. For example, each occurrence of a variable node in the AST is annotated with its occurrence location in the source, as well as the location where it is defined. Locations are presented by the combination of filename, line number, and column number (the standard Erlang lexer and parser had to be modified in order to get the column number). With the binding information, we can easily check whether two variable/function names refer to the same thing just by looking at their defining locations.
- Range information. Each AST node is annotated with its start and end location in the source code. This makes it easier to map a syntax phrase selected from the textual representation in the editor to its AST representation.
- Category information. The original abstract Erlang syntax does distinguish different kinds of syntax categories, such as functions, attributes, if-expressions, etc. The category information introduced here is mainly to distinguish expressions from patterns.
- Type information. Type information is necessary for some refactorings, and some refactorings require even more refined information than the basic data types defined in Erlang. For example, suppose the same atom, `foo` say, is used as both a module name and a function name in the program. In this case, renaming the module name `foo` may need to know whether an occurrence of the atom `foo` refers the module name being renamed or the function name; therefore, simply knowing the type of `foo` is *atom* is not enough. Adding type information to the AST is currently work-in-progress; we are investigating whether the functionalities provided by TypEr [11], a type annotator for Erlang code, can be used to retrieve the necessary type information.

4.2.3 The Implementation Architecture

Figure 8 summaries the implementation architecture of Wrangler.

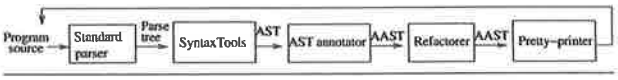


Figure 8. The Implementation Architecture

4.3 The Eötvös Loránd Approach

Instead of annotating the ASTs with information that is necessary for program analysis and transformation, in this approach, we use a relational database, MySQL, to store both abstract Erlang syntax trees and the associated static semantic information, and use SQL to manipulate the stored information. This approach is mainly influenced by the experience from refactoring Clean programs [17, 4].

In the relational database representation, there are two kinds of tables: tables that store the AST, and tables

$$\texttt{gcd}_{30}(\texttt{N}_{15},\ \texttt{M}_{16})\ \texttt{when}\ \texttt{N}_{17} >=_{18} \texttt{M}_{19} \rightarrow \texttt{gcd}_{23}(\ \texttt{N}_{24} -_{15} \texttt{M}_{26},\ \texttt{M}_{28});$$

Figure 9. Source code of the example function clause.

information in the AST	database equivalent	
	table name	record in that table
1st parameter of clause 30 is node 15	clause	30, 0, 1, 15
the name of variable 15 is N	name	15, "N"
2nd parameter of clause 30 is node 16	clause	30, 0, 2, 16
clause 30 has a guard, node 22	clause	30, 1, 1, 22
the left and right operands and the operator of the infix expression 20 are nodes 17, 19 and 18, respectively	infix_expr	20, 17, 18, 19
the body of clause 30 is node 29	clause	30, 2, 1, 29
application 29 applies node 23	application	29, 0, 23
the content of atom 23 is gcd	name	23, "gcd"
1st param. of application 29 is node 27	application	29, 1, 27

Table 1. The representation of the code in Figure 9 in the database.

that store semantic information. The syntax-related tables correspond to the "node types" of the abstract syntax of Erlang as introduced in the Erlang parser. Semantic information, such as scope and visibility of functions and variables, is stored separately in an extensible group of tables. Adding a new feature to the refactoring tool requires the implementation of an additional semantic analysis and the construction of some tables storing the collected semantic information. It is possible to store semantic information of different levels of abstraction in the same database and to support both low-level and high-level transformations.

As an example, consider the code in Figure 9. This is one of the clauses of a function that computes the greatest common divisor of two numbers. Each node of the AST is given a unique id. Every module also has its own id. These ids are written as subscripts in the code.

The database representation of the AST is illustrated in Table 1. The table names *clause*, *name*, *infix_expr* and *application* refer to the corresponding syntactic categories. Without addressing any further technical details, one can observe that each table relates parent nodes of the corresponding type with their child nodes.[1]

In order to make information retrieval faster, an auxiliary table, *node_type* was introduced. This table binds the id of each parent node to the table corresponding to its type. Semantic information about Erlang programs are stored in tables such as *var_visib*,*fun_visib* and *fun_def*. The table *var_visib* stores visibility information on variables, namely which occurrences of a variable name identify the same variable. This table has two columns: *occurrence* and *first_occurrence*. The former is the identifier of a variable occurrence, and the latter is the identifier of the first occurrence of the same variable. The table *fun_visib* stores similar information for function calls, and *fun_def* maintains the arity and the defining clauses of functions.

The *rename variable* and *rename function* transformation is supported with three further table, *forbidden_names*, *scope* and *scope_visib*. The first describes names that are not allowed to use for variables (and for functions). This table contains the reserved words in Erlang, names of the built-in functions, and also user-specified forbidden names. The *scope* table contains the scope of the nodes, what is the most inner scope they are in. The *scope_visib* table stores the hierarchy of the scopes.

As you can observe the resulting data structure is not a tree, but rather a graph, which represents more general connections. We hope it makes easier to implement the refactor steps.

4.3.1 The Implementation Architecture

Figure 10 summaries the implementation architecture of this approach. The refactor step updates the

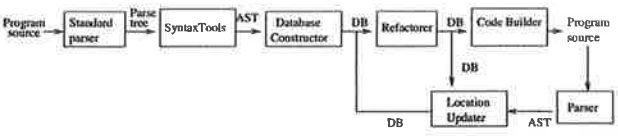


Figure 10. The Implementation Architecture

database (which represents the AST and the semantic information), but the position information might

[1] The price for the separation of tables containing syntactic information from tables containing semantic information is an increased redundancy in the database. For example, the "names" table stores the variable name for each occurrence of the same variable.

no longer reflect the actual positions in the program source. In order to keep the position information up-to-date, we build up the updated syntax tree from the database and use the pretty-printer to refresh the code, then the position information is updated by a simultaneous traversal of the syntax tree represented in the database, and the AST generated by parsing the refreshed code.

4.4 Comparison

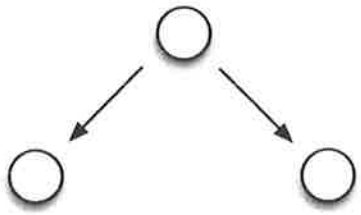
The major difference between the two approaches lies in how the syntactic and semantic information is stored and manipulated. Our first impression is that the second approach needs more time and effort on database designing and the migration of information from abstract Erlang syntax trees to the database; whereas the first approach is relatively light-weight. However, as the second approach tries to avoid reconstruction of the database between two consecutive refactorings by incrementally updating the database so as to keep the stored syntactic and semantic information up-to-date, it may worth the effort. At this stage, it is hard to say which approach is better.

Once both of the two refactoring tools have had support for a number of representative, module-aware refactorings, we would like to test and compare them on some large-scale Erlang programs, and find out the pros and cons of each approach.

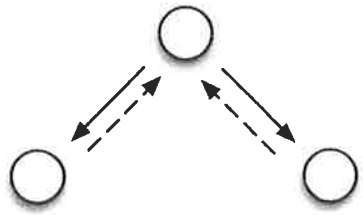
5. Refactorings: The Next Step

The refactorings implemented by both teams thus far are *structural* by nature; we plan also to implement module and data refactorings in line with our work in HaRe. We are also investigating transformations of features characteristic to Erlang, In this section we look at one example, which changes the pattern of communication within a system. We first present a scenario.

A system is constructed in which communication between processes is asynchronous; that is, messages are sent and receipts are not required. It becomes possible to optimise processing within the network by chopping out whole sections; this, however, requires sending a reply back to the sender. As is the case in many software developments, a refactoring can be the first step in modifying the system; in this case, the first step is to make the communication *synchronous*. In pictures, one way communication



is replaced by a two-way, synchronous pattern:



Such a transformation requires a message send to be followed by a receipt, transforming

```
pid!{self(),msg}
```

to

```
pid!{self(),msg},
receive
    {pid, ok}-> ok
```

and in the recipient the code

```
receive {Parent,msg} -> body
```

is replaced by

```
receive {Parent,msg} ->
    Parent!{self(),ok},
    body
```

We envisage implementing other concurrency-related refactorings, and in particular we expect to support transformations of concurrent systems written within the OTP framework; we discuss some other Erlang-specific refactorings now.

Built-in support for concurrency is one of the main features of Erlang. In a well-designed Erlang program, there should be a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world. The following refactoring allows to adjust the process structure in a program.

- *Introduce/remove concurrency* by introducing or removing concurrent processes so as to achieve a better mapping between the parallel processes and the truly parallel activities of the problem being solved. For example, using processes and message passing when a function call can be used instead is a bad programming practice, and this refactoring should help to eliminate the un-desired process and message passing with a function call.

While defensive-style programming is a good programming practice when a sequential programming language is used, non-defensive style programming is the right thing to do when programming with Erlang. Erlang's *worker/supervisor* error handling mechanism allows a clear separation of error recovery code and normal case code. In this mechanism, both *workers* and *supervisors* are processes, where *workers* do the job, and *supervisors* observe the *workers*. If a *worker* crashes, it sends an error signal to its *supervisor*.

- *From defensive-style programming to non-defensive style*. This refactoring helps to transform defensive-style sequential error-handling code written in Erlang into concurrent error handling, typically using supervisor trees.

Erlang programming idioms also expose various refactoring opportunities. Some examples are:

- *Transform a non-tail-recursive function to a tail-recursive function*. In Erlang, all servers must be tail-recursive, otherwise the server will consume memory until the system runs of it.
- *Remove import attributes*. Using import attributes makes it harder to directly see in what module a function is defined. Import attributes can be removed by using remote function call when a call of function defined in another module is needed.
- *From meta to normal function application* by replacing `apply(Module, Fun, Args)` with

  ```
  Module:Fun(Arg1, Arg2,..., ArgN)
  ```

  when the number of elements in the arguments, `Args`, is known at compile-time.
- *Refactoring non-OTP code towards an OTP pattern*. Doing this from pure Erlang code is going to be very challenging, but the whole transformation can be decomposed into a number of elementary refactorings, and each elementary refactoring brings the code a bit closer to the desired OTP pattern.

6. Conclusion

We conclude by surveying related work, and by looking at what we plan to do next.

6.1 Related Work

Programmers used refactoring to make their code more readable, better structured or more apt for further extensions long before the first papers appeared on the topic (e.g. [13]). The field was given much greater prominence with the publication by Fowler's [6], which particularly addressed a wide range of 'manual' refactorings for Java.

Tool support for refactoring is available mostly to object-oriented languages. The first tool was the refactoring browser for Smalltalk [16]. Most tools target Java (IntelliJ Idea, Eclipse, JFactor, Together-J etc.), but there are some for .NET (ReSharper, C# Refactory, Refactor! Pro and JustCode!), C++ (SlickEdit, Ref++ and Xrefactory) and other languages as well. Common refactorings offered by the tools include those that rename program entities (variables, subprograms, modules), those that extract or inline program units, or those that change the static model of classes. A good summary of tools and refactorings can be found at [5], and [12] is an exhaustive survey of the field of software refactoring.

Marcio Lopes Cornèlio formalizes refactorings in an object-oriented language [2]. Some preconditions of refactorings are not simple to compute from the static program text in case of dynamic languages like Smalltalk and Python [16, 15]. The Smalltalk refactoring browser applies dynamic analysis to resolve this problem.

To improve the quality of a code according to a redesign proposal or enforce coding conventions needs support for complex refactoring operations. Planning a sequence of refactoring steps needs refactoring analysis and plan to achieve desirable system structure [14]. Frameworks and libraries change their APIs from time to time. Migrating an application to a new API is tedious work, but typically some eighty percent of the changes will be refactoring steps. Automated detection, record and replay of refactoring steps may support upgrading of components according using the new API [3].

6.2 Future Work

It is a short-term goal for the teams to contrast their approaches on example code bases, to compare the utility of the two approaches. For instance, the ADT approach has the advantage of being more lightweight, but the database representation can offer versioning of code and the concurrent handling of refactoring steps in some cases.

In the medium term, each team will build support for further refactorings, particularly those supporting practising Erlang programmers. In particular we will build refactorings to support the transformation of data representations, changes to patterns of concurrent communication and integration with the OTP framework.

In the longer term we look forward to machine-supported refactoring becoming a valuable part of the Erlang programmers' toolkit.

References

[1] Carlsson, R. . Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3/doc/html/`.

[2] Cornèlio, M.L.: Refactorings as Formal Refinements, PhD thesis, Universidade Federal de Pernambuco, 2004.

[3] Dig, D.: Toward Automatic Upgrading of Component-Based Applications, ECOOP 2006 Doctoral Symposium and PhD Students Workshop, Nantes, Fance, 2006. `http://www.ecoop.org/phdoos/ecoop2006ds/`.

[4] Diviánszky, P. and Szabó-Nacsa, R. and Horváth, Z. Refactoring via Database Representation. In L. Csőke, P. Olajos, P. Szigetváry, and T. Tómács, editors, *The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary*, volume 1, pages 129–135, 2004.

[5] Fowler, M.: Refactoring Home Page, `http://www.refactoring.com/`.

[6] Fowler, M. *et. al.*, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] Refactoring Functional Programs, `http://www.cs.kent.ac.uk/projects/refactor-fp/`.

[8] Gorrie, L.. Distel: Distributed Emacs Lisp (for Erlang).

[9] Li, H. and Reinke, C. and Thompson, S., Tool Support for Refactoring Functional Programs in, ACM SIGPLAN Haskell Workshop 2003, Uppsala, Sweden, Johan Jeuring (ed.), 2003.

[10] Li, H. and Reinke, C. and Thompson, S., The Haskell Refactorer, HaRe, and its API., Electr. Notes Theor. Comput. Sci., 141 (4), 2005.

[11] Lindahl, T. and Sagonas, K. F.. TypEr: a Type Annotator of Erlang Code. In *ACM SIGPLAN Erlang Workshop 2005*, 2005.

[12] Mens, T. and Tourwé, T., A Survey of Software Refactoring, IEEE Trans. Software Eng., 30 (2), 2004.

[13] Opdyke, W.: Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[14] Perez, J. Overview of the Refactoring Discovering Problem, ECOOP 2006 Doctoral Symposium and PhD Students Workshop, Nantes, Fance, 2006. `http://www.ecoop.org/phdoos/ecoop2006ds/`.

[15] Adventures in Refactoring Python. `http://blogs.warwick.ac.uk/refactoring/`, Sep. 24, 2006.

[16] Roberts, D., Brant, J. and Johnson, R. A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems (TAPOS), special issue on software reengineering, 3(4):253–263, 1997.

[17] Szabó-Nacsa, R. and Diviánszky, P. and Horváth, Z. Prototype Environment for Refactoring Clean Programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4*, 2004.

Comparing C++ and Erlang for Motorola Telecoms Software

Phil Trinder & Henry Nyström
Computer Science Department
Heriot-Watt University, UK

Erlang Training & Consulting

David King
Software & Systems Engineering Research
Motorola Labs, UK

High-Level Techniques for Distributed Telecoms Software

- **EPSRC (UK Govt) Project, Dec 2002 – Feb 2006**
- **Collaboration between**
  - Motorola UK Labs
  - Heriot-Watt University

High-Level Techniques for Distributed Telecoms Software

- **EPSRC (UK Govt) Project, Dec 2002 – Feb 2006**
- **Collaboration between**
  - Motorola UK Labs
  - Heriot-Watt University
- **Aim: produce scientific evidence that high-level distributed languages like Erlang or Glasgow distributed Haskell (GdH) can improve distributed software robustness and productivity**
- **Publication: High-Level Distribution for the Rapid Production of Robust Telecoms Software: comparing C++ and Erlang, Concurrency and Computations: Practice & Experience (forthcoming).**

Erlang Comparisons

A number of sequential comparisons, e.g. Computer Language Shootout

Very few distributed system comparisons published!

Ulf Wiger [Wiger01] reports

- **Erlang systems have between 4 and 10 times less code than C/C++/Java/PLEX systems**
- **Similar error rates/line of code**
- **Similar productivity rates**

No direct comparative measurements

Jantsch *et al* compare 6 languages for hardware description [JKS+01]

Research Questions: Potential Benefits

RQ1: Can robust, configurable systems be readily developed?

- Resilience to extreme loads
- Availability in the face of hardware & software failures
- Dynamic reconfigurability on available hardware

RQ2: Can productivity and maintainability be improved?

- How do the sizes of the C++ and Erlang components compare & what language features contribute to size differential?

Research Questions: Feasibility

High-level distributed languages:

- **abrogate control of low-level coordination aspects, so**
  - RQ3 can the required functionality be specified?
- **typically pay space and time penalties for their automatic coordination management.**
  - RQ4 can acceptable performance be achieved?

RQ5 What are the costs of interoperating with conventional technology?

RQ6 Is the technology practical?

Research Strategy

- **Reengineer some telecoms application components in GdH and Erlang**
  - Dispatch Call Controller [NTK04,NTK05]
  - Data Mobility component
- **Compare high-level and Java/C++ implementations for**
  - Performance
  - Robustness
  - Productivity
  - Impact of programming language constructs

1st Component Engineered: Data Mobility Component (DM)

- **Product Component**
- **Communicates with Motorola mobile devices**
- **3000 lines of C++**
- **Uses 18,000 lines of Motorola C library functions**
- **Has a transmitter and a receiver, and 2 message types**
- **Interacts with 5 other components of the system**

2nd Component Engineered: Despatch Call Controller (DCC)

- Handles mobile phone calls
- A process manages each call
- Scalable with multiple servers

Two Erlang DM Implementations

1. Pure Erlang DM
2. Erlang/C DM
   reuses some C DM libraries & drivers

Both interoperate with a C test harness

Combine
- Unix Processes
- Erlang processes
- C Threads (Erlang/C DM)

RQ3 Performance 1: Throughput

- Platform: 167MHz, 128Mb Sun Ultra 1, SunOS 5.8

C++ DM	Erlang/C DM	Pure Erlang DM
480	230	940

Maximum DM Throughput at 100% QoS

RQ3 Performance 1: Throughput

- Platform: 167MHz, 128Mb Sun Ultra 1, SunOS 5.8

C++ DM	Erlang/C DM	Pure Erlang DM
480	230	940

Maximum DM Throughput at 100% QoS

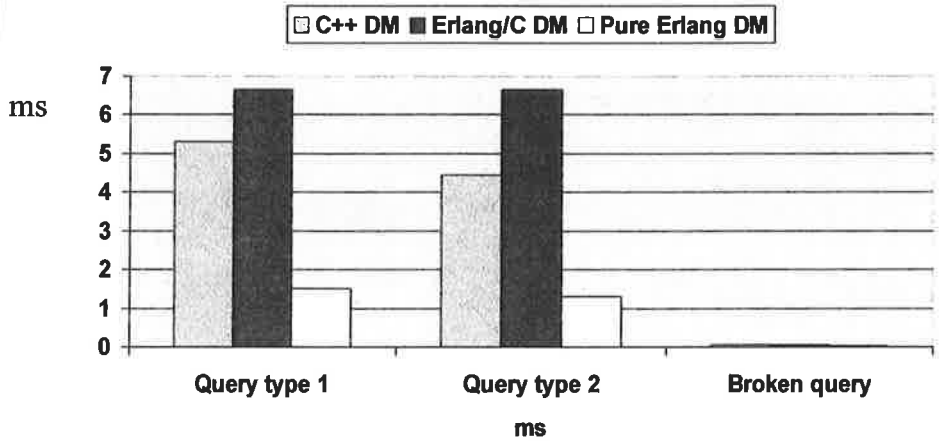
Pure Erlang DM is twice as fast as C++ DM (better process management and marshalling)

Erlang/C DM is ½ speed of C++ DM, but still meets nominal throughput

Performance 2: Round Trip Times

Pure Erlang is approximately 3 times faster

Erlang/C is 26% - 50% slower



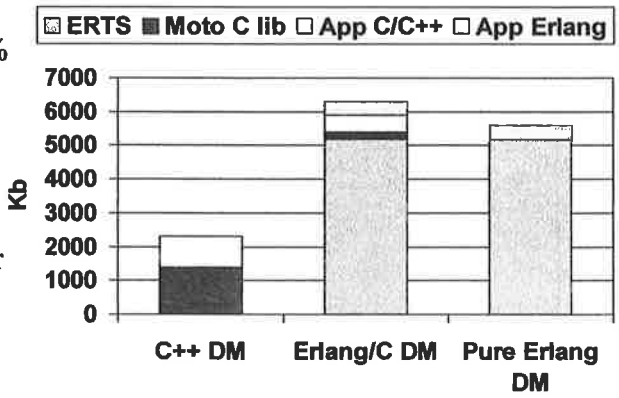
Performance Analysis

Pure Erlang DM is faster due to fast lightweight process management

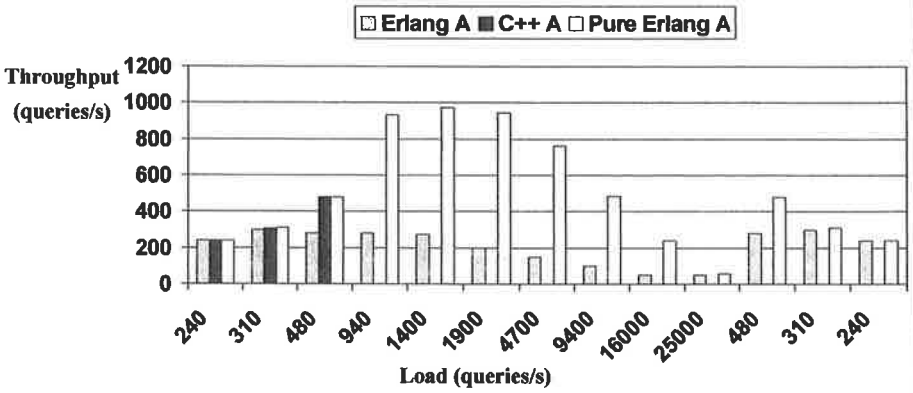
Erlang/C is slower due to additional communication to C components

Performance 3: Memory Residence

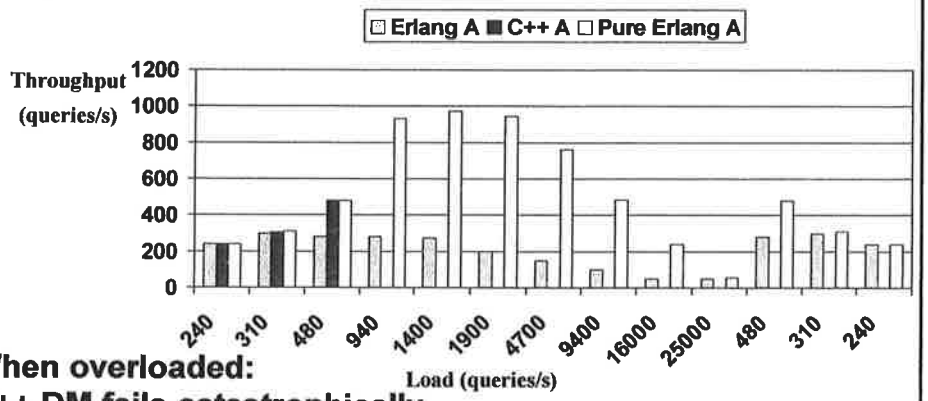
- Erlang DMs use 170% more memory
- Erlang runtime sys (ERTS) has a fixed size
- => would be a smaller % of a larger app.



RQ1 Robustness 1: Resilience



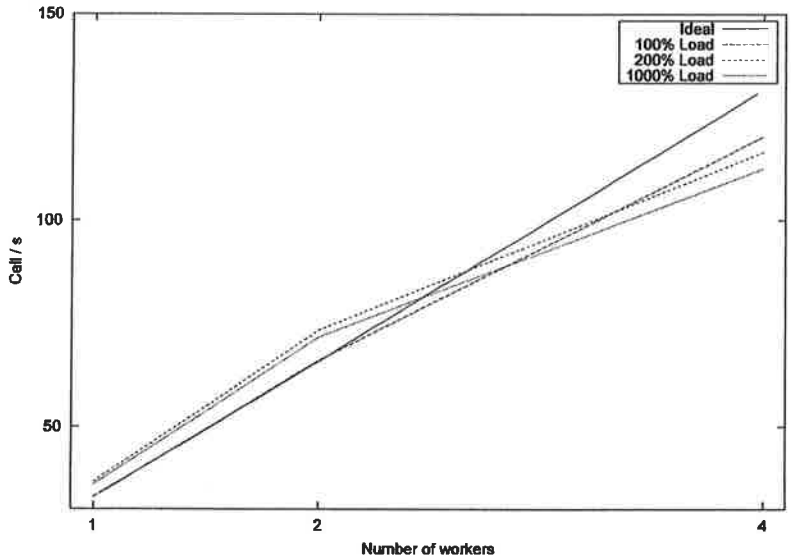
RQ1 Robustness 1: Resilience



When overloaded:
C++ DM fails catastrophically
Pure Erlang & Erlang/C DMs:
- **Throughput degrades**
- **Never completely fails, handling 48 q/s at peak load (25000q/s)**

- Recovers automatically after load drops

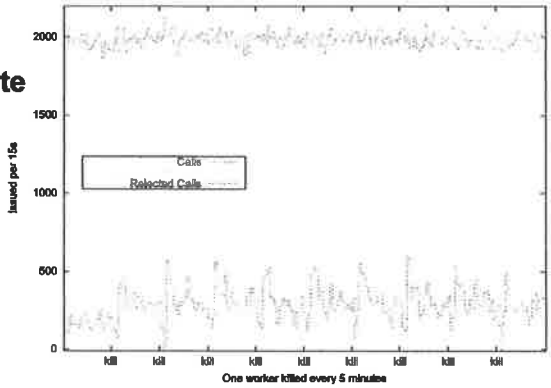
DCC Resilience



Robustness 2: Availability

Erlang systems

- **remain available despite repeated hardware & software failures**
- **performance doesn't degrade with repeated failures**

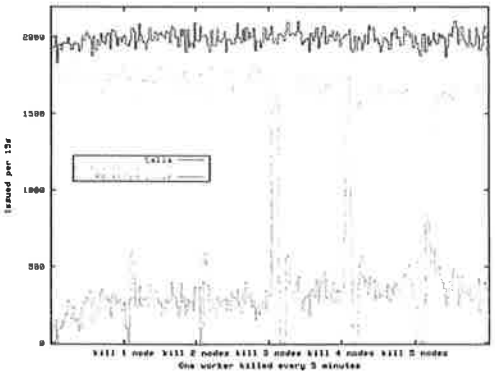


DCC Throughput with Repeated Failures

Robustness 2: Availability

Erlang Systems resists the simultaneous failure of multiple components

When more components fail throughput drops lower & recovery takes longer



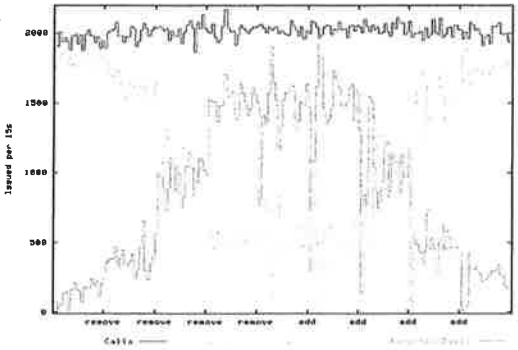
5-processor DCC with Multiple Failures

Robustness 3: Dynamic Configurability

Erlang Systems dynamically adapt to the available hardware resources.

5 processor system:

- remove a processor 4 times
- add a processor 4 times



DCC Throughput
with Varying Numbers of Processors

RQ2: Productivity & Maintainability

Shorter programs are

- **Faster to develop**
- **Contain fewer errors [Wiger01]**
- **Easier to maintain**

The metric we use is Source Lines Of Code (SLOC)

Productivity: Source Code Sizes

Lang.	C/C++	Erlang	Total
C++	3101		3101
Erl./C	247	616	863
Erlang		398	398

DM Implementations

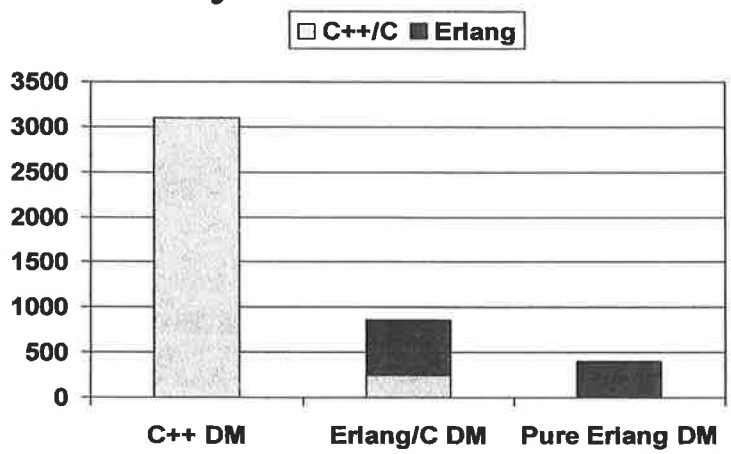
Lang.	C++	IDL	Erl.	Total
C++	14.8K	83		14.9K
Erl.			4882	4882

DCC Implementations

Erlang DCC and DM are less than 1/3rd of size of C++ impl.

Consistent with Wiger & folklore

Productivity: DM Source Code Sizes

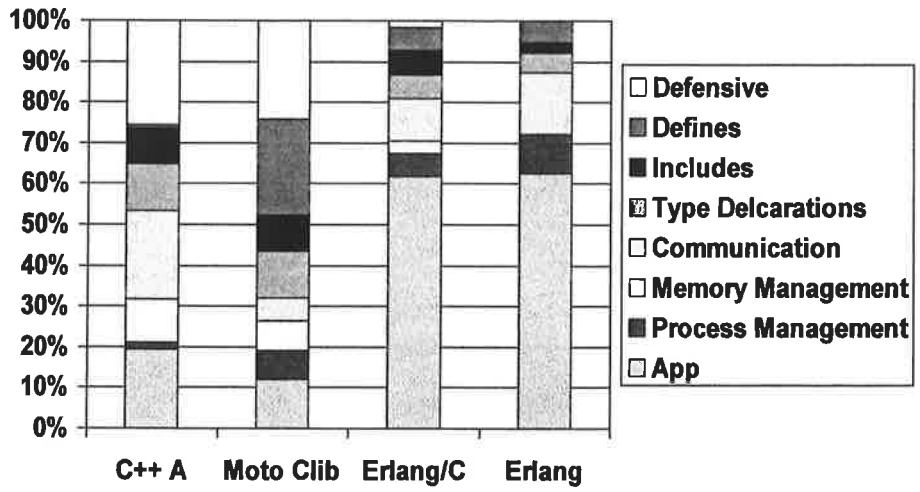


- Erlang/C DM is 1/3rd of the size of the C++ DM
- Pure Erlang DM is 1/7th of the size of the C++ DM
- Erlang/C DM is 1/18th of the size of the C++ DM + libraries

Reasons for difference in Code Size

- **Erlang programmers can**
  - rely on fault tolerance and code for the successful case (27% of C++ DM code is defensive)
  - and have
    - **automatic memory management (11% of C++ DM code)**
    - **high-level communication (23% of C++ DM code)**
    - **Telecom design pattern libraries**

DM Code Breakdown



Code Difference Example

C++

```
void DataMobilityRxProcessor::processUnsupVer(void)
{
  MSG_PTR          msg_buf_ptr;
  MM_DEVICE_INFO_MSG  *msg_ptr;
  RETURN_STATUS      ret_status;
  UINT16           msg_size;

  // Determine size of ici message
  msg_size = sizeof( MM_DEVICE_INFO_MSG);

  // Create ICI message object to send to DMTX so it sends a Device Info
  //   message to VLR and HLR clients
  IciMsg ici_msg_object( MM_DEVICE_INFO_OPC, ICI_DMTX_TASK_ID, msg_size);

  // Retrieve ICI message buffer pointer
  msg_buf_ptr = ici_msg_object.getIciMsgBufPtr();

  // Typecast pointer from (void *) to (MM_DEVICE_INFO_MSG *)
  msg_ptr = (MM_DEVICE_INFO_MSG *)msg_buf_ptr;

  // Populate message buffer
  SET_MM_DEVICE_INFO_DEVICE_TYPE( msg_ptr, SERVER);
  SET_MM_DEVICE_INFO_NUM_VER_SUPPORTED( msg_ptr, NUM_VER_SUPPORTED);
  SET_MM_DEVICE_INFO_FIRST_SUP_PROTO_VERS( msg_ptr, PROTO_VERSION_ONE);
// Send message to the DMTX task
  ret_status = m_ici_io_ptr->send(&ici_msg_object);

  // Check that message was sent successfully
  if (ret_status != SUCCESS)
  {
    // Report problem when sending ICI message
    sz_err_msg( MAJOR, SZ_ERR_MSG_ERR_OPCODE, __FILE__, __LINE__,
            "DataMobilityRxProcessor processUnsupVer: failure sending "
            " device info message to DMTX"),
  }
}
```

Erlang

sz_dme_dmtx:cast(device_info)

Erlang DCC Reusability

Part	SLOC	No. Modules	Percentage
Reusable Platform	2994	26	61%
Specific Services	147	1	3%
Testing/Stat.s	1741	11	36%

Considerable potential for reuse

Summary

- **Investigated high-level distributed language technology for telecoms software**
- **Reengineered two telecoms components in Erlang**
- **Measured & compared the Erlang & C++ components**

RQ1: Robust & Configurable Systems

- **Improved resilience:**
  - Erlang DM and DCC sustain throughput at extreme loads
  - Automatically recover when load drops
  - C++ DM fails catastrophically (predict C++/CORBA DCC would)
- **Improved availability:**
  - Erlang DCC recovers from repeated & multiple failures
  - Predict C++/CORBA DCC would fail catastrophically
- **Dynamic Reconfiguration**
  - Erlang DCC can be dynamically reconfigured to available hardware
  - C++/CORBA DCC can also be dynamically reconfigured using CORBA
- **Potential for hot-code loading (not demonstrated)**

RQ2: Productivity & Maintainability

- **Erlang DM and DCC:**
  - Less than 1/3rd size of C++ implementation
  - Erlang DM 1/18th size of C++ DM with libraries
  - Good reusability
- **Reasons:**
  - Code for successful case – saves 27%
  - Automatic memory management – saves 11%
  - High-level communications – saves 23%
  - Telecom design pattern libraries

RQ3: Distributed Functionality

- **Even though Erlang abstracts over low-level coordination, the required DM and DCC functionality is readily specified.**

RQ4: Performance

- **Time:**
  - Max. throughput at 100% QoS:
    - **Pure Erlang DM is twice as fast as C++ DM**
    - **Erlang/C is ½ as fast as C++ DM , but still exceeds throughput requirements**
  - Roundtrip times
    - **Pure Erlang DM is three times as fast as C++ DM**
    - **Erlang/C is between 26% and 50% slower as C++ DM**
- **Space:**
  - Pure Erlang and Erlang/C both have 170% greater memory residency due to (fixed size) 5Mb runtime system

RQ5: Interoperation Costs

- **Erlang DMs interoperate with a C test harness, and Erlang/C DM incorporates C drivers & library functions.**
- **Costs**
  - Low space cost: an additional 15% residency
  - High time cost:
    - **Erlang/C roundtrip times up to 6 times pure Erlang**
    - **Erlang/C max. throughput ¼ of pure Erlang**
- **Potential for incremental re-engineering of large systems**

RQ6: Pragmatics

- **Erlang is available on several HW/OS platforms, including the Sun/Solaris DM product platform**
- **Well supported with training, consultancy, libraries etc.**

Conclusions

- **Erlang offers robustness & productivity benefits for distributed telecoms software (RQs 1 & 2)**
- **High-level distributed languages like Erlang can deliver the required telecoms functionality and performance (RQs 3 & 4)**
- **Erlang can interoperate with existing technologies and meets pragmatic requirements (RQs 5 and 6)**

Further Information

Web sites/Seminars

Erlang Site: www.erlang.org/
Project Site: www.macs.hw.ac.uk/~dsg/telecoms/

References

[JKS+01] Jantsch A. Kumar S. Sander I. Svantesson B. Oberg J. Hemanic A. Ellervee P. O'Nils M. Comparison of Six Languages for System Level Descriptions of Telecoms Systems, Electronic Chips and System Design, pp 181-192, Kluwer, 2001.

[NTK04] Nystrom J.H. Trinder P.W. King D.J. Evaluating Erlang for Robust Telecoms Software S3S'04, Chicago, July 2004.

[NTK05] Nystrom J.H. Trinder P.W. King D.J. Are High-Level languages suitable for Robust Telecoms Software? SafeComp'05, Fredrikstad, Norway, Sept. 2005.

[Wiger01] *Ulf Wiger,* Workshop on Formal Design of Safety Critical Embedded Systems, Munich, March 2001 http://www.erlang.se/publications/Ulf_Wiger.pdf

Using GNU Autoconf to Configure Erlang Programs

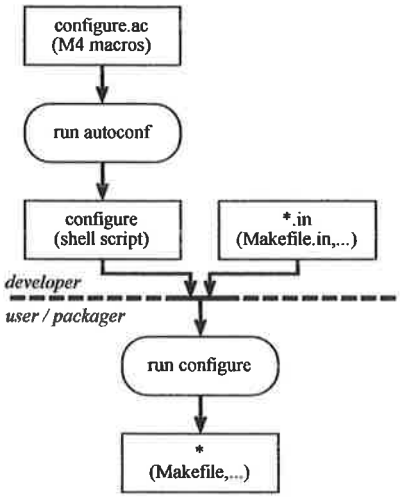
Romain Lenglet
Chiba Shigeru Group
Tokyo Institute of Technology

EUC 2006, Stockholm
2006-11-09

Purpose of this talk

- I will show you
  - How to use existing Autoconf macros to configure Erlang programs
  - How to extend Autoconf by defining new test macros

Overview of Autoconf



- Autoconf deals only with configuration detection
- Generates portable configure Bourne shell scripts that
  - Check the configuration
  - Rewrite files to substitute variables
  - Generate C headers defining constants

Autoconf macros

- Autoconf is essentially a set of M4 macros
  - Autoconf macros = M4 macros
  - configure.ac = shell script with calls to M4 macros
  - Macros are rewritten by Autoconf/M4 to produce pure shell script code that tests configuration
- Autoconf also wraps around M4
  - Caches macro files, etc.

A minimum configure.ac

AC_INIT: meta informations about the project

```
AC_INIT(GTK+ 2 wrapper library for Erlang, 0.20,
            mats.cronqvist@..., gtknode)

AC_PREREQ(2.59c)

AC_COPYRIGHT(Copyright (C) 2005 Mats Cronqvist)

...

AC_CONFIG_FILES([ \
        Makefile \
        src/Makefile \
        ...
        ])

AC_OUTPUT
```

AC_PREREQ: minimum required version of Autoconf (2.59c was the first to contain macros for Erlang)

The actual tests to perform, as calls to test macros

AC_CONFIG_FILES: list of files to rewrite

AC_OUTPUT: mandatory at the end, makes configure rewrite the files

What do tests do?

- What to test?
  - Autoconf philosophy
    - Test for the features that you actually need
    - Do not test version numbers: this is not maintainable!
  - Testable: programs, C features (headers, libs, functions, constants...), etc.
- When executed, a test may
  - Fail (display error message, exit with code > 0)
  - Define substitutions of variables
    - To be substituted in rewritten files (Makefile.in, ...)
  - Define C constants
    - Defined in generated confdefs.h file (cf. Autoheader)

AC_ERLANG_NEED_ERLC

- In configure.ac:

```
...
AC_ERLANG_NEED_ERLC
...
```

- In Makefile.in:

```
...
SUFFIXES = .erl .beam
.erl.beam:
        @ERLC@ -b beam $<
...
```

Variables to substitute must be enclosed in @...@

- This macro
  - Finds the path to erlc
  - Fails if it is not found
  - Substitutes the ERLC variable

AC_ERLANG_CHECK_LIB(*lib*)

- In configure.ac:

```
...
AC_ERLANG_CHECK_LIB(ic)
...
```

Macros can take arguments

- In Makefile.in:

```
...
CFLAGS = -
   I@ERLANG_LIB_DIR_ic@/includ
   e
...
```

- This macro
  - Finds the path to an Erlang library
  - Fails if it is not found
  - Substitutes the ERLANG_LIB_DIR_*lib* variable

AC_ERLANG_SUBST_INSTALL _LIB_SUBDIR(*app*, *version*)

- In configure.ac:

```
...
AC_ERLANG_SUBST_INSTALL_LIB_
SUBDIR(hi, 0.20)
...
```

- In Makefile.in:

```
...
install:
        cp foo.beam
  @ERLANG_INSTALL_LIB_DIR_hi@
  /ebin/
        cp foo.erl
  @ERLANG_INSTALL_LIB_DIR_hi@
  /src/
...
```

- This macro
  - Substitutes the variable for the path to install an Erlang application

Currently available Erlang-related macros (1/2)

- Checks for programs
  - AC_ERLANG_PATH_ERLC(...)
  - AC_ERLANG_NEED_ERLC(...)
  - AC_ERLANG_PATH_ERL(...)
  - AC_ERLANG_NEED_ERL(...)
- Substitutions for installed dirs
  - AC_ERLANG_SUBST_ROOT_DIR
  - AC_ERLANG_SUBST_LIB_DIR
- Checks for installed Erlang libraries
  - AC_ERLANG_CHECK_LIB(...)

Currently available Erlang-related macros (2/2)

- Substitutions for installation dirs
  - AC_ERLANG_SUBST_INSTALL_LIB_DIR
  - AC_ERLANG_SUBST_INSTALL_LIB_SUBDIR(...)
- Support of Erlang as a language to write tests in configure scripts

How to write new test macros?

- Autoconf philosophy
  - To test features, it is better to use directly the programming language of the feature
- Autoconf defines a good framework for supporting multiple languages
  - Used only by writers of new support macros
- Tests can be written in
  - Bourne shell (of course!)
  - C / C++
  - Fortran
  - Erlang
  - ...

How to add support for a new language in Autoconf?

- Autoconf defines conventions for language support macros, that define how to
  - Call the pre-processor (if there is one for the language) on the tests code
  - Compile, and execute tests
  - Pass data between test programs and the configure script (using temporary files)
  - ...
- I wrote those macros for Erlang
  - AC_LANG(Erlang)
  - AC_LANG_PROGRAM(Erlang)
  - AC_LANG_COMPILER(Erlang)
  - ...

Test if a function is exported (1/7)

- Erlang code

```
{[$1], Beam, _Filename} =
code:get_object_code([$1]),

{ok, {[$1], [[{exports,
  Exports}]]}} =
beam_lib:chunks(Beam,
                [[exports]]),

IsExported = lists:member(
{[$2], [$3]}, Exports)
```

The arguments of an M4 macro are called $1, $2, $3, etc.

- Load the code file for Module ($1)
- Get the list of exported functions
- Iterate and test if the Function ($2) (& Arity ($3)) is in there

Test if a function is exported (2/7)

- Erlang code

```
{Module, Beam, _Filename} =
code:get_object_code(Module),
{ok, {Module, [[{exports,
  Exports}]]}} =
beam_lib:chunks(Beam,
                [[exports]]),
IsExported = lists:member(
{Function, Arity}, Exports)
S = if IsExported -> "yes";
       true -> "no" end,
file:write_file(
    "conftest.out", S),
halt(0)
```

- Write the result ("yes" or "no") into the conftest.out temporary file
- Halt the Erlang VM

Using a "conftest.out" temporary file for exchanging data is a convention used in all Autoconf tests

Test if a function is exported (3/7)

- Autoconf code

```
AC_LANG_PROGRAM([], [
{Module, Beam, _Filename} =
code:get_object_code(Module),
{ok, {Module, [[{exports,
  Exports}]]}} =
beam_lib:chunks(Beam,
                [[exports]]),
IsExported = lists:member(
{Function, Arity}, Exports)
S = if IsExported -> "yes";
       true -> "no" end,
file:write_file(
    "conftest.out", S),
halt(0)
])
```

- Use Autoconf/Erlang macro to generate the Erlang module

The test code is the body of the 'start/0' function in the 'conftest' module

Test if a function is exported (4/7)

- Autoconf code

```
AC_LANG_PUSH(Erlang)
AC_RUN_IFELSE(

  [AC_LA
  ,
  [...],
  [...])
AC_LANG_POP(Erlang)
```

Code to execute if the test executed normally; resp. failed

- Compile and execute the Erlang test module

Test if a function is exported (5/7)

- Autoconf code

```
AC_CACHE_CHECK([if ...],
[erlang_cv_foo_$1_$2_$3],
[
AC_LANG_PUSH(Erlang)
AC_RUN_IFELSE(

  [AC_LANG_PROGRAM([],[...])]
  ,
  [erlang_cv_foo_$1_$2_$3
   =`cat conftest.out`],
  [AC_MSG_FAILURE([...])])
AC_LANG_POP(Erlang)
])
```

- Set a shell variable with the result
- Support caching of the test result
  - The test is executed only if the variable is not yet defined
- If test failed, print a test message, and exit the configure script

Test if a function is exported (6/7)

- Complete macro

```
AC_DEFUN([FOO],
[
AC_REQUIRE(
    [AC_ERLANG_PATH_ERLC])
AC_REQUIRE(
    [AC_ERLANG_PATH_ERL])
AC_CACHE_CHECK(...)
AS_IF([test
  "$erlang_cv_foo_$1_$2_$3"
  = "no"], [$5], [$4])
])
```

- Define the macro
- Make sure that erl and erlc are detected
- Execute user code if the function is exported; resp. if it is not

Test if a function is exported (7/7)

- Calls in configure.ac

```
...
FOO(file, write_file, 2,
   [AC_MSG_NOTICE([OK])],
   [AC_MSG_FAILURE([NOT OK])])

FOO(file, write_file, 0,
   [AC_MSG_NOTICE([OK])],
   [AC_MSG_FAILURE([NOT OK])])
...
```

- Output of ./configure

```
checking for erlc...
   /usr/bin/erlc
checking for erl...
   /usr/bin/erl
checking if file:write_file/2
   is exported... yes
configure: OK
checking if file:write_file/0
   is exported... no
configure: error: NOT OK
See `config.log' for more
   details.
```

11)

Conclusion

- Currently: minimal support for Erlang
  - Checks for erl, erlc, installed dirs
  - Substitutions for installing new applications
  - Support for Erlang as a test language
  - Available since January 2006 (version 2.59c)
- Relatively easy to extend
  - By writing new test macros with Erlang code
  - If you have macros, please send them to me!
    - I can submit to GNU Autoconf for inclusion
    - We may start a sub-category in the Autoconf Archive
  - The quickly-written "FOO" macro will be submitted soon (with a better name, of course!)

Please read my blog / Planet Erlang for news

`http://www.csg.is.titech.ac.jp/~lenglet/`
`http://www.planeterlang.org/`

9

Configuration Aware Distributed System Design in Erlang

Gabor Batori, Zoltan Theisz, Domonkos Asztalos
Software Engineering Group, Ericsson Hungary Ltd.
H1037 Laborc u. 1. Budapest, Hungary
{Gabor.Batori, Zoltan.Theisz, Domonkos.Asztalos}@ericsson.com

Abstract

In this paper a new system design concept is described and demonstrated which is based on the innovative combination of meta-model assisted explicit component configuration management and its run-time execution on a causally reflective robust reconfigurable Erlang component system called ErlCOM. Rather than provide a monolithic run-time application structure separate areas of functionalities are packaged into self-contained components that can be individually deployed according to the available hardware/software resources where they can be actively managed during the whole lifetime of the application. The infrastructure extends the approach of ordinary code reuse into higher level where in addition of the shared code base the run-time configuration can be effectively reapplied.

1. Introduction

Future networked distributed systems will have to be able to cope with increased complexity originating from the ever increasing demand of newer communication protocols that should be able to operate in a highly distributed telecom environment and still be able to remain compatible with the already established infrastructure or from novel application fields of telecommunications like wireless sensor networks. The aim of our RUNES [1] (Reconfigurable Ubiquitous Network Embedded Systems) project is to provide the application developer with a proper system design model and a corresponding heterogeneous middleware/platform that enable better application production for networked embedded sensors, actuators and for more powerful devices like embedded gateways or full-fledged application servers regarding time, efforts, maintainability and quality. Our efforts have resulted in the development of ErlCOM [2], which is a causally reflective reconfigurable Erlang component system running anywhere Erlang is available, that is, on gateways and application servers in RUNES, and in the invention and demonstration of a meta-model assisted component configuration management. The component configuration management automatically generates component wrapping code for any functionality written in Erlang and it deploys it later onto ErlCOM. The component configuration is managed either locally by the deployed code or remotely via the meta-modeling environment. Since both the source code and the current component configuration of the running application are available in the meta-modeling environment total application reuse (in contrast with only source code reuse) is easily attainable.

In the remainder of the paper, Section 2 overviews ErlCOM then Section 3 introduces the meta-model assisted component management. In Section 4 the implementation details of the enabling technologies, namely the Deployment Tool and the notification pattern, are explained and finally Section 5 concludes the paper.

2. ErlCOM

ErlCOM provides a super-structure on top of the well-established Erlang/OTP environment. The basic entities of ErlCOM are the components, which can be dynamically created, loaded, updated, unloaded and destroyed, and the bindings, which bind or unbind the receptacles - component egresses - to the interfaces - component ingresses - of the communicating components. Components can be embedded into each other hierarchically and both the components and the bindings are managed by a hierarchical caplet structure where the root caplet - also called capsule - represents the Erlang node. Any ErlCOM entity can possess an unlimited amount of metadata that are stored in a fully distributed repository covering all the caplets. Component configurations can be constrained by building a component framework, however, the constraint enforcement policy is left for the programmer. Since also the component configuration is stored in the distributed repository the component system can be easily reconfigured and the reconfiguration changes are easily tracked. The components communicate to each other via message passing – both synchronous and asynchronous – that can be intercepted at the bindings.

ErlCOM's implementation on top of Erlang is relatively light-weight as components, bindings and caplets are ordinary gen_servers with supervisors, and component communication relies on Erlang message sending. The distributed repository is based on Mnesia, which makes it a little bit heavy-weight, however, it provides a fully distributed robust database solution. ErlCOM's API is described in details in [2].

Since ErlCOM extends the RUNES component meta-model and the middleware CRTK it has been successfully deployed on Lippert [4] gateways and application server PCs in order to deploy distributed applications for wireless sensor networks.

3. Meta-model assisted component configuration management

ErlCOM's basic goal is to provide a useful packaging framework that enables programmers to organize their applications written in Erlang in such a way that they could be easily reconfigured either so that they could adapt in a rapidly changing run-time environment (dynamicity in run-time) or they could be reused - already tested source code and run-time configuration - to satisfy newer change requests for redesign (dynamicity in design-time). However, extra packaging burden might intimidate the programmer to use ErlCOM that's why we have developed a meta-model assisted IDE [2] based on GME [3] that incorporates the ErlCOM meta-model and automatically generates packaging code. More precisely, the IDE contains three models, which are the ErlCOM component meta-model (Figure 1), the componentized application source code inside interconnected RUNES components - according to the logical decomposition of

the application - (Figure 2) and the deployed component configuration of the application (Figure 3).

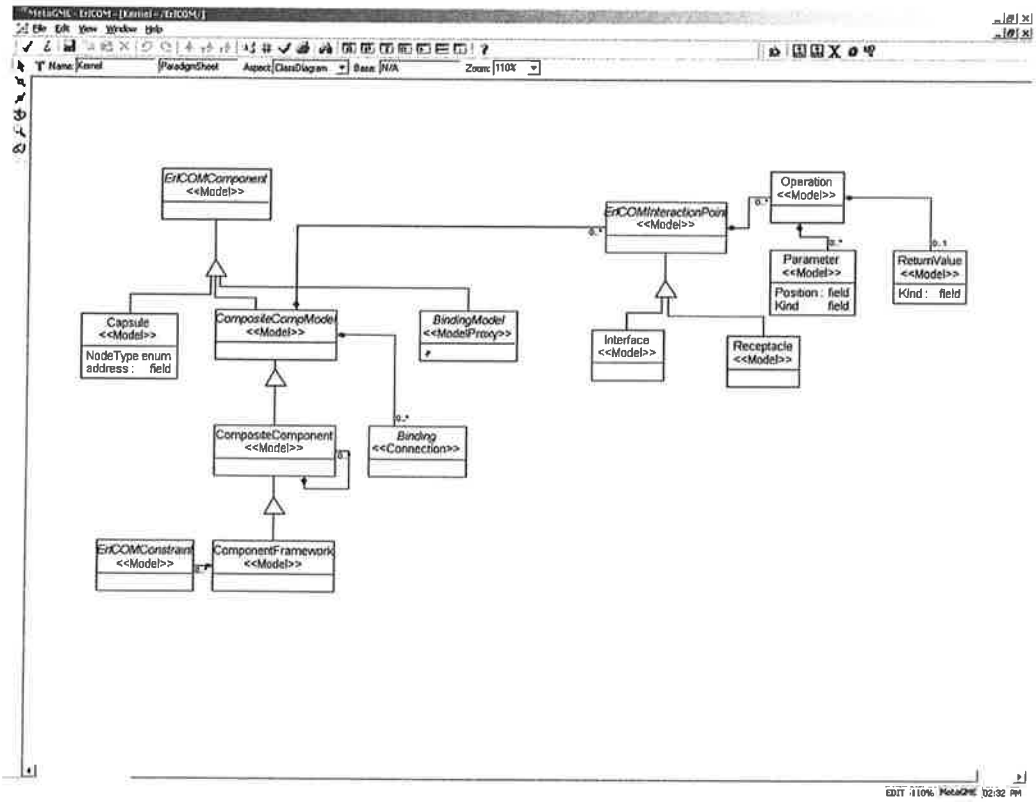


Figure 1: ErlCOM component meta-model

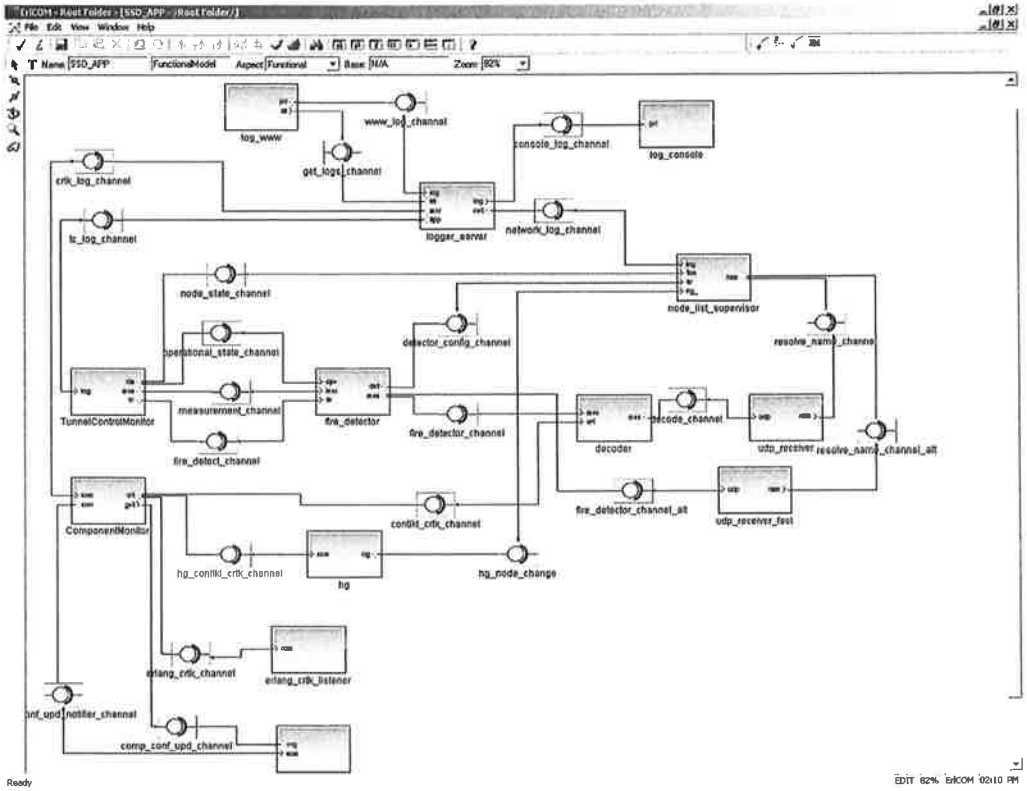


Figure 2: Componentized application source code inside RUNES components

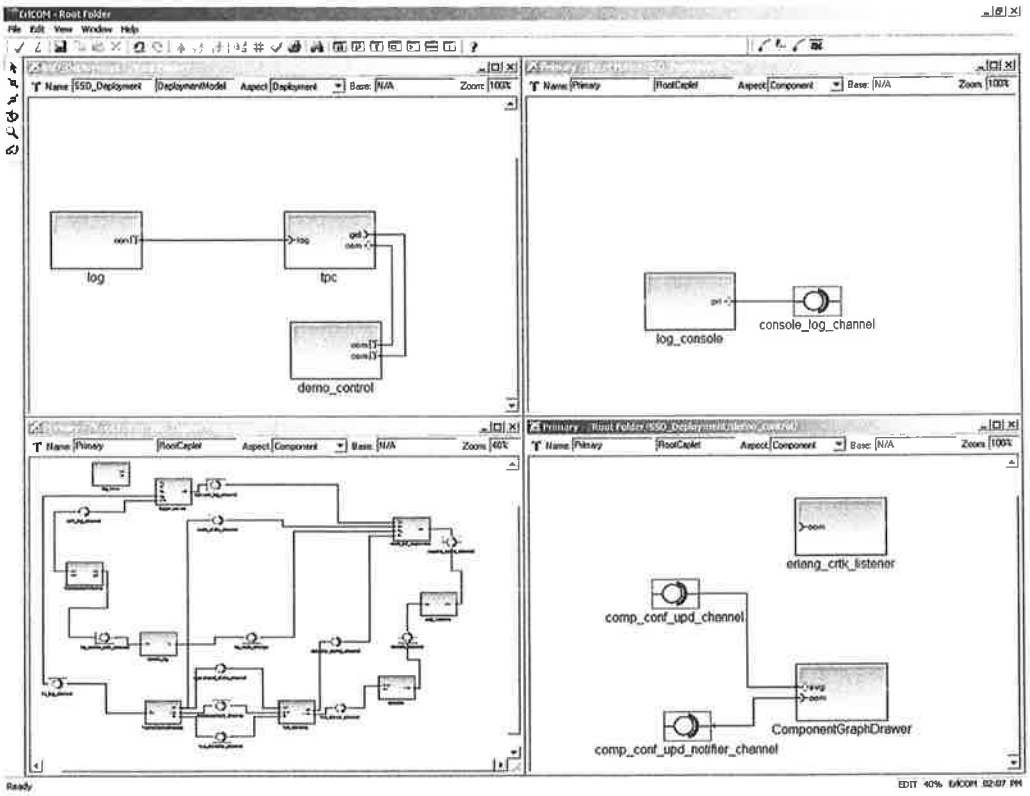


Figure 3: Deployed component configuration

The models are in meta relationship to each other, that is, the deployed component configuration instantiates the componentized application code which is an instance of the ErlCOM component meta-model. Since the ErlCOM component meta-model is fixed in the case of a particular ErlCOM version the programmer can concentrate on and produce the application code in his usual way and the IDE ensures that it will be properly put into interconnected ErlCOM components. The packaging Erlang code is automatically generated and it sets up a wrapper around the application code. After the deployable source code base has been produced the programmer should establish the initial component configuration snapshot of the distributed application by instantiating the componentized source code on the available resources. The IDE provides all the necessary facilities to easily distribute the components via its Deployment Tool, which analyzes the initial component configuration snapshot and creates the ErlCOM elements by activating the corresponding ErlCOM's API operations. It should be emphasized that we talk about only the initial component configuration snapshot since the deployed application can easily reconfigure itself via ErlCOM's reflective API any time, therefore, the initial configuration plays only a temporal role. After the initial deployment has been set up the application starts running and ErlCOM' CRTK administers all the changes of the component configuration and it sends notifications about them to any listener which has subscribed to the changes. The IDE is one of the listeners that's why it is quite straight-forward to show the actual component configuration of the running system. Moreover, the component configurations – including Erlang source code – can be saved from the IDE and later recreated to restore the application to a previous known state.

The novelty of our approach is the way how design-time and run-time software aspects are intertwined. During the software development process the application code is modularized in order to avoid the problems of producing "spaghetti-code", that is, the application code is put into communicating components. Since ErlCOM is causally reflexive the components can be deployed onto it and their current configuration is continuously available and modifiable. The same software development environment can be used for application development and operation and maintenance purposes, too. (Figure 4) Moreover, the approach eliminates the unnatural separation of the functionality and the management aspects of the application; the management layer is anchored to the deployed functionality layer and reflects it via the identity mapping thanks to the meta-modeling environment where everything – including source code and component configuration – is stored in a model database.

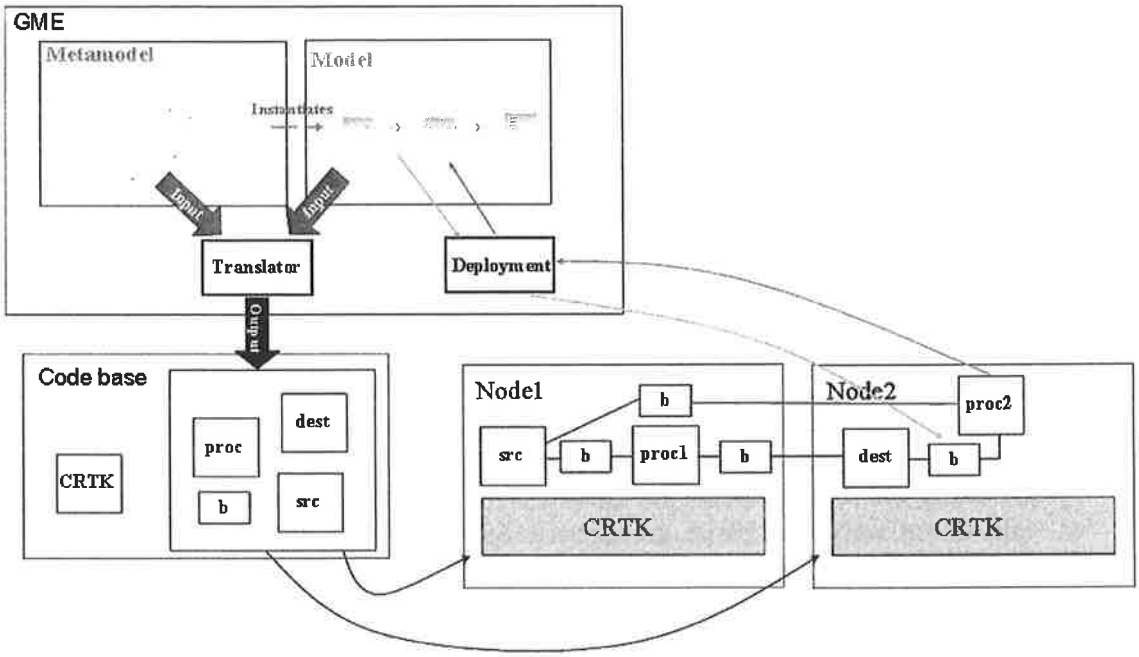


Figure 4: Meta-model assisted component configuration management

The component management has three types of operation:

- **Re-active component reconfiguration:** The application's control logic decides how to reconfigure the currently deployed component configuration to adapt to dynamically changing environmental factors. The IDE only tracks the changes; the control intelligence lies on the side of the application code. The decision-making is based both on the component configuration graph of the application and the current execution state.

- **Pro-active component configuration:** The IDE continuously evaluates the actual component configuration of the deployed application and decides when and how changes should be carried out. The intelligence emanates either from one of the plug-ins of the IDE or from any legacy tool connected to the IDE via a versatile XML importer facility. The decision-making is only based on the component configuration graph of the application.

- **Component behavior change:** The previous two reconfiguration types take effect only on the component configuration, however, the functionality of the component remains the same. The IDE has access to the model database, therefore, the programmer or any intelligent plug-in can modify the Erlang code of any of the components and via the automatic wrapper generation and deployment the functionality of the relevant parts of the application can be changed on the fly without even touching the current component configuration graph. Both the intelligence and the decision-making lie on the IDE side.

Obviously, any of the three operation types can be used separately, however, in most of the cases the pristine cases are combined seamlessly to match the environmental changes.

4. Component frameworks and the Deployment Tool

The main organization concept of the ErlCOM architecture focuses on the provision of system robustness. Therefore, the system is structured hierarchically. The configuration of the deployed components is represented by a graph whose branches are supervisors and whose leaves are gen_servers (communicating entities) or processes (component behaviors). Table 1 summarizes the mappings applied in ErlCOM.

ErlCOM	Erlang
Capsule, Caplet, Component	Supervisor
Interface, Receptacle	gen_server
Component behavior, Notification listener, Pre/post action	process

Table 1 ErlCOM concepts mapping to Erlang

However, the static configuration of the supervisors does not seem to be sufficient to describe the reconfiguration demands corresponding to the changing environment. Furthermore, reconfiguration scenarios may involve ErlCOM entities from different levels of the hierarchy, in which case the supervisor cannot be used. To be able to categorize the reconfiguration scenarios the Component Framework (CF) concept has been introduced into ErlCOM. A Component Framework is the container and manager of logically coherent entities which can be deployed onto different parts of the system. An example of the Component Framework can be seen on Figure 5. The solid lines represent supervisor relationships, the dashed lines mean communication relationships and the dotted lines represent relationships to a CF. A Component Framework always contains two unit of functionality. The Notification Listener part is notified by the CRTK when a reconfiguration action on the entities related to the actual CF has been executed. The reconfiguration scenario which has to be executed in response to the CRTK action is placed in the Reconfiguration Engine. The Reconfiguration Engine part depends heavily on the actual function of the CF and it can apply very complex reconfiguration actions which affect the whole deployment of the configuration graph. On the contrary, the Notification Listener part depends on the CRTK commands so it can be templatized.

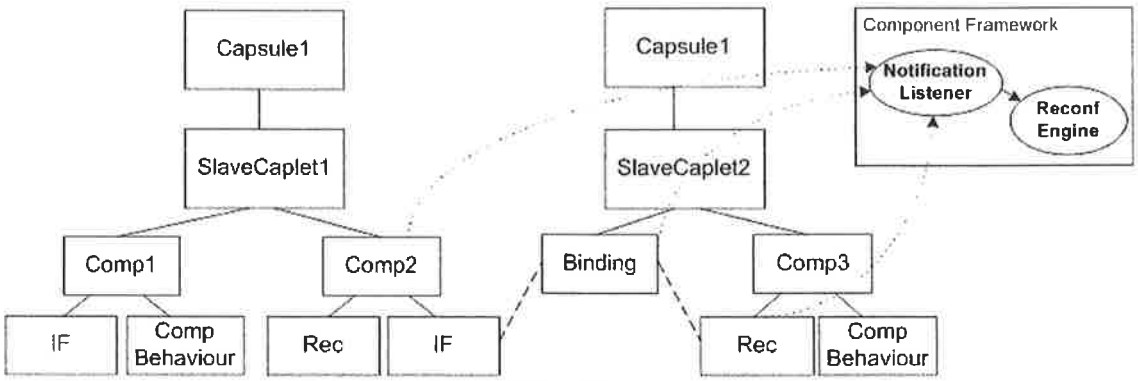


Figure 5 Structure of the Component Framework

The following actions are valid CRTK reconfiguration commands:

- Load component
- Unload component
- Destroy component
- Migrate component to an other caplet
- Bind components
- Unbind components
- Migrate binding to an other caplet
- Add pre/post actions
- Remove pre/post actions
- Add metadata to an ErlCOM entity
- Remove metadata from an ErlCOM entity
- Add a caplet
- Remove a caplet
- Add a new capsule to the ErlCOM system
- Remove a node from the ErlCOM system

Since only these 15 commands can happen on CRTK level a simple template code for notification listeners can be defined. The template code enumerates the valid CRTK commands and their corresponding parameters. The notification listener template code is the following:

```
-module(notification_template).
-export([start/0,loop/0]).

start()->
        Pid=spawn(?MODULE,loop,[]),
        register(notify_gme,Pid),
        Pid.
loop()->
        receive
                stop->true;
                        {Command,Parameters}->
                                prepare_command(Command,Parameters),
                                loop()
                end.
```

```
prepare_command(load,Parameters)->
        [CapletID,LoaderID,ModuleName,CompID,IFIDs,RecIDs]=Parameters;

prepare_command(unload,Parameters)->
        [ComponentID]=Parameters;

prepare_command(migrate_component,Parameters)->
        [OrigCapletID,OrigLoaderID,DestCapletID,DestLoaderID,CompID]=Parameters;

prepare_command(bind,Parameters)->
        [CapletID,BinderID,IFID,RecID,ModuleName,BindingID]=Parameters;

prepare_command(unbind,Parameters)->
        [IFID,RecID,BindingID]=Parameters;

prepare_command(migrate_binding,Parameters)->
        [OrigCapletID,OrigBinderID,DestCapletID,DestBinderID,BindingID]=Params;

prepare_command(addPreActionFirst,Parameters)->
        [BindingID,ModuleName] = Parameters;
prepare_command(addPreActionLast,Parameters)->
        [BindingID,ModuleName] = Parameters;
prepare_command(addPreActionBefore,Parameters)->
        [BindingID,ModuleName,NextModuleName]=Parameters;
prepare_command(addPreActionAfter,Parameters)->
        [BindingID,ModuleName,PreviousModuleName]=Parameters;

prepare_command(deletePreAction,Parameters)->
        [BindingID,PreActionName]=Parameters;

prepare_command(addPostActionFirst,Parameters)->
        [BindingID,ModuleName] = Parameters;
prepare_command(addPostActionLast,Parameters)->
        [BindingID,ModuleName] = Parameters;
prepare_command(addPostActionBefore,Parameters)->
        [BindingID,ModuleName,NextModuleName]=Parameters;
prepare_command(addPostActionAfter,Parameters)->
        [BindingID,ModuleName,PreviousModuleName]=Parameters;

prepare_command(deletePostAction,Parameters)->
        [BindingID,PostActionName]=Parameters;

prepare_command(putprop,Parameters)->
        [MetaDataID,EntityID,PropName,PropType,Value]=Parameters;

prepare_command(deleteprop,Parameters)->
        [MetaDataID,EntityID,PropName]=Parameters;

prepare_command(create_caplet,Parameters)->
        [CapletID,CapletName]=Parameters;

prepare_command(delete_caplet,Parameters)->
        [CapletID]=Parameters;

prepare_command(create_capsule,Parameters)->
        [CapsuleID,CapsuleName]=Parameters;

prepare_command(delete_capsule,Parameters)->
        [CapsuleID]=Parameters;

prepare_command(_,_)->
    unknown_command_error.
```

The earlier mentioned Deployment Tool is an aggregation of a specialized Notification Listener and a CRTK actuator. The notification listener's main tasks are listening to all CRTK events and sending them to the IDE in appropriate format. The CRTK and the IDE have different identifiers for the ErlCOM entities, therefore, the listener manage the IDE-to-CRTK identifier mapping. The actuator receives the commands from the IDE and evaluates them so that it could execute the correct sequence of CRTK commands.

5. Conclusion

The robust reconfigurability of ErlCOM and the versatile component configuration enabled by the Deployment Tool and the notification architecture coupled with the meta-modeling IDE realize the ideas behind our configuration aware distributed system design approach. The approach enables the programmer to concentrate on the application logic and the deployment adaptation logic separately and the infrastructure automatically generates the "intelligent glue" in the form of a dynamically reconfigurable component configuration which contains the application logic and behaves according to the deployment adaptation logic. In the framework of the ongoing RUNES IST project we have successfully used our approach and we hope that other Erlang projects will find the technique valuable both inside and outside Ericsson.

References

[1] RUNES IST Project, http://www.ist-runes.org/
[2] G. Batori, Z. Theisz, D. Asztalos: Robust Reconfigurable Erlang Component System, Erlang User Conference 2005, Stockholm, Sweden
[3] GME Documentation, http://www.isis.vanderbilt.edu/Projects/gme/
[4] Runes Hardware platform definition, http://www.ist-runes.org/docs/deliverables/D3_04.pdf

1 (13)

Eliminating overlapping of pattern matching when verifying Erlang programs in μCRL

Qiang Guo and John Derrick
Department of Computer Science,
The University of Sheffield,
Regent Court, 211 Portobello Street, S1 4DP, UK
{Q.Guo, J.Derrick}@dcs.shef.ac.uk

September 4, 2006

Abstract

When verifying Erlang programs in the process algebra μCRL specification, if there exists overlapping between patterns in the Erlang source codes, the problem of overlapping in pattern matching occurs when translating the Erlang codes into the μCRL specification. This paper investigates the problem and proposes an approach to overcome it. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. Structure Splitting Trees (SSTs) are defined and applied for pattern evaluation. The use of SSTs guarantees that no overlapping patterns will be introduced into the rewritten Erlang code.

Keywords: Erlang language, μCRL specification, Verification, Translation, Pattern matching, Overlapping, SSTs.

1 Introduction

Formal methods are often used for system design and verification. Formal methods are mathematically based techniques. Their mathematical underpinning allows formal methods to specify systems in a more precise, more consistent and non-ambiguous fashion. Model checking [12] is an automatic formal verification technique that has been widely used in verifying requirements and design for a variety of real-time embedded and safety-critical systems.

When verifying systems using model checking based techniques, specification of the system under development is often modelled by a formal specification language such as the process algebra. A model checker is applied to examine the properties that should hold for the system over a finite state system. If the model fails to satisfy some desired properties, faults are determined to exist in the design.

The advantage of using model checking based techniques for system verification is that, when a fault is detected, the model checker can generate a counter example. These faulty traces help system designers to understand the reasons that cause the occurrence of failures and provide clues for fixing the problem.

Two ways might be considered when using model checking based techniques for system verification. In one way, one can use 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; in the other way, one may take the program code as a starting point and abstracts that into a model that can be checked by a model checker. In the second situation, an interpretation mechanism needs to be defined in order that the source code of a programming language can be translated into the formal specification language used for describing the system under development.

Recently, verification of Erlang programs in the process algebra μCRL specification has been studied [10, 7, 8, 15]. The programming language Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. The process

algebra μCRL (micro Common Representation Language) [14] is a formal specification language. It is extended from the process algebra ACP [4] where equational abstract data types [14] are integrated into process specification. When an Erlang program is translated into a μCRL specification, a Labelled Transition System (LTS) can be obtained by using some existing tools such as the CÆSAR/ALDEBARAN Development Package (CADP) [11]. The LTS is used to check the properties that should hold for the system under development.

Benac Earle *et al.* [3, 6] studied the verification of Erlang programs in the process algebra μCRL specification and defined a set of rules for the translation of an Erlang code into μCRL. In their work, translation rules for communication, generic server, supervision tree, functions with side-effects, higher-order functions and pattern matching are defined respectively. They also developed a tool set, *etomcrl*, which automatically translates Erlang codes into a μCRL specification.

However, in the tool set *etomcrl*, pattern matching in an Erlang code are translated in a way where overlapping is not considered. This, however, could cause misinterpretation when translating an Erlang program into the μCRL specification.

In Erlang, evaluation of pattern matching works from top to bottom and from left to right. When a pattern is matched, evaluation is terminated after the corresponding clauses are executed. However, in μCRL, the tool set instantiator does not evaluate rewriting rules in a fixed order. If there exists overlapping between patterns, the problem of overlapping in pattern matching occurs, which could lead to the system being represented by a faulty model. More details about the problem are explained in Section 4.2.

This paper investigated the problem and proposed an approach to overcome it. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. In the counterpart program, functionalities defined in the original program remain unchanged. Structure Splitting Trees (SSTs) are defined and applied for pattern evaluation. The use of SSTs guarantees that no overlapping patterns will be introduced into the rewritten code.

The rest of this paper is organized as follows: Section 2 introduces the Erlang programming language; Section 3 describes the process algebra μCRL; Section 4 discusses the translation of Erlang programs into the process algebra μCRL specification and the problem of overlapping in pattern matching when translating the Erlang programs into μCRL; Section 5 looks at ways to eliminate the problem of overlapping in pattern matching; Section 6 explains the model checking Erlang programs in the μCRL specification with a case study; Conclusions are drawn in Section 7.

2 The Erlang language

The programming language Erlang [1] is a concurrent functional programming language with explicit support for real-time and fault-tolerant distributed systems. Since being developed, it has been used to implement some substantial business critical applications such as the Ericsson AXD 301 high capacity ATM switch [9].

An Erlang program consists of a set of modules, each of which defines a number of functions. A module is uniquely identified by its name as an atom. A function is uniquely identified by the module name, function name and arity (the number of arguments). Two functions with the same name and in the same module, but with different arities are two completely different functions. Functions that are accessible from other modules need to be explicitly declared as *export*. A function named *f_name* in the module *module* and with arity N is often denoted as *module:f_name/N*.

Erlang is a language with light-weight processes. Several concurrent processes can run in the same virtual machine, each of which being called a *node*. Each process has a unique identifier to address the process and a message queue to store the incoming messages. Communication between processes is handled by asynchronous message passing. The receiving process reads the message buffer by a *receive* statement. When reading a message, a process is suspended until a matching message arrives or timeout occurs. A distributed system can be constructed by connecting a number of virtual machines.

An advantage of Erlang is that it uses design patterns (provided by OTP) where a number of generic components are encapsulated. The use of OTP helps to reduce the complexity of system development and testing, while increases the robustness. *Generic server* and *supervisor* are two commonly used generic components in system design. The following briefly reviews these two components.

2.1 Generic server component

The Erlang Open Telecom Platform (OTP) supports a generic implementation of a server by providing the *gen_server* module. The *gen_server* module provides a standard set of interface functions for synchronous and asynchronous communication, debugging support, error and timeout handling, and other administrative tasks. A generic server is implemented by providing a *callback module* where (*callback*) functions are defined specifying the concrete actions of the server such as server state handling and response to messages. When a client wants to synchronously communicate with the server, it calls the standard *gen_server:call* function with a certain message as an argument. If an asynchronous communication is required, the *gen_server:cast* is invoked where no response is expected after a request is sent to the server.

```
-module(client).
-export([start_link/3, init/3]).


start_link(Locker, Resources, Type)- >
    {ok, spawn_link(client, init
        [Locker, Resources, Type])}.

init(Locker, Resources, Type) - >
    loop(Locker, Resources, Type).

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

A: Source code of client

```
-module(locker).
-behaviour(gen_server).
-export([start_link/1, init/1]).

start_link(Request) - >
    gen_server:start_link({local, locker},
            locker, [Request], []).

init(Args) - >
    {ok, Args}.
handle_call(request, Client, Pending) - >
    case Pending of
        [ ] - > {reply, ok, [Client]};
        _ - > {noreply, Pending++[Client]}
    end;
handle_call(release, Client, [_| Pending]) - >
    case Pending of
        [ ]- > {reply, done, Pending};
        _ - > gen_server:reply(hd(Pending), ok),
            {reply, done, Pending}
    end;
handle_call(stop, Client, Requests) - >
    {ok, normal, ok, Request}.

terminate(Reason, Requests) - > {ok}.
```

B: Source code of generic server

Figure 1: The source code of Erlang generic server and client.

Figure 1 illustrates a simple server-client system where a client can acquire the lock by sending a *request* message and release it by sending a *release* message. In the example the server might be called with a *request* or a *release* message. If the message is *request* and *Pending* is an empty list, the server returns the client with *ok*, and the server comes to the new state [*Client*]; otherwise, the reply is postponed and the server goes to a new state where the requesting *Client* is added to the end of *Pending* list. If a *release* message is received, the server will send a *reply* to the first waiting caller in the *Pending* list.

A *terminate* function is defined in the call back module. This function is called by the server when it is about to terminate. It allows the server to do any necessary cleaning up. Its return value is ignored.

2.2 Supervisor component

When developing concurrent and distributed systems using Erlang language, a commonly accepted assumption is that any Erlang process may unexpectedly die due to hardware failure or software errors in the code being executed in the process. Erlang/OTP supports fault-tolerance by using the supervision tree design pattern.

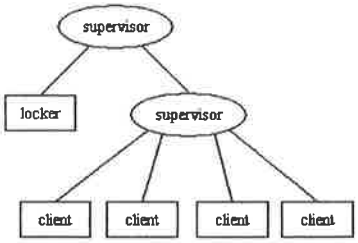


Figure 2: Supervisor tree for locker and clients.

Supervision tree is a structure where the processes in the internal nodes (supervisors) monitor the processes in the external leafs (workers). A supervisor is a process that starts a number of child processes, monitors them, handles termination and stops them on request. The children themselves can also be a supervisor, supervising its children in turn. Figure 2 demonstrates the structure of a supervision tree.

3 The process algebra μCRL

The process algebra μCRL (micro Common Representation Language) [14] is extended from the process algebra ACP [4] where equational *abstract data types* [14] are integrated into process specification.

```
sort
      Bool, N
func
      T,F: → Bool
      0: → N
      S: N → N
      add, times : N × N → N
var
      x,y : N
rew
      add(x,0) = x
      add(x,S(y)) = S(add(x,y))
      times(x,0) = 0
      times(x,S(y)) = add(x, times(x,y))
comm
      in|out = com
proc
      counter(x:N) = p
      buffer = q
```

Figure 3: An example of a μCRL specification.

A μCRL specification is comprised of two parts: the data types and the processes. Processes are declared using the keyword *proc*. A process may contain actions representing elementary activities that can be performed. These actions must be explicitly declared using the keyword *act*.

Data types used in μCRL are specified as the standard abstract data types, using sorts, functions and axioms. Sorts are declared using the key work *sort*, functions are declared using the keyword *func* and *map* is reserved for additional functions. Axioms are declared using the keyword *rew*, referring to the possibility to use rewriting technology for evaluation of terms.

A number of process-algebraic operators are defined in μCRL, these being: sequential composition ($\cdot$), non-deterministic choice ($+$), parallelism ($\|$) and communication ($|$), encapsulation (∂), hiding

(τ), renaming (ρ) and recursive declarations. A conditional expression $true \lhd condition \rhd false$ enables that data elements influence the course of a process, and an alternative quantification operator ($\sum$) provides the possibly infinite choice over some sorts.

In μCRL, parallel processes communicate via synchronization of actions. The keyword *comm* is reserved for communication specification. The communication specification describes which actions may synchronize on the level of the labels of actions. For example, in *comm in|out*, each action $in(t_1, ..., t_k)$ can communicate with $out(t_1', ..., t_k')$ provided $k = m$ and t_1 and t_1' denote the same element for $i = 1, ..., k$.

Figure 3 illustrates an example of a μCRL specification.

4 Translating Erlang into μCRL

In order that an Erlang program can be translated into μCRL, Benac Earle *et al.* [3, 6] defined a set of translation rules. In their work, translation rules for communication, generic server, supervision tree, functions with side-effects, higher-order functions and pattern matching are defined respectively. They also developed a tool set, *etomcrl*, which automatically translates Erlang codes into a μCRL specification.

4.1 Translation rules

The translation from Erlang to μCRL is performed in two stages. First, a source to source transformation is applied, resulting in Erlang code that is optimised for the verification, but has identical behaviour. Second, this code is translated to μCRL.

In μCRL, a data type *Term* is defined where all data types defined in Erlang are embedded. The translation of the Erlang data types to μCRL is then basically a syntactic conversion of constructors as shown in Figure 4.

sort
  Term
func
  pid: Natural → Term
  int: Natural → Term
  nil: → Term
  cons: Term # Term → Term
  tuplenil: Term → Term
  tuple: Term # Term → Term
  true: → Term
  false: → Term

Figure 4: Translation of data types in Erlang to μCRL

Atoms in Erlang are translated to μCRL constructors; *true* and *false* represent the Erlang booleans; *int* is defined for integers; *nil* for the empty list; *cons* for a list with an element (the head) and a rest (the tail); *tuplenil* for a tuple with one element; *tuple* for a tuple with more than one element; and *pid* for process identifiers. For example, a list $[E_1, E_2, ..., E_n]$ is translated to μCRL as $cons(E_1, cons(E_2, cons(..., nil)...))$. A tuple $\{E_1, E_2, ..., E_n\}$ is translated to μCRL as $tuple(E_1, tuple(E_2, ..., tuplenil(E_n)...))$.

Variables in Erlang are mapped directly to variables in μCRL. Operators are also translated directly, specified in a μCRL library. For example, $A + B$ is mapped to *mcrl_plus(A,B)*, where *mcrl_plus(A,B) = int(plus(term_to_nat(A), term_to_nat(B)))*.

High-order functions in an Erlang code are flattened into first-order alternatives. These first-order alternatives are then translated into rewrite rules.

Program transformation is defined to cope with side-effect functions. With a source-to-source transformation, a function with side-effects is either determined as a pure computation or a call to another function with side-effects. *Stacks* are defined in μCRL where *push* and *pop* operations are defined as communication actions. The value of a pure computation is pushed into a stack and is popped when it is called by the function.

Communication between two Erlang processes are translated into two process algebra processes, one of which is defined as a buffer, while the other implements the logic. The synchronous communication is modelled by the synchronizing actions of process algebra. One action pair is defined to synchronize the sender with the buffer of the receiver, while another action pair to synchronize the active receive in the logic part with the buffer. Figure 5 illustrates the translation rules.

```
handle_call({request,Resources,Type},
            Client, ...) ->
case check_availables(Resources,Type,Locks) of
  true ->
   NewLocks =
    map(fun(Lock) -> ...
  false ->
    ...
    case Type of
      exclusive -> ...
      shared -> ...

end.
```

A: Erlang code

```
comm gen_server_call | gscall
   = buffercall ...
proc locker_serverloop(MCRLSelf:Term,State:Term)
=
   sum(Client:Term,
   sum(Resources:Term, ...
     locker_serverloop(MCRLSelf,
       {locker_map_claim_lock(...)...}
     <|equal(locker_check_availables(...)|>
       ...

      <|equal(Type,shared)|>
      delta)))))
```

B: μCRL

Figure 5: Translation of communication in Erlang to μCRL

4.2 The problem of overlapping in pattern matching

However, in the tool set *etomcrl*, pattern matching in an Erlang code is translated in a way where overlapping is not considered. This could cause misinterpretation when translating an Erlang program into μCRL.

```
-module(check_list).
-export([check/1]).
check(List)- >
    case List of
        [ ] - >
            empty_list;
        [1 | _] - >
            head_check;
        [_ | 2, 3] - >
            tail_check
    end.
```

Figure 6: An Erlang program with overlapping patterns.

In Erlang, evaluation of pattern matching works from top to bottom and from left to right. When a pattern is matched, evaluation terminates after the corresponding clauses are executed.

However, the μCRL tool set instantiator does not evaluate rewriting rules in a fixed order. If there exists overlapping between patterns, the problem of overlapping in pattern matching occurs, which could lead to the system being represented by a faulty model.

Figure 6 illustrates an example where a list is checked. If $List = [1, 2, 3]$, the program returns *head_check* when it is executed, although *List* matches $[\_|2, 3]$ as well. However, when translating the code into the μCRL specification, the μCRL tool set instantiator does not evaluate rewriting rules in a fixed order. The return value from the μCRL model checker could be either *head_check* or *tail_check*. The final μCRL specification could represent the Erlang program in an incorrect pattern.

To overcome this problem, guards need to be defined and applied in order that rewriting rules are forced to be evaluated in a fixed order.

5 Eliminating overlapping in pattern matching

Two possible ways might be considered for the elimination of overlapping in pattern matching. In one way, one may introduce a set of guards in rewriting rules and force the μCRL tool set instantiator to evaluate rewriting rules in a fixed order, while, in the other way, one may consider to transform the Erlang source codes and rewrite the pattern matching clauses such as *case* into a series of *case_functions*.

5.1 Applying guards in rewriting rules

Benac Earle [6] proposed a method to overcome the problem of overlapping in pattern matching by introducing a set of guards into μCRL.

$$patterns\_match(P, V, \sigma)$$

$$= \begin{cases} \langle true, \sigma, \cup\{P \mapsto V\}, & var(P) \text{ and } P \notin dom(\sigma) \\ \langle equal(V, \sigma(P)), \sigma\rangle, & var(P) \text{ and } P \in dom(\sigma) \\ \langle is\_list(V) \wedge \phi \wedge \psi, \sigma_t\rangle, & P = [H|T] \\ & \langle \phi, \sigma_h\rangle = patterns\_match(H, hd(V), \sigma) \\ & \langle \psi, \sigma_t\rangle = patterns\_match(T, tl(V), \sigma_h) \\ \langle is\_tuple(V) \wedge \phi_1, \sigma_1\rangle & P = \{P1, ..., P_n\} \\ & \langle \phi_1, \sigma_1\rangle = patterns\_match(P_1, element(1, V), \sigma) \\ & \langle \phi_2, \sigma_2\rangle = patterns\_match(P_2, element(2, V), \sigma_1) \\ & \cdots \\ & \langle \phi_n, \sigma_n\rangle = patterns\_match(P_n, element(n, V), \sigma_{n-1}) \\ \langle equal(P, V), \sigma\rangle & otherwise \end{cases}$$

Figure 7: The definition of patterns_match function.

In the proposed method, a *patterns_match* function is defined (see Figure 7). This function has three arguments: a pattern, an expression and a mapping from variables to expressions, and returns a condition and a new mapping. Inside the function, an auxiliary function $var(P)$ is defined to returns the logic value *true* if P is a variable and *false* otherwise. A guard can then be defined in the μCRL specification.

Figure 8 demonstrates the translation rules where function $cond(P, V, \sigma)$ is the projection of $patterns\_match(P, V, \sigma)$ and $\{V_i \mapsto Vi\}$ represents the mapping from $\{V_1, ..., V_n\}$ to $\{V_1, ..., V_n\}$.

Note that in the μCRL specification (Figure 8-B), a *case* function $case_1$ is invoked when the evaluation of pattern matching starts. Here, $case_1$ is functionally equivalent to the first *case* clause in the Erlang code (Figure 8-A). Function $case_1$ calls another function $case_2$ where $cond(Q_1, E)$ is evaluated. If $cond(Q_1, E)$ returns *true*, it indicates that the first pattern is matched. Clause B_1 is executed and the evaluation terminates; otherwise, if $cond(Q_1, E)$ returns *false*, function $case_3$ is called where pattern Q_2 is evaluated. The evaluation continues in such an order until all patterns have been examined. It can be seen that, by introducing a guard *cond*, the μCRL instantiator evaluates the rewriting rules from top to bottom, which is identical to the order by which the patterns are examined in the Erlang code.

However, the proposed method is not applied in the tool set *etomcrl*.

5.2 Rewriting Erlang source code

The other way to overcome the problem is to rewrite the Erlang code before the translation starts. The rewriting operation rewrites all pattern matching clauses in the original code into some calling functions. A calling function is activated by a guard that is determined by function *patterns_match*.

A: Erlang source code.	B: μCRL specification.
$f\_name(V_1, ..., V_n) ->$ case E of $Q_1 -> B_1;$ ... $Q_m -> B_m$ end.	$f\_name(V_1, ..., V_n) =$ $\text{case1}(V_1, ..., V_n, E).$ $case_1(V_1, ..., V_n, V_{n+1}) =$ $case_2(V_1, ..., V_n, cond(Q_1, V_{n+1}, \{V_i \mapsto V_i\}), V_{n+1}).$ $case_2(V_1, ..., V_n, true, Q_1) = B_1;$ $case_2(V_1, ..., V_n, false, V_{n+1}) =$ $case_3(V_1, ..., V_n, cond(Q_2, V_{n+1}, \{V_i \mapsto V_i\}), V_{n+1}).$ ... $case_N(V_1, ..., V_n, true, Q_m) = B_m.$

Figure 8: Translating Erlang code into μCRL with guards.

Function *patterns_match* takes the predicate of the pattern matching clauses and one pattern as arguments. If the predicate matches the pattern, function *patterns_match* returns *true*; otherwise, *false*.

Figure 9 shows an example where *case* clauses are considered. One can easily extend the method to other pattern matching statements such as *if* and *when*.

```
-module(check_list).
-export([check/1]).

check(Args) ->
   Var1 = 1,
   Var2 = 2,
   case Args of
         P1  ->
               B1;
         P2  ->
               B2;
         P3  ->
               B3
   end.
```

A: Original code

```
-module(check_list).
-export([check/1]).

check(Args) ->
   check_case_0_0(patterns_match(Args, P1), Args, Var_List).

check_case_0_0(true, Args, Var_List) ->
    B1;
check_case_0_0(false, Args, Var_List) ->
    check_case_0_1(patterns_match(Args, P2), Args, Var_List).

check_case_0_1(true, Args, Var_List) ->
   B2;
check_case_0_1(false, Args, Var_List) ->
    check_case_0_2(patterns_match(Args, P3), Var_List).

check_case_0_2(true, Args, Var_List) ->
    B3.
```

B: Reformed code

Figure 9: Rewriting the Erlang code.

Given an Erlang code with three patterns for matching, the program (shown in Figure 9-A) is rewritten into the format as shown in Figure 9-B. Function *check* calls function *check_case_0_0*. Function *check_case_0_0* has three arguments. The first argument is the matching result between the predicate *Args* and the first pattern P_1; the second argument is the predicate *Args*; the last argument is a list of variables. It can be noted that, if $patterns\_match(Args, P_1)$ returns *true*, clauses defined in B_1 are executed; otherwise, function *check_case_0_1* is called where P_2 is evaluated. *Var_List* contains a list of variables that appears before the *case* clause and has been referred in B_1. Before constructing a case function, an analysis of variable dependency is required for the definition of *Var_List*. If a variable appears before the case clause and is referred in the clauses of a pattern, it should be added to *Var_List*. For example, if inside B_1, a clause like $K_1 = 2 \times Var_1$ is defined, one needs to add Var_1 to *Var_List* when constructing function *check_case_0_0*.

The problem now comes to define function *patterns_match*. The function cannot be simply defined as *case E of P − > true*, as it will either introduce new overlapping patterns or cause exception in the runtime.

```
patterns_match(E, P) − >
    case E of
        P− >
                true;
        _ − >
                false
    end.
```

A: Code with overlapping.

```
patterns_match(E, P)− >
    case E of
        P− >
                true
    end.
```

B: Code that causes exception.

Figure 10: Two faulty ways on defining patterns_match function in Erlang program.

Consider two examples shown in Figure 10, if *patterns_match* is defined as the one shown in Figure 10-A, new overlapping patterns will be introduced into the rewritten Erlang code (the question is, if "_" is not considered as a pattern, can we find a suitable expression of $\overline{P}$ such that $\overline{P} \cap P = \phi$ and $\overline{P} \cup P = \{all\ data\ sets\}$?).

If *patterns_match* is defined as the one shown in Figure 10-B, the code is syntactically correct and no overlapping will be introduced. However, the structure of the program will cause system exception if no pattern is matched, which reveals a semantic mistake of the program transformation.

In order to evaluate patterns effectively, we define a Structure Splitting Tree (SST).

Definition 1 *Let D be a datum of complex type. Let $D_{(1,i)}$ is a member of D and $D_{(2,j)}$ is a member of $D_{(1,i)}$. $D_{(1,i)}$ is called a first degree element of D and $D_{(2,j)}$ a second degree element of D. Let $D_{(n,k)}$ be a first degree element of $D_{(n-1,l)}$, $n \geq 2$, $D_{(n,k)}$ is called a nth degree element of D.*

It can be noted that a datum might contain more than one N^{th} degree elements.

An SST is a dependent tree where a datum of complex type is graphically represented. In an SST, each node is labelled with an ID denoted by $N_{(i,j)}$ where i indicates the layer and j the number of the node. Each node contains a datum. The type of a datum is distinguished by a graphic shape. In this work, *atom* is represented by circle, *list* by square and *tuple* diamond. The tree starts with a root node that contains the complete set of data and terminates at a terminal node where the datum is either an atom or a list that contains a "_" character.

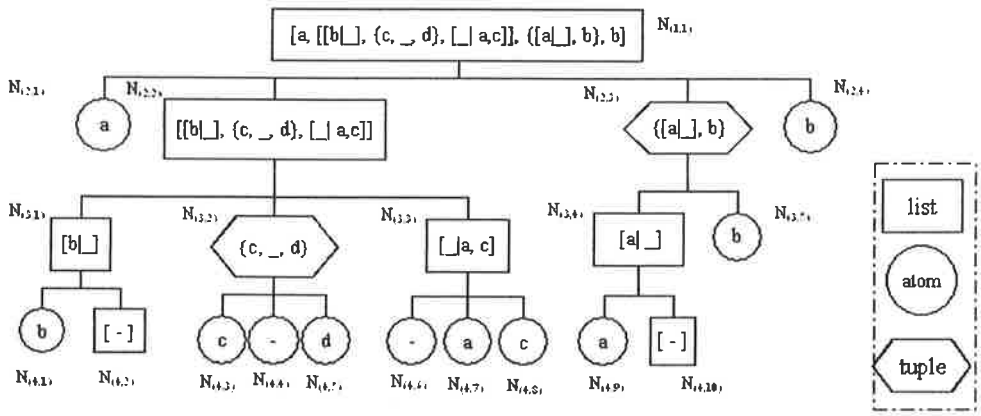


Figure 11: Structure splitting tree of a complex type datum.

Except the root node and the terminal nodes, every node $N_{(i,j)}$ has a parent node $N_{(i-1,g)}$

and several children nodes. The connection between $N_{(i,j)}$ and $N_{(i-1,g)}$ indicates that the datum contained in $N_{(i,j)}$ is a first degree element of $N_{(i-1,j)}$ and an ith degree element of the root node.

Figure 11 illustrates an example where $[a, [[b|\_], \{c, \_, d\}, [\_|a, c]], \{[a|\_], b\}, b]$ is represented by an SST. Note that node $N_{(3,1)}$ contains a list that only the head element is cared. $N_{(3,1)}$ is split into two nodes $N_{(4,1)}$ (contains an atom b) and $N_{(3,2)}$ (contains a list whose elements are not cared), both being terminal nodes.

To check if P_1 matches P_2, one can build the SSTs of P_1 and P_2, and examines the SSTs from top to bottom and from left to right. When a node in one SST is compared with the corresponding node in the other SST, the type of datum is first compared. If two datum types do not match, the evaluation returns *false* and the process of evaluation terminates; otherwise, the two data are further checked. If the datum type is *atom*, and the values of two data are not equal, the evaluation returns *false* and the process of evaluation terminates; otherwise the check of this node is finished and the process of evaluation moves to another node.

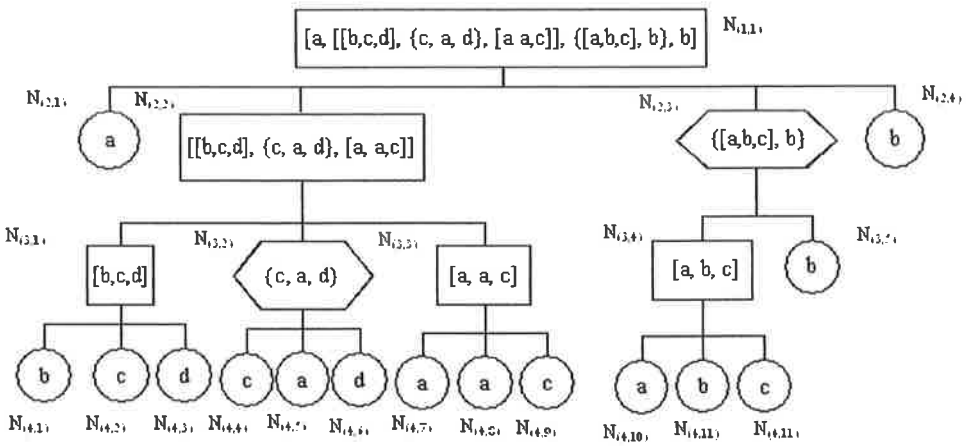


Figure 12: Structure splitting tree of [a,[[b, c, d], {c, a, d}, [a, a, c]], {[a, b, c], b}, b].

For example, to check if $P_1 = [a, [[b, c, d], \{c, a, d\}, [a, a, c]], \{[a, b, c], b\}, b]$ matches $P_2 = [a, [[b|\_], \{c, \_, d\}, [\_|a, c]], \{[a|\_], b\}, b]$, the SSTs of P_1 and P_2 are constructed, shown in Figure 12 and Figure 11 respectively. Nodes in Figure 12 are compared with the corresponding nodes in Figure 11 from top to bottom and from left to right. Root nodes of the SSTs are compared first. It can be seen that both root nodes contain a non-empty list, which indicates that the first layer comparison is matched. The evaluation moves on to the second layer where nodes $N_{(2,1)}$, $N_{(2,2)}$, $N_{(2,3)}$ and $N_{(2,4)}$ in Figure 12 are checked in sequence. $N_{(2,1)}$ contains an atom datum and its value needs to be compared with that of $N_{(2,1)}$ in Figure 11. Once the checking for the second layer is completed, the evaluation moves on to the next layer.

Note that, when evaluation comes to the fourth layer, the checking upon nodes $N_{(4,2)}$ and $N_{(4,3)}$ should be ignored since node $N_{(4,2)}$ in Figure 11 contains a list whose elements match any possible data.

The process of evaluation continues until all nodes in Figure 12 have been examined. It can be seen that, since the evaluation only check datum type (two types are the same or not) and the values of atoms (the values are equal or not), there should be no overlapping in the pattern matching and no exception will be caused during the runtime.

It is easy to see that the problem of evaluating the pattern of an SST is equivalent to that of searching nodes in a tree. A breadth-first search algorithm [5] or a depth-first search algorithm [5] can therefore be applied to solve the problem.

5.3 Comparison between two methods

It is easy to see that the two methods proposed above are functionally equivalent but realize the elimination of overlapping in pattern matching at different stages.

In the tool set *etomcrl*, before translation starts, some pre-processes are made where an Erlang program is transformed into a μErlang program. The structure of the μErlang program is closer to that of μCRL specification, which makes the translation easier.

The first method discussed in Section 5.1 considers the use of guards in the translation rules. The use of guards forces the rewriting rules to be evaluated in a fixed order. The first method copes with the problem at the stage of translation

The second method discussed in Section 5.2 considers the transformation of an Erlang program with overlapping patterns into one without overlapping patterns. Pattern matching clauses in the original code are replaced by a series of case functions. These functions are guarded by a *patterns_match* function. The second method tackles the problem at the stage of pre-process.

In this work, the second method is used as it involves in less effort in modifying the source codes of *etomcrl*.

6 Model checking Erlang in μCRL

Once Erlang programs are translated into a μCRL specification, an LTS can be derived by using some existing tool sets such as CADP. The properties of the system can then be examined through checking all transitions in the LTS.

We took a case study where a simplified version of resource manager is used. The resource manager is based on a real implementation in the control software of the AXD 301 ATM switch. It contains a *locker* and a number of *clients*. Locker provides access to an arbitrary number of resources for an arbitrary number of client processes. The clients may ask access to the resources either in a *shared* way or an *exclusive* way. For more about the resource manager, see [9].

Before translating the Erlang programs into the μCRL specification, some pre-processes are made. All functions that contain overlapping patterns are rewritten as discussed in Section 5.2. We also added an additional function (shown in Figure 6) to the original code. This is intended to evaluate the function *patterns_match* defined in Section 5.

After applying the *etomcrl* tool set to the rewritten codes, a μCRL specification file is obtained. An LTS is then generated by using CADP. Total 120 states and 193 transitions are explored by CADP. Figure 13 shows the LTS derived from the μCRL specification. The checking result implies that the model is correct, which suggests that the method proposed in this paper is capable of coping with the problem of overlapping in pattern matching.

7 Conclusions and future work

When verifying Erlang programs in the process algebra μCRL specification, if there exist overlapping patterns in the Erlang source codes, the problem of overlapping in pattern matching occurs when translating Erlang codes into the process algebra μCRL. The problem is caused due to the fact that the μCRL instantiator does not evaluate rewriting rules in a fixed order. This problem could lead to the Erlang programs being represented by a faulty μCRL model.

This paper investigated the problem and proposed an approach to overcome the problem. The proposed method rewrites an Erlang program with overlapping patterns into a counterpart program that has no overlapping patterns. The functionalities defined in the original program remain unchanged in the counterpart program. SSTs are defined and applied for pattern evaluation. An SST graphically represents a complex datum in a dependent tree.

When evaluating whether pattern P_a matches pattern P_b, the SSTs of P_a and P_b are constructed and compared. If the pattern of P_a's SST is identical to that of P_b' SST, the pattern matching evaluation returns *true;* otherwise, *false*. During the comparison of two SSTs, only the types of complex data and the values of atom data are evaluated. This guarantees that no overlapping pattern will be introduced into the rewritten Erlang codes.

A case study was carried out to evaluate the effectiveness of the proposed method. The evaluation result suggests that the proposed method is capable of coping with the problem of overlapping in pattern matching.

The evaluation of the proposed method in this paper considered the use of a comparatively simple example. More complicated systems are required to experimentally evaluate this method. This, however, remains a research topic in the future work.

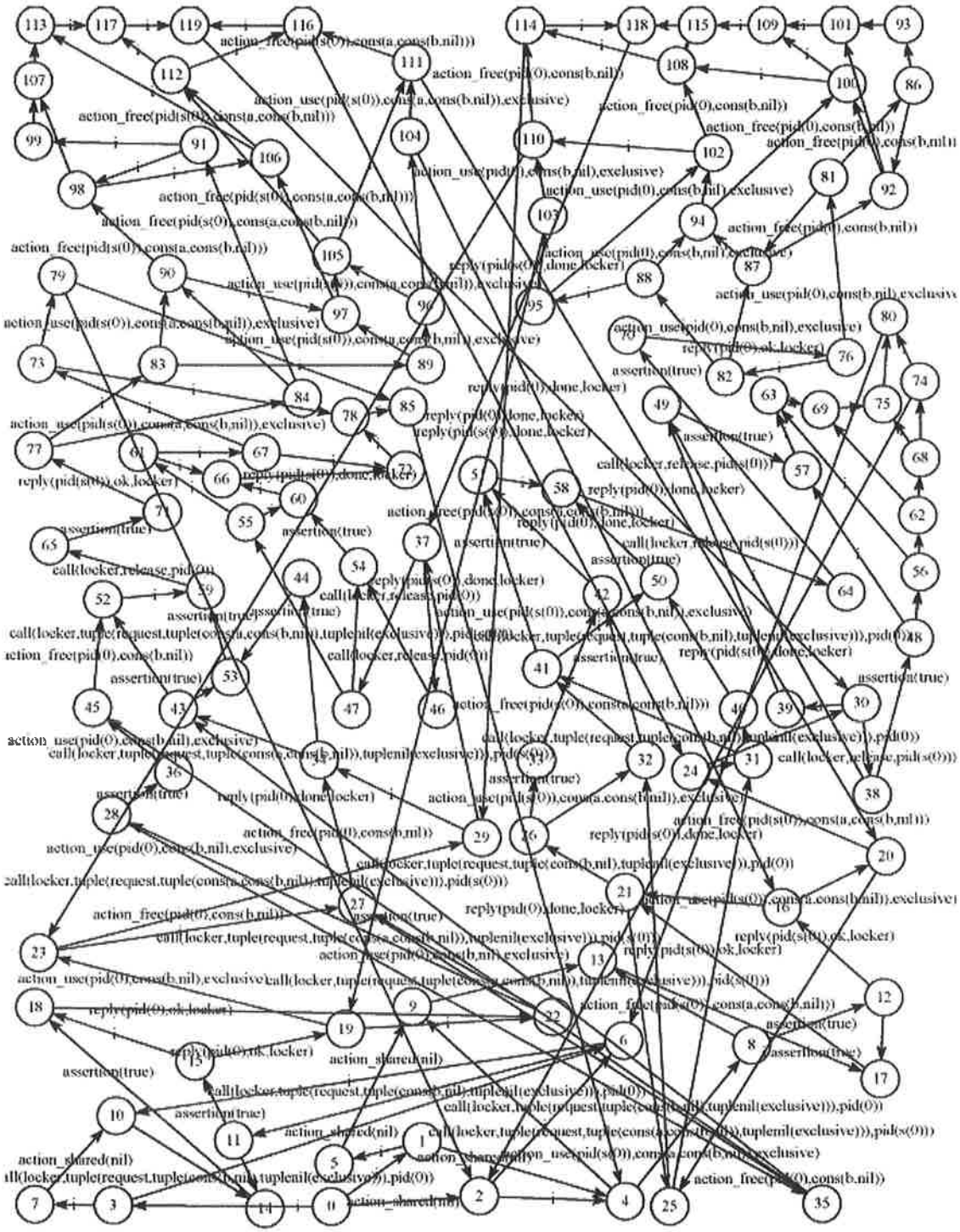


Figure 13: Labelled transition system generated from the locker system

Acknowledgements

We would like to thank Clara Benac Earle for her generous help throughout this work. We would also like to thank the developers of the tool sets of μCRL and CADP for allowing us to use the tool sets for system verification. This work is funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/C525000/1.

References

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

[2] T. Arts, C. Benac Earle, and J. Derrick. Verifying erlang code: A resource locker case-study. In *FME*, pages 184–203, 2002.

[3] T. Arts, C. Benac Earle, and Juan José Sánchez Penas. Translating erlang to μcrl. *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 135–144, 2004.

[4] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, 1990.

[5] J. Bang-Jensen and G. Gutin. *Digraphs: Theory Algorithms and Applications.* Springer-Verlag, London, 2001.

[6] C. Benac Earle. *Model check the interaction of Erlang components.* PhD thesis, The University of Kent, Canterbury, Department of Computer Science, 2006.

[7] C. Benac Earle and L. Å. Fredlund. Verification of language based fault-tolerance. In *EUROCAST*, pages 140–149, 2005.

[8] C. Benac Earle, L. Å. Fredlund, and J. Derrick. Verifying fault-tolerant erlang programs. In *Erlang Workshop*, pages 26–34, 2005.

[9] J. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. Axd 301: A new generation atm switching system. *Computer Networks*, 31:559–582, 1999.

[10] J. Blom and B. Jonsson. Automated test generation for industrial erlang applications. In *Erlang Workshop*, pages 8–14, 2003.

[11] CADP. *http://www.inrialpes.fr/vasy/cadp/.*

[12] E. Clarke, O. Grumberg, and D. Long. *Model Checking.* MIT Press, 1999.

[13] L. Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for erlang. *International Journal on Software Tools for Technology Transfer.*, 4:405–420, 2003.

[14] J. F. Groote and A. Ponse. The syntax and sematics of μcrl. In *Algebra of Communicating Processes 1994, Workshop in Computing*, pages 26–62, 1995.

[15] F. Huch. Verification of erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.

ErlHive
Safe Erlang Reloaded

An angle on community web development

Ulf Wiger, Ericsson AB

The Goal

- Web-based multi-user information management
- Blog, forum, wiki, chat, access control, ...
- What set of abstractions could allow us to treat these as convenient building blocks?

- (Not having to install a separate web server + unique version of perl for each)

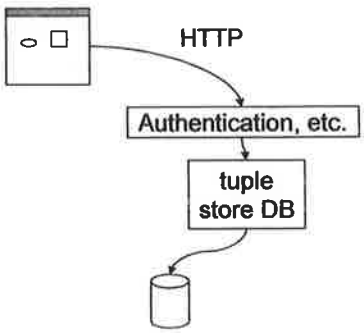
The Frustration

- Installed and tested lots of blogs, wikis and forums
- Surprisingly difficult
- Not particularly modular
- Picky about perl/python/php/mysql versions
- Esp. multi-user versions worked poorly

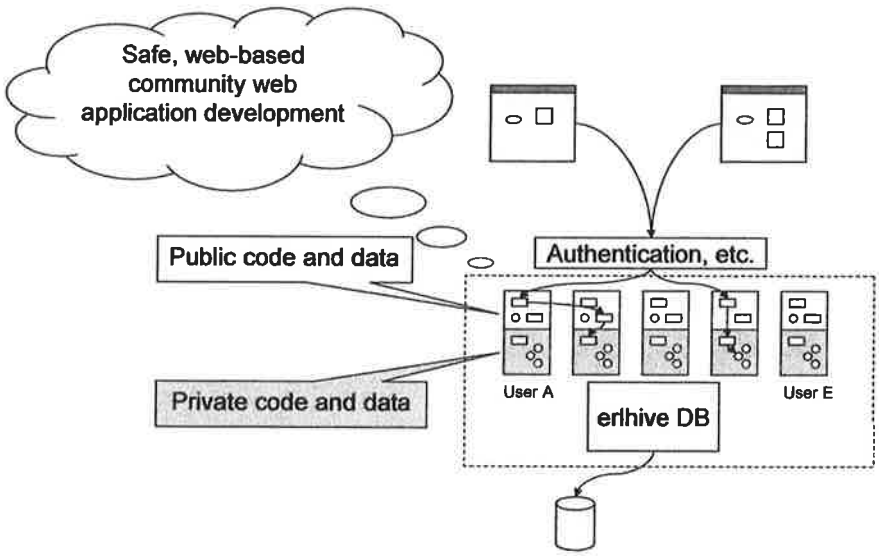
- Obvious room for improvement
- But with Erlang, lots of assembly required also.

The Tuple Store

- Joe Armstrong's idea
- A simple on-line database for web development
- Storing objects, sets and streams per user
- Joe wrote the front-end
- I wrote the back-end



Stored modules extend the vision



Back-end Concepts

- Each account contains:
  - Variable declarations (scalars, arrays, streams, and modules)
  - Areas (public and private)

Classes of Variable

- **Scalar** – can be of any type
  (a type grammar exists and is enforced)
- **Array** – an associative array (ordered set)
- **Stream** – like an inbox
  (append, lookup, delete)
- **Module** – a safe-compiled Erlang module

Access control

In the public area:

	Owner	Other Users
Scalars & arrays	Read/write/delete	Read
Streams	Append/read/delete	Append
Modules	Read/write/delete/call	Read/call

- Data and code in the private area accessible only to the owner
- The owner's public modules can call the owner's private modules

Safe code execution

- Only side-effects allowed are through the erlhive API
- Allow calls to modules/functions known to be safe (lists, ordsets, calendar, etc.)
- No spawn, send, receive, link, etc.
- Meta calls filtered at run-time (and possibly blocked)
- Everything runs in mnesia transactions
- Otherwise, no restrictions

Code example

```
-module(ex3_pub).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    {time, calendar:universal_time()},
    {caller, erlhive.user:caller()},
    {from_module, erlhive.user:from_module()},
    {owner, erlhive.user:owner()},
    {ex3_priv, ex3_priv:f()}].
```

```
-module(ex3_priv).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...].
```

Owned by user <<"ulf">>

```
-module(ex3_joe).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...,
    {ex3_pub, erlhive.ulf.ex3_pub:f()}].
```

Owned by user <<"joe">>

```
1> erlhive:with_user(
     <<"ulf">>,
     fun(M) ->
          M:put_schema(ex3_pub, [{class, module},{area, public}]),
          M:store_module(ex3_pub, "-module(ex3_pub).\n...")
     end).
```

Code example

```
-module(ex3_pub).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    {time, calendar:universal_time()},
    {caller, erlhive.user:caller()},
    {from_module, erlhive.user:from_module()},
    {owner, erlhive.user:owner()},
    {ex3_priv, ex3_priv:f()}].
```

```
-module(ex3_priv).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...].
```

```
-module(ex3_joe).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...,
    {ex3_pub, erlhive.ulf.ex3_pub:f()}].
```

"Meta functions" for introspection

Package syntax for calling other users' modules

Execution

```
-module(ex3_pub).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    {time, calendar:universal_time()},
    {caller, erlhive.user:caller()},
    {from_module, erlhive.user:from_module()},
    {owner, erlhive.user:owner()},
    {ex3_priv, ex3_
```

```
-module(ex3_priv).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...].
```

```
-module(ex3_joe).
-export([f/0]).
f() ->
   [{'?MODULE', ?MO
    ...,
    {ex3_pub, erlhi
```

```
2> erlhive:with_user(
     <<"joe">>,
     fun(M) ->
         M:apply(erlhive.joe.ex3_joe, f, [])
     end).
[{'?MODULE', 'erlhive.joe.ex3_joe'},
 {caller, <<"joe">>},
 {from_module, 'erlhive.user'},
 {owner, <<"joe">>},
 {ex3_pub, [{'?MODULE', 'erlhive.ulf.ex3_pub'},
            {time, {{2006,11,9},{16,53,17}},
            {caller, <<"joe">>},
            {owner, <<"ulf">>},
            {ex3_priv, [{'?MODULE', 'erlhive.ulf.ex3_priv'},
                        {caller, <<"ulf">>},
                        {from_module, 'erlhive.ulf.ex3_pub'},
                        {owner, <<"ulf">>}]}]}]
```

Execution

```
-module(ex3_pub).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    {caller, erlhive.user:caller()},
    {from_module, erlhive.user:from_module()},
    {owner, erlhive.user:owner()},
    {ex3_priv, ex3_priv:f()}]
```

```
-module(ex3_priv).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...].
```

```
-module(ex3_joe).
-export([f/0]).
f() ->
   [{'?MODULE', ?MO
    ...,
    {ex3_pub, erlhi
```

```
2> erlhive:with_user(
     <<"joe">>,
     fun(M) ->
         M:apply(erlhive.ulf.ex3_priv, f, [])
     end).

** exited: {aborted, {{undef,[{{'erlhive.ulf.ex3_priv',
                                <<"joe">>,
                                'erlhive.user'}, f, 0},
                               {erlhive,with_watchdog,1},
                               ...]}, ...}
```

Cannot call another user's private modules.
Restricted calls appear as undefs.

Profiling

```
-module(ex3_pub).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    {caller, erlhive.user:caller()},
    {from_module, erlhive.user:from_module()},
    {owner, erlhive.user:owner()},
    {ex3_priv, ex3_priv:f()}]
```

```
-module(ex3_priv).
-export([f/0]).
f() ->
   [{'?MODULE', ?MODULE},
    ...].
```

```
-module(ex3_joe).
-export([f/0]).
f() ->
   [{'?MODULE', ?MO
    ...,
    {ex3_pub, erlhi
```

```
2> erlhive:profile(
     <<"joe">>,
     fun(M) ->
         M:apply(ex3_joe, f, [])
     end).

{[{'?MODULE','erlhive.joe.ex3_joe'},
  ...],
[{trace,<0.403.0>,call,{erlhive_user,apply,4}},
 {trace,<0.403.0>,call,{'erlhive.joe.ex3_joe',f,1}},
 {trace,<0.403.0>,call,{'erlhive.ulf.ex3_pub',f,1}},
 {trace,<0.403.0>,return_to,{'erlhive.ulf.ex3_pub',f,1}},
 {trace ,<0.403.0>,return_to,{'erlhive.ulf.ex3_pub',f,1}}]}
```

A censored call trace. Can be followed by a specific trace on 'visible' modules. (work in progress...)

Status

- Beta version at Sourceforge
  http://erlhive.sourceforge.net
- Authenticating web server front-end
- Components
  - Simple web-based management front-end
  - Blog with threaded comments
  - Wiki code syntax library
  - Role-Based Access Control library

ERLANG

Erlang/OTP Development at Ericsson

ERICSSON
TAKING YOU FORWARD

ERLANG

OTP R11B-2 released

- Emulator
  - **Kernel poll support can now be combined with SMP support.** Currently the following kernel poll versions exist: /dev/poll, epoll, and kqueue. Linux kpoll has been replaced with epoll. Some time in the future there will also be a kernel poll version using Solaris event ports.
  - **The SMP emulator now avoids locking for the following operations:** atom_to_list/1, atom comparision, atom hashing, erlang:apply/3.

2

OTP R11B-2 released

- Stdlib and Compiler
  - Option 'strict_record_tests' in now made default that is, reading a field from a record using the `Record#record_tag.field` syntax will fail if `Record` is not a record of the correct type.
    Can be shut off with option 'no_strict_record_tests' or with environment variable ERL_COMPILER_OPTIONS.
- Inets
  - Enhancements regarding aynchronous HTTP-request
  - New option to support PROXY-Authorization
  - Bug corrections regading parsing of URI's, chunked decoding, empty body, ...

OTP R11B-2 released

- Dialyzer
  - Dialyzer's building of PLT is now based on a different type inference algorithm. More specifically, Dialyzer uses inferrence of refined success typings to infer function prototypes.
    As a result, Dialyzer bases its analysis on a significantly more powerful basis and thus is able to detect more discrepancies. In particular, Dialyzer is now able to find more discrepancies in the form of malformed uses of Erlang/OTP library functions. The downside is that building the PLT is a considerably slower process. We will work on improving that, but let us assure you that the time building the PLT is well spent.
  - Dialyzer takes into account the BEAM compiler directive -compile({nowarn_unused_function, {F,A}}). and suppresses the warning that function F/A will never be called.
  - Dialyzer's default initial PLT now also includes "mnesia".
- QLC now with support for faster join of 2 tables.
  - Support for two kinds of join:
    - lookup join, that uses existing indices
    - merge join, that takes 2 sorted inputs
  - Several other enhancements

Ongoing work

Documentation

- Plan to release "Docbuilder" as an application in OTP together with the OTP documentation sources in XML. Docbuilder is pure Erlang and produces HTML.
- Edoc is better integrated to get the same format as all other OTP documentation.

Ongoing work

SMP support

- Removing the big lock around IO is ongoing, allowing for parallell execution in drivers.
  A new API for drivers which support parallell invokation will be introduced. Old driver API still supported.
- Probable release in Q2 2007.

4(4)

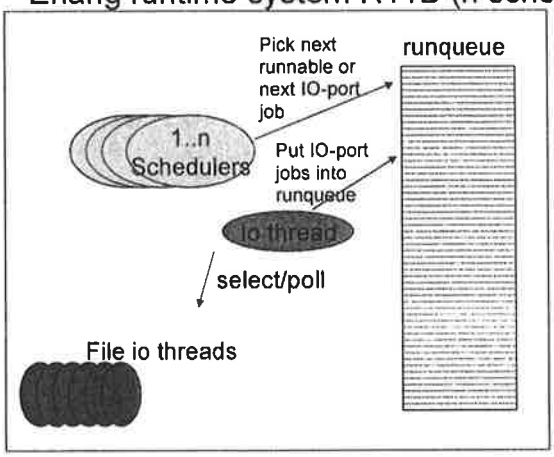
Ongoing work

Misc

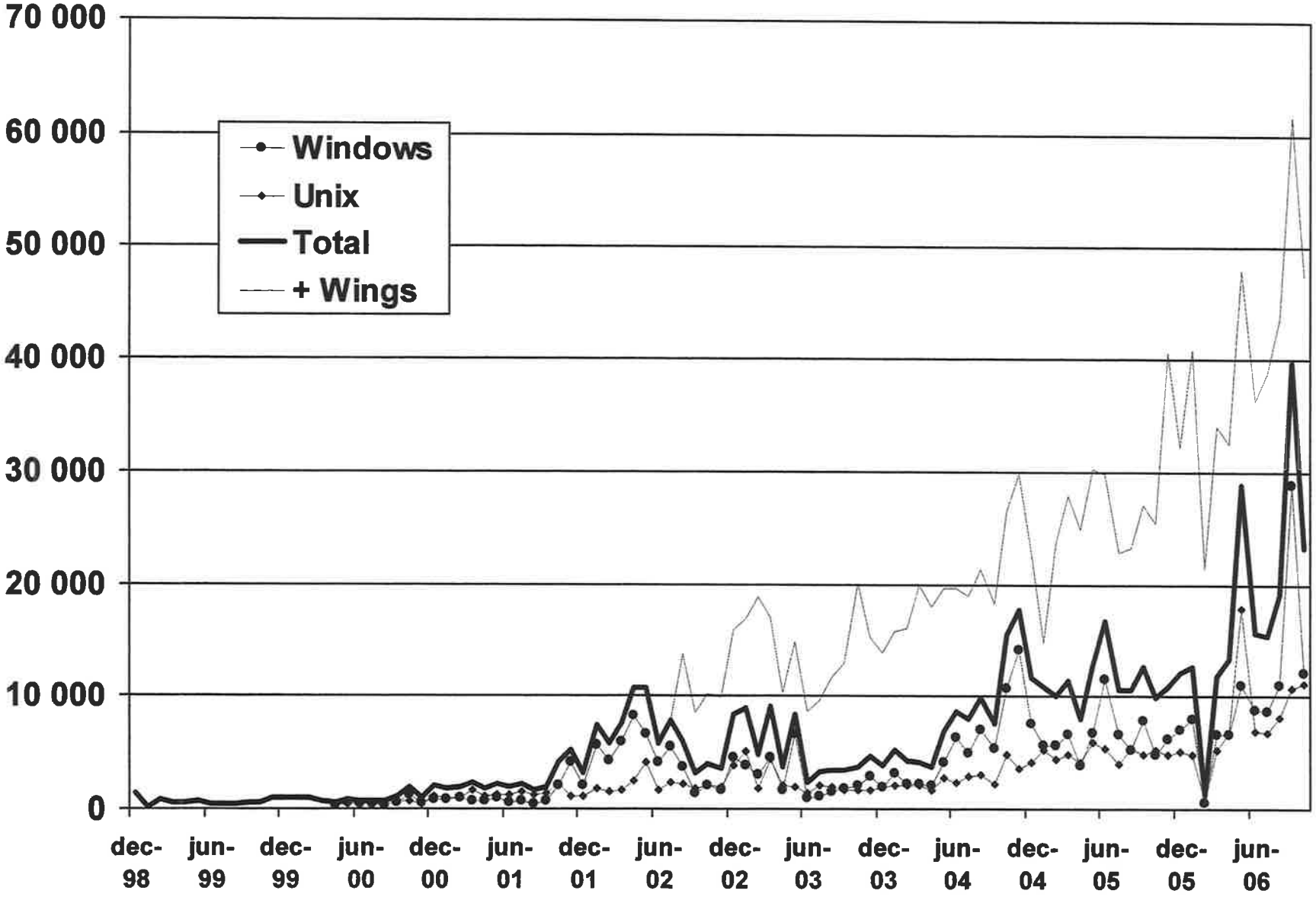
- Constant pool per module e.g.
  f() -> {a,b,c,[10,20,...],.....very big constant term...}.
  The term will be built by the compiler once and for all.
  Today the term is built every time function f is called.
- Support for Bitstr (bit sized binaries i.e not a multiple of 8 bits) and binary comprehensions in the compiler
- XMERL with XMLschema validation

Multiprocessor support

Erlang runtime system R11B (n schedulers)

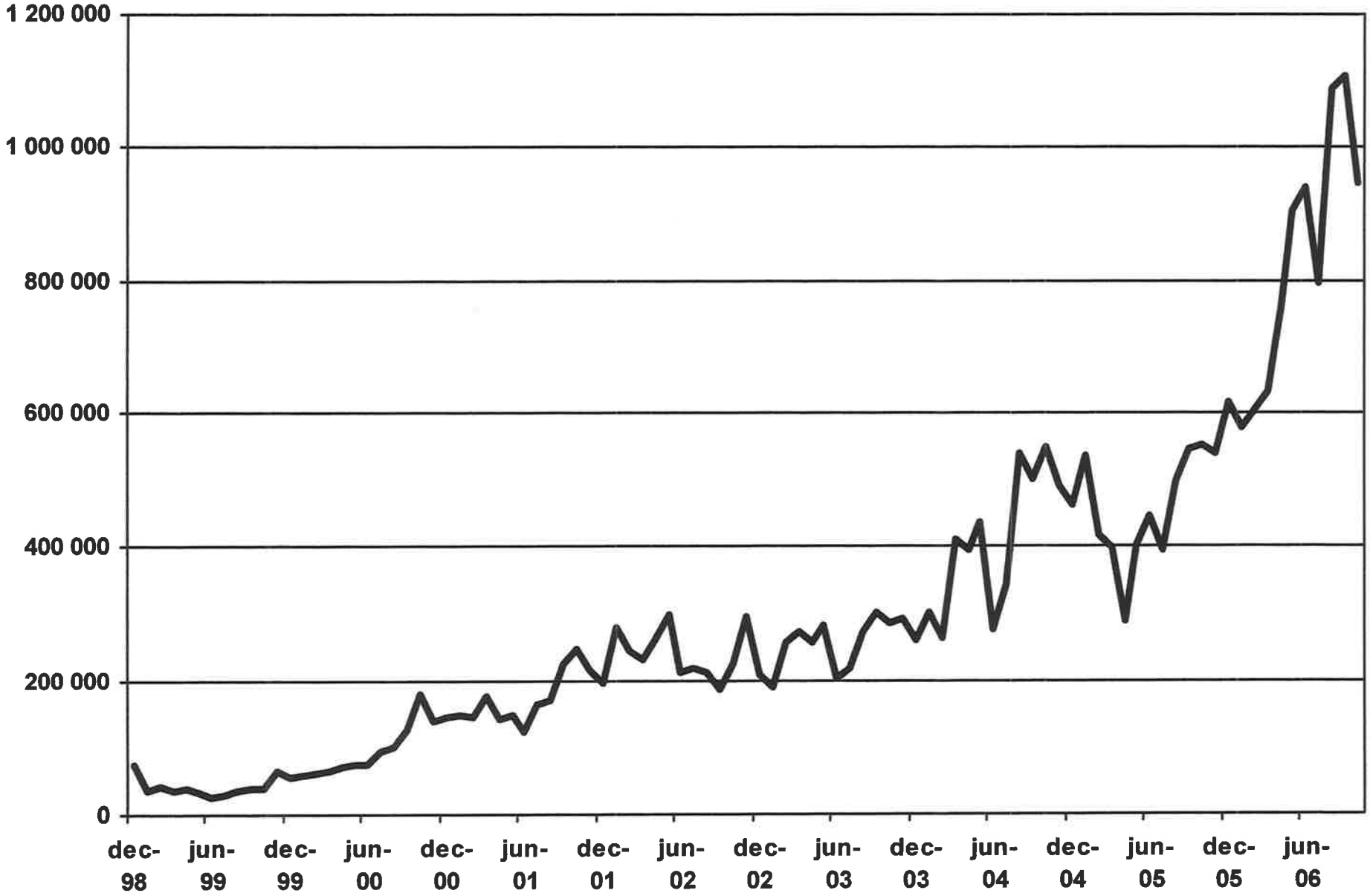


Downloads/month from `www.erlang.org` or bundled with Wings

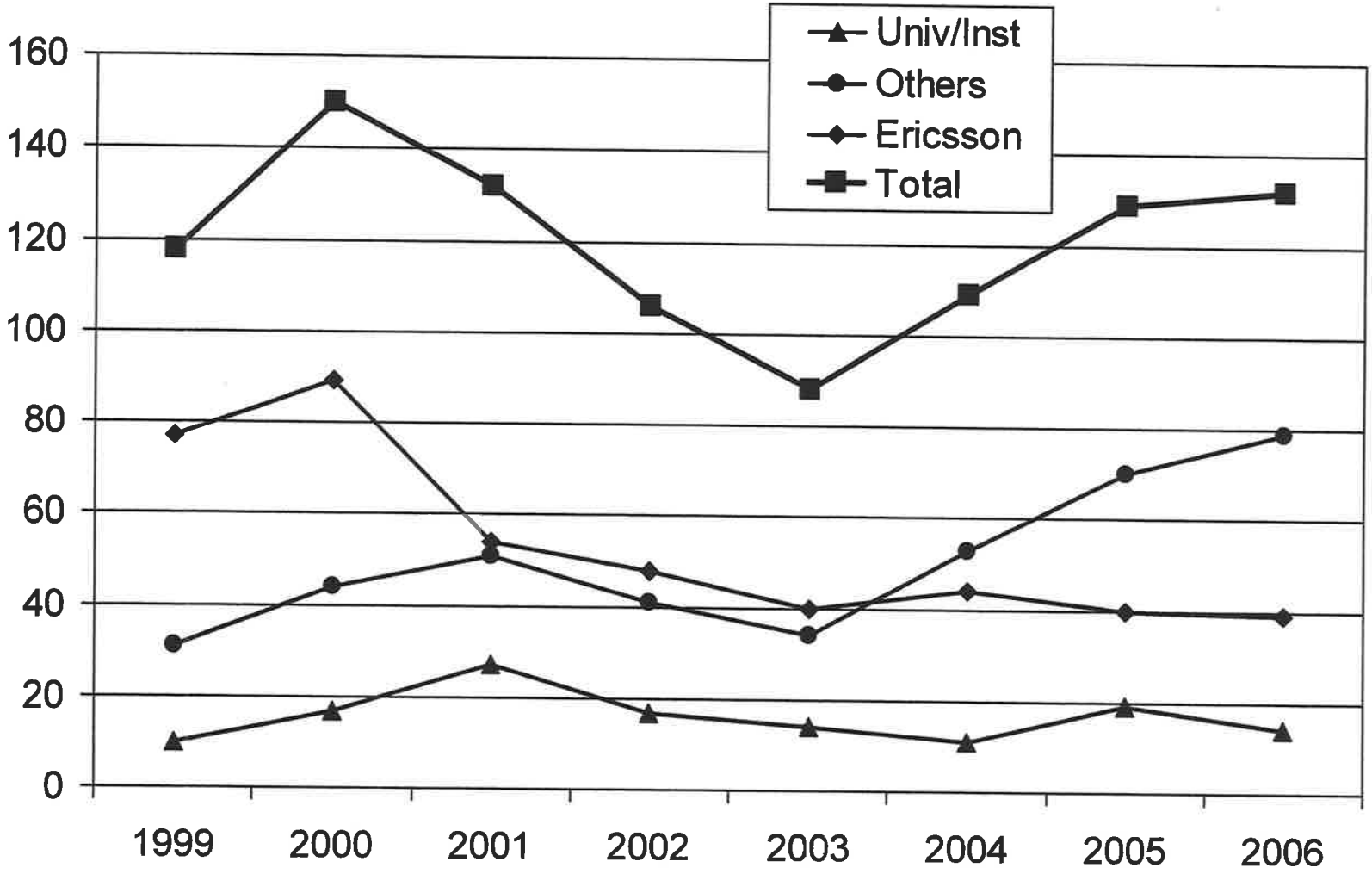


1(3)

Requests/month to www.erlang.org



EUC participation



Erlang/OTP User Conference 2006

Speakers and chairman

John	Hughes	Chalmers University of Technology	Göteborg	Sweden	john.hughes@quviq.com
Bjarne	Däcker	cs-lab.org	Segeltorp	Sweden	bjarne@cs-lab.org
Domonkos	Asztalos	Ericsson	Budapest	Hungary	Domonkos.Asztalos@ericsson.com
Gabor	Batori	Ericsson	Budapest	Hungary	Gabor.Batori@ericsson.com
Zoltán	Theisz	Ericsson	Budapest	Hungary	Zoltan.Theisz@ericsson.com
Ulf	Wiger	Ericsson	Stockholm	Sweden	ulf.wiger@ericsson.com
Kenneth	Lundin	Ericsson OTP team	Stockholm	Sweden	kenneth.lundin@ericsson.com
Jan Henry	Nyström	Erlang Training and Consulting	Uppsala	Sweden	jan@erlang-consulting.com
Thomas	Arts	IT University of Göteborg	Göteborg	Sweden	arts@ituniv.se
Erik	Stenman	Kreditor	Stockholm	Sweden	Erik.Stenman@kreditor.se
Mickaël	Rémond	Process-one	Paris	France	mickael.remond@process-one.net
Christophe	Romain	Process-one	Paris	France	
Jérôme	Sautret	Process-one	Paris	France	
Romain	Lenglet	Tokyo Institute of Technology	Tokyo	Japan	rlenglet@users.forge.objectweb.org
Zoltan	Horvath	University Eötvös Loránd	Budapest	Hungary	hz@inf.elte.hu
Vincenzo	Nicosia	University of Catania	Catania	Italy	me@katolaz.homeunix.net
Huiqing	Li	University of Kent	Canterbury	England	H.Li@kent.ac.uk
Simon	Thompson	University of Kent	Canterbury	England	S.J.Thompson@kent.ac.uk
John	Derrick	University of Sheffield	Sheffield	England	J.Derrick@dcs.shef.ac.uk
Qiang	Guo	University of Sheffield	Sheffield	England	Q.Guo@dcs.shef.ac.uk
Dale	Harvey	vixo.com	Edinburgh	Scotland	harveyd@gmail.com
Conrad	Levitt	vixo.com	Edinburgh	Scotland	benefitsdragon@gmail.com

Participants

Dmitri	Girenko	Akumiitti Oy	Helsinki	Finland	Dmitri.Girenko@akumiitti.com
Patrik	Winroth	bwi systems	Stockholm	Sweden	patrik@bwi.se
Tee	Teoh	Canadian Bank Note Co	Ottawa	Canada	tteoh@cbnco.com
Geoff	Cant	Catalyst IT	Wellington	New Zealand	geoff@catalyst.net.nz
Pascal	Brisset	Cellicium	Bagneux	France	pascal.brisset@cellicium.com
Dominic	Williams	Cellicium	Bagneux	France	dominic.williams@cellicium.com
Matthias	Läng	Corelatus	Stockholm	Sweden	matthias@corelatus.se
Ulf	Svarte Bagge	Corelatus	Stockholm	Sweden	ulf@corelatus.se
Mikael	Karlsson	Creado Systems	Stockholm	Sweden	mikael.karlsson@creado.com
Taavi	Talvik	Elisa	Tallinn	Estonia	taavi.talvik@elisa.ee

Participants cont.					
Ola	Andersson	Ericsson	Stockholm	Sweden	ola.a.andersson@ericsson.com
Joe	Armstrong	Ericsson	Stockholm	Sweden	joe.armstrong@ericsson.com
John-Olof	Bauner	Ericsson	Stockholm	Sweden	john-olof.bauner@ericsson.com
Éva	Bihari	Ericsson	Budapest	Hungary	eva.bihari@ericsson.com
Mats	Cronqvist	Ericsson	Budapest	Hungary	mats.cronqvist@ericsson.com
Graham	Crowe	Ericsson	Stockholm	Sweden	graham.crowe@ericsson.com
Anders	Danne	Ericsson	Stockholm	Sweden	anders.danne@gmail.com
David	Haglund	Ericsson	Linköping	Sweden	david.xa.haglund@ericsson.com
Håkan	Huss	Ericsson	Stockholm	Sweden	hakan.huss@ericsson.com
Joakim	Johansson	Ericsson	Stockholm	Sweden	joakim.l.johansson@ericsson.com
Angela	Johansson	Ericsson	Linköping	Sweden	angela.xa.johansson@ericsson.com
Bengt	Kleberg	Ericsson	Stockholm	Sweden	bengt.kleberg@ericsson.com
Tomas	Langer	Ericsson	Stockholm	Sweden	tomas.langer@ericsson.com
Leslaw	Lopacki	Ericsson	Göteborg	Sweden	leslaw.a.lopacki@ericsson.com
Hans	Nilsson	Ericsson	Stockholm	Sweden	hans.r.nilsson@ericsson.com
Lars	Thorsén	Ericsson	Stockholm	Sweden	lars.thorsen@ericsson.com
András	Vajda	Ericsson	Jorvas	Finland	andras.vajda@ericsson.com
Chris	Williams	Ericsson	Stockholm	Sweden	chris.williams@ericsson.com
Peter	Andersson	Ericsson OTP team	Stockholm	Sweden	peppe@erix.ericsson.se
Ingela	Andin Anderton	Ericsson OTP team	Stockholm	Sweden	
Gunilla	Arendt	Ericsson OTP team	Stockholm	Sweden	gunilla@erix.ericsson.se
Hans	Bolinder	Ericsson OTP team	Stockholm	Sweden	
Jakob	Cederlund	Ericsson OTP team	Stockholm	Sweden	jakob@erix.ericsson.se
Björn-Egil	Dahlberg	Ericsson OTP team	Stockholm	Sweden	bjorn-egil.dahlberg@ericsson.com
Niclas	Eklund	Ericsson OTP team	Stockholm	Sweden	
Richard	Green	Ericsson OTP team	Stockholm	Sweden	
Dan	Gudmundsson	Ericsson OTP team	Stockholm	Sweden	
Björn	Gustavsson	Ericsson OTP team	Stockholm	Sweden	bjorn@erix.ericsson.se
Micael	Karlberg	Ericsson OTP team	Stockholm	Sweden	micael.karlberg@ericsson.com
Bertil	Karlsson	Ericsson OTP team	Stockholm	Sweden	
Håkan	Mattsson	Ericsson OTP team	Stockholm	Sweden	hakan@erix.ericsson.se
Raimo	Niskanen	Ericsson OTP team	Stockholm	Sweden	raimo@erix.ericsson.se
Patrik	Nyblom	Ericsson OTP team	Stockholm	Sweden	pan@erix.ericsson.se
Göran	Stupalo	Ericsson OTP team	Stockholm	Sweden	
Martin	Carlson	Erlang Training and Consulting	London	England	

Participants cont.

Alfonso Rivero	Cebrian	Erlang Training and Consulting	London	England	
Francesco	Cesarini	Erlang Training and Consulting	London	England	francesco@erlang-consulting.com
Francesca	Gangemi	Erlang Training and Consulting	London	England	
Mazen	Harke	Erlang Training and Consulting	London	England	
Oscar	Hellström	Erlang Training and Consulting	London	England	
Andreas	Hillkvist	Erlang Training and Consulting	London	England	
Ludvig	Johanson	Erlang Training and Consulting	London	England	
Lukas	Larsson	Erlang Training and Consulting	London	England	
Adam	Lindberg	Erlang Training and Consulting	London	England	
Laurent	Picouleau	Erlang Training and Consulting	London	England	
Michal	Slaski	Erlang Training and Consulting	London	England	
Marcus	Taylor	Erlang Training and Consulting	London	England	marcus@erlang-consulting.com
Gillan	Ward	Erlang Training and Consulting	London	England	
Yariv	Sadan	erlyweb.org	Boston	USA	yarivvv@gmail.com
Gordon	Levitt	Heriott-Watt University	Edinburgh	Scotland	gl20@hw.ac.uk
Niklas	Hanberger	HiQ	Stockholm	Sweden	Niklas.Hanberger@hiq.se
Magnus	Fröberg	Kreditor	Stockholm	Sweden	
Mikael	Lindmark	Kreditor	Stockholm	Sweden	
Daniel	Luna	Kreditor	Stockholm	Sweden	
Håkan	Stenholm	Kreditor	Stockholm	Sweden	
Torbjörn	Törnkvist	Kreditor	Stockholm	Sweden	
Jane	Walerud	Kreditor	Stockholm	Sweden	jane@walerud.com
Marcus	Arendt	Marcus Arendt AB	Stockholm	Sweden	marcus@arendt.se
Nils	Decker	Media Consult International GmbH	Hamburg	Germany	n.decker@mci-broadcast.com
Thomas	Lindgren	Millpond Services Ltd	London	England	thomasl_erlang@yahoo.com
Bahram	Bahar	Mobile Arts	Stockholm	Sweden	bahram.bahar@mobilearts.com
Johan	Blom	Mobile Arts	Stockholm	Sweden	johan.blom@mobilearts.com
Göran	Båge	Mobile Arts	Stockholm	Sweden	goran.bage@mobilearts.com
Jonas	Falkevik	Mobile Arts	Stockholm	Sweden	jonas.falkevik@mobilearts.com
Alexander	Harju	Mobile Arts	Stockholm	Sweden	alexander.harju@mobilearts.com
Dragan	Havelka	Mobile Arts	Stockholm	Sweden	dragan.havelka@mobilearts.com
Rikard	Johansson	Mobile Arts	Stockholm	Sweden	rikard.johansson@mobilearts.com
Martin	Kjellin	Mobile Arts	Stockholm	Sweden	martin.kjellin@mobilearts.com
Thomas	Mattison	Mobile Arts	Stockholm	Sweden	thomas.mattisson@mobilearts.com
Göran	Oettinger	Mobile Arts	Stockholm	Sweden	goran.oettinger@mobilearts.com

Participants cont.

Esbjörn	Dominque	Optimobile AB	Stockholm	Sweden	
Pekka	Hedqvist	Optimobile AB	Stockholm	Sweden	Pekka.Hedqvist@Optimobile.se
Johan	Montelius	Royal Institue of Technology	Stockholm	Sweden	johanmon@kth.se
Amir	Payberah	Royal Institue of Technology	Stockholm	Sweden	payberah@kth.se
Fatemeh	Rahimian	Royal Institue of Technology	Stockholm	Sweden	rahimian@kth.se
Filippo	Pacini	S.G. Consulting	Rome	Italy	pacini@sgconsulting.it
Gösta	Ask	SalveLinus	Stockholm	Sweden	g.ask@telia.com
Lennart	Öhman	Sjöland & Thyselius Telecom AB	Stockholm	Sweden	lennart.ohman@st.se
Morgan	Eriksson	SoftCM	Stockholm	Sweden	morgan.eriksson@comhem.se
Robert	Virding	Swedish Defence Materiel Administration	Stockholm	Sweden	robert.virding@telia.com
Kristoffer	Andersson	Synapse Mobile Networks	Stockholm	Sweden	toffe@synap.se
Per	Hallin	Synapse Mobile Networks	Stockholm	Sweden	perhal@synap.se
Peter	Lund	Synapse Mobile Networks	Stockholm	Sweden	peterl@synap.se
Johan	Bevemyr	Tail-f	Stockholm	Sweden	jb@tail-f.com
Martin	Björklund	Tail-f	Stockholm	Sweden	mbj@tail-f.com
Joakim	Grebenö	Tail-f	Stockholm	Sweden	
Per	Hedeland	Tail-f	Stockholm	Sweden	
Håkan	Millroth	Tail-f	Stockholm	Sweden	hakanm@tail-f.com
Ola	Samuelsson	Tail-f	Stockholm	Sweden	
Sebastian	Strollo	Tail-f	Stockholm	Sweden	seb@strollo.org
Claes	Wikström	Tail-f	Stockholm	Sweden	klacke@tail-f.com
Adnan	Shafi	Telegia Technologies	Stockholm	Sweden	adnan@optimobile.se
Peter-Henry	Mander	T-Mobile	Hatfield	England	erlang@manderp.freeserve.co.uk
Chandrashekhar	Mullaparthi	T-Mobile	Hatfield	England	Chandrashekhar.Mullaparthi@t-mobile.co.uk
Fredrik	Thulin	University of Stockholm	Stockholm	Sweden	ft@it.su.se
Per	Gustafsson	University of Uppsala	Uppsala	Sweden	per.gustafsson@it.uu.se
Tobias	Lindahl	University of Uppsala	Uppsala	Sweden	Tobias.Lindahl@it.uu.se
Gordon	Guthrie		Edinburgh	Scotland	gordonguthrie@backawinner.gg
Per Einar	Strömme		Stockholm	Sweden	stromme@telia.com
Göran	Östlund		Stockholm	Sweden	goran.may@chello.se

Updated 2006-11-02