14th International Erlang/OTP User Conference

Stockholm, November 13, 2008

Proceedings

http://www.erlang.se/euc/08/

QuviQ

corelatus

Mobile Arts

ERICSSON

Erlang Training and Consulting

KREDITOR

tail-f

process one INSTANT MESSAGES CREATIVE BUSINESSES

Sjöland&Thyselius

synapse mobile networks s.a.

Conference Programme

08.30 *Registration.*

Session I

09.00 **Update on the EU project ProTest.**
Thomas Arts, Quviq, Göteborg,
Francesco Cesarini, Erlang Training & Consulting, London, and
Simon Thompson, University of Kent.

09.45 **Progress of the RefactorErl Project.**
Zoltán Horváth, Eötvös Loránd University, Budapest, Hungary.

10.05 **Formal Specifications for Free!**
John Hughes, Quviq, Göteborg.

10.30 *Coffee.*

Session II

11.00 **The Erlang Web - an Open Source Fault Tolerant and Scalable Web Framework.**
Michal Slaski, Erlang Training & Consulting, London.

11.30 **Ad Serving with Erlang.**
Bob Ippolito, Mochi Media, San Francisco.

12.00 **Erlang DTRace.**
Garry Bulmer, Coventry.

12.30 *Lunch.*

Session III

14.00 **Gradual Typing of Erlang Programs.**
Kostis Sagonas, University of Uppsala.

14.30 **Testing a SIP decoder with QuickCheck.**
Hans Nilsson, Ericsson.

15.00 **An ABNF (Augmented Backus-Naur Form) Parser Generator for Erlang.**
Anders Nygren, Telexpertise de Mexico.

--.-- **Autocoding State Machine in Erlang: A Case Study of Model-Driven Software Development.**
Yu Guo, University of Southern Denmark, Sønderborg,
Torben Hoffman and Nicholas Gunder, Motorola, Glostrup.
(Only in the proceedings.)

15.30 *Coffee.*

Session IV

16.00 **ErlIDE - The Erlang Eclipse Plugin.**
Jacob Cederlund, Ericsson, and
Vlad Dumitrescu, HiQ, Stockholm.

16.30 **LFE (Lisp Flavoured Erlang).**
Robert Virding, Stockholm.

17.00 **Inside the Erlang VM, focusing on SMP (plus Erlang/OTP News).**
Kenneth Lundin, Ericsson.

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

Update on EU project ProTest

ProTest

property based testing

University of Kent Computing

Erlang

QuviQ

UNIVERSIDADE DA CORUÑA

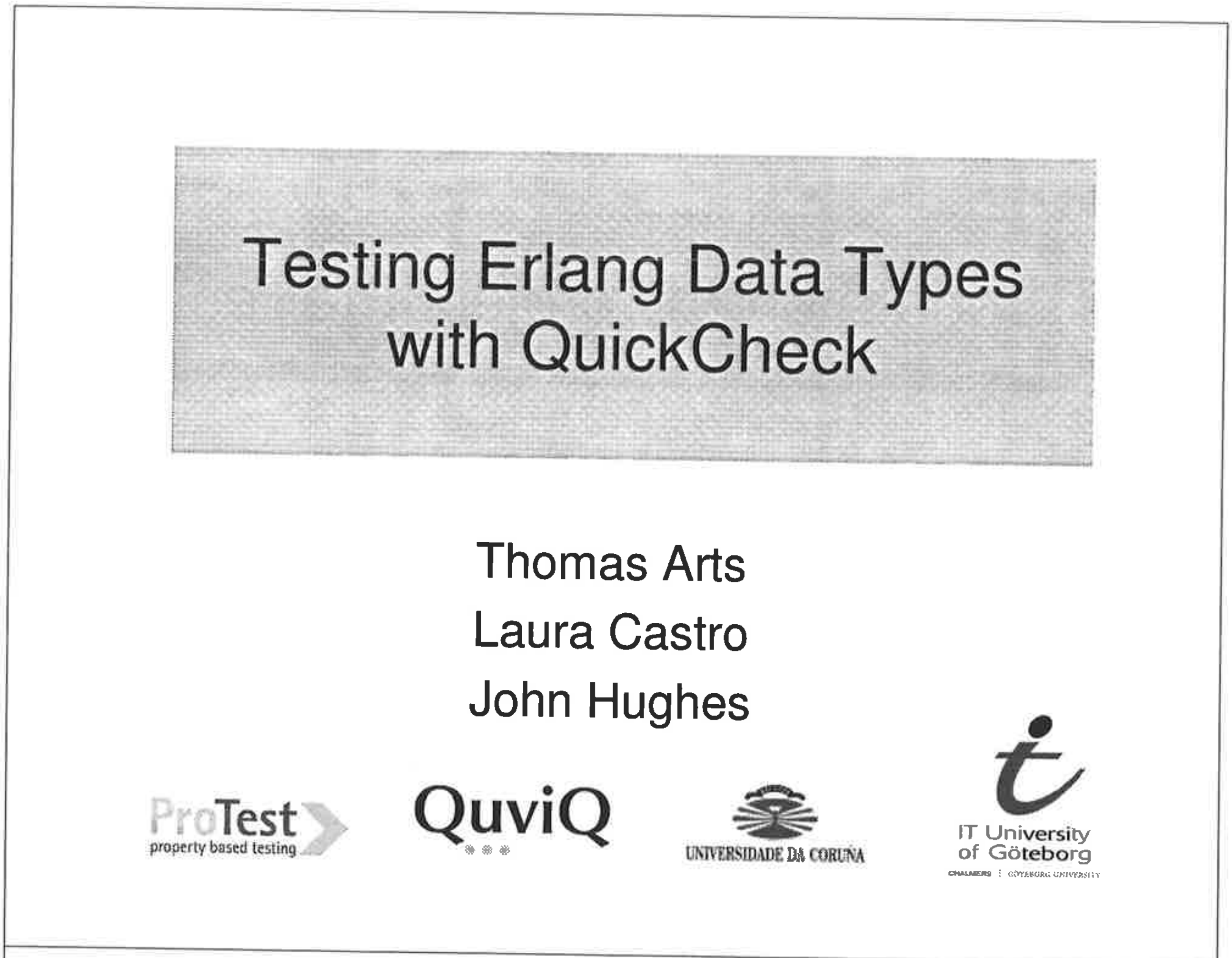
IT University of Göteborg

http://www.protest-project.eu/

Some results from Q1 + Q2

- Testing Erlang Data Types with QuickCheck
- Refactoring with Wrangler
- Early fault detection with model-based testing
- Erlang Testing and Tools Survey

http://www.protest-project.eu/



Testing Erlang Data Types with QuickCheck

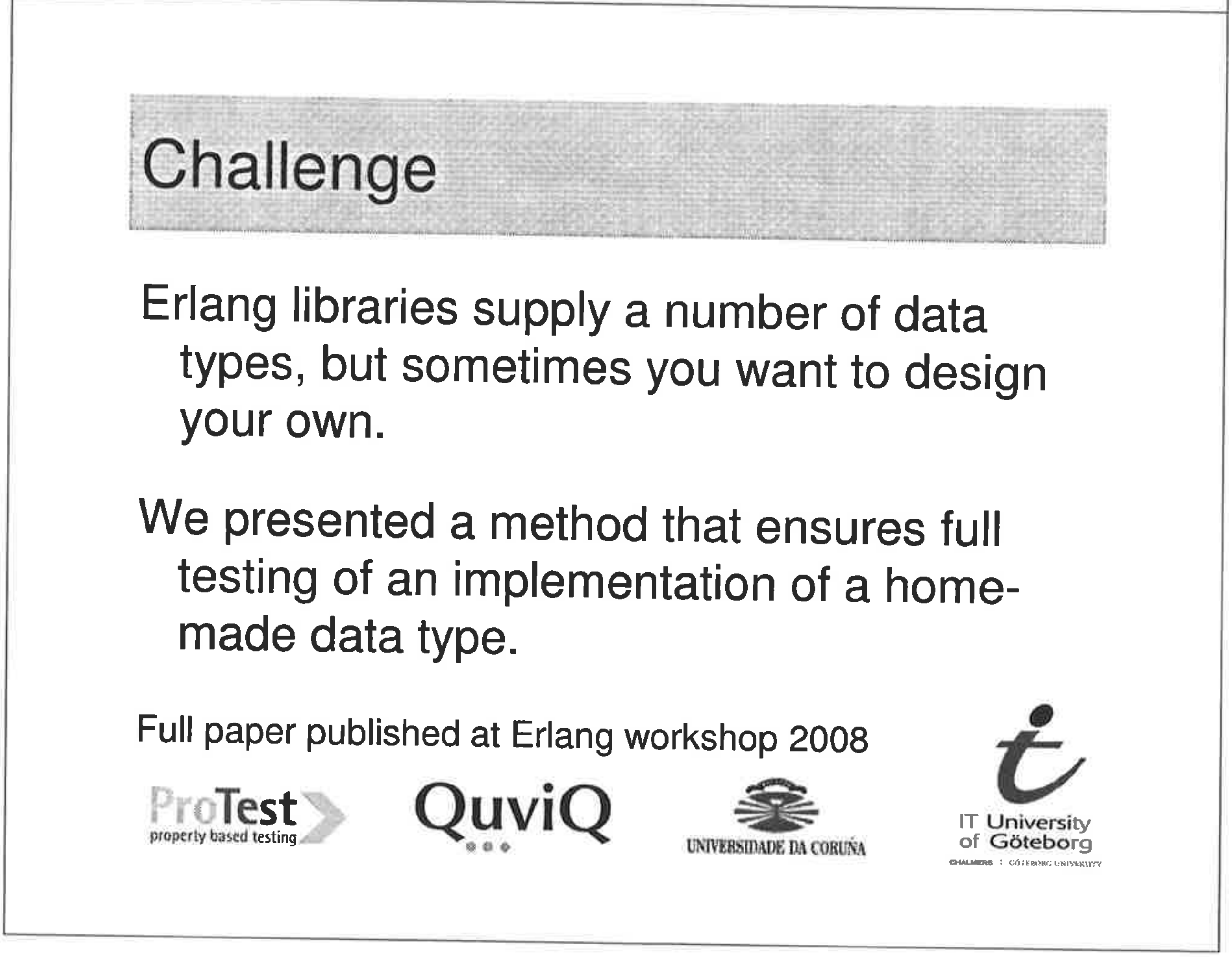
Thomas Arts
Laura Castro
John Hughes

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg
CHALMERS : GÖTEBORG UNIVERSITY



Challenge

Erlang libraries supply a number of data types, but sometimes you want to design your own.

We presented a method that ensures full testing of an implementation of a home-made data type.

Full paper published at Erlang workshop 2008

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg
CHALMERS : GÖTEBORG UNIVERSITY

Testing

Implementation of data type for decimals

How to test this implementation?

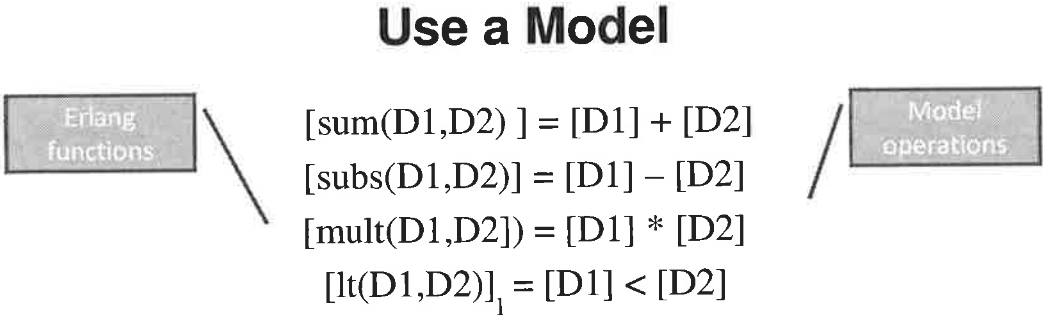
```
decimal() ->
  ?LET(Tuple, {int(),nat()}, new(Tuple)).

prop_sum_comm() ->
  ?FORALL({D1,D2}, {decimal(),decimal()},
          sum(D1,D2) == sum(D2,D1)).
```

QuickCheck generates thousands of tests

Testing

Which other properties do we add?
When do we have sufficiently many properties?

Use a Model

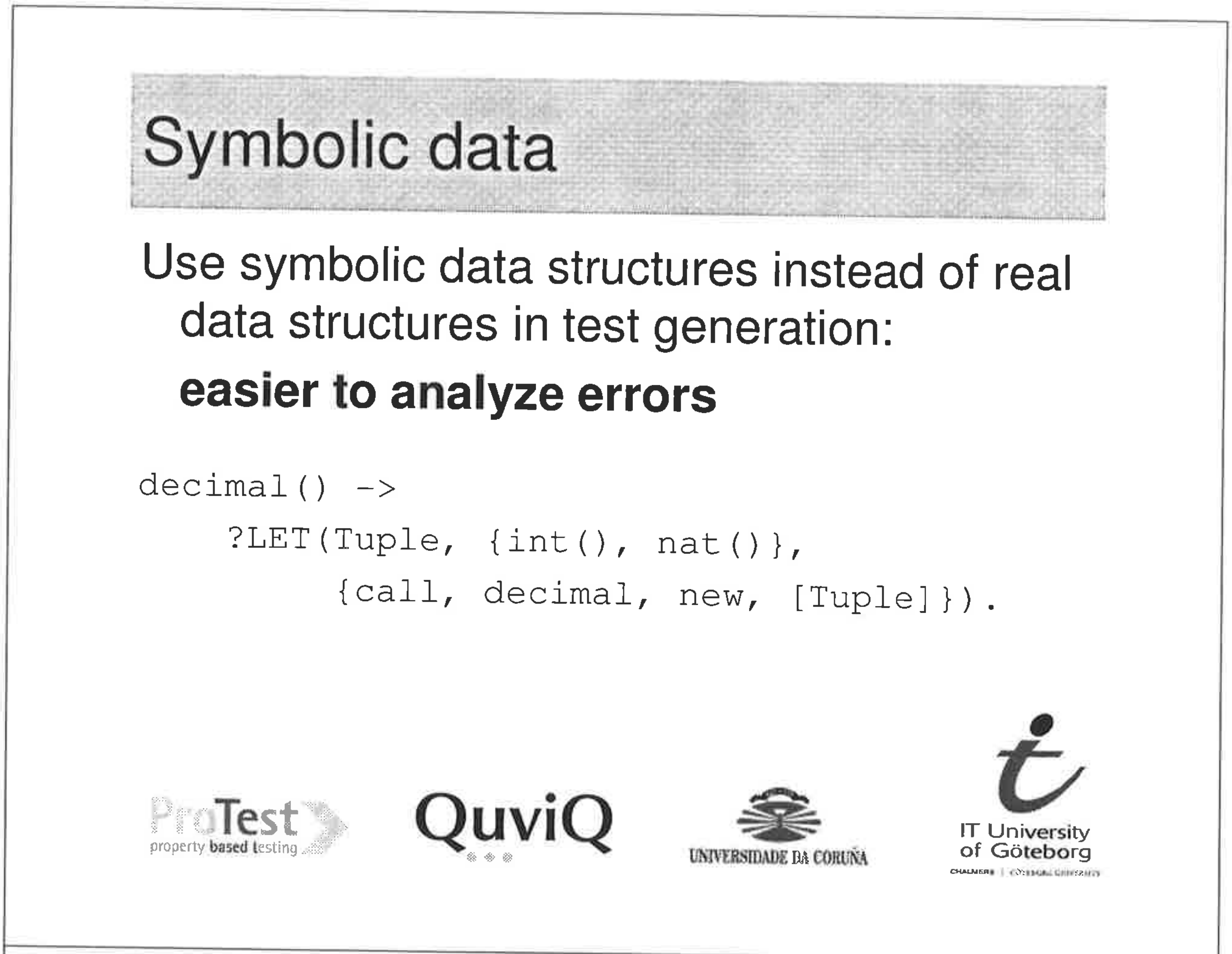


Erlang functions

$$[sum(D1,D2)] = [D1] + [D2]$$
$$[subs(D1,D2)] = [D1] - [D2]$$
$$[mult(D1,D2]) = [D1] * [D2]$$
$$[lt(D1,D2)]_1 = [D1] < [D2]$$

Model operations

.....



Symbolic data

Use symbolic data structures instead of real data structures in test generation:

easier to analyze errors

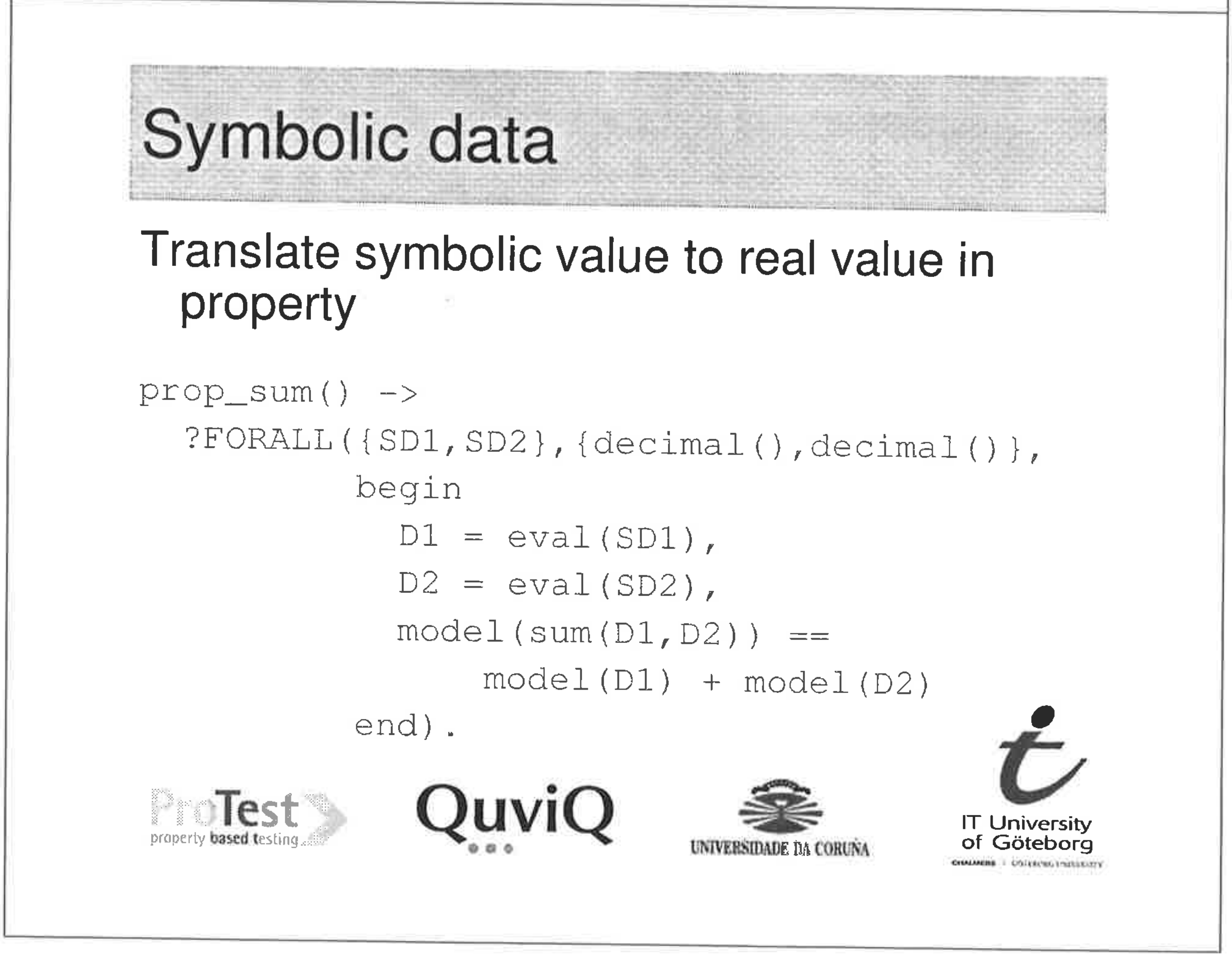
```
decimal() ->
    ?LET(Tuple, {int(), nat()},
         {call, decimal, new, [Tuple]}).
```

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg



Symbolic data

Translate symbolic value to real value in property

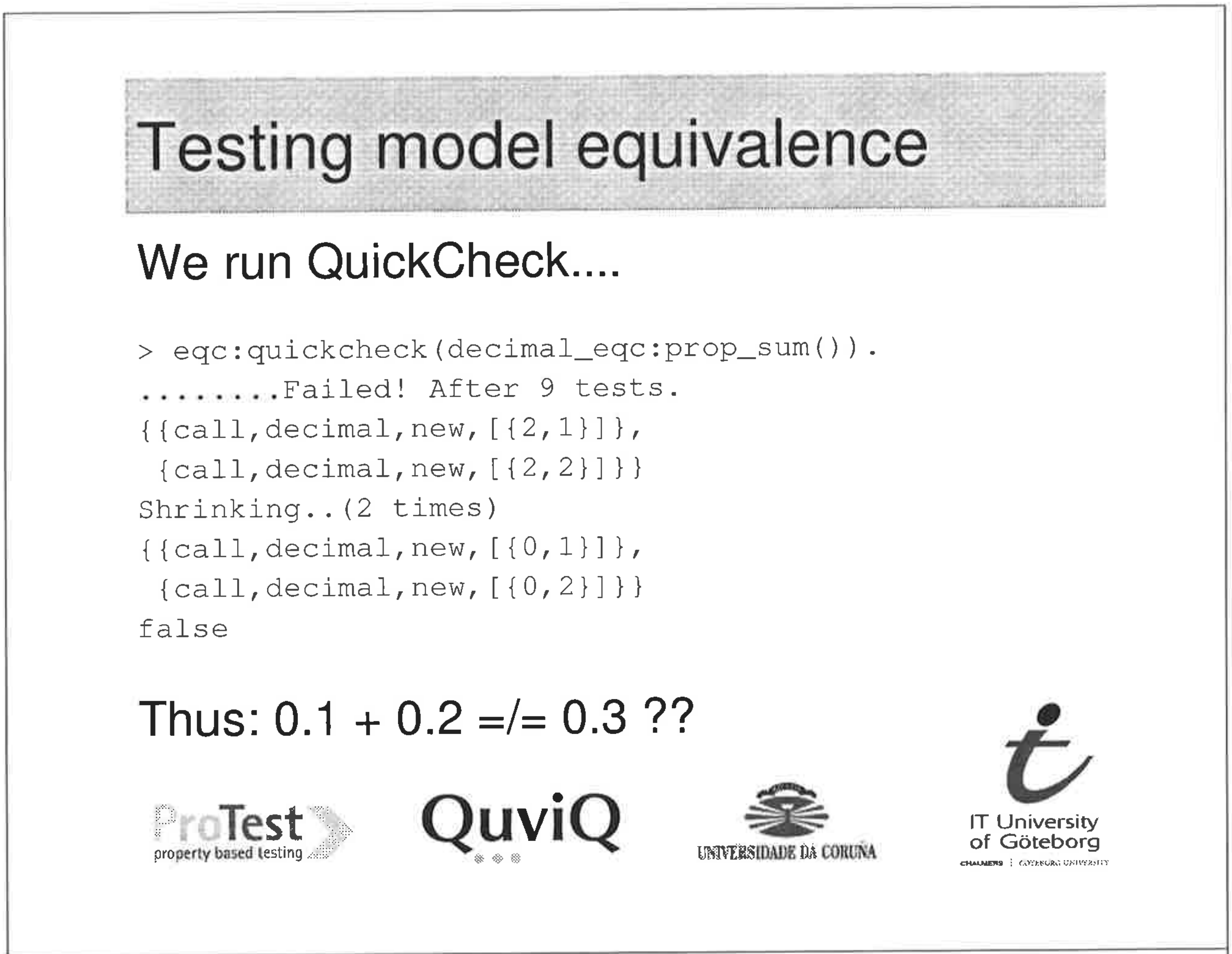
```
prop_sum() ->
  ?FORALL({SD1,SD2},{decimal(),decimal()},
         begin
           D1 = eval(SD1),
           D2 = eval(SD2),
           model(sum(D1,D2)) ==
               model(D1) + model(D2)
         end).
```

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg

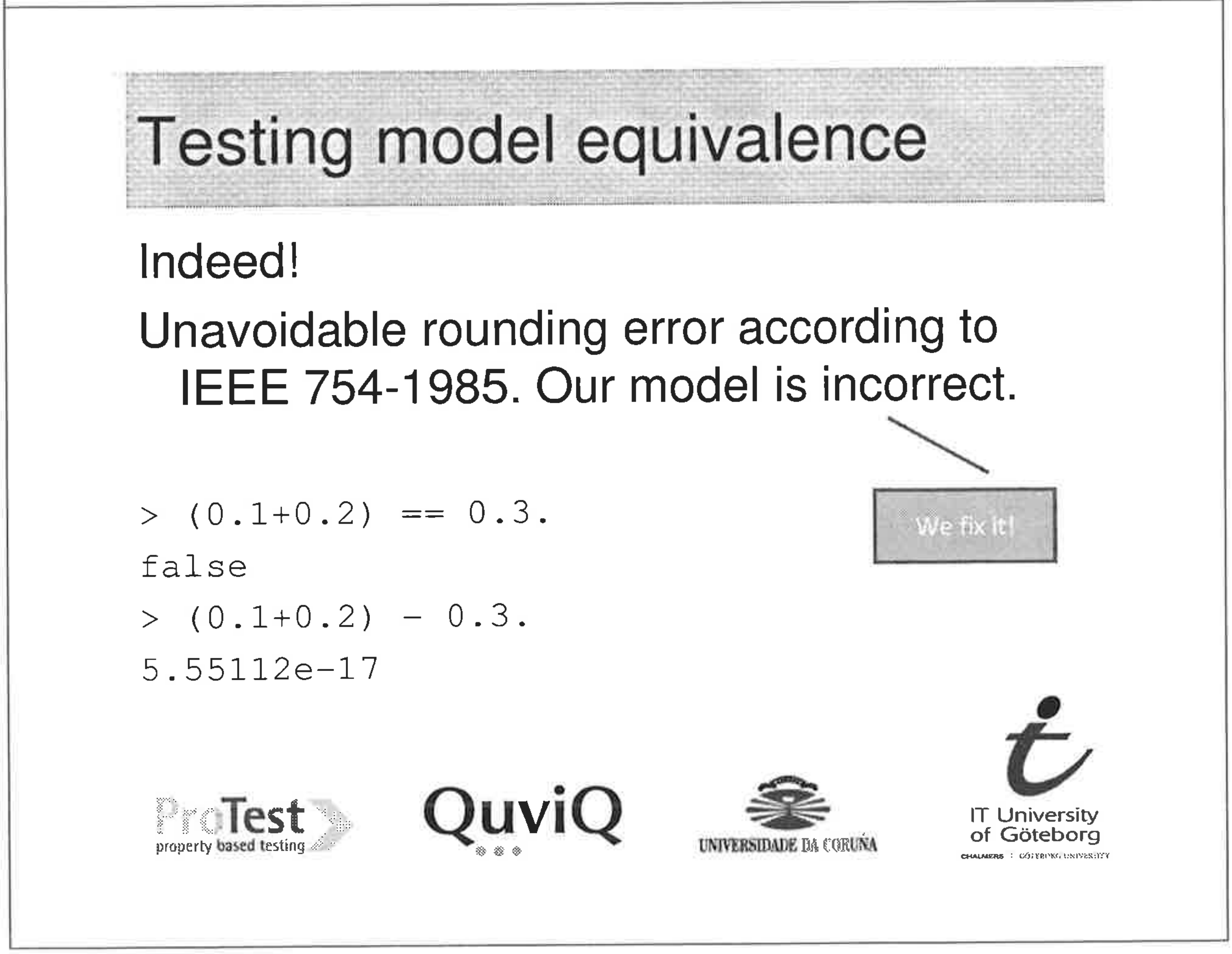


Testing model equivalence

We run QuickCheck....

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
........Failed! After 9 tests.
{{call,decimal,new,[{2,1}]},
 {call,decimal,new,[{2,2}]}}
Shrinking..(2 times)
{{call,decimal,new,[{0,1}]},
 {call,decimal,new,[{0,2}]}}
false
```

Thus: 0.1 + 0.2 =/= 0.3 ??



Testing model equivalence

Indeed!

Unavoidable rounding error according to IEEE 754-1985. Our model is incorrect.

We fix it!

```
> (0.1+0.2) == 0.3.
false
> (0.1+0.2) - 0.3.
5.55112e-17
```

Testing model equivalence

Property prop_sum() passes thousands of test cases.

Similarly, we can add a property prop_mult().

But... although we will obtain 100% code coverage, we miss testing combinations of mult and sum!

Recursive generators

```
decimal() ->
   ?SIZED(Size, decimal(Size)).

decimal(0) ->
   {call, decimal, new, [{int(),nat()}]};
decimal(Size) ->
   Smaller = decimal(Size div 2),
   oneof([
      decimal(0),
      ?LETSHRINK([D1, D2], [Smaller, Smaller],
                 {call, decimal, sum, [D1, D2]}),
      ?LETSHRINK([D1, D2], [Smaller, Smaller],
                 {call, decimal, mult, [D1, D2]}
   ]).
```

Testing model equivalence

Add subs and divs to generator and test **same property** again:

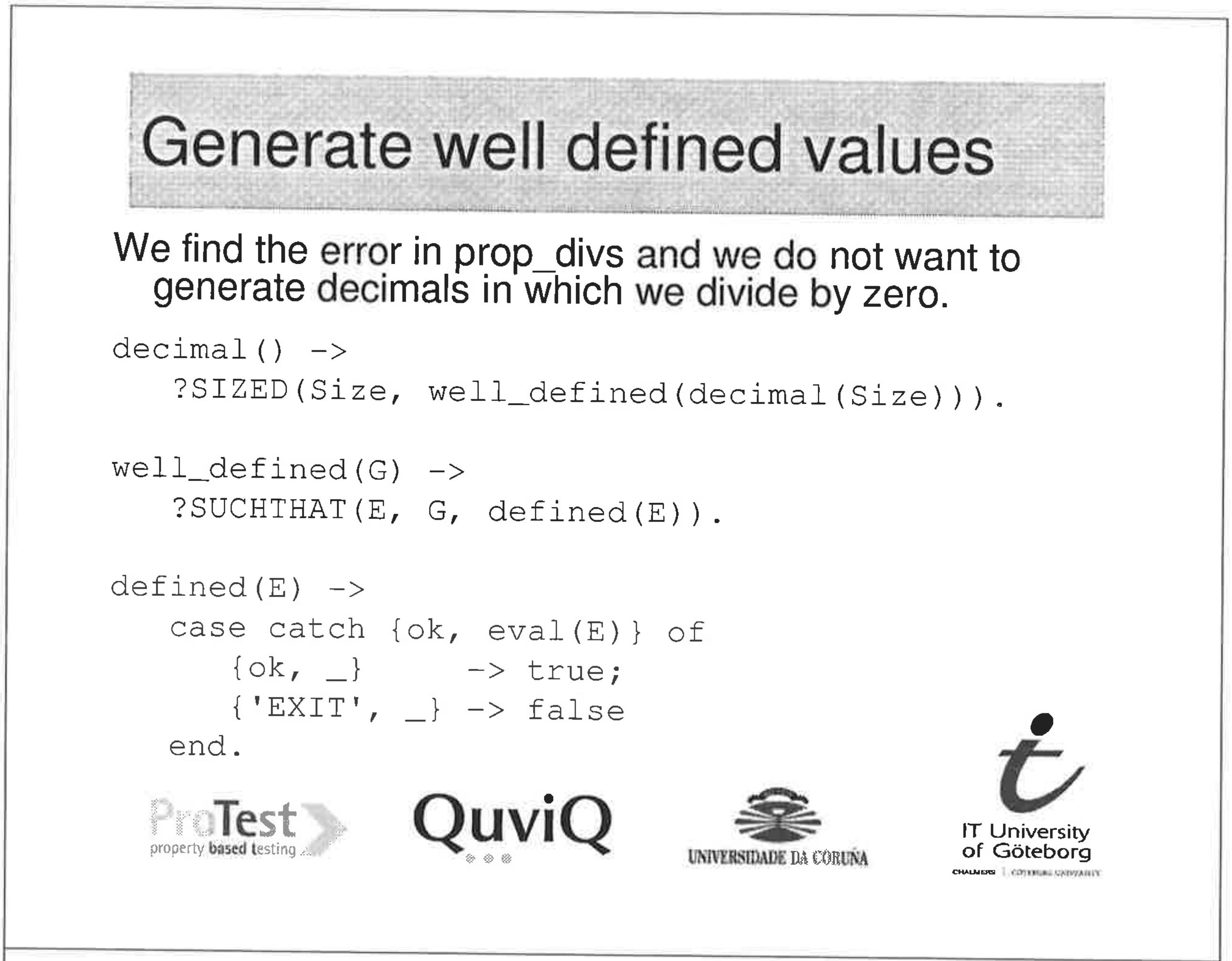
```
> eqc:quickcheck(decimal_eqc:prop_sum()).
............Failed!
After 13 tests.
Shrinking....(4 times)
Reason:
{'EXIT',{{not_ok,{error,decimal_error}},
        [...]}}
{{call,decimal,divs,
   [{call,decimal,new,[{0,0}]},
    {call,decimal,new,[{0,0}]}]},
 {call,decimal,new,[{0,0}]}}
false
```

division by zero

Negative testing

We do want to test that division by zero results in an error... *in prop_divs, not in prop_sum*

```
prop_divs() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
      begin
         D1 = eval(SD1),
         D2 = eval(SD2),
         case catch (model(D1)/model(D2)) of
            {'EXIT',_} ->
                      is_error(divs(D1, D2));
            Value ->
               equiv(model(divs(D1, D2)),
                     Value)
         end
      end).
```



Generate well defined values

We find the error in prop_divs and we do not want to generate decimals in which we divide by zero.

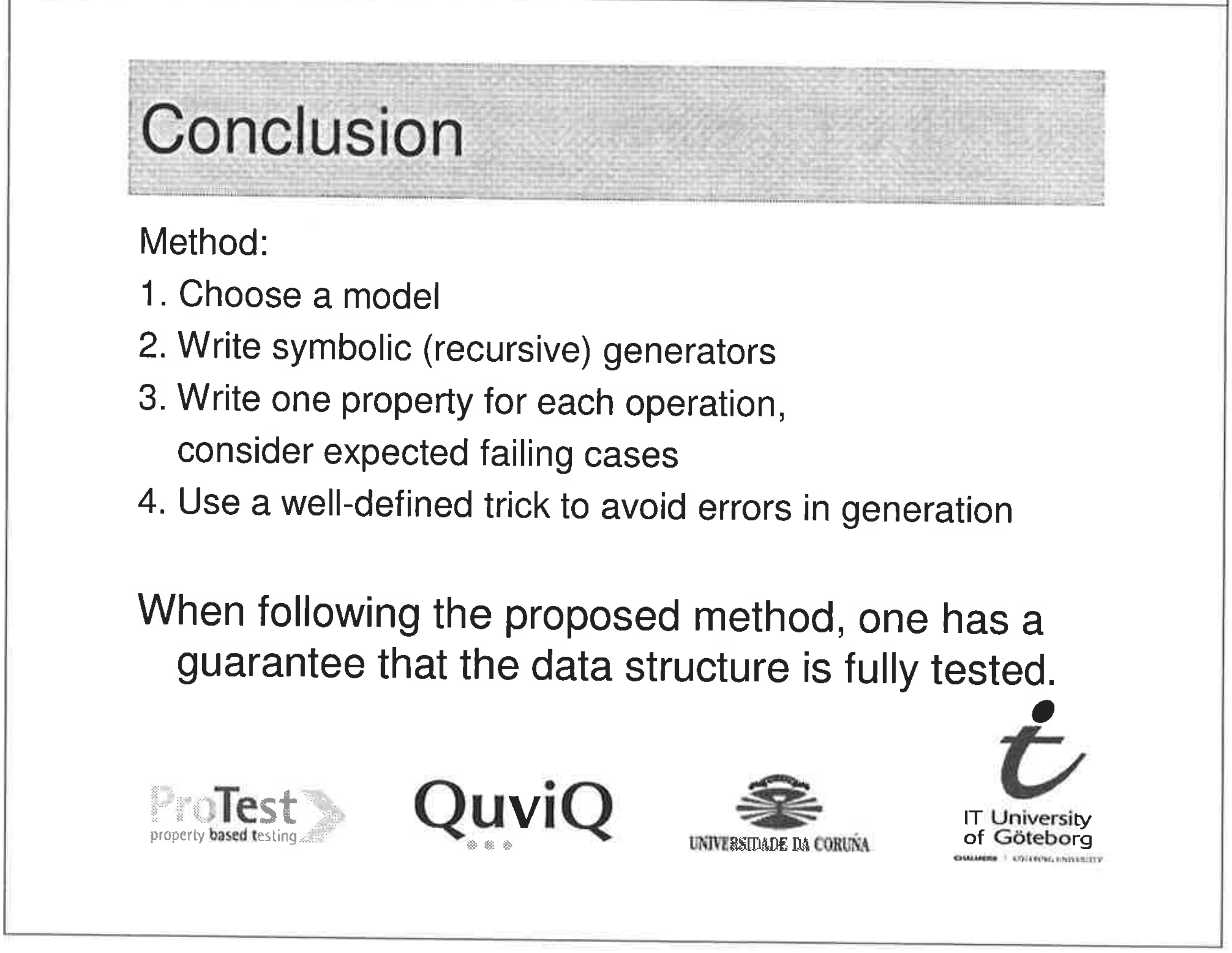
```
decimal() ->
   ?SIZED(Size, well_defined(decimal(Size))).

well_defined(G) ->
   ?SUCHTHAT(E, G, defined(E)).

defined(E) ->
   case catch {ok, eval(E)} of
      {ok, _}      -> true;
      {'EXIT', _} -> false
   end.
```

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg



Conclusion

Method:

1. Choose a model
2. Write symbolic (recursive) generators
3. Write one property for each operation, consider expected failing cases
4. Use a well-defined trick to avoid errors in generation

When following the proposed method, one has a guarantee that the data structure is fully tested.

ProTest property based testing

QuviQ

UNIVERSIDADE DA CORUÑA

IT University of Göteborg

Refactoring with Wrangler

Huiqing Li
Simon Thompson
University of Kent

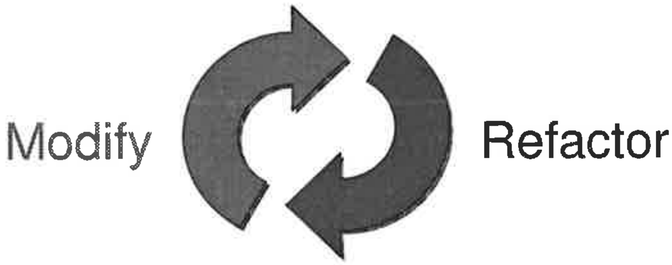
Melinda Tóth
George Orosz
Eötvos Loránd Univ

ProTest property based testing

University of Kent Computing

Refactoring

Refactoring means changing the design or structure of a program … without changing its behaviour.



Modify

Refactor

ProTest property based testing

University of Kent Computing

Generalisation and renaming

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

add_one ([H|T]) ->
  [H+1 | add_one(T)];

add_one ([]) -> [].

f(X) -> add_one(X).
```

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

add_int (N, [H|T]) ->
 [H+N | add_int(N,T)];

add_int (N,[]) -> [].

f(X) -> add_int(1, X).
```

ProTest
property based testing

University of Kent Computing

Generalisation

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

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


printList([1,2,3])
```

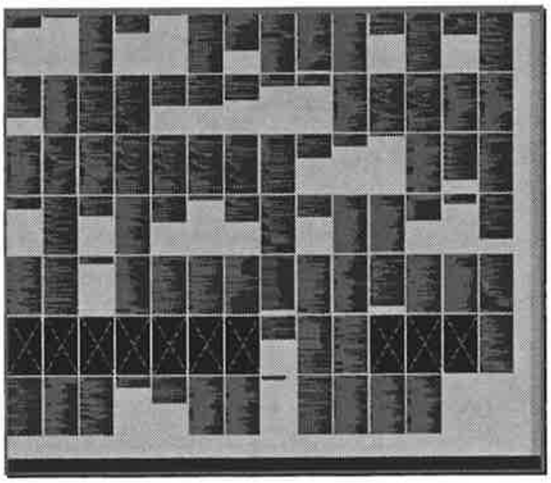
```
-export([printList/2]).

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


printList(
  fun(H) ->
    io:format("~p\n", [H])
  end,
  [1,2,3]).
```

ProTest
property based testing

University of Kent Computing

Refactoring tool support

- Bureaucratic and diffuse.
- Tedious and error prone.
- Semantics: scopes, types, modules, …
- Undo/redo
- Enhanced creativity



ProTest property based testing

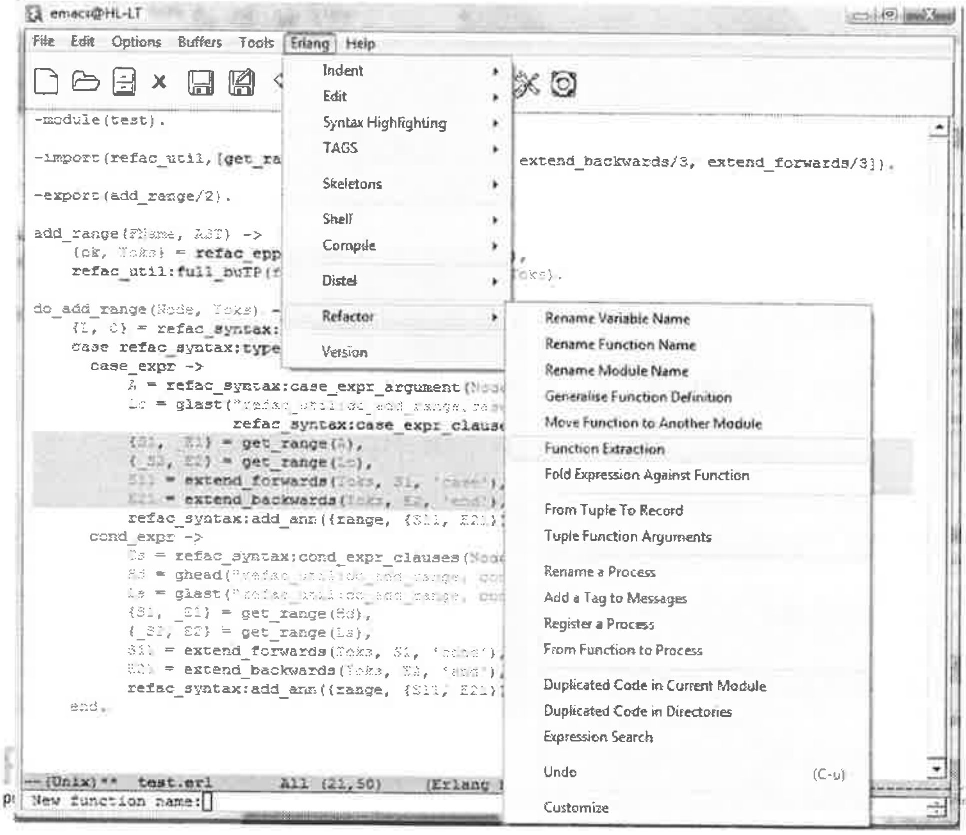
University of Kent Computing

Wrangler

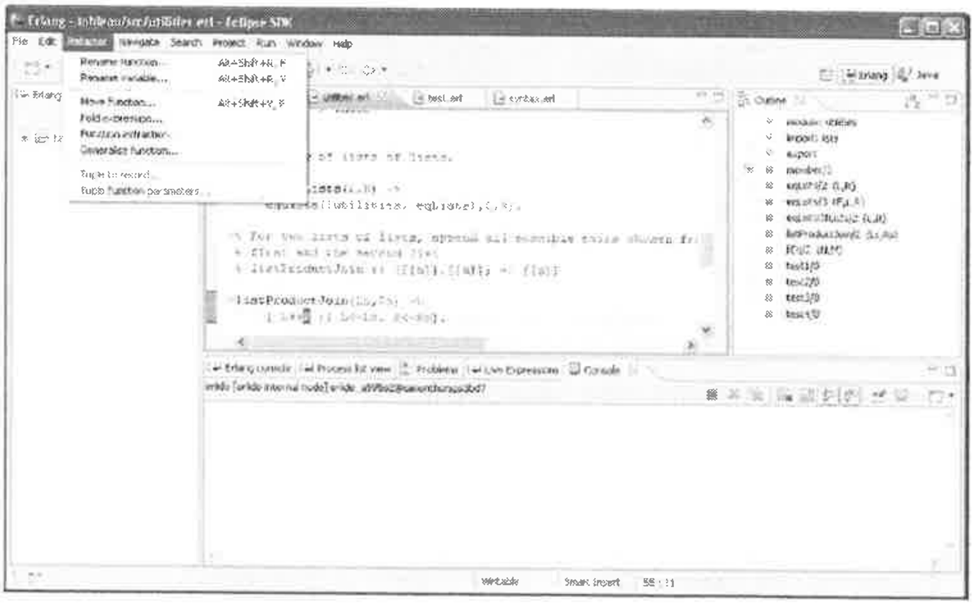
- Embedded in Emacs and Eclipse.
- Structural, data type and module refactorings.
- AAST-based analysis and transformation.
- Works with multiple modules.
- Supports undo of refactorings
- Preserves layout and comments as much as possible.
- Respects aspects of the macro system.

ProTest property based testing

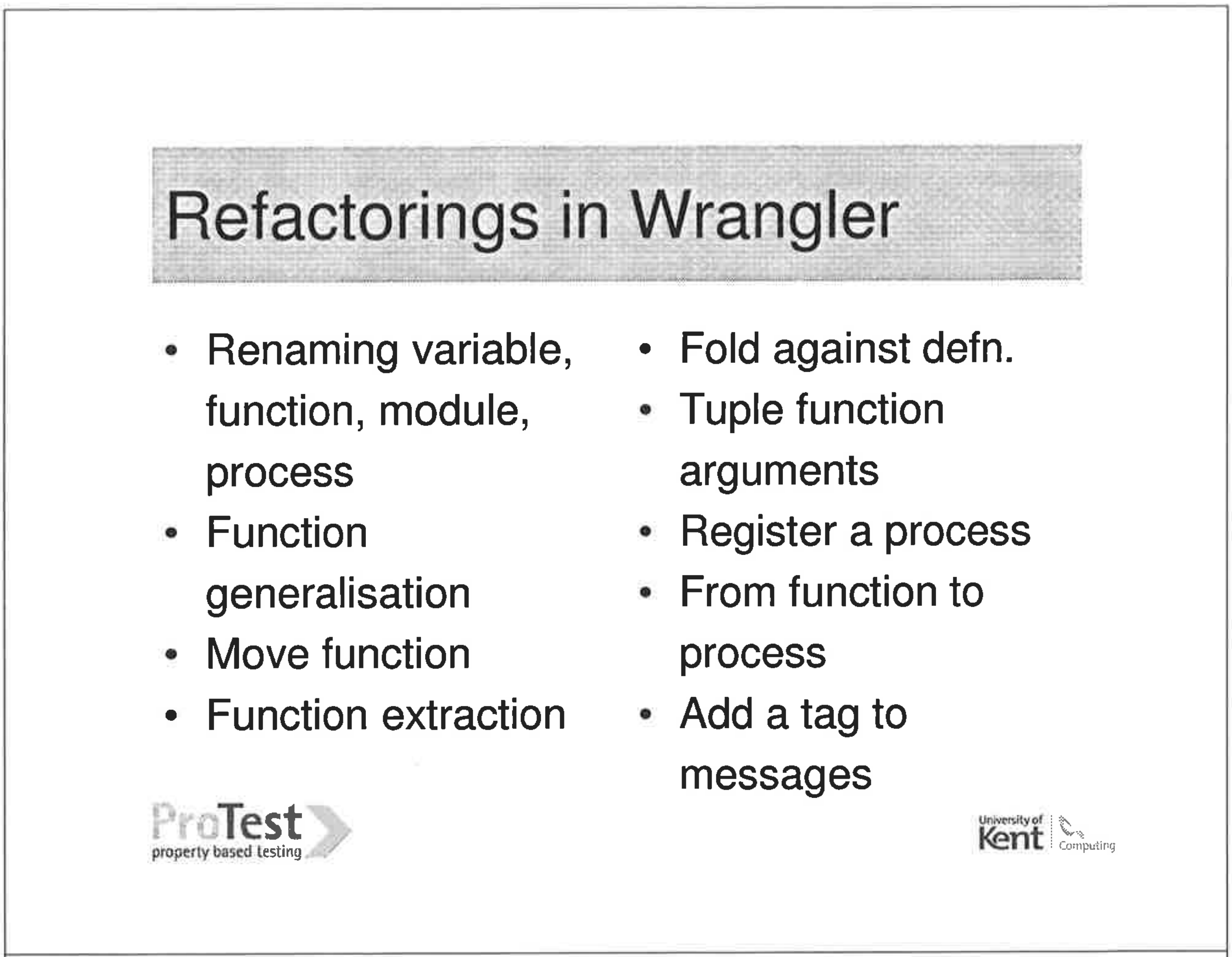
University of Kent Computing



```
-module(test).

-import(refac_util,[get_ra                         extend_backwards/3, extend_forwards/3]).

-export(add_range/2).

add_range(FName, AST) ->
    {ok, Toks} = refac_epp                         ,
    refac_util:full_buTP(f                         Toks).

do_add_range(Node, Toks) -
    {L, C} = refac_syntax:
    case refac_syntax:type
      case_expr ->
          A = refac_syntax:case_expr_argument(Node
          Lc = glast("refac_util:do_add_range,case
                     refac_syntax:case_expr_clause
          {S1, _E1} = get_range(A),
          {_S2, E2} = get_range(Lc),
          S11 = extend_forwards(Toks, S1, 'case'),
          E21 = extend_backwards(Toks, E2, 'end'),
          refac_syntax:add_ann({range, {S11, E21}
      cond_expr ->
          Cs = refac_syntax:cond_expr_clauses(Node
          Hd = ghead("refac_util:do_add_range, con
          Ls = glast("refac_util:do_add_range, con
          {S1, _E1} = get_range(Hd),
          {_S2, E2} = get_range(Ls),
          S11 = extend_forwards(Toks, S1, 'cond'),
          E21 = extend_backwards(Toks, E2, 'end'),
          refac_syntax:add_ann({range, {S11, E21}
    end.

--(Unix)**  test.erl      All (21,50)     (Erlang
New function name:
```



```
listProductJoin(Ls,Ts) ->
```

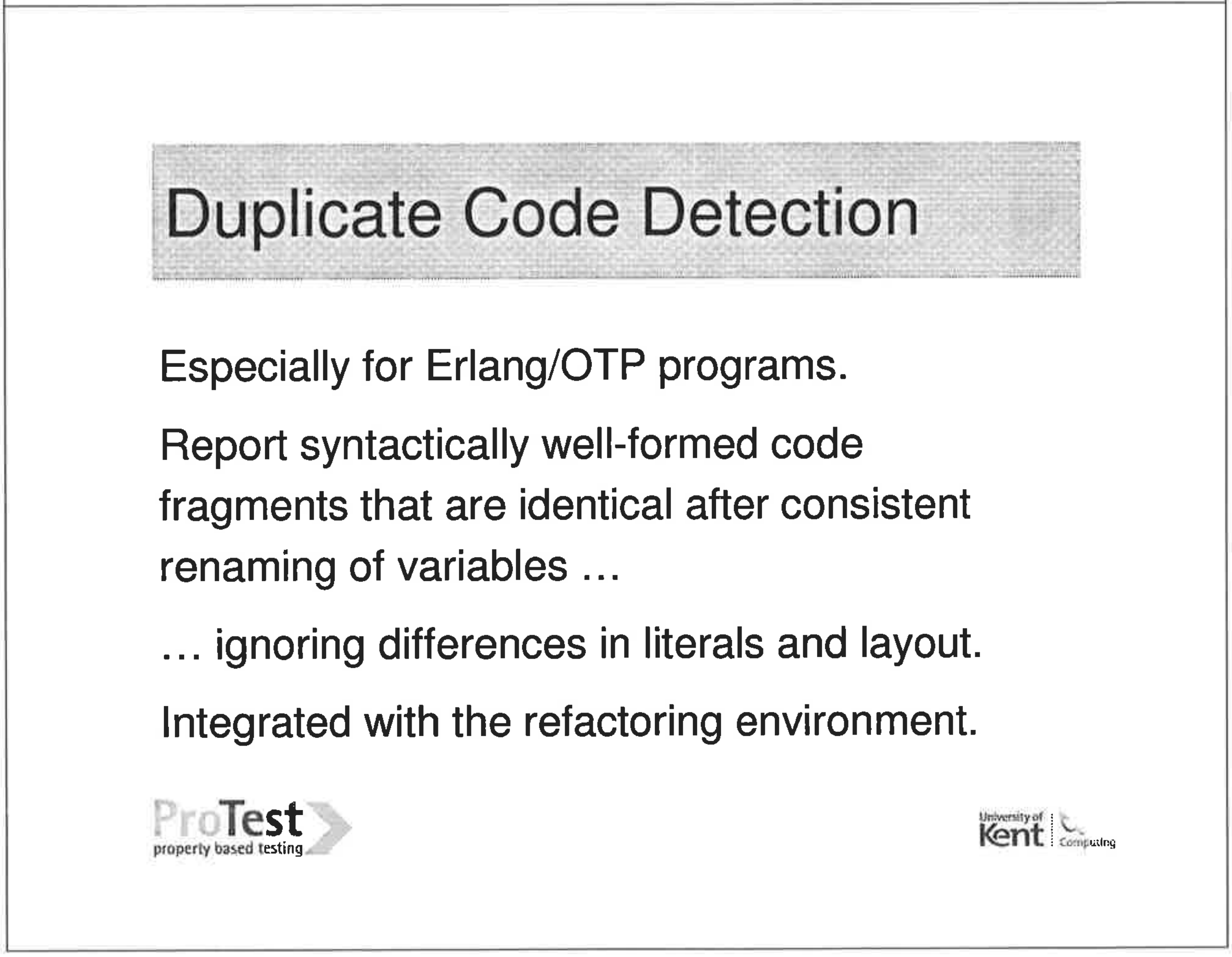


Refactorings in Wrangler

- Renaming variable, function, module, process
- Function generalisation
- Move function
- Function extraction
- Fold against defn.
- Tuple function arguments
- Register a process
- From function to process
- Add a tag to messages

ProTest property based testing

University of Kent Computing



Duplicate Code Detection

Especially for Erlang/OTP programs.

Report syntactically well-formed code fragments that are identical after consistent renaming of variables …

… ignoring differences in literals and layout.

Integrated with the refactoring environment.

ProTest property based testing

University of Kent Computing

Code Inspection Support

- Variable use/binding information.
- Caller functions.
- Caller/callee modules.
- Case/if/receive expressions nested more than a specified level.
- Long function/modules.
- Non tail-recursive servers.
- Non-flushed unknown messages
- . . .

Ongoing and Future work

- Continue the integration of Wrangler with Eclipse + Erlide
- More refactorings are being added including introduce macros, from module to process, etc.
- To investigate the use of trace information to help the refactoring process, especially process-related refactorings.

http://www.cs.kent.ac.uk/projects/forse/

ProTest property based testing

University of Kent Computing

Protest Survey on Erlang Testing Tools

Aniko Nagyné Víg
Tamás Nagy
Francesco Cesarini
Erlang Training and Consulting

ProTest property based testing

Erlang

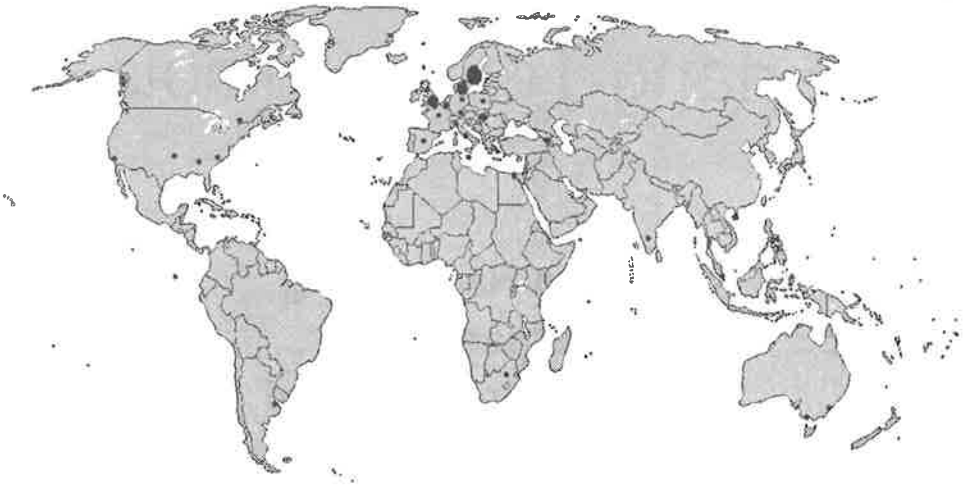
Research Method

- Published an online survey
- Advertised it by email:
  - Erlang Questions, approx. 1000 users
  - Erlang Training and Consulting Newsletter list, approx. 1000 users
  - Smaller Erlang related mailing lists
    - Trapexit User Group, 500 users
    - ProTest Mailing List, 50 users
    - London / Stockholm Erlang User Groups, 100 users
- 200 direct emails to relevant contacts at ETC
  - Merged with the main survey after the results were similar
- 40-45% of total(200) responses were from developers

ProTest
property based testing

Erlang

Geographical Diversion



ProTest
property based testing

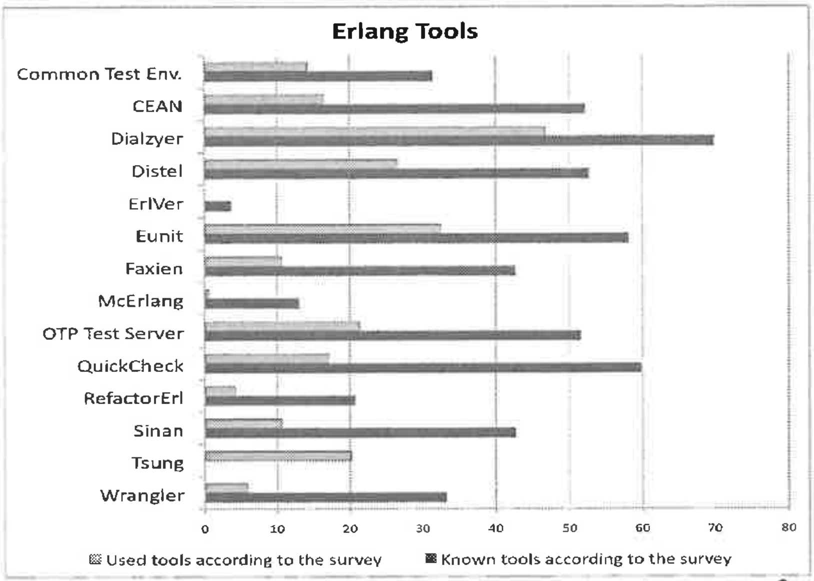
Erlang

Survey Structure

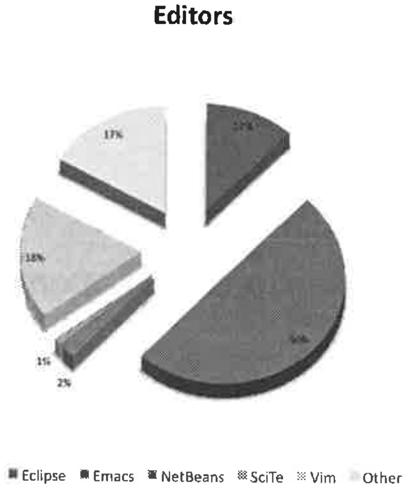
We asked 20 questions about

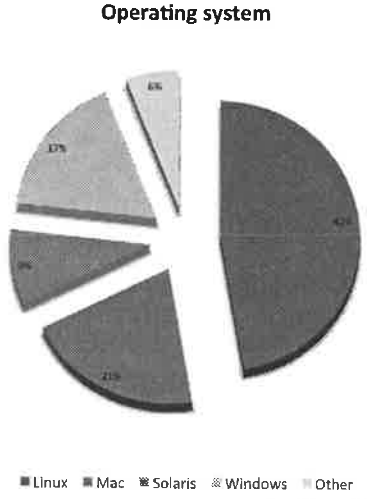
- The Erlang development environment
- Usage and Knowledge of existing tools and open source applications
- Submitter's job role and Erlang background
- Identify common processes to improve tools support

Erlang Tools Knowledge / Usage



Editors and Operating Systems





Problems Identified by the Survey

- Weaknesses of most Erlang tools and projects were found to be
  - Lack of documentation
  - Lack of examples and tutorials
  - Incomplete and untested tools
- Design issues included
  - Badly layered software
  - Not extensible and not structured
- Doubts about sustainability & support
- Hard to install and use
  - Especially for non Erlang users
  - Extensive manual configuration required

Missing Functionality

- No tools for stub generation
- Testing tools lack high quality results display
  - Web interface or dashbord
- Load testers are not available for all requirements
  - Especially state based protocols
- Continous integration
  - Hooks towards version control systems
  - Integrated into a general framework
- A complete framework that integrates different tools

ProTest
property based testing

Erlang

Conclusion

What are the key factors for building a successful Erlang tool?

- Reliable software
- User friendliness
- Good documentation
- Support
- Well promoted!!

ProTest
property based testing

Erlang

Further Reading

Paper from the ACM SIGPLAN Erlang Workshop and Complete Survey Results are available at

www.protest-project.eu/publications.html

ProTest property based testing

Erlang

Questions?

ProTest property based testing

ProTest property based testing

University of Kent Computing

Erlang

QuviQ

IT University of Göteborg

CHALMERS | GÖTEBORG UNIVERSITY

Subscribe to our RSS feeds and newsletter @ www.protest-project.eu

ProTest - Property Based Testing for Erlang - Mozilla Firefox

File Edit View History Bookmarks Tools Help
http://www.protest-project.eu/index.html
Most Visited Latest Headlines Erlang Consulting Slashdot
ProTest property based testing
Home Project Publications Tutorials Tools Partners
Done

In 2007 the ProTest Consortium was established with a financial contribution of 2.5 million Euros from the European Commission to help fund the project.

Project Goals

- develop software engineering approaches to improve reliability of service oriented networks
- support fault-finding and diagnosis based on specified properties of the system
- build automated tools that will generate and run tests, monitor execution at run time, and log events for analysis.

Consortium Partners

IT University of Göteborg
University of Kent
Universidad Politécnica de Madrid
University of Sheffield
Ericsson
Erlang Training and Consulting Ltd
LambdaStream
Quviq

ProTest News

5 September 2008: Erlang testing tools survey results
17 June 2008: Using QuickCheck to test Erlang programs tutorial
15 May 2008: Instruction videos for QuickCheck

More News

ProTest Events

[illegible]: QuickCheck course (Stockholm, Sweden)
[illegible]: Erlang/OTP User Conference, Stockholm, Sweden
[illegible] 2nd IEEE International

ProTest property based testing

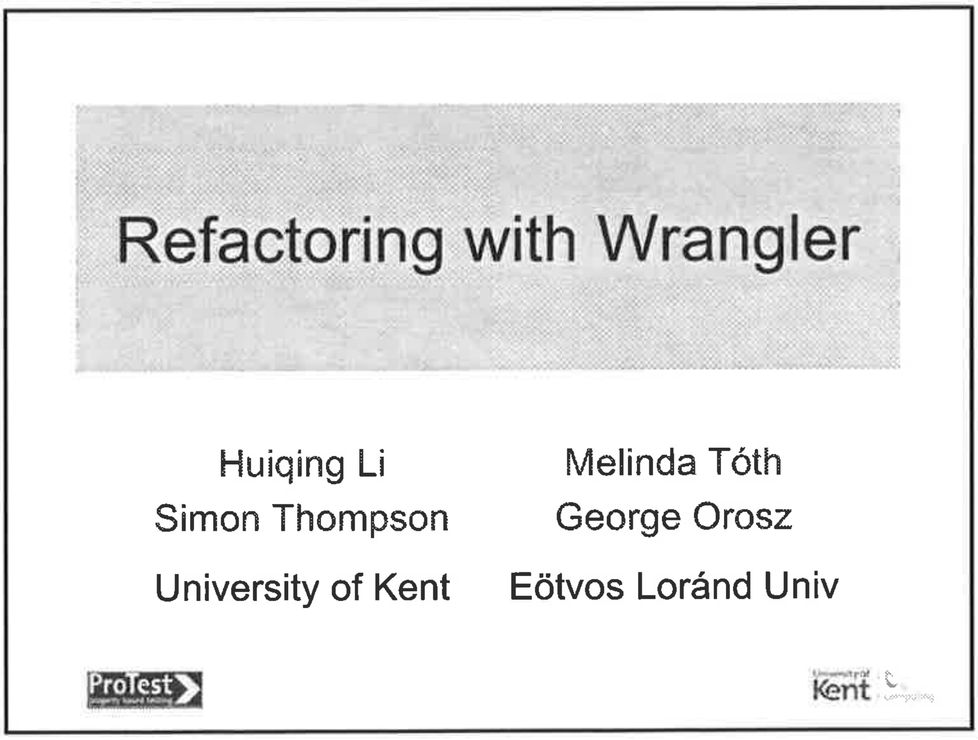
University of Kent Computing

Erlang

QuviQ

IT University of Göteborg

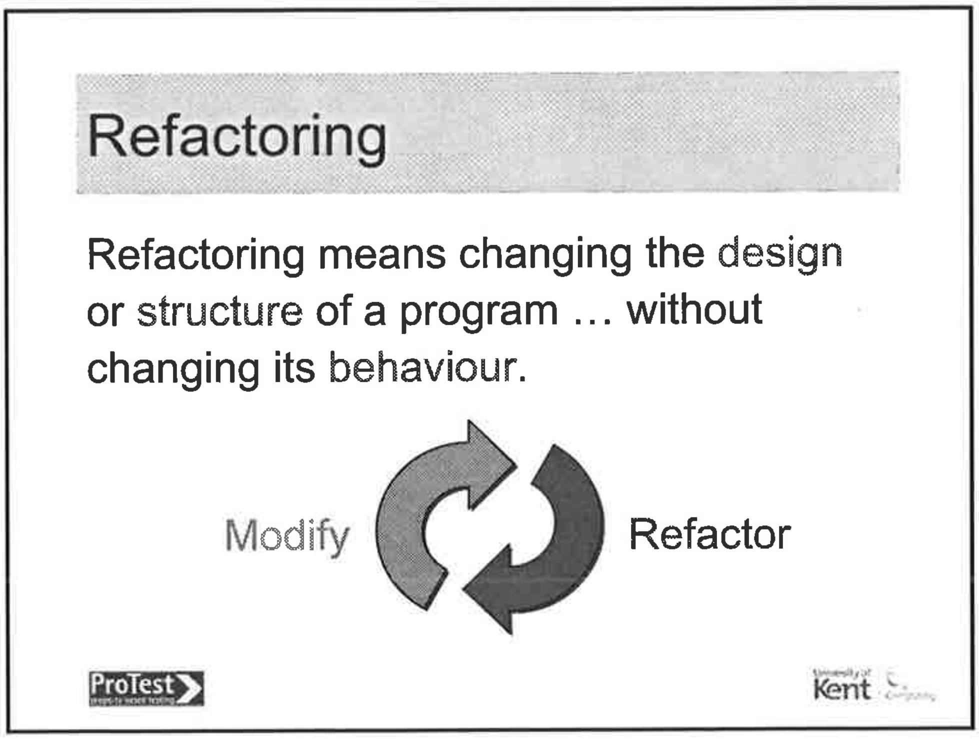
CHALMERS | GÖTEBORG UNIVERSITY



Refactoring with Wrangler

Huiqing Li
Simon Thompson
University of Kent

Melinda Tóth
George Orosz
Eötvos Loránd Univ

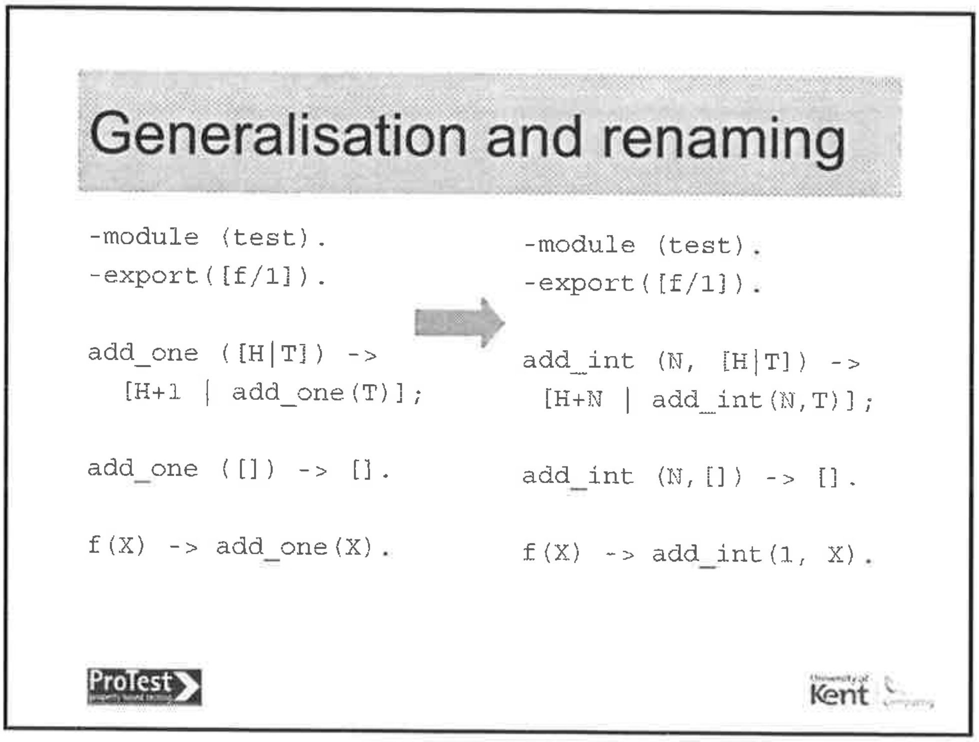


Refactoring

Refactoring means changing the design or structure of a program … without changing its behaviour.

Modify

Refactor



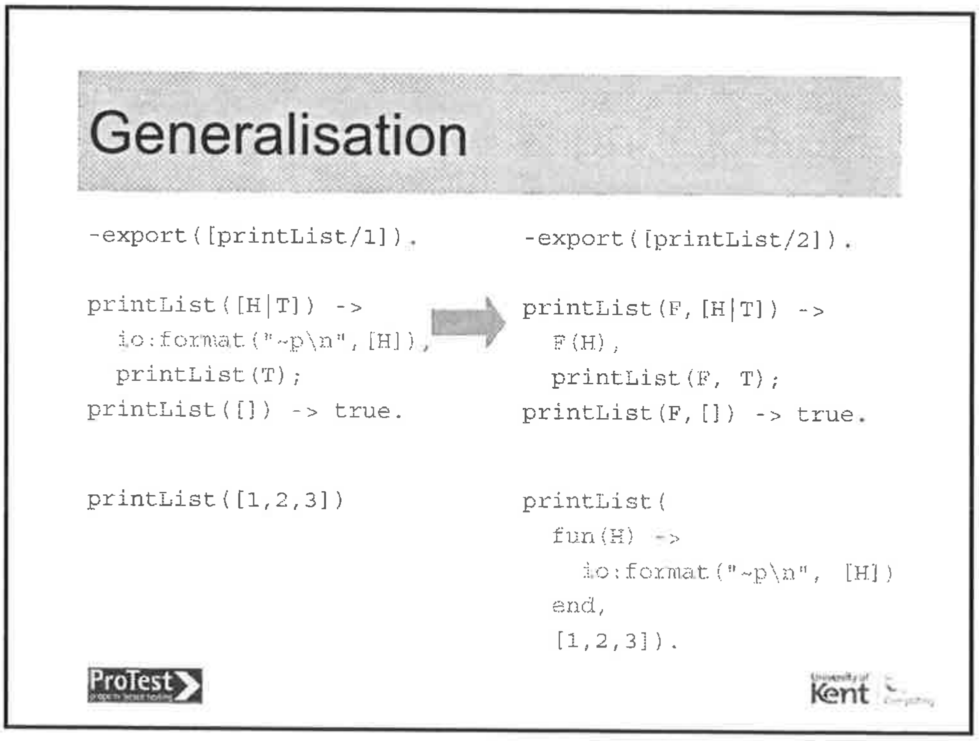
Generalisation and renaming

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

add_one ([H|T]) ->
  [H+1 | add_one(T)];

add_one ([]) -> [].

f(X) -> add_one(X).
```

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

add_int (N, [H|T]) ->
 [H+N | add_int(N,T)];

add_int (N,[]) -> [].

f(X) -> add_int(1, X).
```

ProTest property based testing

University of Kent Computing



Generalisation

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

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


printList([1,2,3])
```

```
-export([printList/2]).

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


printList(
  fun(H) ->
    io:format("~p\n", [H])
  end,
  [1,2,3]).
```

ProTest property based testing

University of Kent Computing

Refactoring tool support

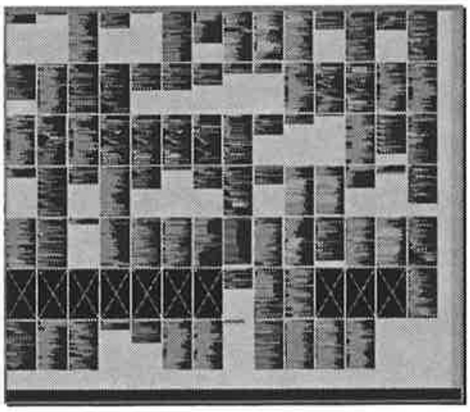
Bureaucratic and diffuse.

Tedious and error prone.

Semantics: scopes, types, modules, …

Undo/redo

Enhanced creativity



Wrangler

Embedded in Emacs and Eclipse.

Structural, data type and module refactorings.

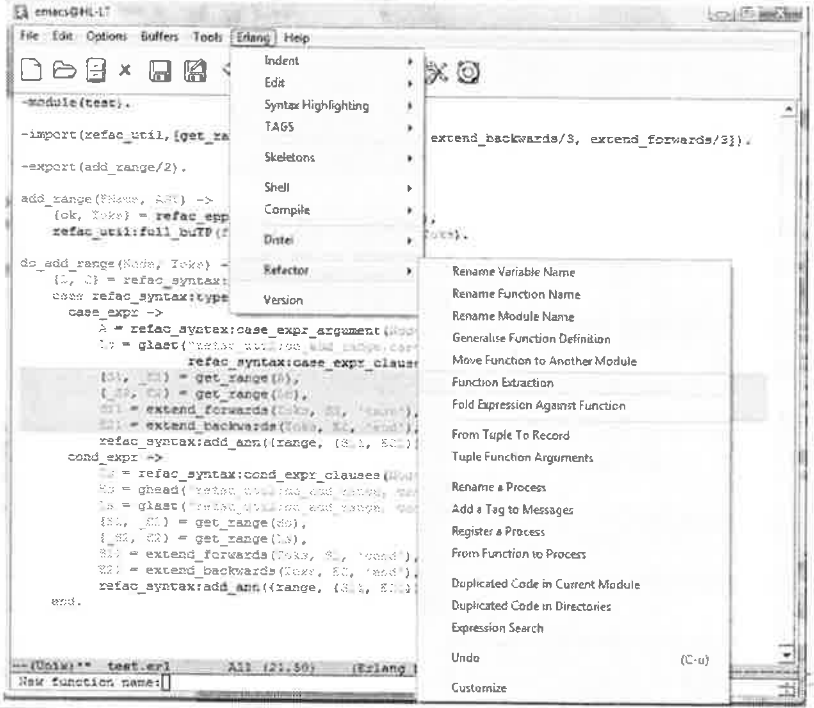
AAST-based analysis and transformation.

Works with multiple modules.

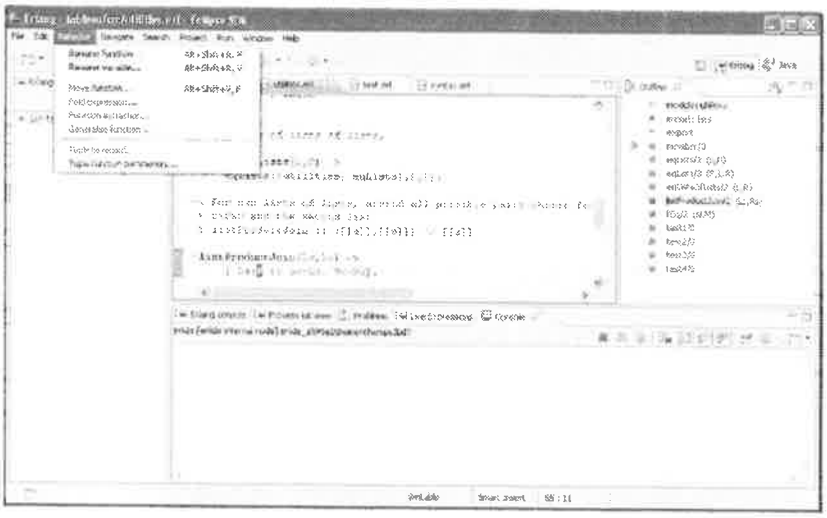
Supports undo of refactorings

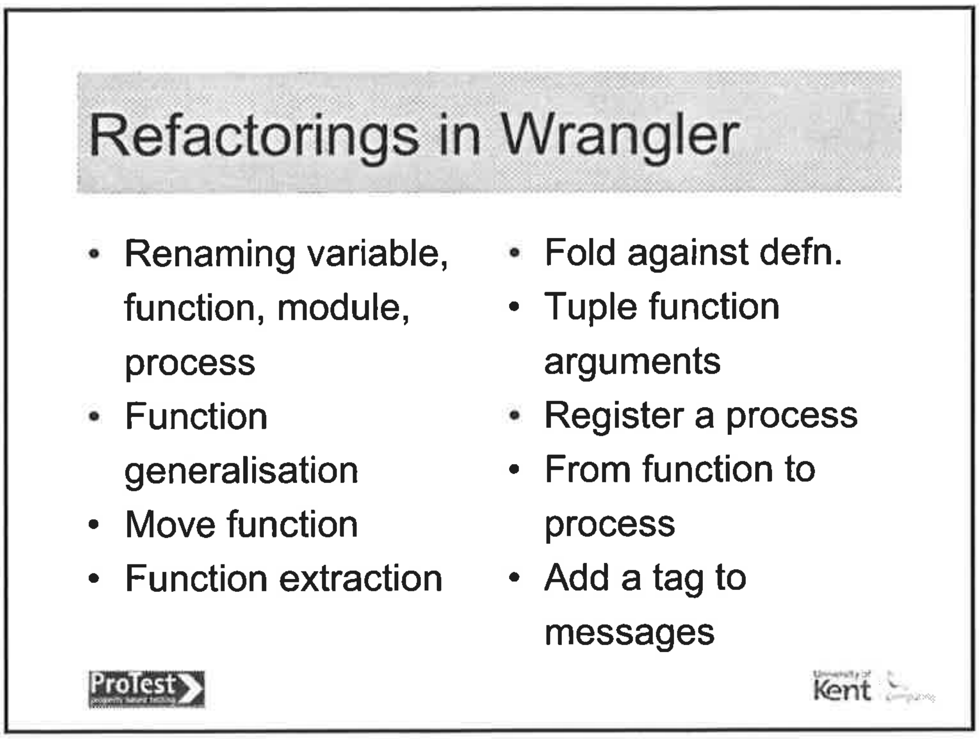
Preserves layout and comments as much as possible.

Respects aspects of the macro system.



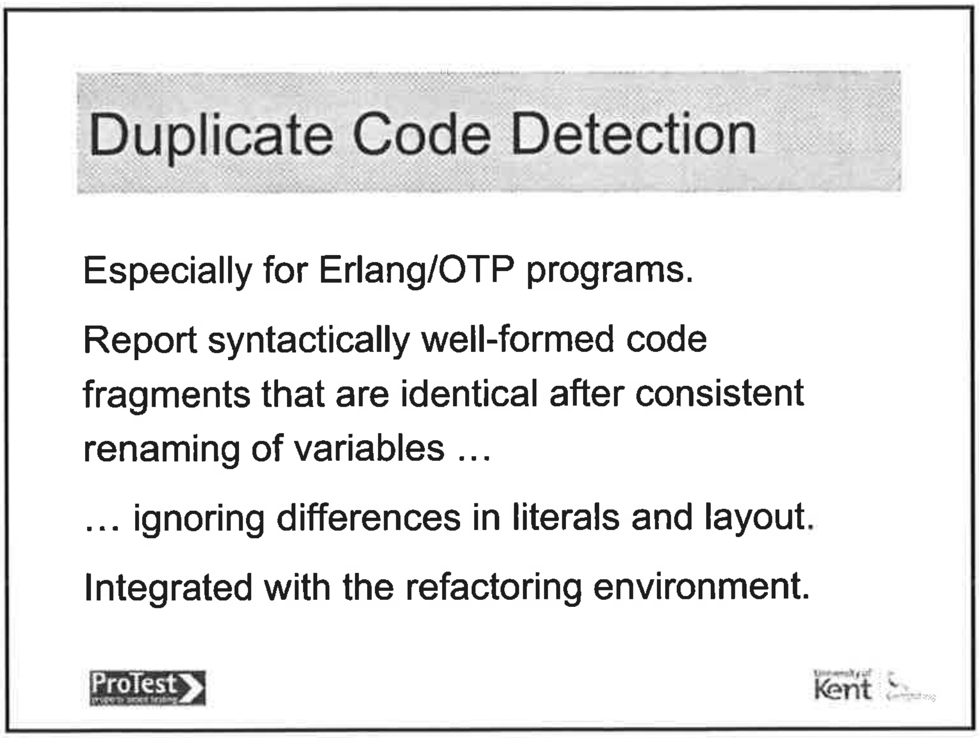
```
-module(test).

-import(refac_util,[get_ra                       extend_backwards/3, extend_forwards/3]).

-export(add_range/2).

add_range(FName, AST) ->
    {ok, Toks} = refac_epp                       ),
    refac_util:full_buTP(f                       Toks).

do_add_range(Node, Toks) ->
    {L, C} = refac_syntax:
    case refac_syntax:type
      case_expr ->
          A = refac_syntax:case_expr_argument(Nod
          Ls = glast("refac_util:do_add_range,cas
                      refac_syntax:case_expr_clause
          {S1, _E1} = get_range(A),
          {_S2, E2} = get_range(Ls),
          S11 = extend_forwards(Toks, S1, 'case'),
          E21 = extend_backwards(Toks, E2, 'end'),
          refac_syntax:add_ann({range, {S11, E21}
      cond_expr ->
          Cs = refac_syntax:cond_expr_clauses(Nod
          Hs = ghead("refac_util:do_add_range, Cs
          Ls = glast("refac_util:do_add_range, Cs
          {S1, _E1} = get_range(Hs),
          {_S2, E2} = get_range(Ls),
          S11 = extend_forwards(Toks, S1, 'cond'),
          E21 = extend_backwards(Toks, E2, 'end'),
          refac_syntax:add_ann({range, {S11, E21}
    end.

--(Unix)**  test.erl        All (21,50)     (Erlang
New function name:
```





Refactorings in Wrangler

- Renaming variable, function, module, process
- Function generalisation
- Move function
- Function extraction
- Fold against defn.
- Tuple function arguments
- Register a process
- From function to process
- Add a tag to messages



Duplicate Code Detection

Especially for Erlang/OTP programs.

Report syntactically well-formed code fragments that are identical after consistent renaming of variables …

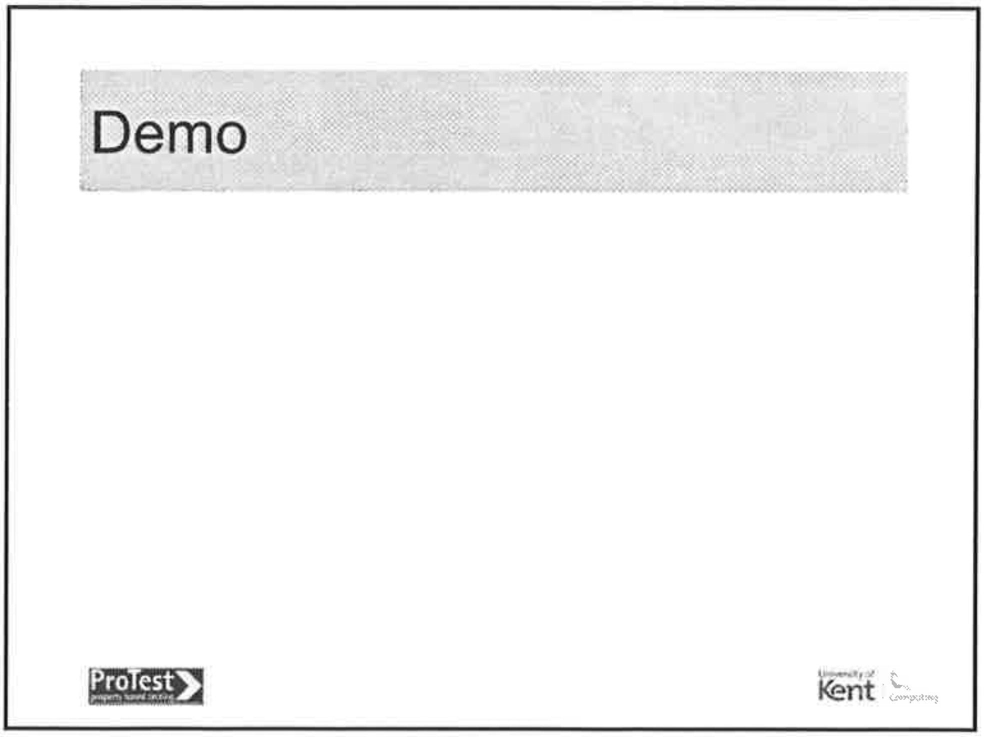
… ignoring differences in literals and layout.

Integrated with the refactoring environment.

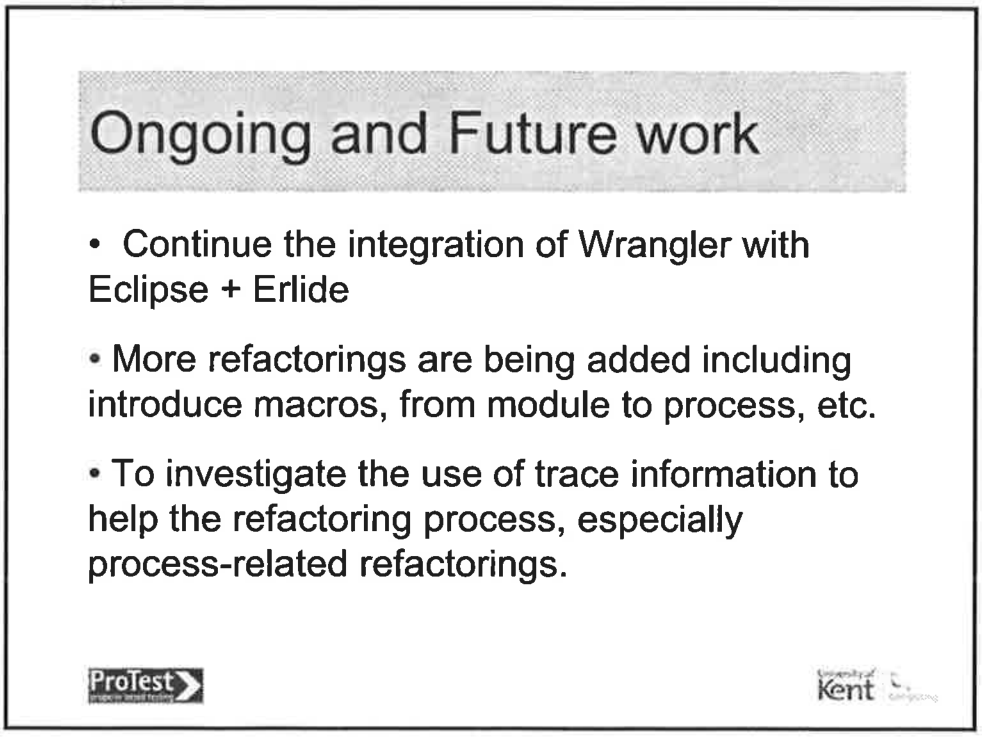
Demo

Code Inspection Support

- Variable use/binding information.
- Caller functions.
- Caller/callee modules.
- Case/if/receive expressions nested more than a specified level.
- Long function/modules.
- Non tail-recursive servers.
- Non-flushed unknown messages
- . . .



Demo



Ongoing and Future work

- Continue the integration of Wrangler with Eclipse + Erlide
- More refactorings are being added including introduce macros, from module to process, etc.
- To investigate the use of trace information to help the refactoring process, especially process-related refactorings.

http://www.cs.kent.ac.uk/projects/forse/

ProTest
property based testing
University of Kent
Computing

Automated syntax manipulation in RefactorErl*

Róbert Kitlei László Lövei Melinda Tóth
Zoltán Horváth Tamás Kozsik Roland Király
István Bozó Csaba Hoch Dániel Horpácsi

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
e-mail: {hz, kto, kitlei, lovei,kiralyroland,bozo_i,hoch,toth_m,daniel_h}@inf.elte.hu

Abstract

Refactorings often have to change the source code by adding, changing or replacing parts of the syntax tree. It is important to make these changes convenient and secure for the developer of refactorings.

In this paper, we introduce a method that helps us create, replace and insert syntactically correct subtrees in an Erlang refactoring tool.

1 Introduction

Refactoring is the systematic changing of source code while retaining the semantics of the code. Some refactorings, e.g. renaming a variable or a function, do not change the shape of the syntax tree, only update the information in the nodes, while most of the other refactorings construct, delete, move, insert or replace subtrees in the syntax tree. Deletion does not pose a problem, and moving a subtree is equivalent to its removal and reinsertion, therefore the most intriguing questions of the above are the construction, insertion and replacement of subtrees.

In addition to the above, refactorings have to gather additional information about semantic aspects of the source code as well. Since these bits of information can only be collected by visiting diverse parts of a syntax tree, another representation design may prove more efficient. A novel graph representation is proposed by the Erlang refactoring group at Eötvös Loránd University (ELTE, Budapest, Hungary). The ELTE group proposed this representation after previous experience with refactoring [5, 8]. Details about the representation and the refactoring tool RefactorErl are found in [4].

The structure of this paper is as follows. Section 2 and 3 describe methods that facilitate the creation, insertion and replacement of subtrees.

*Supported by ELTE IKKK and Ericsson Hungary.

Section 4 describes our experience with the above methods through a case study.

2 Subtree construction

One possible solution for constructing AST subtrees would be to use the parser itself by providing the source code that corresponds to that of the desired subtree. This approach would require the user to manually fill in all the punctuation, and would require separate grammars for each nonterminal to be generated. Another possibility is to do all the construction by hand, which is tedious and error-prone. Here we present another alternative.

The user has to supply two pieces of information for the node creation algorithm. One is the contents of the newly created node, which also contains its node type. By supplying the type of the node, the relevant rule structure can be selected from the grammar. The other piece of information required is a description of the desired contents of the node.

Since keywords and separator tokens can be automatically generated, and the skeleton determines the structure of the nodes, these tokens are not included in the description. All other tokens (e.g. variable names or function names) and all symbols have to be listed in order.

The algorithm processes the rule description and the content description. If the rule prescribes an automatically created token, it is created; otherwise, one or more elements supplied by the user are consumed when creating the next symbol or construct from the rule description.

Subtrees can be created by repeated use of the above algorithm. When constructing a new node, previously created nodes can be used as well as nodes that were already present in the graph.

3 Subtree replacement or insertion

Node replacement is done in a similar way to that of node construction. As parameters, the new nodes to be inserted and the place of insertion or replacement has to be specified. One of the following can be specified.

- Replacement of all or part of nodes of the same type.
- Replacement of a range of nodes.
- Insertion before or after a node.

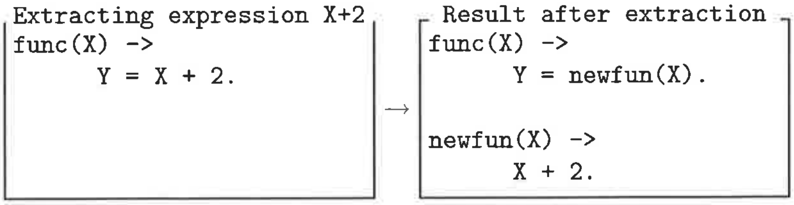
The insertion-replacement algorithm scans the grammar description, the given description and the actual structure. It determines the affected part in the syntax tree, makes the change and controls whether the resulting structure conforms to the grammar description.

All of the algorithms described above use an automatically generated scanner to check whether the tokens given in the descriptions are valid.

4 Case study: extract function

Refactorings usually consist of three different parts: collection of necessary information, checking preconditions and performing the transformation. The first and second part are used to reject the refactoring if its preconditions are not met, and the requested actions would change the behaviour of the code. The first and second part are necessary to guarantee the semantic consistency and preserve the externally observable behaviour. When performing the transformation, new code parts are created from the previously collected data. The new code parts strongly depend on the representation of the source, and have to be syntactically valid. Most of the time, the parts to be created and inserted cannot be composed only of parts that were present in the original code.

The "extract function" transformation extracts an expression or a sequence of expressions to a new function. Its parameters are the name of the containing file, the selection that will be extracted and the name of the new function. The transformation creates a new function definition, and replaces the selection with a function application. The variables which are used inside but bound outside the selection become the formal parameters of the new function. (See in Figure: 1).



```
Extracting expression X+2
func(X) ->
      Y = X + 2.
```

```
Result after extraction
func(X) ->
      Y = newfun(X).

newfun(X) ->
      X + 2.
```

Figure 1: Extracting expression X+2 into function newfun.

The created function definition (See in Figure: 3) contains opening and closing parentheses, an arrow and a stop token as punctuation, and the function name, the parameters of the function and the expressions of the body of the function. Also, if multiple parameters or body expressions are present, additional separating comma tokens are present. The function application contains a function name, opening and closing parentheses and the parameters with separating commas. (See in Figure: 4)

Most of the above tokens can be automatically generated. Creating the function definition requires only the function name, parameter names and the body expressions; creating the function application requires the function name and the actual parameters. The syntactical and lexical representation of the body of the extracted function are available from the selection, but the other parts have to be constructed.

Previously, this refactoring constructed all parts using manually crafted code. Such code proved to be long, hard to maintain and error prone. In contrast, using the utilities in section 2, the corresponding code consists of only a few lines. Furthermore, the code is much more readable, as it highlights those parts that cannot be automatically generated.

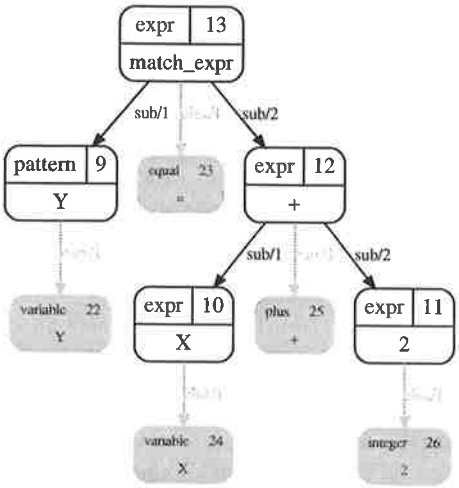


Figure 2: The representation of X+2, which is about to be extracted.

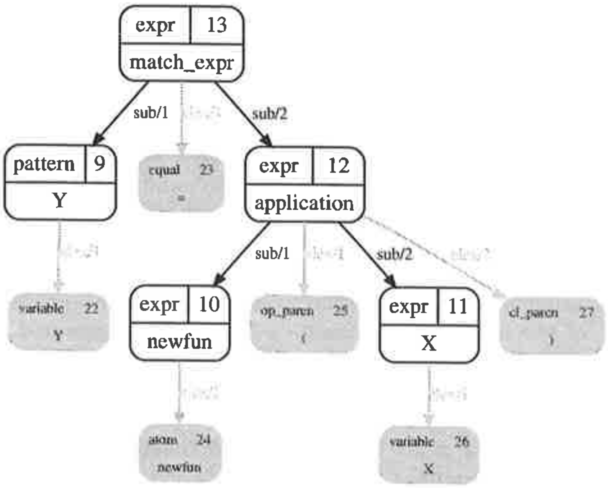


Figure 3: The representation of the function application that replaces the selection.

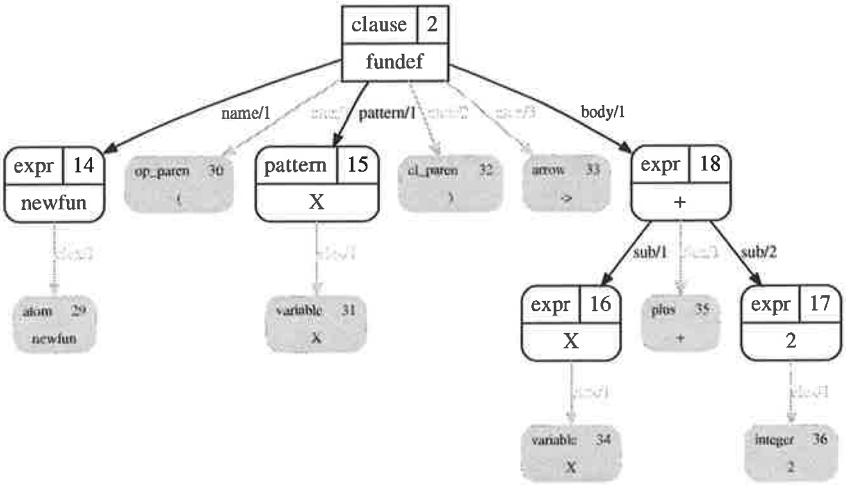


Figure 4: The representation of the newly created function definition.

5 Conclusion

In this paper, we have presented methods for creating, replacing and inserting subtrees in the RefactorErl refactorer. Our experience shows that it greatly enhances the readability and reliability of the invocation of these operations, in contrast to their hand coded equivalents. Using these methods, we were able to implement rudimentary functionality of twelve new refactorings in a time frame of two months, compared to only two before. The two already implemented refactorings were *Extract function* and *Move function definition*, which were improved using the methods described in this paper. The new refactorings are the following: *Rename variable*, *Rename function*, *Rename module*, *Rename record*, *Rename record field*, *Reorder function arguments*, *Tuple function arguments*, *Eliminate variable*, *Merge subexpression duplicates*, *Move record*, *Inline function*, *Generalize function*.

References

[1] G. Fischer, J. Lusiardi, and J. Wolff v. Gudenberg, *Abstract syntax trees and their role in model driven software development.* In ICSEA online proceedings. IEEE, 2007.

[2] J. Barklund and R. Virding, *Erlang Reference Manual*, 1999, Available from http://www.erlang.org/download/erl_spec47.ps.gz.

[3] Greg J. Badros, *Javaml: a markup language for java source code.* In Proceedings of the 9th international World Wide Web conference

on Computer networks: the international journal of computer and telecommunications networking, pages 159177. North-Holland Publishing Co. Amsterdam, The Netherlands, The Netherlands, 2000.

[4] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, Tamás Kozsik, *Preprocessor and whitespace-aware toolset for Erlang source code manipulation.* Abstract submitted to the 20th International Symposium on the Implementation and Application of Functional Languages, Hatfield UK.

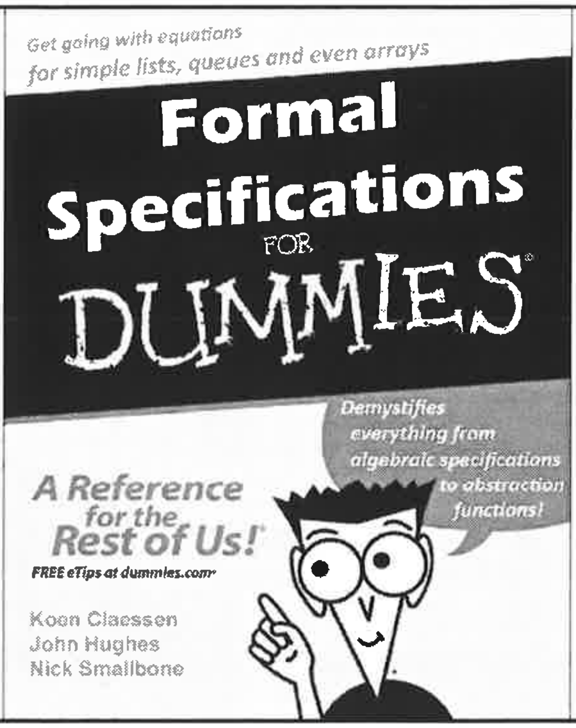
[5] R. Szabó-Nacsa, P. Divinszky, and Z. Horváth, *Prototype environment for refactoring Clean programs.* In The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4, 2004.

[6] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy, *Refactoring Erlang Programs.* In Proceedings of the 12th International Erlang/OTP User Conference, November 2006.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[8] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A., and Nagy T, *Refactoring in Erlang, a Dynamic Functional Language.* In Proceedings of the 1st Workshop on Refactoring Tools, Berlin, Germany, July 2007, pp. 45-46.

[9] Jonathan I. Maletic, Michael L. Collard, and Adrian Marcus, *Source code files as structured documents.* In Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), pp. 289–292. IEEE Computer Society Washington, DC, USA, 2002.

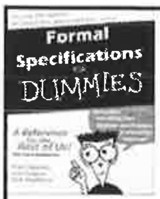

Get going with equations
for simple lists, queues and even arrays
Formal Specifications
FOR
DUMMIES
Demystifies everything from algebraic specifications to abstraction functions!
A Reference for the Rest of Us!
FREE eTips at dummies.com
Koen Claessen
John Hughes
Nick Smallbone

Algebraic Specifications

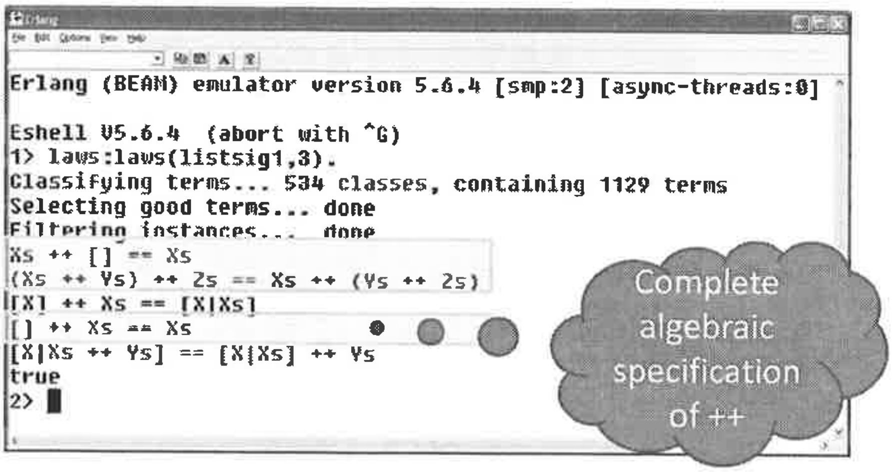
- Relate functions in an API to each other
  - Capture useful properties
  - Don't really say what they do

```
Xs ++ [] == Xs
[] ++ Xs == Xs
(Xs++Ys) ++ Zs == Xs ++ (Ys++Zs)
```

- Good for data-structures
- Only *pure functions* need apply!

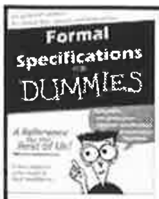


DEMO

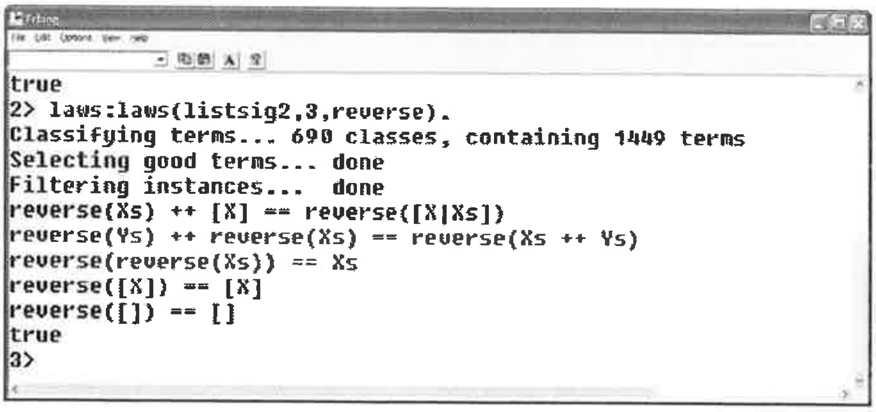


```
Erlang (BEAM) emulator version 5.6.4 [smp:2] [async-threads:0]

Eshell V5.6.4  (abort with ^G)
1> laws:laws(listsig1,3).
Classifying terms... 534 classes, containing 1129 terms
Selecting good terms... done
Filtering instances...  done
Xs ++ [] == Xs
(Xs ++ Ys) ++ Zs == Xs ++ (Ys ++ Zs)
[X] ++ Xs == [X|Xs]
[] ++ Xs == Xs
[X|Xs ++ Ys] == [X|Xs] ++ Ys
true
2>
```

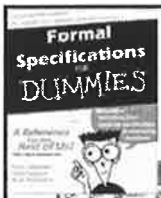
Complete algebraic specification of ++



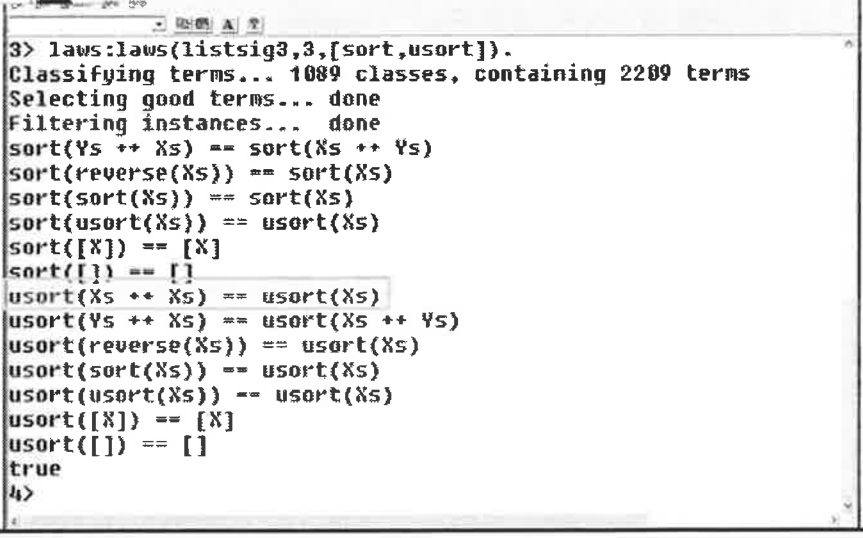
Add reverse to the mix



```
true
2> laws:laws(listsig2,3,reverse).
Classifying terms... 690 classes, containing 1449 terms
Selecting good terms... done
Filtering instances...  done
reverse(Xs) ++ [X] == reverse([X|Xs])
reverse(Ys) ++ reverse(Xs) == reverse(Xs ++ Ys)
reverse(reverse(Xs)) == Xs
reverse([X]) == [X]
reverse([]) == []
true
3>
```

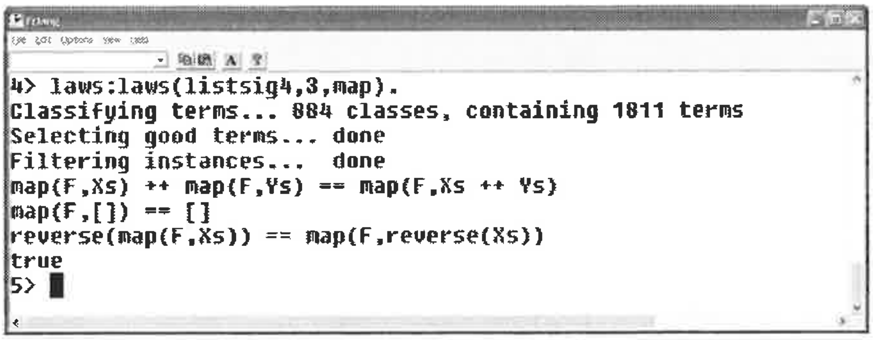


What about sorting?



```
3> laws:laws(listsig3,3,[sort,usort]).
Classifying terms... 1089 classes, containing 2209 terms
Selecting good terms... done
Filtering instances...  done
sort(Ys ++ Xs) == sort(Xs ++ Ys)
sort(reverse(Xs)) == sort(Xs)
sort(sort(Xs)) == sort(Xs)
sort(usort(Xs)) == usort(Xs)
sort([X]) == [X]
sort([]) == []
usort(Xs ++ Xs) == usort(Xs)
usort(Ys ++ Xs) == usort(Xs ++ Ys)
usort(reverse(Xs)) == usort(Xs)
usort(sort(Xs)) == usort(Xs)
usort(usort(Xs)) == usort(Xs)
usort([X]) == [X]
usort([]) == []
true
4>
```

What about map?



```
4> laws:laws(listsig4,3,map).
Classifying terms... 884 classes, containing 1811 terms
Selecting good terms... done
Filtering instances...  done
map(F,Xs) ++ map(F,Ys) == map(F,Xs ++ Ys)
map(F,[]) == []
reverse(map(F,Xs)) == map(F,reverse(Xs))
true
5> █
```

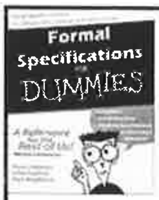
The new array library

- Flexible arrays, indexed from 0
- Purely functional updates

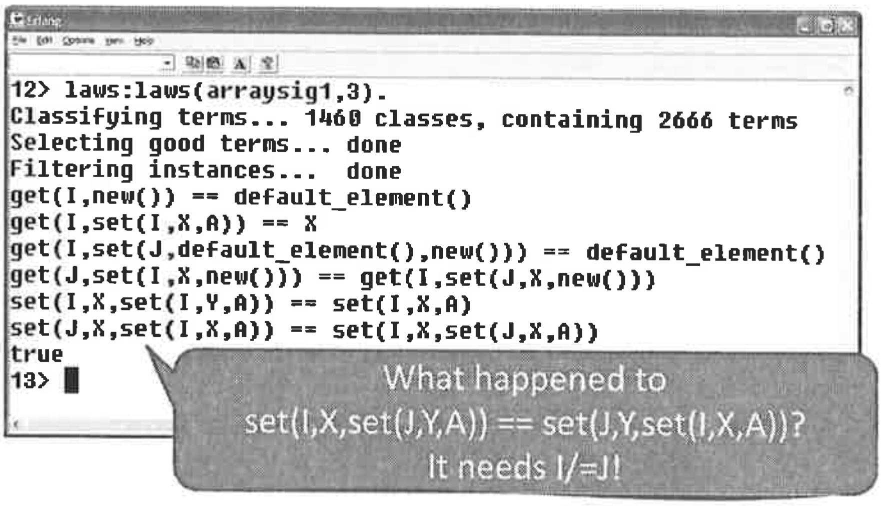
new() -> array()
Create a new, extendible array with initial size zero

get(I::integer(), Array::array()) -> term()
Get the value of entry I.

set(I::integer(), Value::term(), Array::array()) -> array()
Set entry I of the array to Value.



Get and Set



```
12> laws:laws(arraysig1,3).
Classifying terms... 1460 classes, containing 2666 terms
Selecting good terms... done
Filtering instances...  done
get(I,new()) == default_element()
get(I,set(I,X,A)) == X
get(I,set(J,default_element(),new())) == default_element()
get(J,set(I,X,new())) == get(I,set(J,X,new()))
set(I,X,set(I,Y,A)) == set(I,X,A)
set(J,X,set(I,X,A)) == set(I,X,set(J,X,A))
true
13> 
```

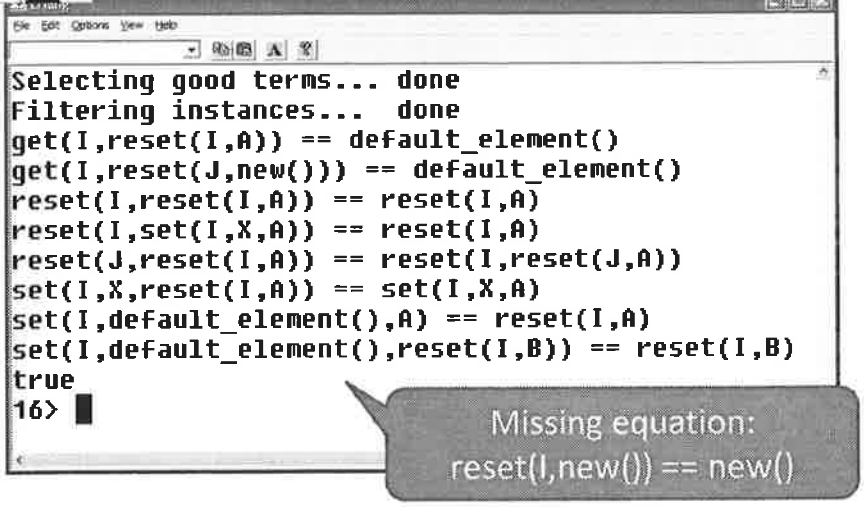
What happened to
set(I,X,set(J,Y,A)) == set(J,Y,set(I,X,A))?
It needs I/=J!

Adding Reset

- Elements can be reset to the default value

reset(I::integer(), Array::array()) -> array()
Reset entry I to the default value for the array.

Specifying reset



```
Selecting good terms... done
Filtering instances...  done
get(I,reset(I,A)) == default_element()
get(I,reset(J,new())) == default_element()
reset(I,reset(I,A)) == reset(I,A)
reset(I,set(I,X,A)) == reset(I,A)
reset(J,reset(I,A)) == reset(I,reset(J,A))
set(I,X,reset(I,A)) == set(I,X,A)
set(I,default_element(),A) == reset(I,A)
set(I,default_element(),reset(I,B)) == reset(I,B)
true
16> █
```

Missing equation:
reset(I,new()) == new()

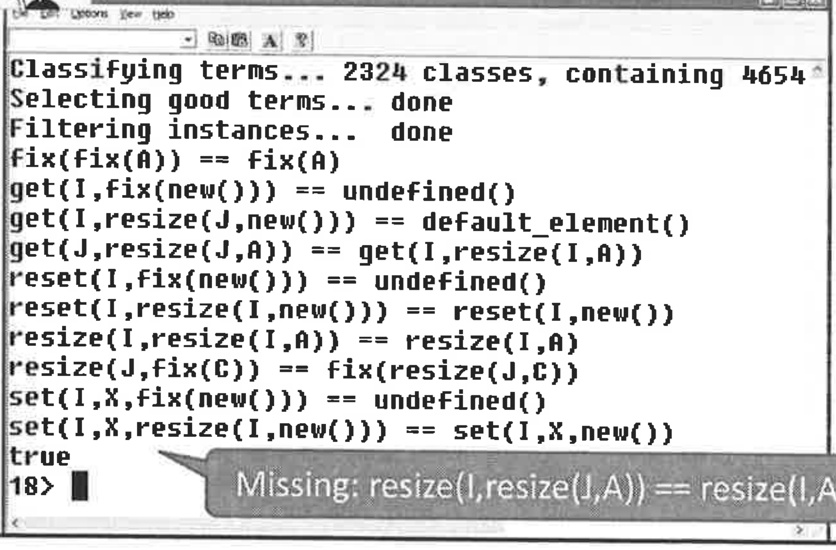
Fix and Resize

- It is possible to *fix* the size of an array…

fix(Array::array()) -> array()
Fix the size of the array.

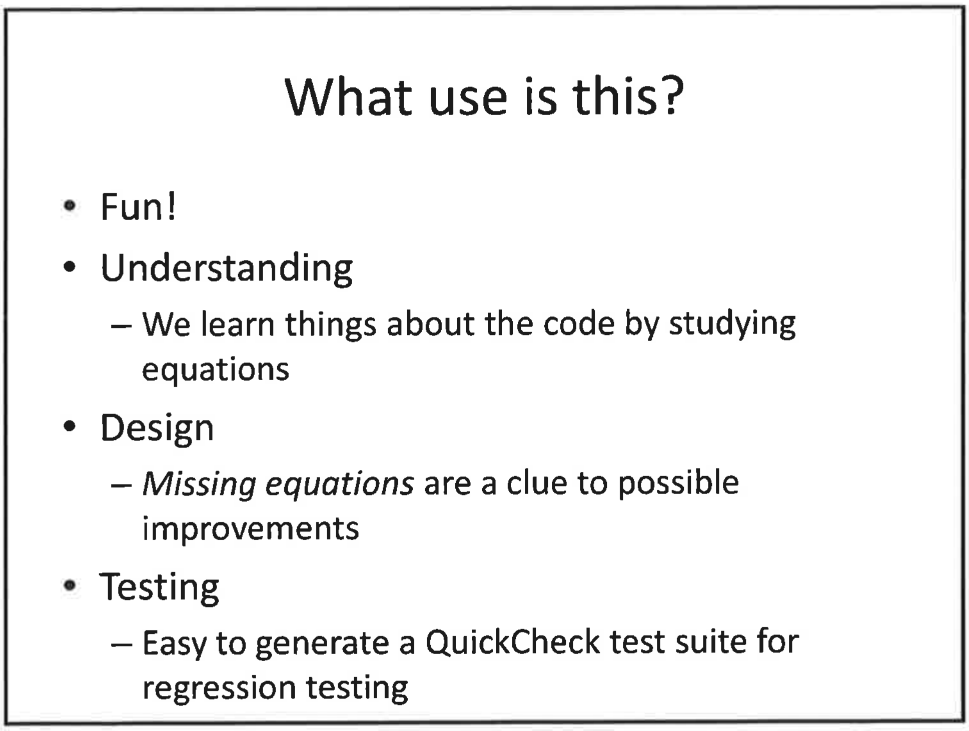
resize(Size::integer(), Array::array()) -> array()
Change the size of the array.

Specifying Fix and Resize



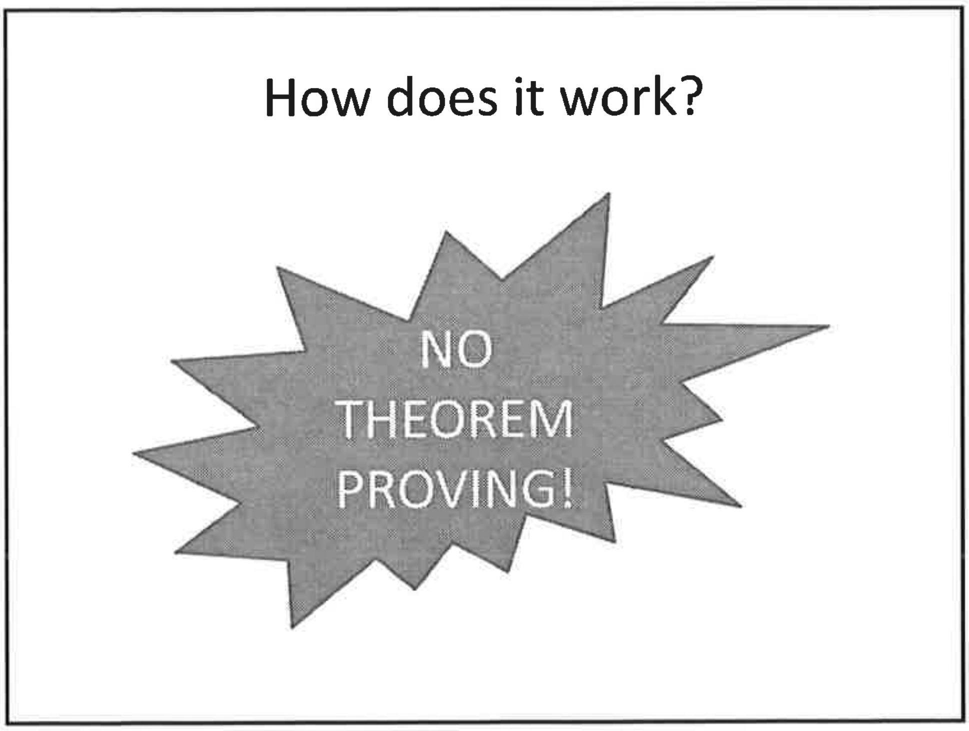
```
Classifying terms... 2324 classes, containing 4654
Selecting good terms... done
Filtering instances...  done
fix(fix(A)) == fix(A)
get(I,fix(new())) == undefined()
get(I,resize(J,new())) == default_element()
get(J,resize(J,A)) == get(I,resize(I,A))
reset(I,fix(new())) == undefined()
reset(I,resize(I,new())) == reset(I,new())
resize(I,resize(I,A)) == resize(I,A)
resize(J,fix(C)) == fix(resize(J,C))
set(I,X,fix(new())) == undefined()
set(I,X,resize(I,new())) == set(I,X,new())
true
18> █
```

Missing: resize(I,resize(J,A)) == resize(I,A)



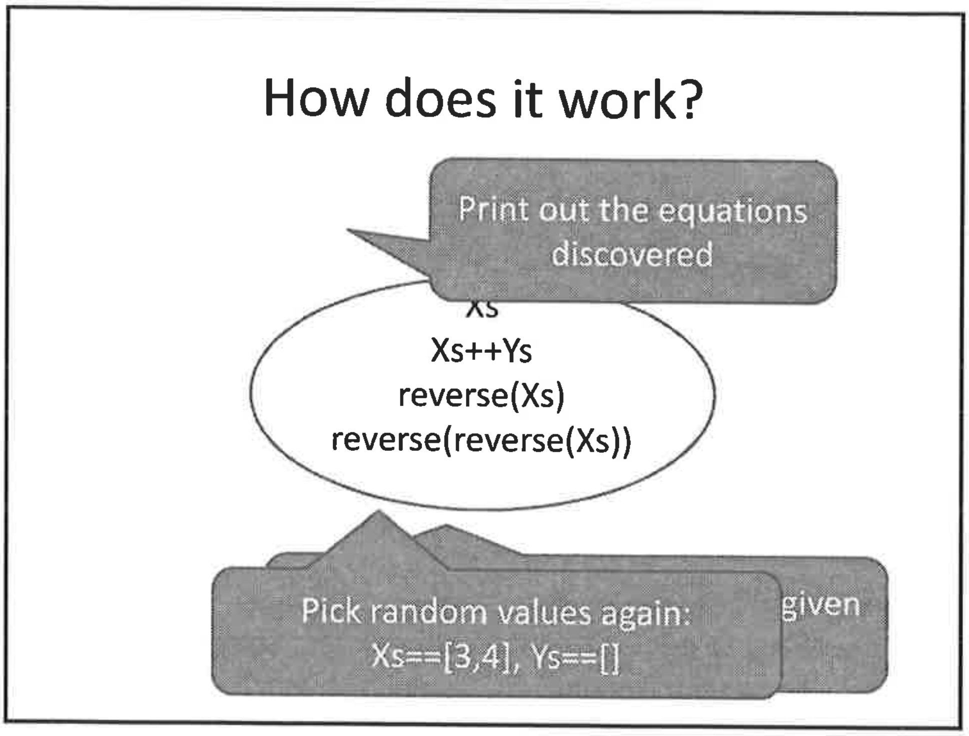
What use is this?

- Fun!
- Understanding
  - We learn things about the code by studying equations
- Design
  - *Missing equations* are a clue to possible improvements
- Testing
  - Easy to generate a QuickCheck test suite for regression testing



How does it work?

NO
THEOREM
PROVING!



How does it work?

Print out the equations discovered

Xs

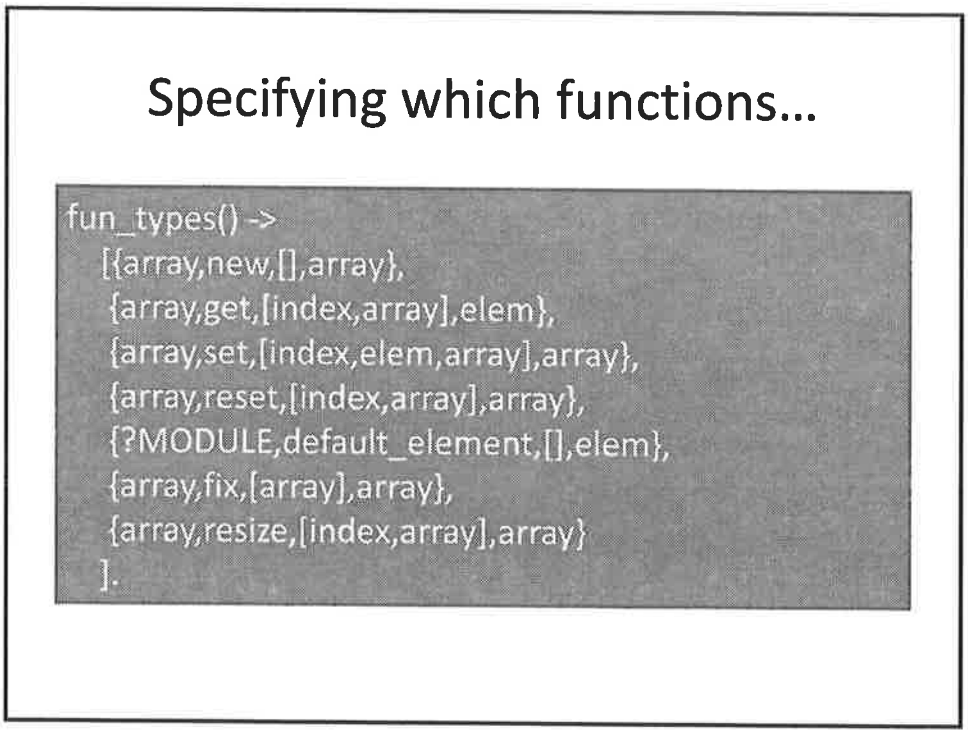
Xs++Ys

reverse(Xs)

reverse(reverse(Xs))

given

Pick random values again:
Xs==[3,4], Ys==[]



Specifying which functions…

```
fun_types() ->
   [{array,new,[],array},
    {array,get,[index,array],elem},
    {array,set,[index,elem,array],array},
    {array,reset,[index,array],array},
    {?MODULE,default_element,[],elem},
    {array,fix,[array],array},
    {array,resize,[index,array],array}
   ].
```

Specifying which variables…

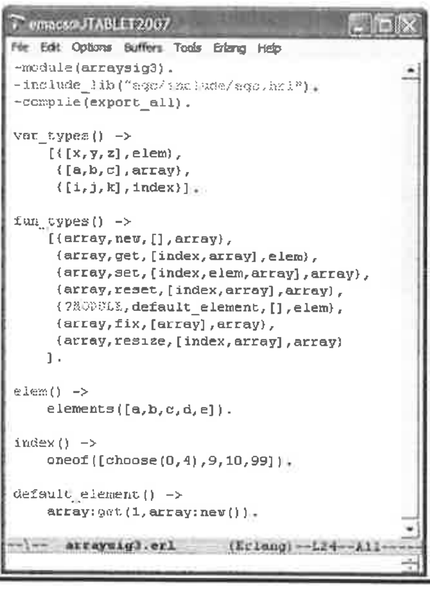
```
var_types() ->
   [{[x,y,z],elem},
    {[a,b,c],array},
    {[i,j,k],index}].
```

A few QuickCheck generators…

```
elem() ->
   elements([a,b,c,d,e]).

index() ->
   oneof([choose(0,4),9,10,99]).
```

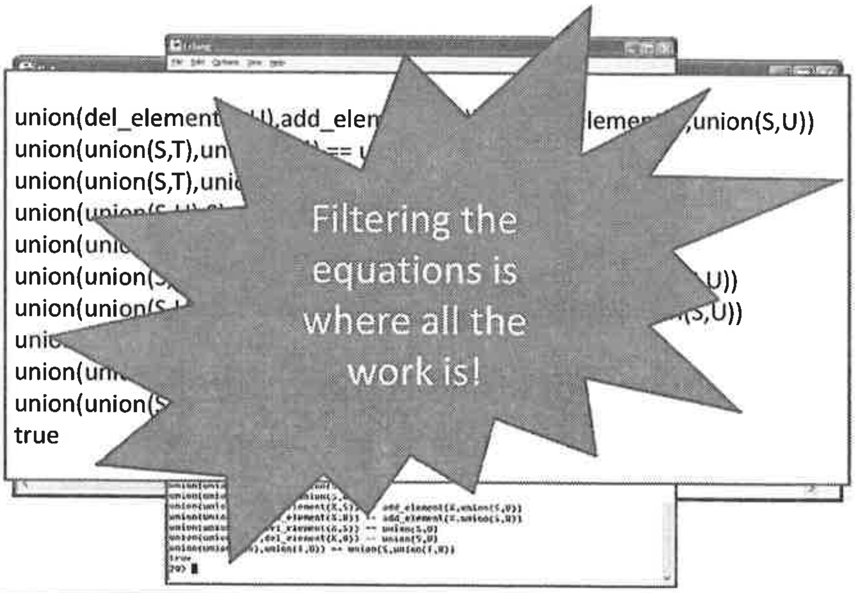
- A generator for arrays is constructed automatically

That's It!



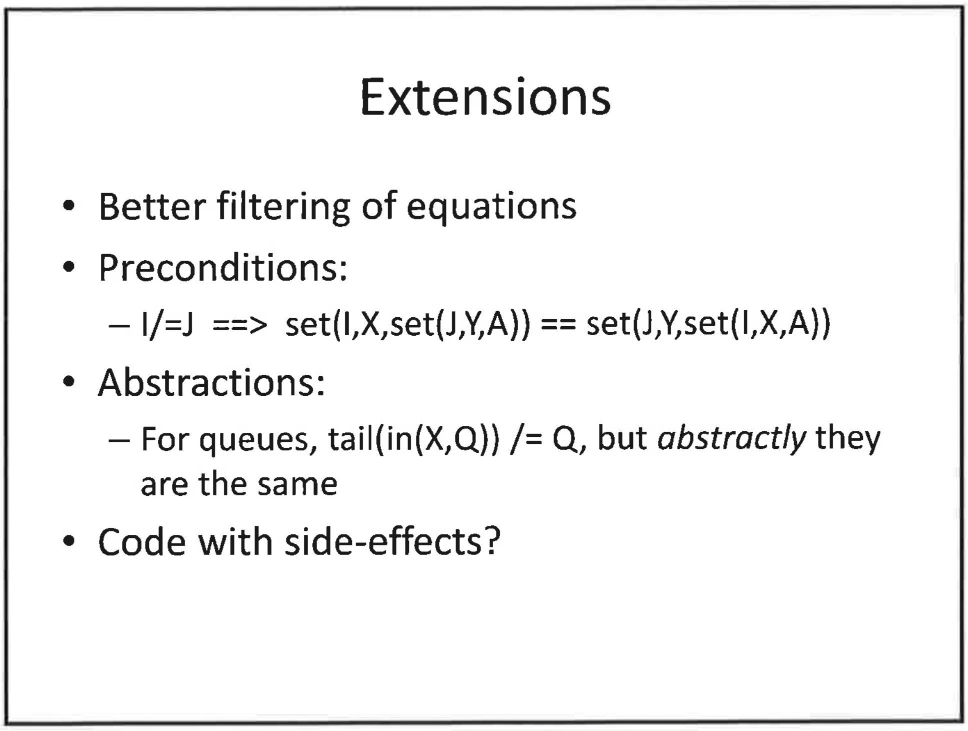
```
-module(arraysig3).
-include_lib("eqc/include/eqc.hrl").
-compile(export_all).

var_types() ->
    [{[x,y,z],elem},
     {[a,b,c],array},
     {[i,j,k],index}].

fun_types() ->
    [{array,new,[],array},
     {array,get,[index,array],elem},
     {array,set,[index,elem,array],array},
     {array,reset,[index,array],array},
     {?MODULE,default_element,[],elem},
     {array,fix,[array],array},
     {array,resize,[index,array],array}
    ].

elem() ->
    elements([a,b,c,d,e]).

index() ->
    oneof([choose(0,4),9,10,99]).

default_element() ->
    array:get(1,array:new()).
```

The Hard Part…



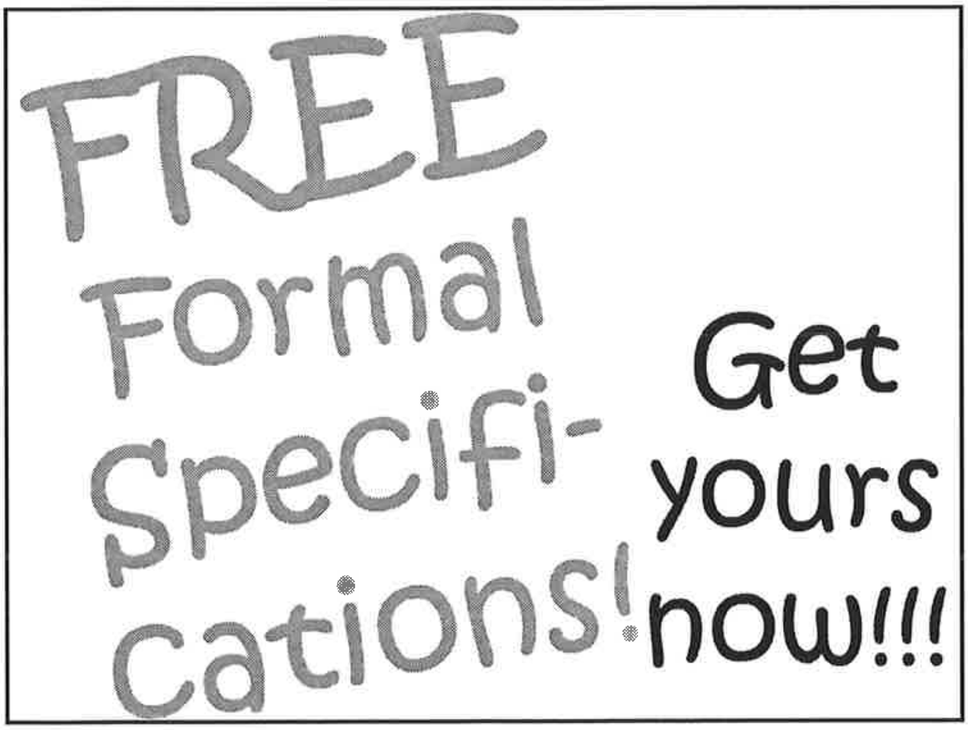
```
union(del_element  U),add_elem                lemen  ,union(S,U))
union(union(S,T),un        == u
union(union(S,T),unic
union(
union(unic
union(union(                                          U))
union(union(S                                         S,U))
unic
union(un
union(union(S
true
```

Filtering the equations is where all the work is!



Extensions

- Better filtering of equations
- Preconditions:
  - I/=J ==> set(I,X,set(J,Y,A)) == set(J,Y,set(I,X,A))
- Abstractions:
  - For queues, tail(in(X,Q)) /= Q, but *abstractly* they are the same
- Code with side-effects?



FREE Formal Specifications!

Get yours now!!!

The Erlang Web

an Open Source Fault Tolerant and Scalable Web Framework

Michal Ptaszek, Michal Slaski, Michal Zajda
Erlang Training and Consulting Ltd

Erlang Web

The Erlang Web is an open source framework for applications based on HTTP protocols, giving the developer better control of content management. With Erlang Web's simple but extensible concept of including both static and dynamic content in pages, libraries of reusable components can be built. Currently it supports INETS and Yaws servers, but others are planned in the future. The Erlang Web platform has been developed by Erlang Training & Consulting Ltd. for the past three years and has been used in many commercial and high profile projects.

Key features

In the Erlang Web generation of dynamic pages is done by merging XHTML code with a dedicated XML tag called **wpart**. With the concept of wparts, it is possible to develop pieces of functionality that can be reused in different pages, providing solid framework for developing dynamic web services. A set of frequently used patterns has been implemented as a common library of wparts. Some examples of wparts include:

- retrieving data
- forms building
- manipulating and iterating over a list of Erlang terms
- translating a string
- branching constructions

Moreover, the **wtype** mechanism allows formatting basic types like date, time, numbers and validating the data that came with GET/POST request. Formatting and validation can be done also for user defined complex types. Such complex types can consist of basic or other complex types and can be defined using Erlang record syntax. This is convenient when data is kept in the Mnesia database, however the data can be kept in other databases like MySQL or PostgreSQL as well.

To make URLs easier to remember, Erlang Web provides a **dispatcher** engine based on regular expressions which maps URLs to controller calls. Each call to the controller function can be preceded with and followed by calls to the so-called **dataflow functions**, which allows adding aspects. With the **template inheritance** and support for **multilingual** sites, the Erlang Web is a complete tool for building web applications.

Wparts

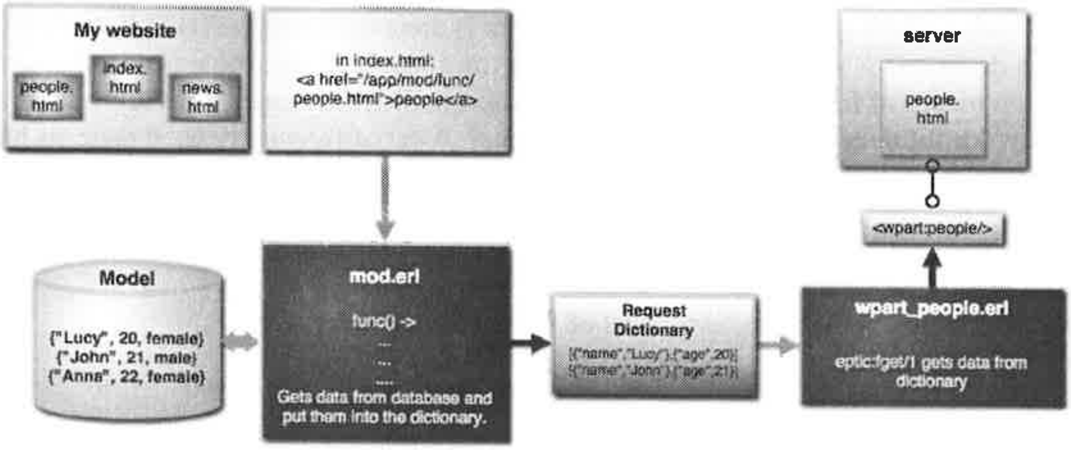
Having wparts defined as XML elements allows the Erlang Web to validate a page with an XML parser ensuring that what is sent to the client browser is free of XHTML errors. A parsed file is converted to Erlang binary and cached on disk or kept in the memory.

Wtypes

Wtype module is responsible for validating and formatting the data of some type. The Erlang Web provides over 10 simple basic types like dates, integers, strings, etc. and it is easy to add new ones. Developer can build his own complex types on top of them. Without additional development effort validation and generation of forms for create/update operations can be automated.

Example Code – simple case

The below diagram shows an example of wpart *people*, that is used to add dynamic content to the page. When "/app/mod/func/people.html" URL is visited, the controller function mod:func() is called. The function reads data from the model and prepares it for the wpart *people*. During page rendering, wpart tag is expanded with the dynamic data.



people.html – renders data from the model

```
<html>
  <head>
    <title>Erlang Web Sample Page 4</title>
  </head>
  <body>
    <center>
      <wpart:people rows="2" key="list">
          <tr>
            <td><wpart:lookup key="item:person" /></td>
            <td><wpart:lookup key="item:age" format="age"/></td>
            <td><wpart:lookup key="item:sex" format="sex"/></td>
          </tr>
      </wpart:people>

      <img src="/images/powered.gif" />
    </center>
  </body>
</html>
```

install.erl – initialises mnesia database and puts sample data into it.

```
-module(install).
-export([install/0]).
install() ->
    mnesia:create_schema([node()]),
    application:start(mnesia),
    mnesia:create_table(person,
                             [{disc_copies, [node()]},
                              {attributes, record_info(fields,person)}]),
    mnesia:dirty_write(#person{name = "Lucy", age = "20", sex="female"}),
    mnesia:dirty_write(#person{name = "John", age = "22", sex="male"}),
    mnesia:dirty_write(#person{name = "Anna", age = "22", sex="female"}),
```

mod.erl – controller function writes data into the request dictionary

```
-module(mod).
-export([validate/1,func/0,]).
-export([install/0]).

-record(person, {name, age, sex}).

validate(func) ->
    {ok, []};

func() ->
    Keys = mnesia:dirty_all_keys(person),
    Records = [mnesia:dirty_read(person, Key) || Key <- Keys],
    Persons = lists:map(fun([#person{name=Name, age=Age, sex=Sex}]) ->
                                       [{"person",Name},
                                        {"age",Age},
                                        {"sex",Sex}]
                             end,
                             Records),
    eptic:fset("list", Persons),
    template.
```

wpart_people.erl – wpart module handles formatting and retrieves data from the request dictionary

```
-module(wpart_people).
-export([handle_call/1]).
-include_lib("xmerl/include/xmerl.hrl").

handle_call(E) ->
    Start = case catch list_to_integer(eptic:fget("get", "start")) of
                   S when is_integer(S) -> S;
                   _ -> 1
              end,
    Rows = case catch list_to_integer(wpart:has_attribute("attribute::rows", E)) of
                 R when is_integer(R) -> R;
                 _ -> 10
             end,
    Key = wpart:has_attribute("attribute::key", E),
    List = eptic:fget(Key),

    Prev = if Start-Rows > 0 -> link_prev(Start-Rows);
                Start == 1     -> "Prev | ";
                true           -> link_prev(1)
             end,
    Next = if Start+Rows > length(List) -> "Next";
                true                       -> link_next(Start + Rows)
             end,

    F = fun(Item, {N,Start,End,Acc}) when N>=Start, N<End ->
                   eptic:fset("item", Item),
                   {N+1,Start,End,[wpart:eval(E#xmlElement.content)|Acc]};
             (_, {N,Start,End,Acc}) ->
                   {N+1,Start,End,Acc}
          end,
    {_,_,_,TableRows} = lists:foldl(F, {1,Start,Start+Rows,[]}, List),
    [#xmlText{value = "<table>", type=cdata},
     lists:reverse(TableRows),
     #xmlText{value = "</table>", type=cdata},
     #xmlText{value = Prev, type=cdata},
     #xmlText{value = Next, type=cdata}].

link_prev(Start) ->
    ["<a href=\"?start=",integer_to_list(Start),"\">Prev</a> | "].
link_next(Start) ->
    ["<a href=\"?start=",integer_to_list(Start),"\">Next</a>"].
```

Example Code – listing the person with given Id and adding new person

The example illustrates dispatching request with the dispatcher, logging the request with a dataflow function, validating, automatic form building and error handling.

dispatch.conf – dispatcher configuration file that defines regular expressions

```
{dynamic, “^/person”, {people, list}}.
{static, “^/add_person$”, “add_person.html”}.
{dynamic, “^/create_person$”, {people, add}}.
{static, “^/index.html$”, “index.html”}.
```

index.html – contains a link to the page that lists the person description stored in the database

```
<a href=”/person”>List the first person</a>
```

and to the page where a new person can be added to the database:

```
<a href=”/add_person”>Add a person</a>
```

person.hrl – contains person and person_types records

```
-record(person, {id, name, sex, age}).

-record(person_types,
         {id = {integer, [{description, “Person ID”},
                          {primary_key},
                          {min, 1}]},
         name = {string, [{description, “Person name”},
                          {min_length, 3},
                          {max_length, 20}],
         sex = {bool, [{description, “Is person male?”}]}}).
```

wtype_person.erl – defines the wtype person using Erlang records

```
-module(wtype_person).
-import(“person.hrl”).

-export([get_record_info/1, validate/1]).

get_record_info(person_types) -> #person_types{};
get_record_info(person) -> #person{}.

validate(From) ->
         wpart_valid:validate(get_record_info(person),
                              get_record_info(person_types),
                              From ++ [“person”]).
```

people.erl – implements controller function. Call to the function will be proceded with calls to the *log* function and *get_arg* or *validate* functions.

```
-module(people.erl).
-export([dataflow/1, error/2]).
```

```
-export([log/2, get_arg/2, validate/2]).
-export([list/1, add/1]).

-include(“person.hrl”).

dataflow(list) → [log, get_arg];
dataflow(add)  → [log, validate].

error(_, {cannot_write_to_log, _LogName, _Fun} = Reason) →
          error_logger:error_msg(“~p module, error: ~p~n”, [?MODULE, Reason]),
          {redirect, “/index.html”};
error(add, not_valid) →
          Err = wpart:fget(“__error”),
          Msg = “ERROR: Incomplete input or wrong type in form! Reason: “ ++ Err,
          wpart:fset("error_message", Msg),
          {template, “error_add.html”}.

log(Fun, _) →
          case my_logger:log(“people.log”, {?MODULE, Fun}) of
                    true → {ok, []};
                    false → {error, {cannot_write_to_log, “people.log”, Fun}}
          end.

get_arg(list, _) →
          case catch list_to_integer( wpart:fget(“get”, “id”) of
                    P when is_integer(P) → {ok, [P]};
                    _ → {ok, [1]}
          end.

validate(add, _) →
          validate_tool:validate_cu(?MODULE, add).

list(Id) →
          [Person] = mnesia:dirty_read(person, Id),
          wpart:fset(“person:name”, Person#person.name),
          wpart:fset(“person:sex”, Person#person.sex),
          wpart:fset(“person:age”, Person#person.age),

          Prev = if
                    Id > 1 →
                              “<a href=”?id=\”” ++ integer_to_list(Id-1) ++ “\”>Prev</
a> | ”;
                    true →
                              “Prev | “
          end,
          Next = “<a href=”?id=\”” ++ integer_to_list(Id+1) ++ “\”>Next</a>”,

          wpart:fset(“person:next”, Next),
          wpart:fset(“person:prev”, Prev),

          {template, “person.html”}.

add(Person) →
          mnesia:dirty_write(Person),
          {redirect, “/person?id=” ++ integer_to_list(Person#person.id)”}.
```

person.html – renders a person

```
<html>
  <head>
    <title>Erlang Web Example Page</title>
  </head>
  <body>
    <center>
          Person (id = <wpart:lookup key=”person:id”/>) details:
            Name: <wpart:lookup key="person:name" /><br/>
            Age: <wpart:lookup key="person:age" format="age"/><br/>
            Sex: <wpart:lookup key="person:sex" format="sex"/><br/>

          <wpart:lookup key=”person:prev”/><wpart:lookup key=”person:next”/>
    </center>
  </body>
</html>
```

add_person.html – generates form used to create a new person

```
<html>
  <head>
    <title>Erlang Web Example Page</title>
  </head>
  <body>
    <center>
          <wpart:form type=”person” action=”/create_person”/>
    </center>
  </body>
</html>
```

error_add.html – displays error

```
<html>
  <head>
    <title>Erlang Web Example Page</title>
  </head>
  <body>
    <center>
          <wpart:lookup key=”error_message”/>
    </center>
  </body>
</html>
```

install.erl – initialises mnesia database and puts sample data into it

```
-module(install).
-export([install/0]).
-import(“person.hrl”).

install() ->
    mnesia:create_schema([node()]),
    application:start(mnesia),
    mnesia:create_table(person,
                             [{disc_copies, [node()]},
                              {attributes, record_info(fields,person)}]),
    mnesia:dirty_write(#person{name = "Lucy", age = "20", sex="female"}),
    mnesia:dirty_write(#person{name = "John", age = "22", sex="male"}),
    mnesia:dirty_write(#person{name = "Anna", age = "22", sex="female"}),
```

Ad Serving in Erlang

Bob Ippolito
Mochi Media, Inc.

Erlang User Conference 2008 - Stockholm

November 13, 2008

Author: Bob Ippolito
Date: November 2008
Venue: Erlang User Conference 2008

What's MochiAds?

MochiAds:

- Monetization platform for Flash game ecosystem
- Advertising solution for game developers
- Revenue share and distribution for publishers

Road to Erlang

MochiBot:

- (Originally) Python w/ Twisted
- Fast, but not fast enough (CPU bound)
- Wanted easy multi-node distribution

Previous Experience

Python:

- Non-blocking sockets are tedious
- Threads are too heavy

C/C++:

- Too low-level

Why Erlang?

- Performance
- Concurrency
- Distribution
- Fault Tolerance

(Bad) Benchmarks

ab -c 50 -n 10000 localhost

- Apache: 1x
- Twisted: 1.12x
- mochiweb: 2.5x
- nginx: 3.9x

Erlang at Mochi Media

MochiAds, MochiBot

- High-performance HTTP servers
- Ad targeting
- Real-time analytics
- Social gaming
- Lots of internal use

MochiAds Service

- Front-end
- Data warehouse
- Ad server

What's Not Erlang

Front-end:

- Python and PostgreSQL

Data warehouse:

- Python and Vertica

Ad Serving Platform

- Juniper Routers
- Cisco Switches
- OpenBSD load balancer
- Nginx HTTP load balancer (Linux)
- Erlang/OTP R12B-3 (Linux)

Ad Server Stack

- Erlang/OTP R12B-3
- mochiweb (http)
- egeoip (geolocation)
- eswf (SWF file format)

Ad Server

- Gather information about client
- Choose an ad
- Log impression data
- Log click data, redirection

Gathering Client Info

- Mostly client-side Flash code
- SharedObject (like cookies)
- Feeds targeting decisions for ad choice

Choose an ad

- Fold over in-memory data structure
- Filter out ads that don't match targeting info
- Weight the rest
- Choose a random number; 0 <= N < sum(Weights)

Log impression data

- Validation
- Stream to hourly disk log
- Increment counters in RAM db

Log click data

- Validation
- Stream to hourly disk log
- Increment counters in RAM db
- Redirect (HTTP 302) to destination URL

Cheap Tricks

- Ad request data in URL
- Long-term state in SharedObject

Short-term Feedback Loop

- Client state
- RAM db counters from previous serves that day

Analytics Feedback Loop

- ETL processes adjust per hour
- Campaign weights, budget adjustments, etc.

Lessons Learned (part 1)

- Network partitioning sucks
- Network partitioning sucks
- Network partitioning sucks

Lessons Learned (part 2)

- pg2 is broken
- mnesia is too slow (for us)
- Inter-node distribution protocol can be flaky
- Erlang open source not always robust
- Lists are not a good data type for strings

Favorite Erlang Features

- Module reloading
- Pattern matching
- Binaries
- Lightweight processes
- Concise but not cryptic

More Erlang at Mochi Media

- MochiBot
- MochiCrypt
- MochiScore
- IRC bot
- SVN deployment
- Monitoring system
- Node discovery

Erlang-DTrace

Garry Bulmer

Team DTrace: Tim Becker

What I'm going to talk about

- Introduction to DTrace & DTrace Architecture
- Demo of DTrace with 'one liners'
- Erlang + Dtrace = ?
- Erlang VM Architecture
- Current Erlang DTrace Scope
- Erlang-DTrace Demo
- Questions

What is DTrace?

"DTrace is a comprehensive dynamic tracing facility ... that can be used by administrators and developers on ***live production systems*** *to examine the behavior of both* ***user programs*** *and of the* ***operating system*** *itself.*

DTrace enables you to ***explore*** *your system to understand how it works, track down performance problems* ***across many layers of software****, or locate the cause of aberrant behavior.*

DTrace lets you ***create*** *your own* ***custom programs*** *to* ***dynamically instrument*** *the system and provide immediate,* ***concise answers to arbitrary questions****"*

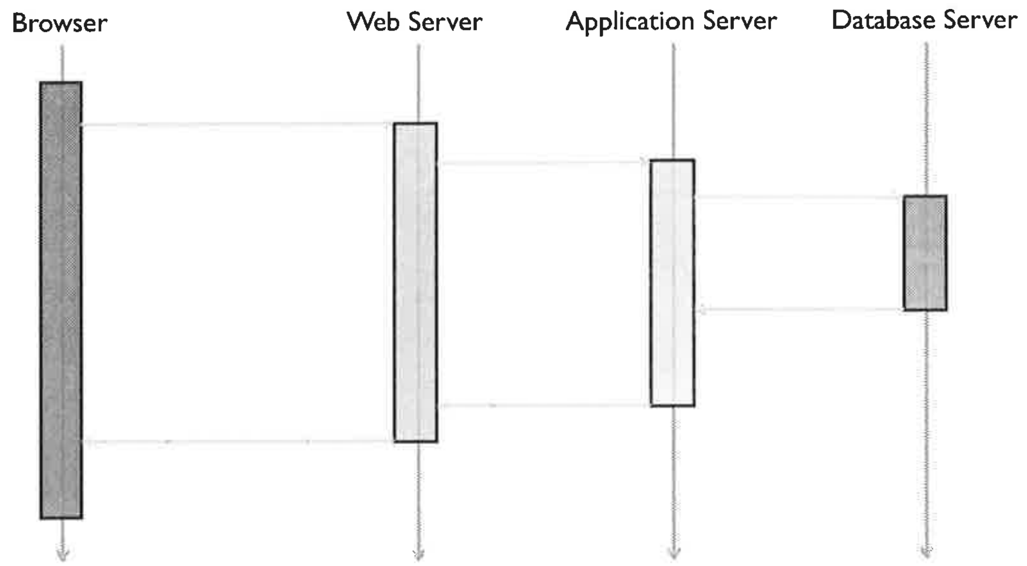
Source: Sun Microsystems "Solaris Dynamic Tracing Guide"

How does DTrace work?

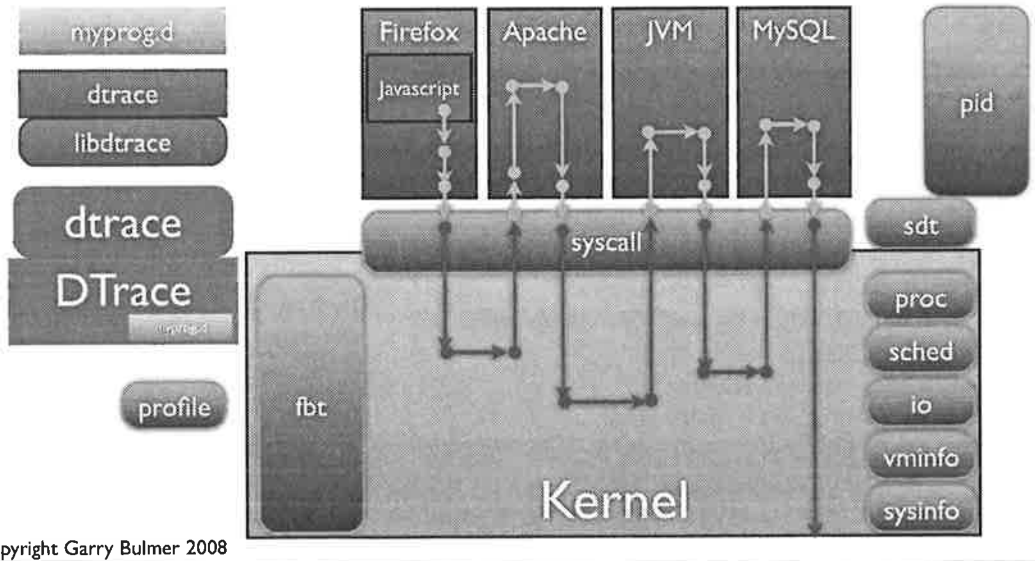
- KEY: Dynamically enabled probes - 'safe' for Production
- **Probes** observe function entry, exit & parameters
  - Probes observe events, and capture data
  - Probes in OS kernel ≈ 'Zero cost' when disabled *
- '**Providers**' - subsystem managing a group of Probes
  - Providers forward events and data to 'D programs'
- 'PID' Provider observes User applications

* SUN say cost < 0.5%

When is DTrace Useful?



DTrace End-to-End



DTrace D-Scripts

```
probe                    Provider:Module:Function:Name
/ optional predicates /
{
    optional action statements;
}
```

```
syscall:::entry
/ pid == 660 /
{
    printf(%-15s\n”, probefunc);
}
```

Provider = syscall
Module = *
Function = *
Name = entry

Copyright Garry Bulmer 2008

Providers

pid - userland processes, function entry, exit or instruction
sdt - Statically Defined Tracing
- programmer defined probes

syscall - entry and return of every system call
profile - time-driven probes, nanosecs. to days, across CPU’s
fbt - entry & return of almost all kernel functions

sched - scheduling thread on/off CPU, sleep/wake, ...
io - I/O events, start/complete/wait
proc - process creation and lifecycle
vminfo - uses vm kstat updates
sysinfo - uses sys kstat

Copyright Garry Bulmer 2008

DTrace - ‘one liners’

Question: Which applications are making the most system calls?

```
dtrace -n 'syscall:::entry { @num[execname] = count(); }'
```

Question: Which system calls is the Erlang VM making?

```
dtrace -n 'syscall:::entry /execname == "beam.smp"/
{ @num[probefunc] = count(); }'
```

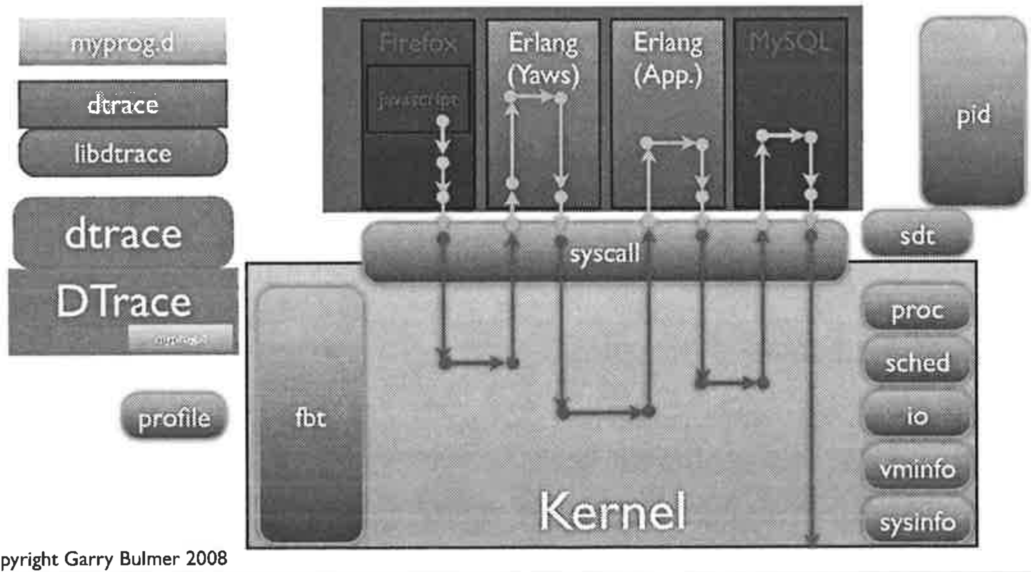
Question: What functions is the erlang VM calling?

```
dtrace -n 'pid*:::entry /execname == "beam.smp"/
{ @num[probefunc] = count(); }'
```

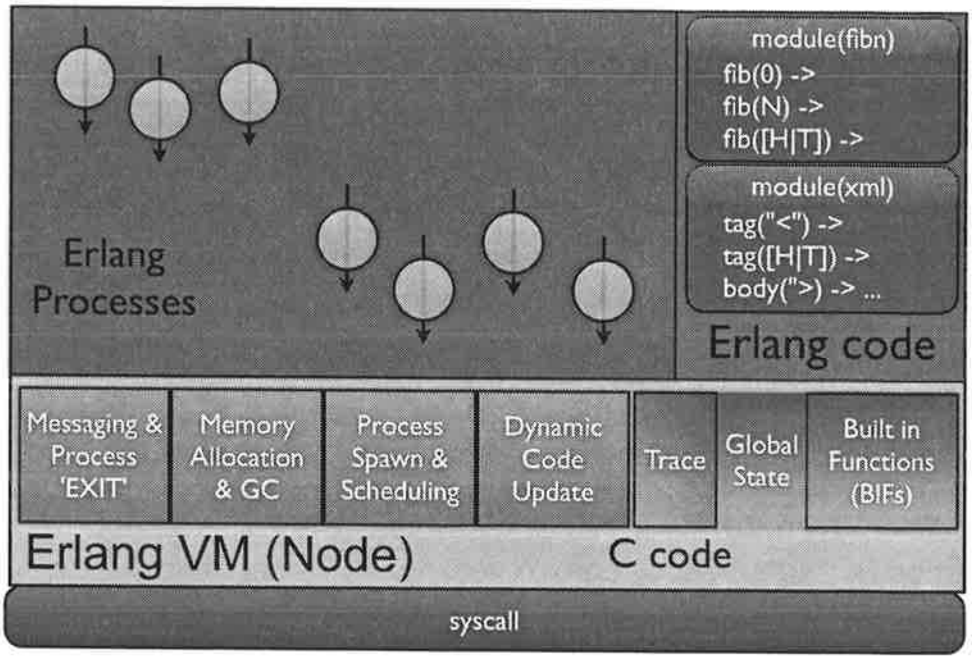
Question: How much memory is beam.smp allocating?

```
dtrace -n 'pid*::malloc*:entry /execname == "beam.smp"/
{ @num[probefunc] = sum(arg0); }'
```

Erlang-DTrace End-to-End



Erlang VM Architecture



module(fibn)
fib(0) ->
fib(N) ->
fib([H|T]) ->

module(xml)
tag("<") ->
tag([H|T]) ->
body(">) -> ...

Erlang Processes

Erlang code

Messaging & Process 'EXIT'

Memory Allocation & GC

Process Spawn & Scheduling

Dynamic Code Update

Trace

Global State

Built in Functions (BIFs)

Erlang VM (Node)

C code

syscall

Erlang's DTrace 'Fit'

- DTrace 'PID' Provider can observe C programs
  - Good: Erlang VM is C
  - Bad: user needs to understand Erlang VM internals !
- Erlang VM-managed, Fine-Grain 'Process'
  - Erlang Process ≈ 'opaque data' ∴ invisible to DTrace
- Erlang data is dynamically typed
  - DTrace uses static 'C-style' data types
- Erlang scripts are 'opaque data' to DTrace

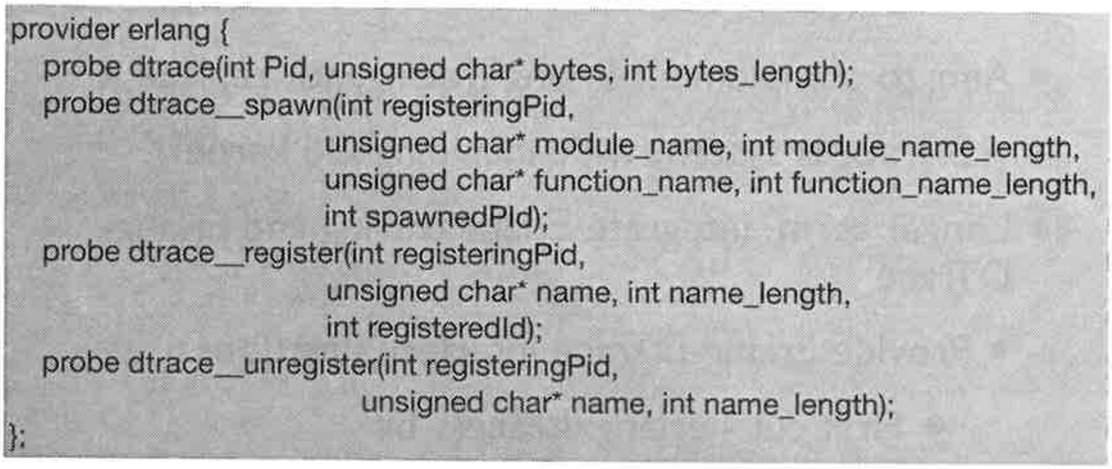
Erlang has Dynamic Tracing

- Aim: to complement Erlang Tracing, not replace it
  - DTrace is system-wide including OS kernel
- Longer term integrate Erlang tracing and Erlang-DTrace
  - Provide Erlang-DTrace interface functions
    - First cut - erlang:dtrace() ‘bif’

Erlang DTrace Implementation

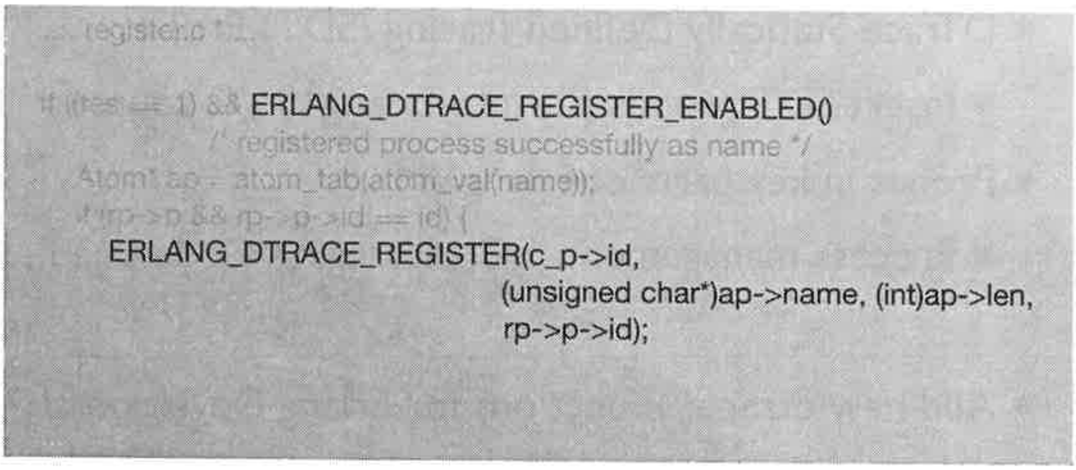
- DTrace Statically Defined Tracing (SDT) Probes
  - Insert SDT probes (C) into Erlang VM C source
- Probes in key parts of Erlang VM
  - Process management, GC, Messaging, Code Load ...
- Add new dtrace( ) functions for Erlang Developers

SDT - Erlang DTrace Provider



```
provider erlang {
   probe dtrace(int Pid, unsigned char* bytes, int bytes_length);
   probe dtrace__spawn(int registeringPid,
                         unsigned char* module_name, int module_name_length,
                         unsigned char* function_name, int function_name_length,
                         int spawnedPId);
   probe dtrace__register(int registeringPid,
                         unsigned char* name, int name_length,
                         int registeredId);
   probe dtrace__unregister(int registeringPid,
                            unsigned char* name, int name_length);
};
```

SDT - Erlang Source addition



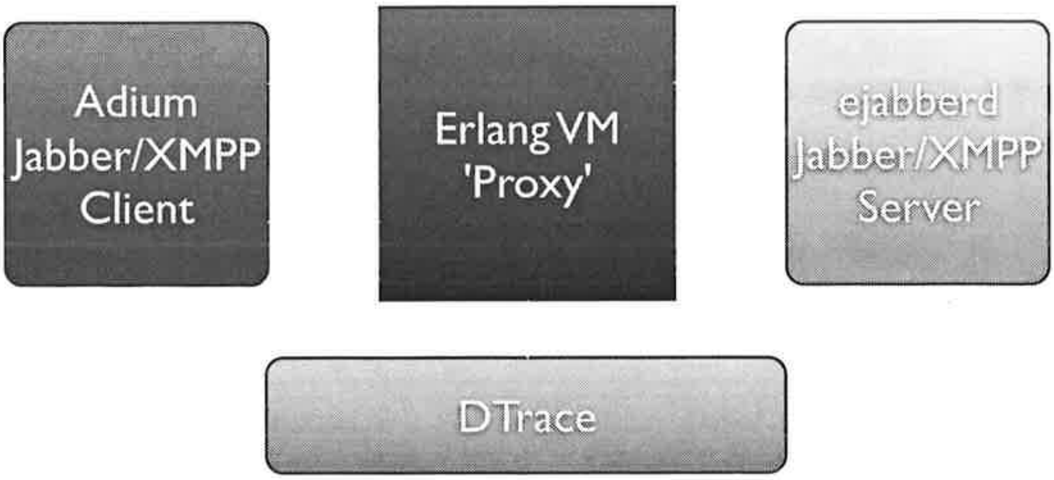
```
... register.c *...

if ((res == 1) && ERLANG_DTRACE_REGISTER_ENABLED()
           /* registered process successfully as name */
   Atom* ap = atom_tab(atom_val(name));
   if (rp->p && rp->p->id == id) {
     ERLANG_DTRACE_REGISTER(c_p->id,
                             (unsigned char*)ap->name, (int)ap->len,
                             rp->p->id);
```

* source is trimmed down from the original for readability

V002 Erlang-DTrace Scope

- Statically Defined Tracing Probes added to Erlang VM
    - Processes (spawn), Memory (GC),
    - Global State (Registry)
- New DTrace BIFs (explicitly use DTrace probes in Erlang)

Erlang-Dtrace Demo

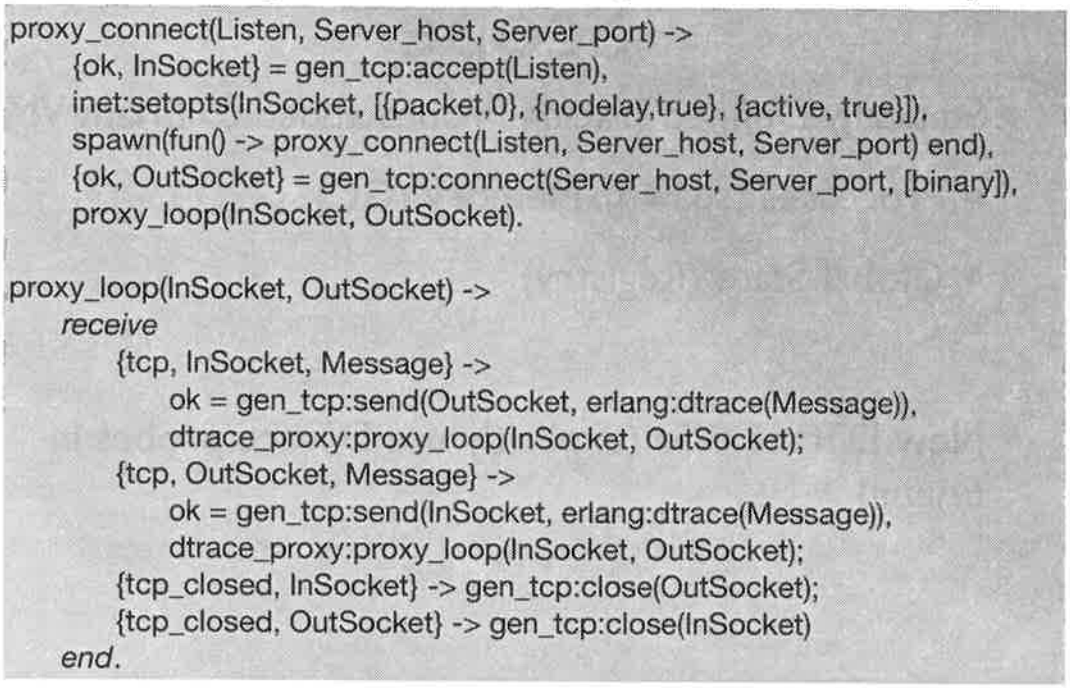

Adium Jabber/XMPP Client

Erlang VM 'Proxy'

ejabberd Jabber/XMPP Server

DTrace

'Proxy' Code (trimmed)



```
proxy_connect(Listen, Server_host, Server_port) ->
    {ok, InSocket} = gen_tcp:accept(Listen),
    inet:setopts(InSocket, [{packet,0}, {nodelay,true}, {active, true}]),
    spawn(fun() -> proxy_connect(Listen, Server_host, Server_port) end),
    {ok, OutSocket} = gen_tcp:connect(Server_host, Server_port, [binary]),
    proxy_loop(InSocket, OutSocket).

proxy_loop(InSocket, OutSocket) ->
    receive
        {tcp, InSocket, Message} ->
            ok = gen_tcp:send(OutSocket, erlang:dtrace(Message)),
            dtrace_proxy:proxy_loop(InSocket, OutSocket);
        {tcp, OutSocket, Message} ->
            ok = gen_tcp:send(InSocket, erlang:dtrace(Message)),
            dtrace_proxy:proxy_loop(InSocket, OutSocket);
        {tcp_closed, InSocket} -> gen_tcp:close(OutSocket);
        {tcp_closed, OutSocket} -> gen_tcp:close(InSocket)
    end.
```

Future Directions

- Better use of existing Erlang Trace facilities
  - Dynamic DTrace Probes
- Correlate Messages across Erlang Processes
- Extend to Erlang Data Types (e.g. Lists) in DTrace ...
  - ... and not flatten to strings in probe code
  - Dynamic DTrace Type extensions
- Distributed/Clustered DTrace (one day ...)

Where we are now?

- It appears to work, and showing some promise
  - Lots more to do, and looking for help
- Google Group: Erlang-DTrace
- Source will be at opensolaris.org
- Tim Becker & Garry Bulmer can be reached at that group
- Thanks to Bryan Cantrill, Sun Microsystems for encouragement and support

DTrace

- Mac OS X 10.5
- Solari/OpenSolaris
- FreeBSD
- QNX

- Maybe Vtrace

Questions or Feedback?

Gradual Typing of Erlang Programs

Kostis Sagonas

Main message

This talk aims to document and promote a different mode of Erlang program development:

- one where most typos, interface abuses, type errors, etc. are *identified automatically* using static analyzers
- one where *type information becomes part of the code* and *checked for definite violations* after program modifications
- one where all the above are *optional*, can take place *gradually*, and can be *refined at any point* to the extent desired by the programmer

Practice and experience

- We have been practicing this development mode in large Erlang code bases:
  - `dialyzer`
  - `typer`
  - `hipe` (a very large part)
  - `stdlib` & `kernel` (many key modules)
- Also tried it in code with which we were not familiar – see the paper @ Erlang'08

Step #1

Use Dialyzer

Wrangler 0.1

- Released January 2007
- 25 modules
- 35,000 lines of code
- Many modules are slight modifications or clones of Erlang/OTP ones – mainly of `syntax_tools`

Dialyzer on Wrangler 0.1

- Run as simply as

```
> cd distel-wrangler-0.1/wrangler
> dialyzer --src -c *.erl
```

- 67 warnings in less than 2 minutes
- about 50 of them due to abuse of `file:open/2`

`file:open(Name,read)` vs. `file:open(Name,[read])`

- After fixing this and one similar interface abuse, 15 warnings remain
  - all genuine bugs

Can you spot the bug?

```
handle_call(Call, DefinedVars, State) ->
  ...
  case is_c_atom(Mod) andalso is_c_atom(Fun) of
    true ->
      M = atom_val(Mod),
      ...
      case {M_Loc, Call_Loc} of
        {{L1, C1}, {L2, C2}} ->
            if (L1 < L2) or
               ((L1==L2) and ((C2-C1) > length(M)))
      ...
```

```
refac_atom_info.erl:715:
  Guard test length(M::atom()) can never succeed
```

Can you spot the bug?

```
get_new_name(Sub, NewRegExp) ->
  Index = string:str(NewRegExp, "*"),
  case Index of
    0 -> NewRegExp;
    N ->
      Prefix = string:sub_string(NewRegExp, 1, N-1),
      case Sub of
        [] -> exit(error, "Cannot infer ...");
        _ -> Sub1 = hd(Sub),
             get_new_name(tl(Sub), Prefix++Sub1++...)
      end
  end.
```

```
refac_batch_rename_mod.erl:161:
  The call erlang:exit('error',string()) will fail
  since it differs in argument 1 from the success
  typing arguments (pid() | port(),string())
```

Can you spot the bug?

```
expand_files([File|Left], Ext, Acc)  ->
  case filelib:is_dir(File) of
    true ->
       ...
    false ->
       case filelib:is_regular(File) and
            filename:extension(File) == Ext of
         true -> expand_files(Left,  Ext, [File|Acc]);
         false -> expand_files(Left, Ext, [File])
       end
  end;
```

```
refac_util.erl:1322:
   The call erlang:and(bool(),[integer()]) will fail
   since it differs in argument position 2
   from the success typing arguments: (bool(),bool())
```

Wrangler 0.3

- Released January 2008 – one year after 0.1
- 25 modules
- 27,000 lines of code

Dialyzer on Wrangler 0.3

- Run as simply as

```
> cd distel-wrangler-0.3/wrangler/erl
> dialyzer --src -I ../hrl -c *.erl
```

- Analysis takes 50 secs – produces many warnings
- Many due to `file:open/2` and due to confusing `lists:concat/1` with `lists:append/1`
- After fixing these, 10 warnings remain
  - all genuine bugs
  - two of them are remains from Wrangler 0.1
  - not very surprising: they are in uncommon code paths

Step #2

Expose type information:
make it part of the code

Exposing type information

Can happen in either of the following ways:

- Add explicit type guards in key places in the code
  - Ensures the validity of the information
  - Has a runtime cost – typically small
  - Programs may not be prepared to handle failures
- Add type declarations and contracts
  - Documents functions and module interfaces
  - Incurs no runtime overhead
  - Can be used by dialyzer to detect contract violations

Turning @specs into -specs

Often Edoc @spec annotations

```
%% @spec batch_rename_mod(OldNamePattern::string(),
%%                        NewNamePattern::string(),
%%                        SearchPaths::[string()]) ->
%%        ok | {error, string()}
```

Can easily be turned into -spec declarations

```
   -spec batch_rename_mod(OldNamePattern::string(),
                          NewNamePattern::string(),
                          SearchPaths::[string()]) ->
          'ok' | {'error', string()}.
```

Turning @specs into -specs

In some other cases

```
%% @spec duplicated_code(FileName :: filename(),
%%                       MinLines :: integer(),
%%                       MinClones:: integer()) -> term()
```

Type declarations are also required

```
 -type filename() :: string().
 -spec duplicated_code(FileName :: filename(),
                       MinLines :: integer(),
                       MinClones:: integer()) -> term().
```

Turning @specs into -specs

A problem with Edoc annotations is that often they are not in accordance with the code

- Not surprising – they are comments after all!

For example, to be correct, let alone precise, the previous case should read:

```
-type filename() :: string().
-spec duplicated_code(FileNames::[filename()],
                      MinLines ::[integer()],
                      MinClones::[integer()]) -> term().
```

How to turn @specs into -specs

Option 1: Convert @specs into -specs in one go

- Brave and quick
- Typically not a good idea: results in many Dialyzer warnings which may be hard to debug

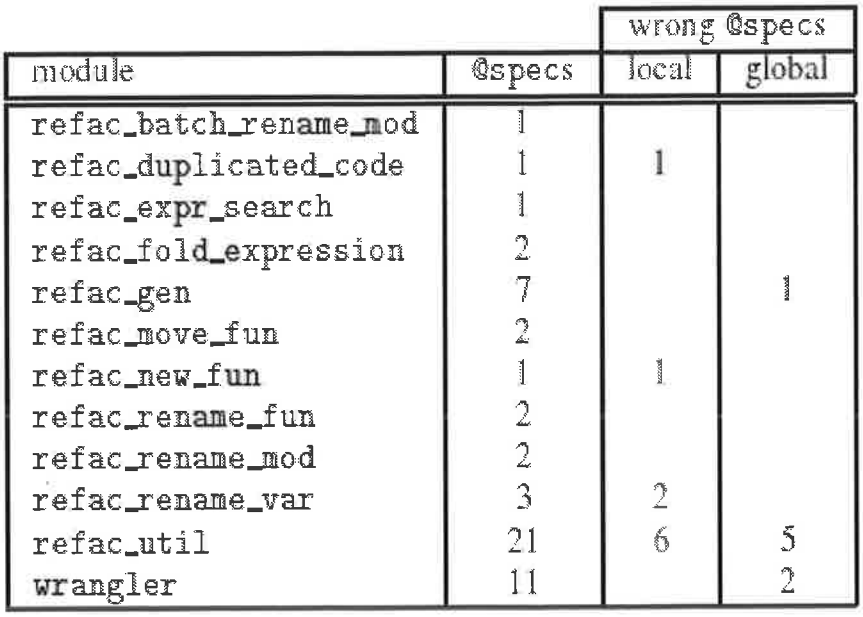
Experiment: 162 warnings on the code of Wrangler 0.3

Option 2: Convert @specs gradually and fix the erroneous ones using Dialyzer

- First locally (on a module-by-module basis)
- Then globally

→ **We strongly recommend Option 2**

Wrong @specs in Wrangler 0.3



		wrong @specs	
module	@specs	local	global
refac_batch_rename_mod	1		
refac_duplicated_code	1	1	
refac_expr_search	1		
refac_fold_expression	2		
refac_gen	7		1
refac_move_fun	2		
refac_new_fun	1	1	
refac_rename_fun	2		
refac_rename_mod	2		
refac_rename_var	3	2	
refac_util	21	6	5
wrangler	11		2

Table 2. Wrong @specs in Wrangler 0.3; blank entries denote 0

Step #3

Fix bugs exposed by
`-spec` declarations

Step #4

Strengthen and factor
`-type` declarations

Strengthening -type declarations

- Type declarations can be refined to the extent desired by the programmer

```
-type pos() :: any().
```

```
-type pos() :: tuple().
```

```
-type pos() :: {any(), any()}.
```

```
-type pos() :: {number(), number()}.
```

```
-type pos() :: {integer(), integer()}.
```

```
-type pos() :: {0..1000000, 0..200}.
```

Step #5

Strengthen underspecified -spec declarations

Strengthening underspecified -specs

Can take place semi-automatically using Dialyzer

```
> dialyzer -Wunderspecs --src -I ../hrl -c *.erl
```

```
refac_duplicated_code.erl:53:
  Type specification for duplicated_code/3 ::
    ([filename()],[integer()],[integer()]) -> term()
  is a supertype of the success typing:
    ([string()],[integer()],[integer()]) -> {'ok',string()}
```

Step #6

Add -spec declarations
for all exported functions

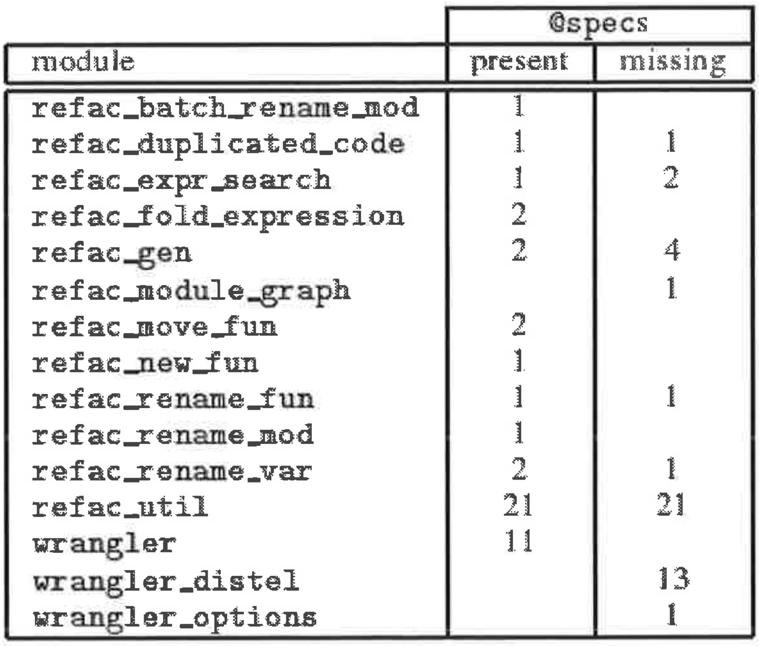
Adding missing -specs

Can take place semi-automatically using Typer

```
> erlc +warn_missing_spec -I../hrl refac_rename_var.erl
./refac_rename_var.erl:166: Warning:
    missing specification for function pre_cond_check/4
```

```
> typer --show-exported -I../hrl refac_rename_var.erl

%% File: "refac_rename_var.erl"
%% ----------------------------
-spec pre_cond_check(tuple(),integer(),integer(),atom()) -> bool().
-spec rename(syntaxTree(),pos(),atom()) -> {syntaxTree(),bool()}.
-spec rename_var(filename(),...,[string()]) ->
              {'ok',string()} | {'error',string()}.
```

Missing @specs in Wrangler 0.3



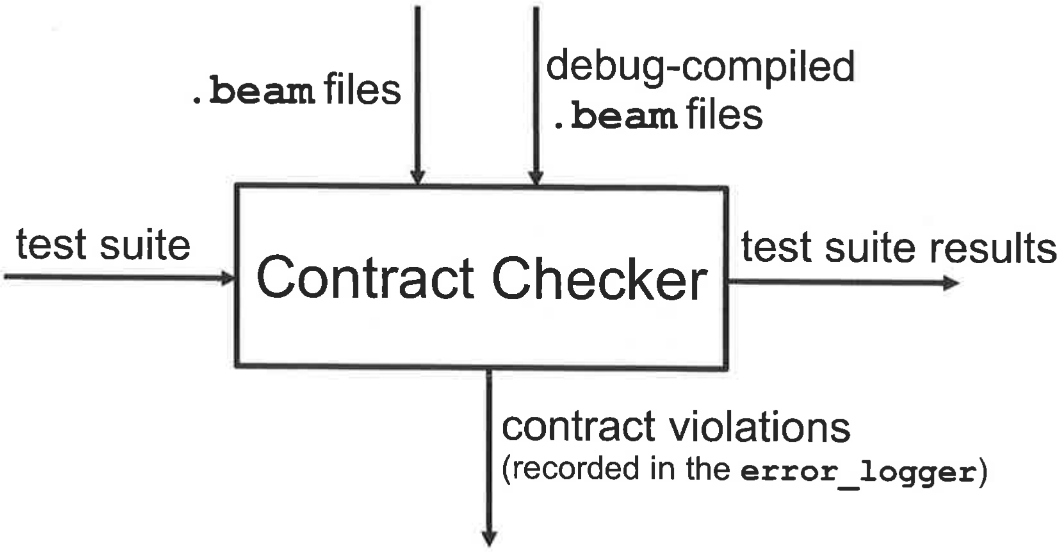
	@specs	
module	present	missing
refac_batch_rename_mod	1	
refac_duplicated_code	1	1
refac_expr_search	1	2
refac_fold_expression	2	
refac_gen	2	4
refac_module_graph		1
refac_move_fun	2	
refac_new_fun	1	
refac_rename_fun	1	1
refac_rename_mod	1	
refac_rename_var	2	1
refac_util	21	21
wrangler	11	
wrangler_distel		13
wrangler_options		1

Table 3. Number of existing and missing specs for all exported functions of Wrangler 0.3 modules; blank entries denote 0

Step #7

Test the validity of contracts using runtime monitoring

Testing for contract violations



Out of the 106 **`-spec`** declarations of Wrangler

- 55 were exercised by the test suite
- 4 of them were detected as erroneous

Concluding remarks

- Described a methodology for how to:
  - use static analysis for detecting definite type errors
  - add type information to existing Erlang applications
  - become confident about the validity of that information
- Showed both the benefits and common pitfalls of the approach on a non-trivial case study
- Type information is not a panacea but makes code more robust, easier to understand and maintain

Testing a SIP decoder with QuickCheck - extended abstract

Hans Nilsson, Ericsson
Hans.R.Nilsson@ericsson.com

October 30, 2008

1 Introduction

SIP[4] is a protocol that recent years has gained high attention in the telecoms industry for services in connection with telephony over IP.

The messages in the protocol are text based and the syntax is defined by Augmented BNF[2] in RFCs. Unfortunately, the SIP grammar is not suitable as input to a traditional parser generator without radical re-writing. Left for the decoder/encoder implementor is a monotone programming session lasting for weeks.

The result is like all such code usually full of errors which makes systematic testing necessary. This paper describes a successful automatic test of a SIP decoder/encoder using QuickCheck[1]. The test code was generated from the BNF grammar and was therefore free from the test case implementators miss-understandings and errors.

In traditional testing, a number of test cases are programmed where the test object is given known input and the result is compared to the known output in the test case. This has some disadvantages. Only the cases that the tester can imagine and write as a program are tested. Usually only a few parameters in the test cases are varied in such code and only within the limits the tester believes are relevant.

There are alternatives gaining a growing interest. One is *property based testing* where the tester specifies the properties the test object shall have. The test system generates test cases to show that the properties are fulfilled. In the case of QuickCheck, the test input is randomly generated according to the specification. When an error is found, so called *shrinking* is applied and the tester is presented a *minimal* example that triggers that error.

Property based testing therefore generates a large number of test cases, often with value combinations that are a surprise for a human reviewing the generated test data. In that way a very good coverage is obtained.

2 The test

With QuickCheck the tester has to supply two things:

1. a *generator* that specifies the type of test input data that QuickCheck shall randomly generate

2. a *property* that decides if a result is valid or not with the actual input

In a simple example - a square function - the generator could be the QuickCheck built-in function generating integers. Those integers are given as arguments to the square function to be tested. The property could be very simple, for example just testing that the function return value is a positive integer or be more exact and really checking that the value is the square of the input.

For the SIP decoder/encoder there were two alternatives: either generate the internal Erlang form or the external text form. The text form was selected because that generator could be automatically obtained from the BNF in the SIP specifications.

The property shall test, that for all SIP messages M:

1. the semantics of M in text format is the same as the semantics of M decoded into the internal representation

2. the semantics of M in the internal representation is the same as M encoded into the text format

Why not just compare the syntax? Obviously the internal form differs from the text form. Two different messages in text form could however be semantically equal due to differences in case, number of blanks, line breaks and even line ordering. By some kind of normalization it could be possible to compare the syntax. Such a normalization is not trivial, and is probably error prone.

To avoid trying to extract the semantics out of both text form and internal form or to write a normalizer, a simpler but anyway useful middle way was chosen:

```
decode(M) == decode(encode(decode(M)))
```

where M is a message generated in text format by QuickCheck.

This is actually a syntactical normalization of the messages, since the decoder is such that differences in case etc are lost. Surely some test cases are missed but the testing is hopefully "good enough" - more about this later.

2.1 Example of generators and properties

The BNF is large - about 1000 lines in 20 RFCs. As an example of the direct generation[1] of code from BNF we could look at a sligthly edited example from [4]:

```
SIP-message     =  Request / Response
Request         =  Request-Line
                   *( message-header )
                   CRLF
                  [ message-body ]
Request-Line    =  Method SP Request-URI SP SIP-Version CRLF
Method          = REGISTERm / INVITEm / ACKm / OPTIONSm
                 / BYEm / CANCELm / REGISTERm / extension-method
INVITEm            =  %x49.4E.56.49.54.45 ; INVITE in caps
```

[1] Thanks to Joe Armstrong who gave me a BNF parser

```
extension-method  =  token
token             =  1*tok-char
tok-char          = (alphanum / "-" / "." / "!" / "%" / "*" / "_"
                    / "+" / "`" / "'" / "~" )
```

Naively converted to generators this will be :

```
eSIP_message() -> oneof([eRequest(), eResponse()]).
eRequest() -> [eRequest_Line(),
               list_of(emessage_header()),
               eCRLF(),
               oneof([[], [emessage_body()]])
              ].
eRequest_Line() -> [eMethod(), eSP(), eRequest_URI(),
                    eSP(), eSIP_Version(), eCRLF()].
eMethod() -> oneof([eREGISTERm(),eINVITEm(),eACKm(),eOPTIONS(),
                    eBYE(),eCANCEL(),eREGISTER(),eextension_method()]).
eINVITEm() -> "INVITE".
etoken() -> [etok_char() | list_of(etok_char())].
etok_char() -> oneof([ealphanum(),45,46,33,37,42,95,43,96,39,126]).
eextension_method() -> etoken().
```

where `oneof/1` is a QuickCheck function that selects on of the elements in the list in the argument and `list_of/1` generates a list (maybe empty) where the elements is generated by the generator in the argument of `list_of/1`.

The property to test is basically:

```
prop_sip_decode_encode() ->
     ?FORALL(M, eSIP_message(),
             decode(M) == decode(encode(decode(M))) ).
```

The `?FORALL(Var, Generator, Property)` macro is a QuickCheck provided macro that generates a value by calling `Generator`, assigns it to `Var` and finally calls `Property` which returns `true` if the property is fulfilled.

3 Problems

There are of course some problems with the naive generation:

1. The BNF is not always correct. `"1:FF"` is for example a correct IPv6 address according to both the SIP grammar[4] and the IPv6 grammar[3]!
2. Some constructs results in infinite loops
3. Different branches in the BNF tree have different sizes of the sets of possible values, but the branches have same probability to be chosen when a value is to be generated. This gives the unwanted situation that some test cases will have higher probability than others. In the worst case, some constructs could be left untested while others are tested more than once.

4. When calling `eSIP_message/0`, *all* generator functions will be called. The resulting data structure and fun's will have the possibility to generate *any* SIP construct. Out from this, QuickCheck will only use one path and the rest will be thrown away.

The obvious solution for the BNF problems (1-2) is to re-write the BNF. For the unbalanced probability problem 3), the solution[2] is to add weights depending on the sub tree sizes. The size was simply calculated as 1 for a leaf and as the sum of all sub trees for an internal node. The QuickCheck function `frequency/1` takes as argument a list of {`Probability, Generator`} as argument, and selects one `Generator` depending on the `Probability`.

Problem 4) was solved with inserting the `LAZY` macro around all generator bodies. The result is that for a generator `g()` as argument, a `fun() -> g() end` will be returned instead. QuickCheck will then only eval `g()` if needed.

The resulting generator code for the example is:

```
eSIP_message() -> ?LAZY(frequency([{463,eRequest()},
                                   {377,eResponse()}])).
eRequest() -> ?LAZY([eRequest_Line(),
                     list_of_smaller(emessage_header()),
                     eCRLF(),
                     oneof([[], [emessage_body()]])
                    ]).
eRequest_Line() -> ?LAZY([eMethod(), eSP(), eRequest_URI(),
                          eSP(), eSIP_Version(), eCRLF()]).
eMethod() -> ?LAZY(oneof([eREGISTERm(),eINVITEm(),eACKm(),eOPTIONS(),
                          eBYE(),eCANCEL(),eREGISTER(),eextension_method()])).
eINVITEm() -> ?LAZY("INVITE").
etoken() -> ?LAZY([etok_char()|list_of_smaller(etok_char())]).
etok_char() -> ?LAZY(frequency([{5,ealphanum()},{1,45},{1,46},
                                {1,33},{1,37},{1,42},{1,95},
                                {1,43},{1,96},{1,39},{1,126}])).
eextension_method() -> ?LAZY(etoken()).
```

4 Results and discussion

Most important: a lot of errors was found. Many of them was in legal messages of strange sorts that normally would not have been tested. Even if such messages do not occur in practice, some components are sometime present. If such an error would have been left in the decoder, there would have been some sporadic and hard-to-catch errors left.

It is interesting that no errors was found in the encoder/decoder neither when it was tested conventionally nor during heavy usage in labs and at customer premises. Another part of the SIP stack was not tested by QuickCheck and was written by the same author. Conventional testing found some errors, but when tested by QuickCheck later, additional errors was found

The QuickCheck testing was extremely valuable and saved time and therefore money also for other parts of the system, since the decoder/encoder is very

[2] Thanks to John Hughes who actually pointed out this problem and the one in 4) for me and also solved them

central in the whole system. An error here will definitely stop most other test cases.

The approach of generating the generators from the "formal" BNF specification gave test code that is as correct as possible. The limited test in the property seem to have had no impact of the final test quality.

References

[1] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

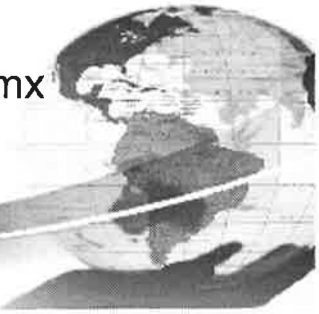
[2] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF, 1997.

[3] R. Hinden and S. Deering. IP version 6 addressing architecture (RFC2373), 1998.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session initiation protocol (RFC3261), 2002.

An Augmented Backus-Naur Format, (ABNF), Parser Generator for Erlang

Anders Nygren
anygren@txm.com.mx



www.txm.com.mx

Contents

- ABNF
- Using abnfc
- Implementation
- Todo

txm

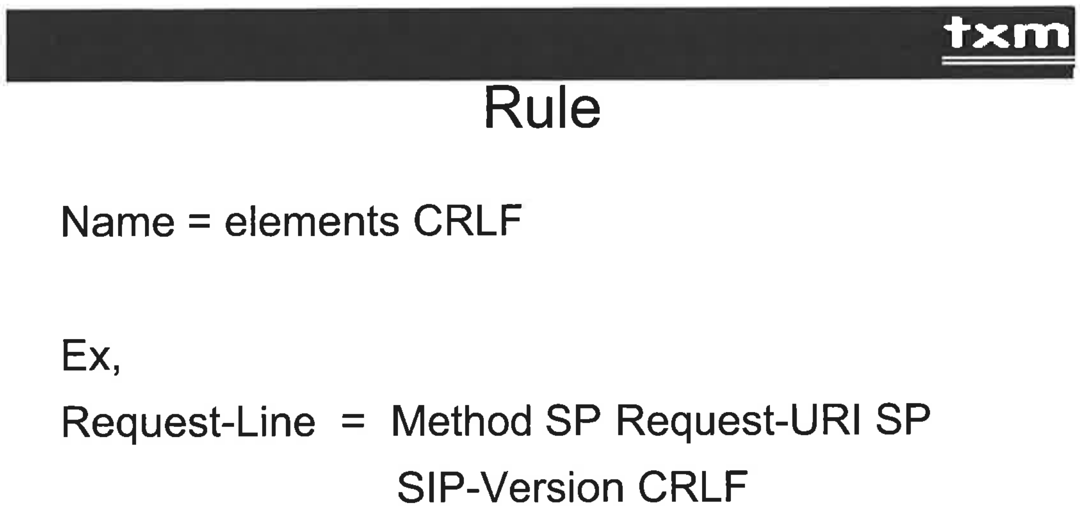
Why abnfc?

- ABNF used for specifying many important protocols, e.g. HTTP, SIP, SDP
- Handwritten parsers are
  - A lot of work
  - Error prone
- Not practical, (impossible?), to use yecc

txm

What is ABNF?

- Augmented Backus-Naur Form
- Used by IETF for specifying protocols
- Initially informally defined in the RFCs where it was used
- later defined in a series of RFCs, currently in RFC 5234

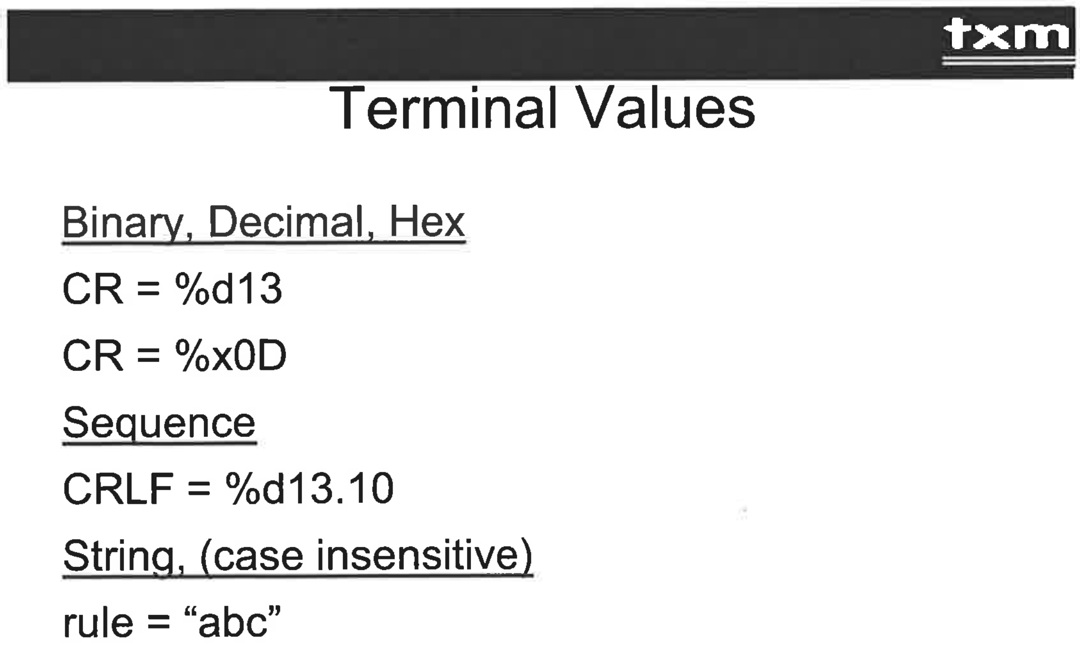


Rule

Name = elements CRLF

Ex,

Request-Line = Method SP Request-URI SP
SIP-Version CRLF



Terminal Values

Binary, Decimal, Hex

CR = %d13

CR = %x0D

Sequence

CRLF = %d13.10

String, (case insensitive)

rule = “abc”

Concatenation

Rule = Rule1 Rule2 … RuleN

```
foo = %x61    ; a
bar = %x62    ; b
mumble= foo bar foo
```

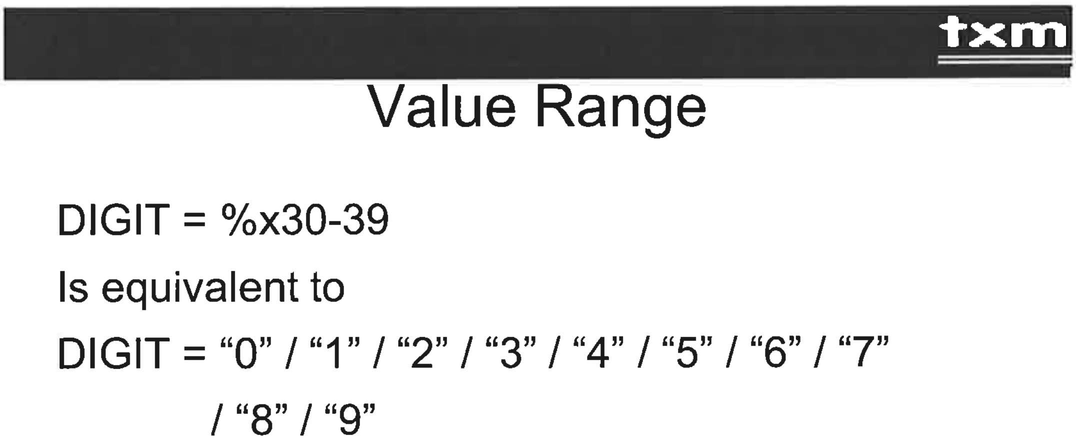
Accepts “aba”

Alternation

Rule = Rule1 / Rule2 / … / RuleN

```
foo = %x61    ; a
bar = %x62    ; b
A-or-b = foo / bar
```

Accepts “a” and “b”

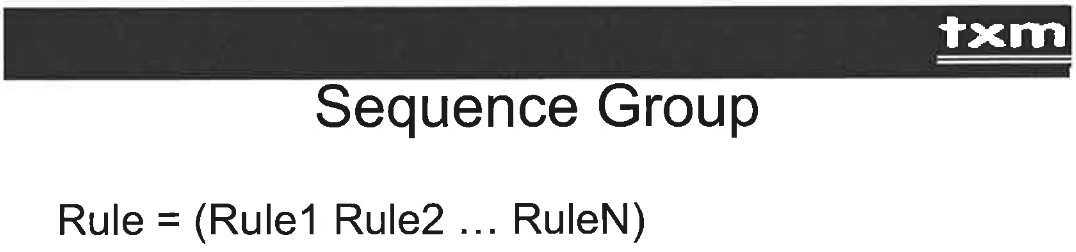


Value Range

DIGIT = %x30-39

Is equivalent to

DIGIT = “0” / “1” / “2” / “3” / “4” / “5” / “6” / “7”
/ “8” / “9”



Sequence Group

Rule = (Rule1 Rule2 … RuleN)

Repetition

*rule	; 0 or many occurrences
<n>*rule	; n or more occurrences
*<m>rule	; 0 to m occurrences
<n>*<m>rule	; n to m occurrences
<n>rule	; exactly n occurrences, equivalent to <n>*<n>rule

Optional

[foo bar]

Equivalent to

0*1(foo bar)

“Imported” Rules

Rules defined in other RFCs are frequently reused.

But there is no formal way of specifying the imports. Normally it is done with

- A comment in the ABNF specification
- A note in the RFC text

Using abnfc

Grammar definition file: my_mod.abnf

Name = elements : Erlang_Code.

Ex.

```
callid   =  word [ "@" word ] : lists:flatten(_YY).
```

Variable Bindings

In the Erlang code the following variables are available

- _YY : Bound to the complete match for the rule
- _YY1 to _YYn : Bound to each part of the match

ex.

opt-name = “(“ name “)” : {opt_name, _YY2}.

Variable Bindings

```
Allow  =  "Allow" HCOLON [Method *(COMMA
  Method)] :
     Allowed = case _YY3 of
              [] -> [];
        [[M1,Ms]] -> [M1|[M||['COMMA',M]<-Ms]]
              end,
     {'Allow', Allowed}.
```

Implementation

Three main parts

- Syntax specification parser
- Transformations/optimizations
- Code generation

Syntax Spec. Parser

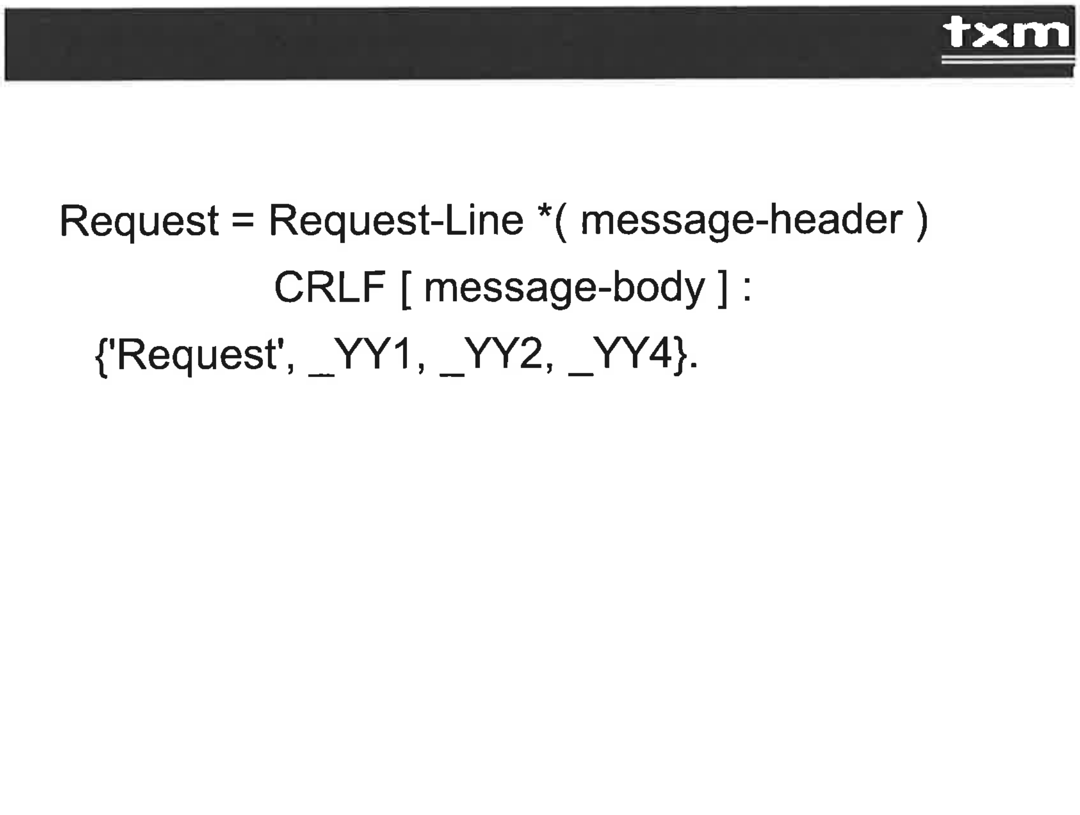
- Originally hand written using parser combinators
- Later generated by abnfc, (ABNF is specified in ABNF)
- Generates a simple AST

Transformations

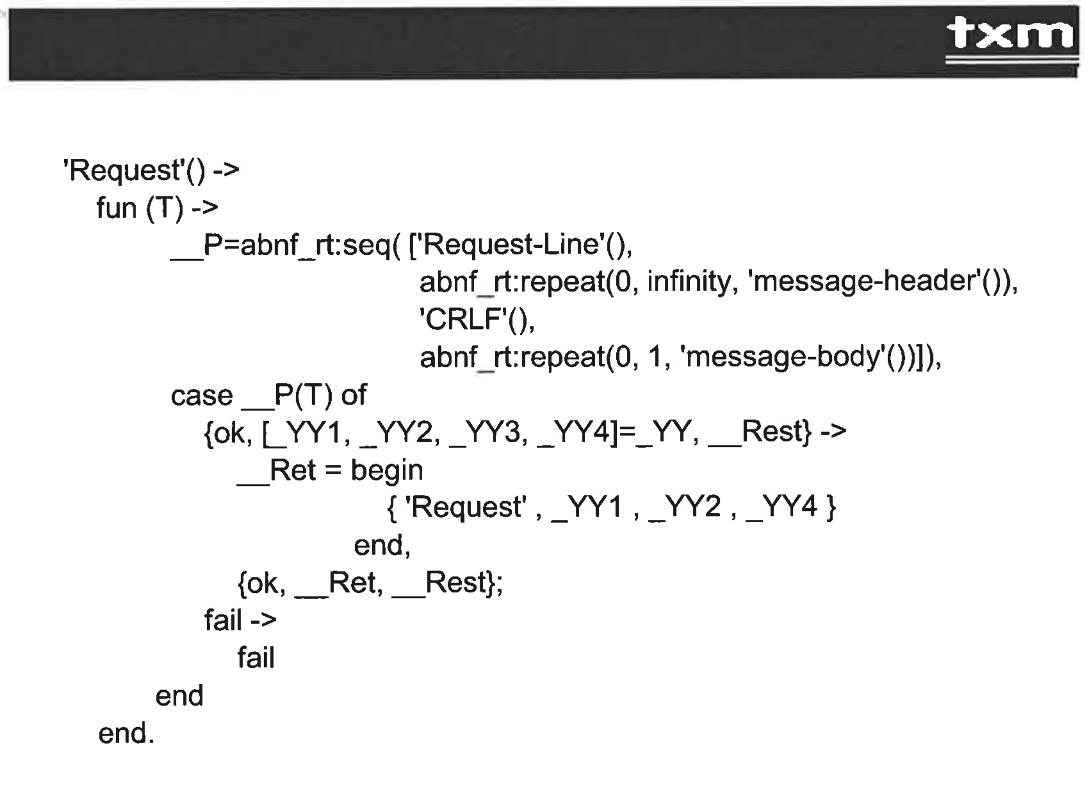
- Convert AST to an internal representation
- A number of passes that performs transformations on the internal format
  - Remove {repeat, 1, 1}
  - Remove alternation with only one element
  - Remove concatenation with only one element
  - Merge num_val elements, (pending)
  - Inline imported rules when possible to improve optimizations, (pending)

Code generation

- The generated code currently uses parser combinators
  - Simple
  - Not very efficient
- Parser functions for all rules are exported
- A my_mod.hrl is always included, this makes it possible to import rules from other modules



```
Request = Request-Line *( message-header )
                CRLF [ message-body ] :
  {'Request', _YY1, _YY2, _YY4}.
```



```
'Request'() ->
   fun (T) ->
         __P=abnf_rt:seq( ['Request-Line'(),
                           abnf_rt:repeat(0, infinity, 'message-header'()),
                           'CRLF'(),
                           abnf_rt:repeat(0, 1, 'message-body'())]),
         case __P(T) of
            {ok, [_YY1, _YY2, _YY3, _YY4]=_YY, __Rest} ->
               __Ret = begin
                          { 'Request' , _YY1 , _YY2 , _YY4 }
                       end,
               {ok, __Ret, __Rest};
            fail ->
               fail
        end
   end.
```

txm

Limitations

- Limited backtracking
  - Some workarounds
    - Reorder alternatives
    - Handwritten parsers for difficult cases
    - Modify rules and do validation in Erlang
- No error messages

txm

Todo

- More transformations/optimizations
- Better code generation
- Support older ABNF syntax
- Generate encoding functions (?)

Expertise Makes it simple

Anders Nygren
SL, product development and solutions
Telexpertise de México S.A. de C.V.
Phone: +52 (844)438-4800
anygren@txm.com.mx
www.txm.com.mx

txm www.txm.com.mx

Autocoding State Machine in Erlang: A Case Study of Model-Driven Software Development

Yu Guo

Mads Clausen Institute
University of Southern Denmark
Soenderborg, Denmark

Torben Hoffmann Nicholas Gunder

Motorola A/S
Glostrup, Denmark

Abstract

This paper presents an autocoding tool suit, which supports development of state machine in a model-driven fashion, where models are central to all phases of the development process. The tool suit, which is built on the Eclipse platform, provides facilities for the graphical specification of a state machine model. Once the state machine is specified, it is used as input to a code generation engine that generates source code in Erlang.

Keywords: model-driven development, state machine, autocoding, Eclipse, tools

1 Introduction

In Motorola's TWSD organisation, we have been working with Erlang/OTP for a while now and the need for a higher level of abstraction than code has surfaced a couple of times.

The Erlang/OTP code is very clean, but sometimes it is a lot easier to communicate using pictures and models. It is always a practical problem to keep the pictures and models in sync with the code, so any tool support which can help out with that would be appreciated. Even though writing Erlang/OTP code is a lot faster than doing the same code in other languages, you still have to write some boilerplate code to implement a component using Erlang/OTP. As a first step, a state machine model is typically conceptualized in some form before this is done. So a method that could auto-generate code from a state machine model offers some benefits to an Erlang developer.

The semantic gap between Erlang/OTP state machines and formal models of state machines is quite small, and this poses a tough requirement on any modeling tool: To make the tool more useful than writing the code by hand.

The scientific work of Mads Clausen Institute for Product Innovation, University of Southern Denmark was brought to the attention of Motorola, and the potential for solving the problems outlined above seemed so promising that a case study was initiated. [11]

Traditional methods used to develop software are plagued by the problem that the implementation is not always consistent with the specification. Model-driven software development [1][3] seems promising to give the solution, since the implementation can be derived from, or generated directly from the specification. The method requires a modeling language for specifying the application and a code generation engine that translates application models into code. However, it needs adequate tools that automate the steps between specification and implementation.

This paper is intended to give a solution of above problem. When thinking of model-driven software development, the immediate understanding is that models drive the development of the state machine, in the sense that the state machine is constructed by transforming models from higher levels of abstraction to the point where we reach a piece of executable code.

The *autocoding tool suit* is built on the Eclipse platform. Thanks to the wealth of modeling approaches the platform supports, most of which are based on well-established and popular projects. We use the Eclipse Modeling Framework (EMF) project as the modeling facility and the Eclipse Graphical Modeling Framework (GMF) project to provide a graphical modeling environment. The tool supports both a textual notation as well as a visual one. The Acceleo can be fruitfully exploited for a transformation engine to develop the tool for the code generation. It has built-in facilities to read models that support the smooth integration with modeling tools in the Eclipse. Developed in such way, the state machine autocoding tool suit contains a set of Eclipse plug-ins, therefore, a uniform development environment can be obtained.

The rest of the paper is structured as follows: Section 2 presents state machine used in Erlang. Section 3 deals with the metamodel and constraints of the state machine. Section 4 describes the generation template. The implementation in Eclipse is presented in Section 5. Section 6 discusses the development process using the tool. A future work is discussed in Section 7, and a summary is given in the concluding section of the paper.

2 Finite state machine in Erlang

The finite state machine used in Erlang is described as a set of relations of the form:

```
State(S) x Event(E) -> Actions(A), State(S')
```

The relations are interpreted as meaning: If we are in state `S` and the event `E` occurs, we should perform the actions `A` and make a transition to the state `S'`.[4]

The finite state machine can be further considered as a *mealy state machine* [14], where an output (or action) is generated based on its current state, and an input (or event). A transition is marked with a trigging event, a guard expression and an action. The guard is a boolean expression. A developer should think in the following manner: if an event associated to a transition occurs, and the guard on the transition is satisfied, the transition is fired. Consequently, the state machine reacts to the event by performing the action on the transition; state maybe changed, too.

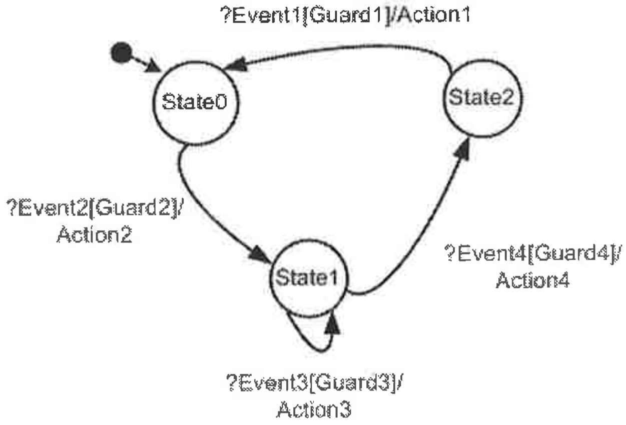


Fig. 1. The mealy state machine model

According this state machine model, for each transition of a state, a transition clause should conform to the following convention, when using the Erlang gen_fsm behaviour to implement the state machine:

```
StateName(Event, StateData) when Guard ->
    ...  code for actions here ...
{next_state, NextStateName, NextStateData}
```

3 Metamodeling the state machine

Metamodeling plays a fundamental role when using the model-driven software development approach. A metamodel describes possible structure of models, by defining the constructs and their relationship of the modeling language, as well as constraints. It is also the basis for building tools, concerning construction of the state machine model, validation of models against constraints, as well as generation of code. A constraint specifies a restriction of the metamodel element it is applied to. It can be written in natural language or in the Object Constraint Language (OCL) [5].

To define a metamodel, a metamodeling language is required. The Eclipse Modeling Framework Project provides facilities to create a metamodel, with the support of a meta-meatmodeling language – *the Ecore metamodel.* The language is considered to be at the M3 layer of the Meta-Object Facility (MOF) Four Layer Metadata Architectures [2]. In EMF, a metamodel described by the Ecore metamodel, known as *Ecore model*, contains structural requirements and constraints for the model, which are information contents that the model editor will manipulate. It extends the Ecore metamodel by instantiating classes, in the sense that a new class with new attributes in the Ecore model is created as an instance of an existing one defined in the Ecore metamodel. A metamodel is at the M2 layer of the MOF Four Layer Metadata Architectures. (see Fig. 2)

The Fig. 3 presents the metamodel of the mealy state machine. The metamodel is fairly intuitive and easy to understand. The state machine container is the root

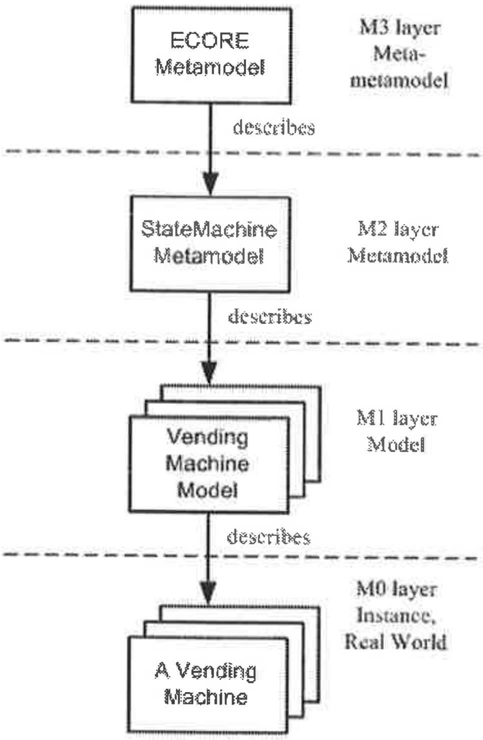


Fig. 2. MOF Four Layer Metadata Architectures

element of all instances. It must contain one state machine instance, and an arbitrary number of action as well as event's instances. The instances of the action and the event are referenced by transitions of the state machine model.

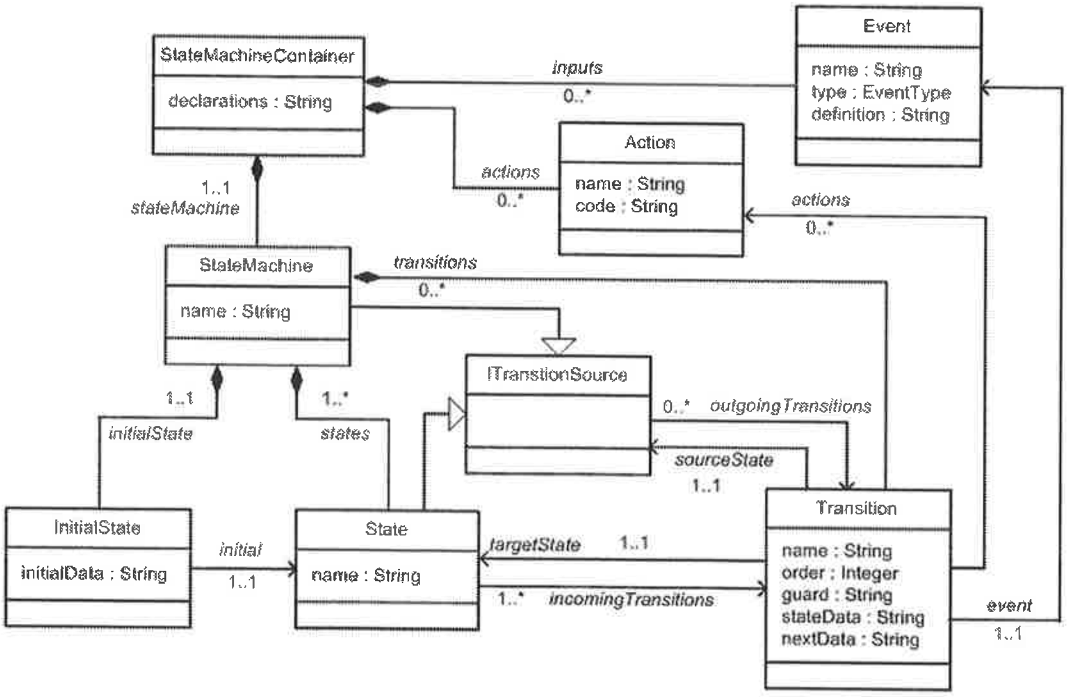


Fig. 3. The state machine metamodel

Typically, a transition has state instance as "sourceState" or "targetState". But, the state machine in Erlang has the feature that an event can arrive at any state. The event is sent with the `gen_fsm:send_all_state_event/2` instead of sending the event with `gen_fsm:send_event/2`, so that only one clause `Module:handle_event/3` is needed to handle the event [4]. In order to model this feature, an event-triggered transition can have the state machine as source, meanwhile, the type of the event on this kind of transitions must be "all_state_event". That is why both the "StateMachine" and the "State" class implement the interface "ITransitionSource".

The state machine in Erlang is deterministic, as the sequence of the clause for each transition matters while executing the program. The clauses are to be matched according to the order of precedence in the source file. If the first match failed, this second one will be picked. This is why an "order" attribute is used on the transition of the state machine model. It is an integer value. All outgoing transitions of a state must have different order. It determines the appearance sequence of the transition clauses in the source file at the generation stage.

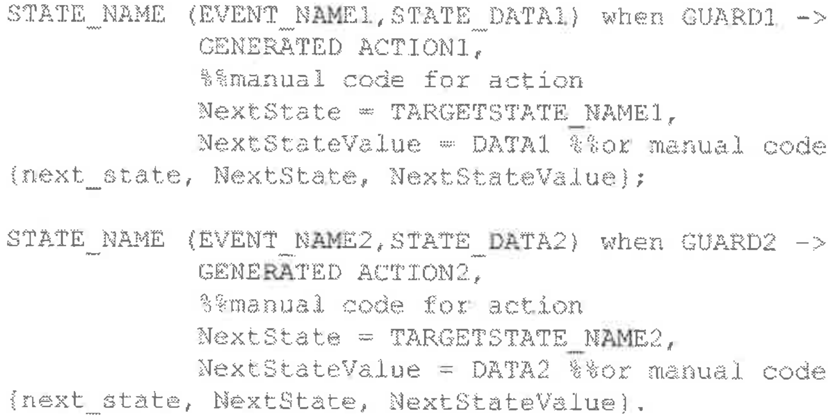
To obtain a complete domain model, the metamodel needs to be accompanied with constraints. It will not allow you to perform a code generation without doing the series of static checks. The constraints in a model-driven fashion are categorized into two levels: the platform independent level and the platform specific level. The platform independent constraints have not concerns with which target code is going to be generated, whereas the platform specific ones are bound to the target language, in this case study – the Erlang language. But in case of creating an Erlang state machine specific model, constraints from two levels can be combined.

For instance: according to the metamodel, both "State" and "StateMachine" are subclass of the "ITransitionSource" so that they can have outgoing transitions. However, the state machine turns into dead end if there were not any outgoing transition from a state. Thus a constraint like "each state has at least one outgoing transition" is a platform independent one. On the other hand, "the names of the states and the events must start with lowercase letter" is an Erlang platform specific constraint, since this rule has to be satisfied to make the generated code compliable. In case of C code generation from the state machine model, the constraint is not necessary.

4 Generation Template

After the metamodel has been built, a model, which is instance of the metamodel, can be used together with the templates for generation. Having models as central to all phases of the development, the generation process is independent from the concrete syntax of the model. No matter which format a model is stored as, the metamodel is always of special significance in the context of model-driven development. Generation templates should not be written based on some specific format that the model stored as, but on the metamodel.

To make the autocoding feasible, the way of programming a finite state machine needs to be normalized. The result of the normalization plays the role as the generation template. This step requires the answer of the question: where the static code, dynamic code and manual code are located in the template. The static code is always the same for all different models, whereas the dynamic code is the one transformed from the model.



```
STATE_NAME  (EVENT_NAME1,STATE_DATA1) when GUARD1 ->
            GENERATED ACTION1,
            %%manual code for action
            NextState = TARGETSTATE_NAME1,
            NextStateValue = DATA1 %%or manual code
(next_state, NextState, NextStateValue);

STATE_NAME  (EVENT_NAME2,STATE_DATA2) when GUARD2 ->
            GENERATED ACTION2,
            %%manual code for action
            NextState = TARGETSTATE_NAME2,
            NextStateValue = DATA2 %%or manual code
(next_state, NextState, NextStateValue).
```

Fig. 4. The generation template

The piece of pseudo code (Fig. 4) reveals the structure for one state with two outgoing transitions in the generated code. Words all in uppercase are dynamic code, which will be generated from the model. As mentioned in the previous section, the sequence of the two clauses depends on the value of the order attribute of outgoing transitions. The value of the variable `NextStateValue` could be generated from the “nextData” attribute of the transition if specified, or be written manually if it is empty, which is dependent of whether or not this value should be calculated by manual code. The `STATE_NAME` for two transition sections must be identical because they represent the same state. The rest dynamic code such as `EVENT_NAME`, `STATE_DATA` or `GUARD` could be either the same or different for each outgoing transitions. No manual code is allowed between the assignment of NextStateValue and the statement `next_state, NextState, NextStateValue`. It is only allowed to add manual code to the specified place. Otherwise, it will be lost in case of regeneration.

5 Implementation

The model-driven software development approach has been adopted to develop the autocoding tool suit in the Eclipse platform (www.eclipse.org), so as to reduce the amount of manual work needed to develop tools in a conventional manner. The Eclipse Graphical Modeling Framework project provides means to ease and speed up the development of graphical editors for modeling, which can be used for the rapid development of standardized Eclipse graphical modeling editors, by providing a generative component and runtime infrastructure for developing graphical editors [10].

The majority of the state machine model can be created graphically, but there is something that a graphical editor does not necessarily be applied. The Eclipse provides property sheet as complementary to editors, with basically two functionalities: set or display property of a model; create model instances that cannot (necessarily) be created using the graphical editors. Therefore models like actions and events can be instantiated within property sheets, otherwise the diagram that is supposed to emphasize the relation between states and transitions will be polluted with those less important ones to the visualization.

The Acceleo (www.acceleo.org) is an Eclipse based code generator transforming models into code. It works with any metamodel, implementing MOF as specified by the OMG (The Object Management Group). It needs templates that describe the information required to generate source code from a metamodel. Acceleo deals with incremental generation, by defining specific protected area. Code, which is modified by developers, is surrounded by special tags in the protected area. These tags do not pollute target code because they are implemented as explicit comments. On the next generation, the whole text in the protected area is kept.

A nice feature that the Acceleo offers is that Java functions can be invoked as services within the generation template. Acceleo is a hybrid with respect to the generation template. The advantages of an expressive template language and the power of the Java language are well combined, in such a way that parts of the template is written in the template language, while the complicated algorithm are implemented in Java.

Fig. 5 shows the graphical development environment in the Eclipse. A vending machine model is built with the editor. In order for a correct code generation process, the model has to be validated. The Fig. 6 presents the case when two states are identically named, thus, with the support of the GMF runtime, the models that violate the constraint are highlighted.

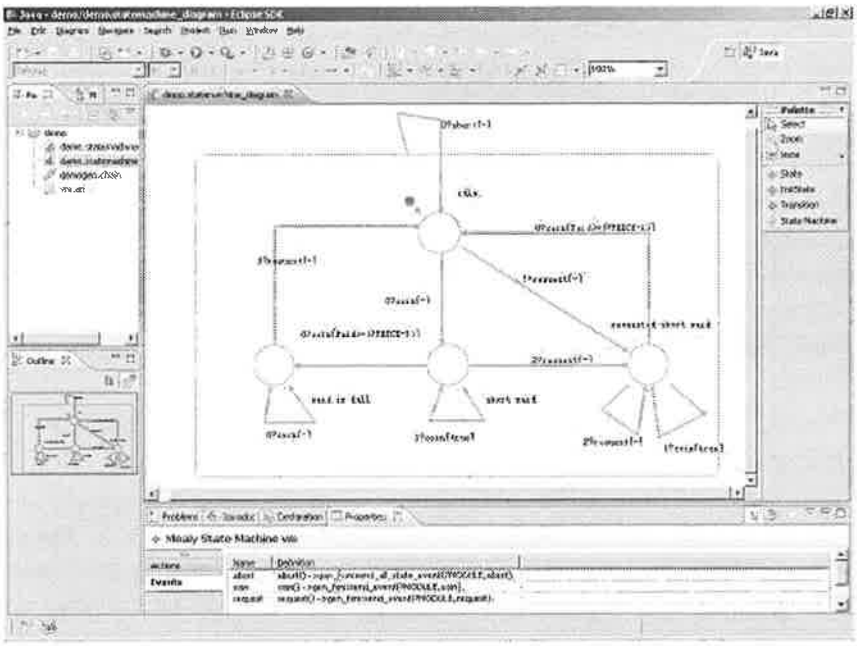
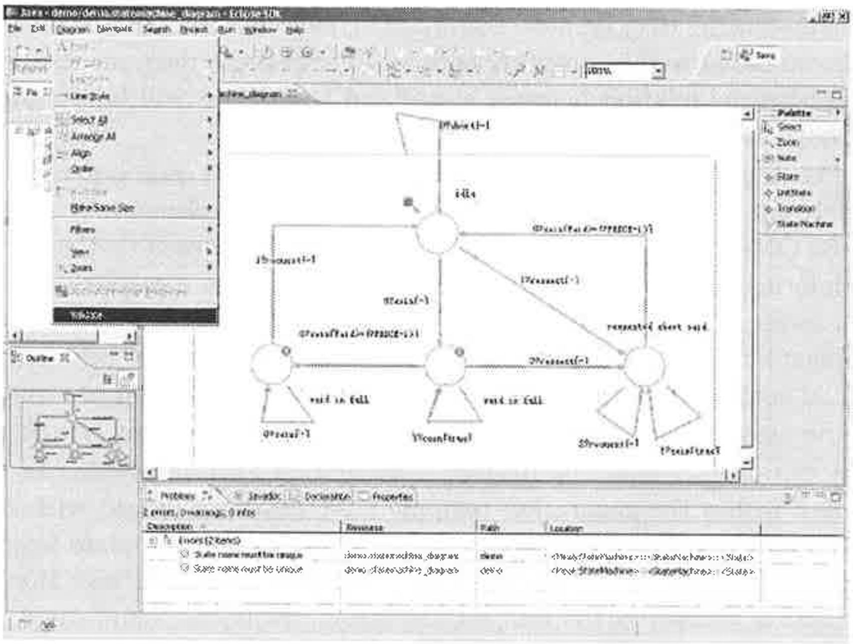


Fig. 5. The graphical development environment

Once the state machine model has been built, the tool can deserialize it into XML format, which gives the ability to maintain interoperability among tools based on the metamodel, as well as the 3rd party tools.



2 errors, 0 warnings, 0 infos

Description	Resource	Path	Location
Errors (2 items)			
State name must be unique	[illegible]	demo	[illegible]
State name must be unique	[illegible]	demo	[illegible]

Fig. 6. Violation of a constraint

State machine code will be generated then from the model. The vending machine code and model, as a reference, used for code generation in this case study is not something that Motorola markets, but the vending machine problem and original code is quite similar to most of the state machine code Motorola has written so the results from the case study applies to the real world code as well.

6 Development Process

It is well known that, even when using the model-driven development approach, some code still has to be handwritten. The handwritten code could come from any legacy code that has not been integrated into the model. While the state machine model can express when the actions are invoked, the implementations and algorithms of those actions are not represented. So, after generating the skeletons of the state machine, developer still needs to fill those actions skeletons with business logic.

Usually a balanced approach is considered to be the best, in the sense that generating the initial code base from the model and then just start from there. It can save some of the initial tediousness of translating the model to code. Users should not modify the generated code unless the changes are in the protected area. Otherwise, these changes will be lost in the next generation iteration.

The obvious way to lower the burden of the developer is to generate 100% code from the model. To achieve this goal, an iterative development process can be considered when using the tools. At the beginning, the developer should concentrate on the application logic to design the model. He needs to specify states, transitions, events as well as the names of the action, and put some dummy code (like comments) inside the action method. Once the state machine skeleton has been generated, he needs to fill out the actions method with the business logic code in the protected area.

When the state machine code passed tests, the handwritten code can be further integrated back into the model by replacing the dummy action code. (This has to be done manually, at the moment.) Consequently these code will be generated in the next generation. Finally, the developer will end up with a state machine model integrated with all action code. In this manner, it is possible to generate complete code if all actions are just predefined function calls from some library.

However, bear in mind that the complete model is not a platform independent model anymore. It becomes a platform independent model mixed with the platform specific code. In case of generation of another target code from this model, for example C code, the platform specific part in the model has to be changed.

It is possible to make improvement by introducing some kind of textual language that specifies the action. In this case the language has to be defined or even be invented. Meanwhile all the tooling needs to be created, too. It is not a trivial task since the semantics of the language has to be correct. As a result, the flexibility is obtained at the cost of the increased complexity.[3]

7 Future Work

The work that has been described in this paper is an initial step towards development tools for application in Erlang/OTP, in a model-driven fashion. It can be continued in several possible directions to further improve developer productivity by introducing more features as described in following sections.

7.1 Configuration Management

One of the biggest practical problems Motorola has experienced with various model driven approaches has been the management of different versions of a model component.

Being large organization with many development centers in the world, Motorola is challenged to share common tools and code with other teams who are not necessarily located in the same region. Typically, these teams have their own additions and changes, sharing a common code base. One problem deals with the task of being able to integrate individually created functionality, as well as that created by separate development centers, into a common code base. The fundamental problem resulting in this becomes an issue of potential rework and many headaches when the same code is changed by different people. Working at the source code level, this is a merging problem which requires a significant about of labor to do consistently. When moving to model-driven approaches, the problem does not go away – it is merely lifted to a higher level of abstraction, where models and changes to models need to be merged. This is a problem that so far has not been solved well enough to be of practical use. The practical problems in this has so far resulted in a situation where models are used to generate the first version of a component, and then all additions and changes are done at the source code level.

Clearly, there are two basic problems that need to be solved in order to make a model-driven approach a good fit for Motorola:

- Integration of manual changes back into the model. (Fig. 7)
- Merging of a new version of a model with the source code based on a previous model plus some manual changes. (Fig. 8)

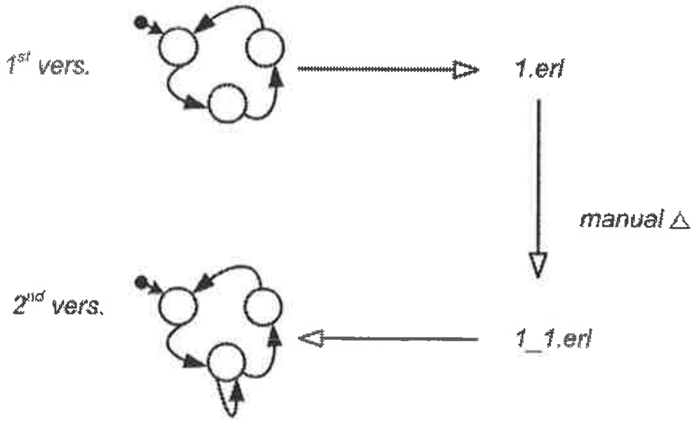


Fig. 7. Integration of manual changes

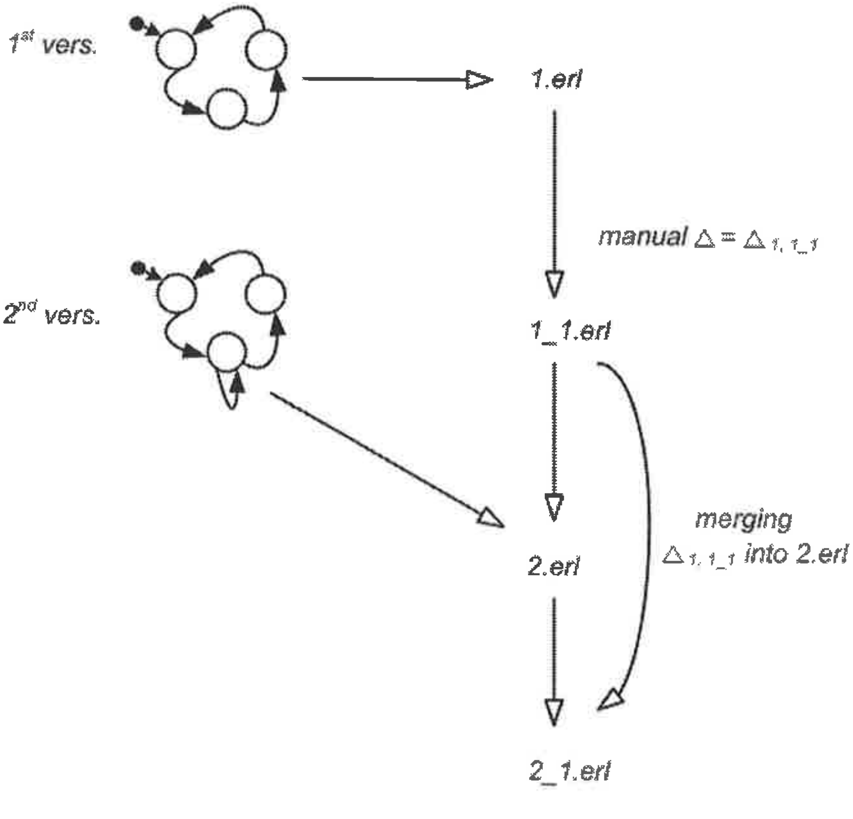


Fig. 8. Merging of a new version of a model with the source code

When these two problems are solved, it becomes possible to solve the general problem of merging two changes to the same model. (Fig. 9)

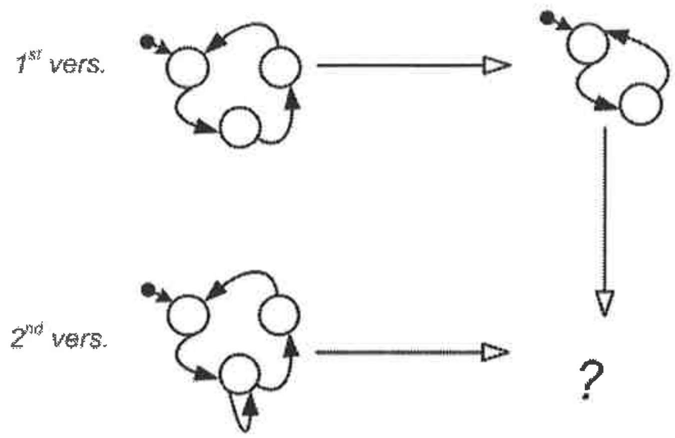


Fig. 9. Merging two changes to the same model

7.2 *Refactoring*

Sometimes you have an existing code base and would like to introduce a model-driven approach – more often than not, this is a major practical issue since most freely written code does not easily fit with a model framework.

In order to overcome this obstacle it would be interesting to investigate the use of refactoring tools that assist the programmer in transforming a legacy code base to a format that allows easy reverse engineering of the code into a model.

In this case, a given state machine code needs to be transformed back into the model that is at a higher level of abstraction. Adding reverse engineering for the current tool suit makes it possible to offer a visual content of the state machine from legacy code as higher-level document. Applying model based analysis technique i.e., model checking [7] to the model is also feasible, once the model has been built from the legacy code.

Wrangler [8] is a refactoring tool providing a collection of basic refactorings to the program. With the help of the tool, the legacy code can be refactored as close as possible to the generation template. Subsequently, A text-to-model transform engine can be applied to take the refactored program as input and produce the model correspondingly.

7.3 *Artifact generation*

Besides the state machine source code, more artifacts can be generated from the model such as supervisor, application, makefile, etc., as they require certain information contained in the state machine model, too. This information can be exploited as much as possible in order to minimize the amount of manual work. However, the current state machine model does not necessarily contain all information for these artifacts. Depending on what to generate, proper metamodels, editors and generators could be added to the current development environment. Following the introduced approach, it is not hard to extend the current tool suit with the new components.

If a complete metamodel that covers all the concepts of the domain was developed, equipped with right tools, a component framework and a set of predefined components, the developer is able to model the whole telecommunication application and generate 100% code from it. The ErlCOM [12], as a domain specific language for robust reconfigurable components, tries to introduce a component layer on the top of Erlang environment, so that the developer concentrate on the system design at a higher level. It provides 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 runtime environment or they could be reused. [13] Modeling and code generation tools have been developed based on the Generic Modeling Environment. However, its modeling concept mainly focuses on one of the static aspects of an application – component configuration management. The dynamic aspect of the application, that could be modeled using state machine, is not yet covered.

7.4 Model debugging

The current tool suit enables the developer to visualize the state machine as a diagram. The diagram and model could be further exploited within a debugging session by providing animation functionality. An executing state machine could send certain information, which is embedded into the generated code, back to the diagram. This information is used to synchronize the graphical notation to trigger the animation. States and transitions with the synchronous data should be highlighted. If the developer recognizes a mistake in the diagram while debugging, he can instantly change the model, and generate the code once again. However, the current stage of the development of the Eclipse modeling projects does not yet provide such a framework for the graphical debugging according to a given metamodel. More research needs to be done in this area.

8 Conclusion

The paper has presented an autocoding tool suit specifically designed to support finite state machine development. It facilitates the development by providing the developer with a specialized graphical editing environment to specify the state machine model, and a code generator to produce Erlang code. An initial prototype of the tools has been developed on the Eclipse platform following the model-driven software development philosophy. The model-driven approach for tool development has been experimented, in which models are central to all phases of the development process.

References

[1] The Object Management Group, *MDA Guide Version 1.0.1*, 12th June 2003

[2] The Object Management Group, *Meta Object Facility(MOF) Specification, Version 1.4*, April 2002

[3] Stahl,T., M. Völter, J. Bettin, A. Haase, S. Helsen, K. Czarnecki (Foreword by), B. von Stockfleth (Translated by), "Model-Driven Software Development: Technology, Engineering, Management", ISBN: 978-0-470-02570-3, Wiley, 2006

[4] Ericsson AB, *OTP Design Principles Version 5.6.4*

[5] Warmer, J, A. Kleppe, "Object Constraint Language, Getting Your Models Ready for MDA", Second Edition, Publisher : Addison Wesley, Pub Date : August 29, 2003, ISBN : 0-321-17936-6

[6] Chikofsky, E.J.; J.H. Cross II, *Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software*, IEEE Computer Society: 13C17. January 1990

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen, *A Tutorial on Uppaal*, Updated 17th November 2004. Department of Computer Science, Aalborg University

[8] Huiqing Li, Simon Thompson, George Orosz, and Melinda Toth, *Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse*, Proceedings of the Seventh ACM SIGPLAN Erlang Workshop, page 12pp. ACM Press, September 2008.

[9] Frank Budinsky, Dave Steinberg, Ed Merks, Ray Ellersick, Timothy J. Grose, "Eclipse Modeling Framework", Published Aug 11, 2003 by Addison-Wesley Professional

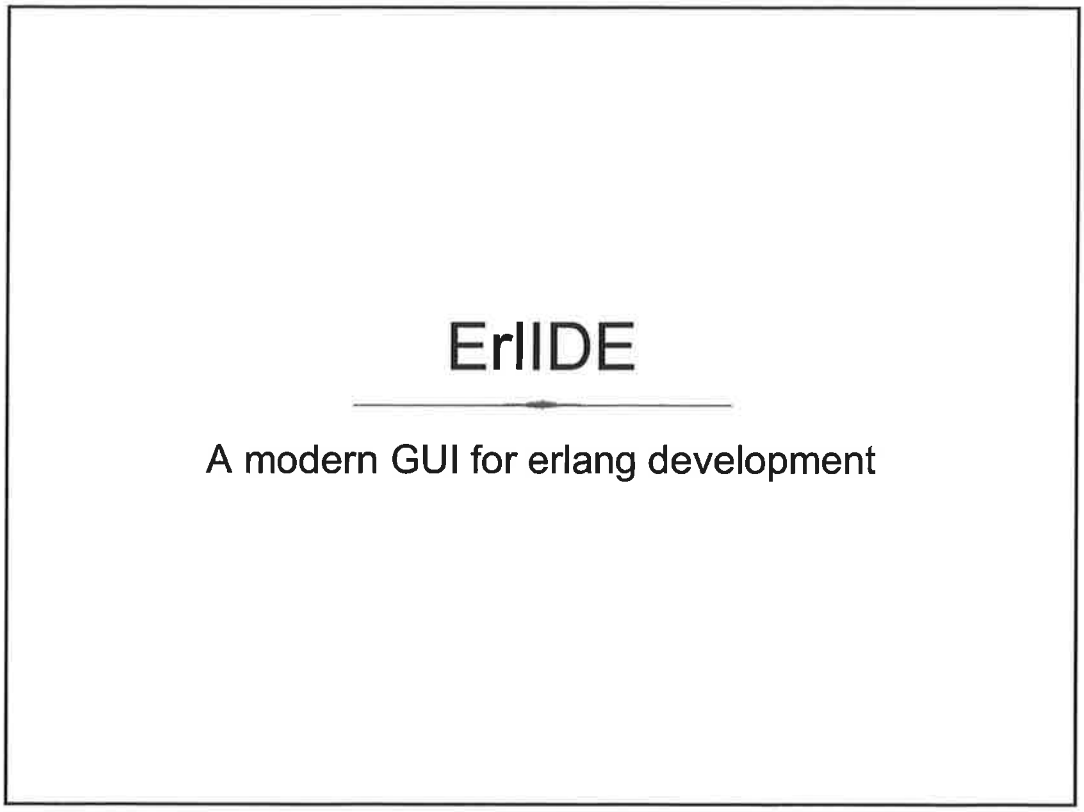
[10] Frederic Plante, *Introducing the GMF Runtime*, Eclipse Corner Article, January 16th, 2006

[11] C. Angelov, Xu Ke, Yu Guo and K. Sierszecki, *Reconfigurable State Machine Components for Embedded Applications*, Proc. of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2008, Parma, Italy, Sept 2008

[12] Gabor Batori, Zoltan Theisz and Domonkos Asztalos, *Robust Reconfigurable Erlang Component System*, In The Proceedings of 11th International Erlang/OTP User Conference, Stockholm, Sweden, November 2005.

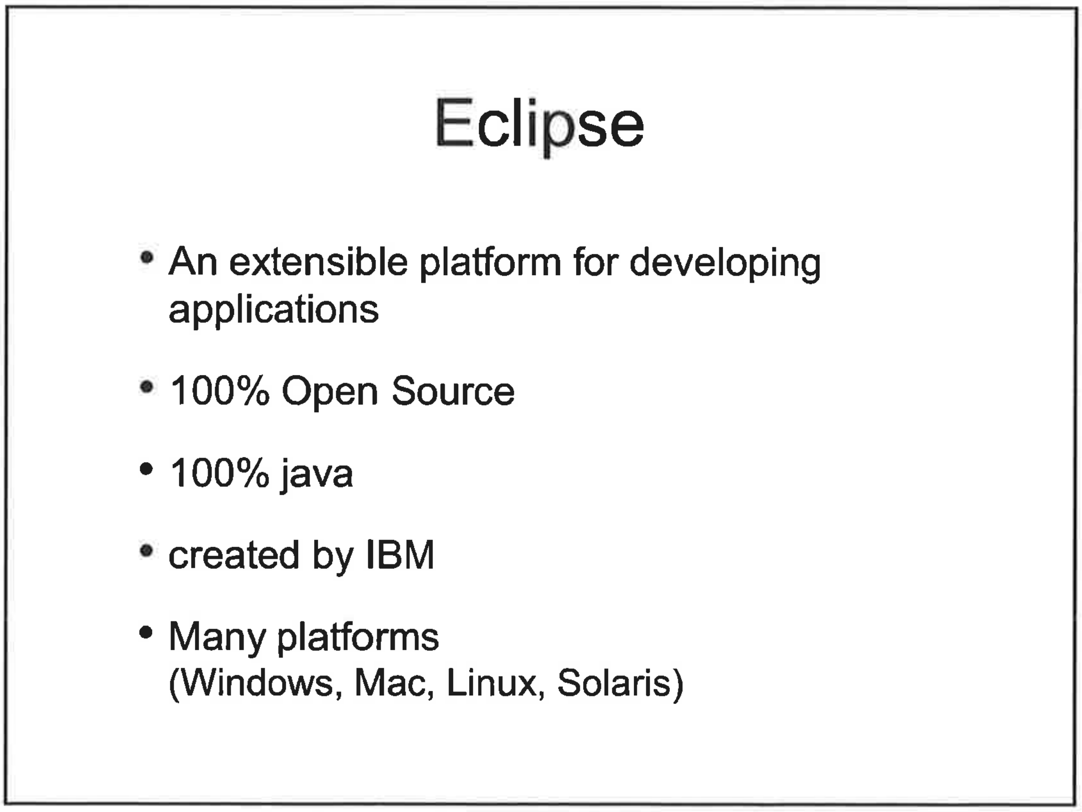
[13] Gabor Batori, Zoltan Theisz, and Domonkos Asztalos, *Configuration Aware Distributed System Design in Erlang*, In The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden, November 2006.

[14] Mealy, GH, *A Method to Synthesizing Sequential Circuits.* Bell System Technical J, 1045-1079. 1955



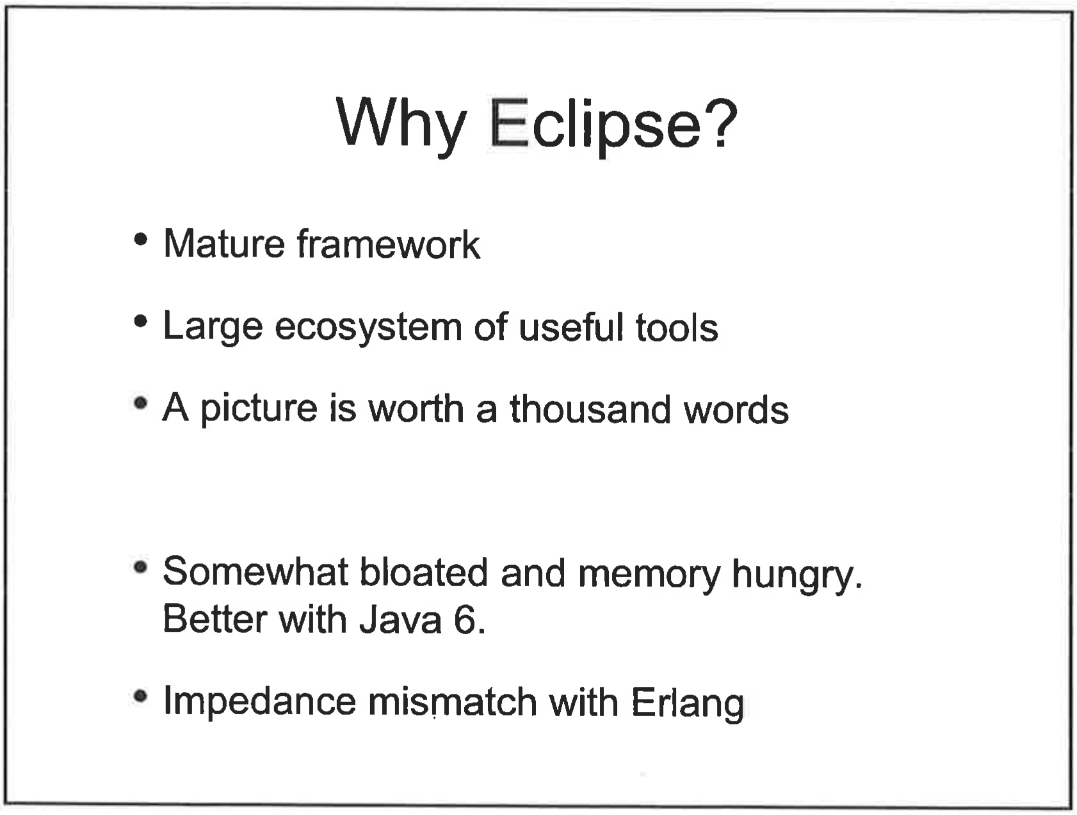
ErlIDE

A modern GUI for erlang development



Eclipse

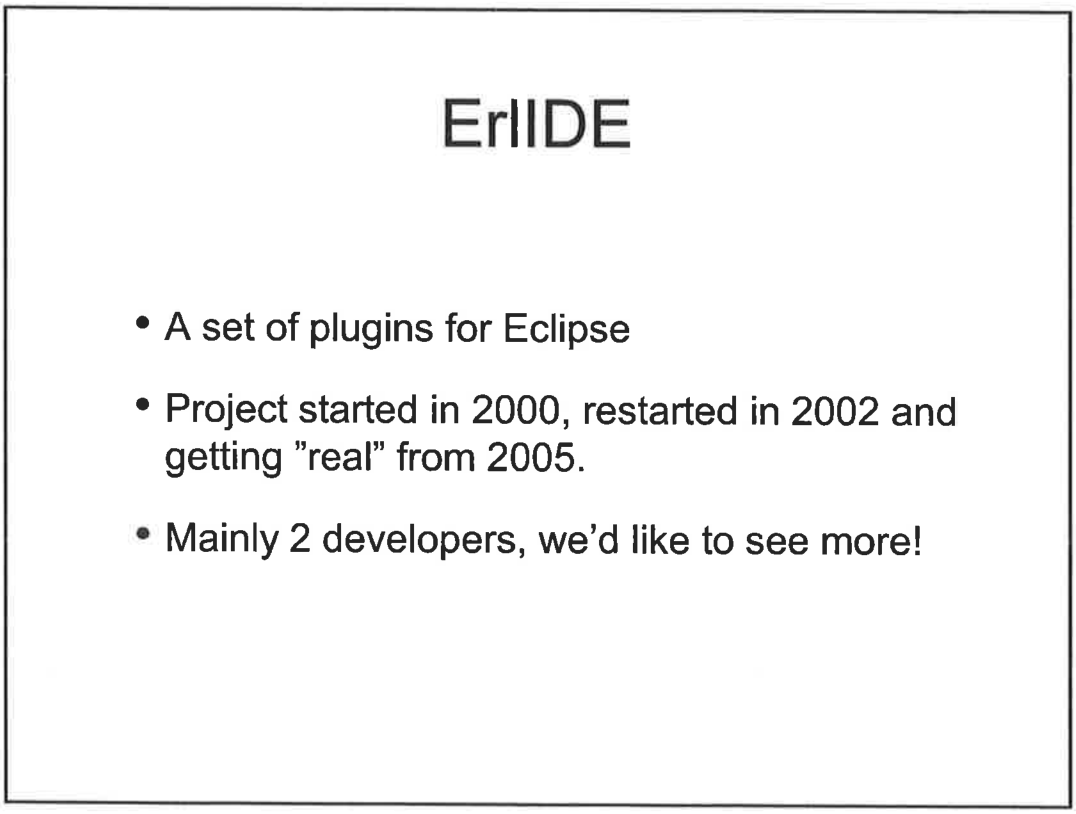
- An extensible platform for developing applications
- 100% Open Source
- 100% java
- created by IBM
- Many platforms (Windows, Mac, Linux, Solaris)



Why Eclipse?

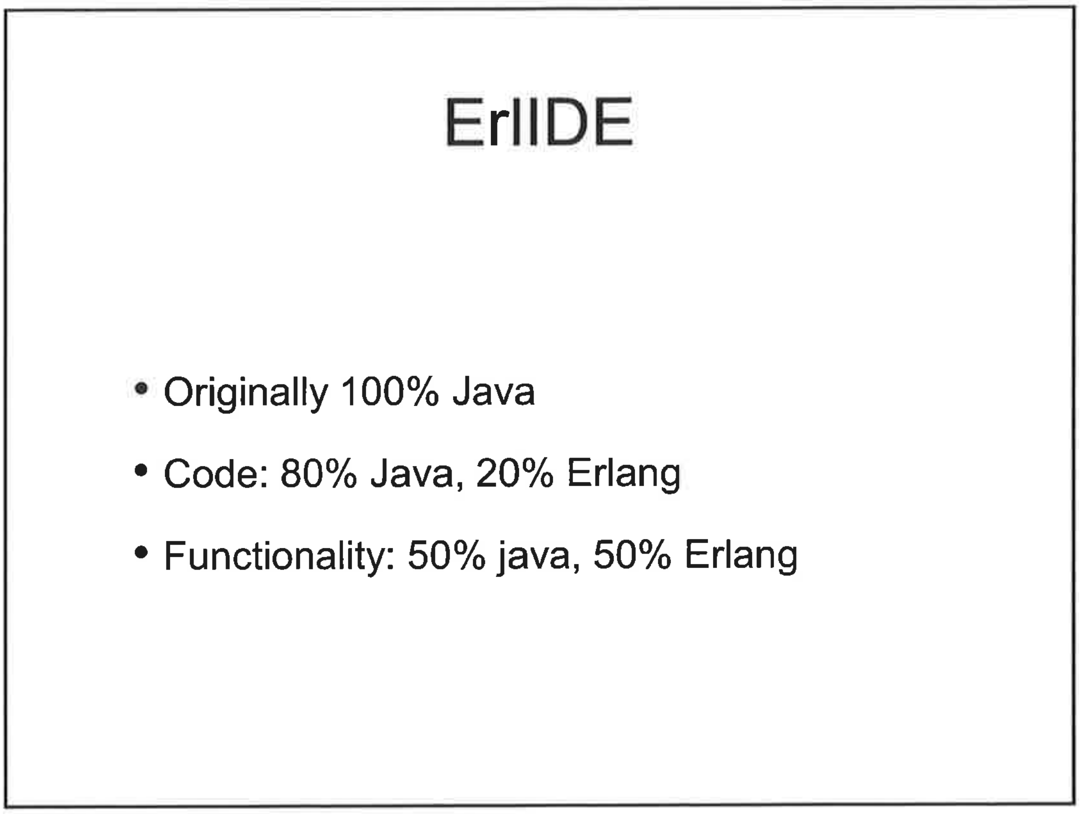
- Mature framework
- Large ecosystem of useful tools
- A picture is worth a thousand words

- Somewhat bloated and memory hungry. Better with Java 6.
- Impedance mismatch with Erlang



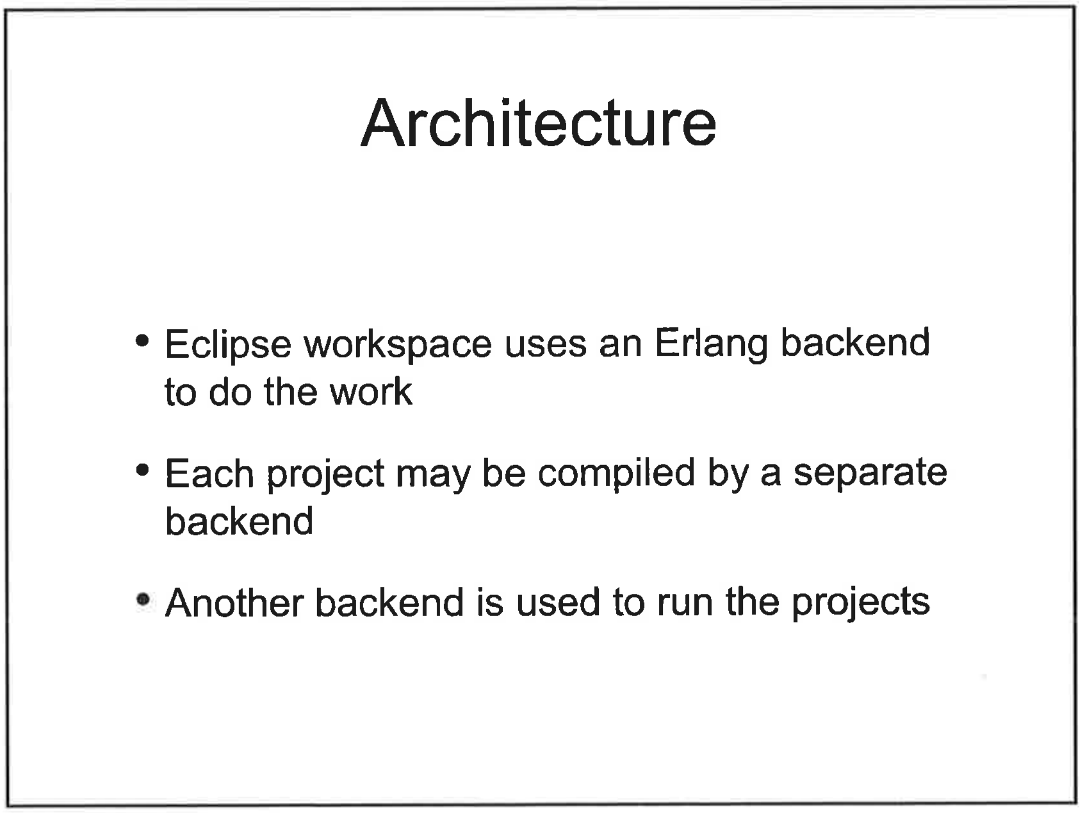
ErlIDE

- A set of plugins for Eclipse
- Project started in 2000, restarted in 2002 and getting ”real” from 2005.
- Mainly 2 developers, we’d like to see more!



ErlIDE

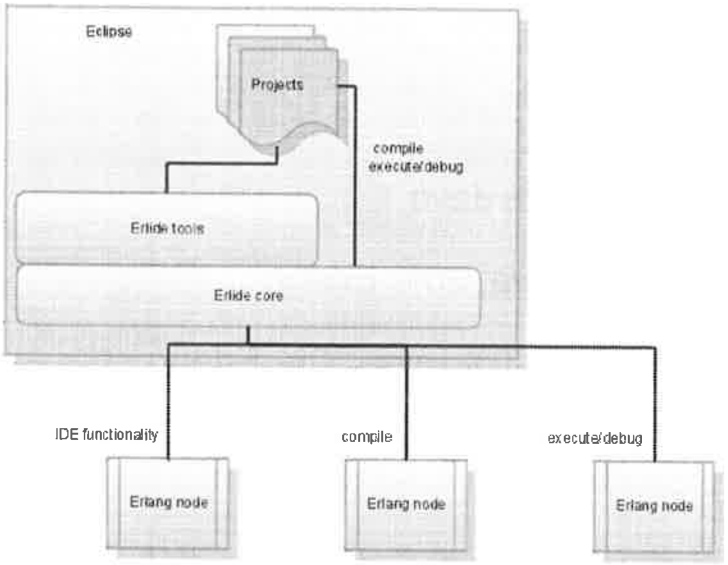
- Originally 100% Java
- Code: 80% Java, 20% Erlang
- Functionality: 50% java, 50% Erlang



Architecture

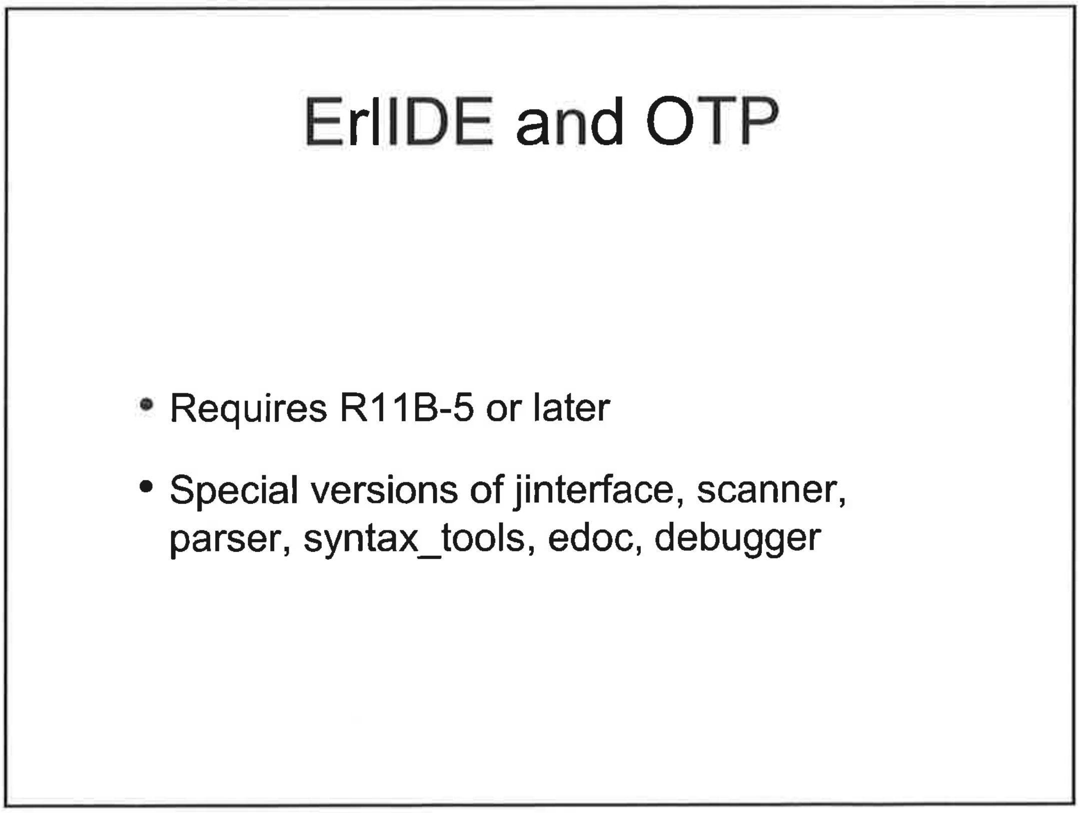
- Eclipse workspace uses an Erlang backend to do the work
- Each project may be compiled by a separate backend
- Another backend is used to run the projects

Architecture



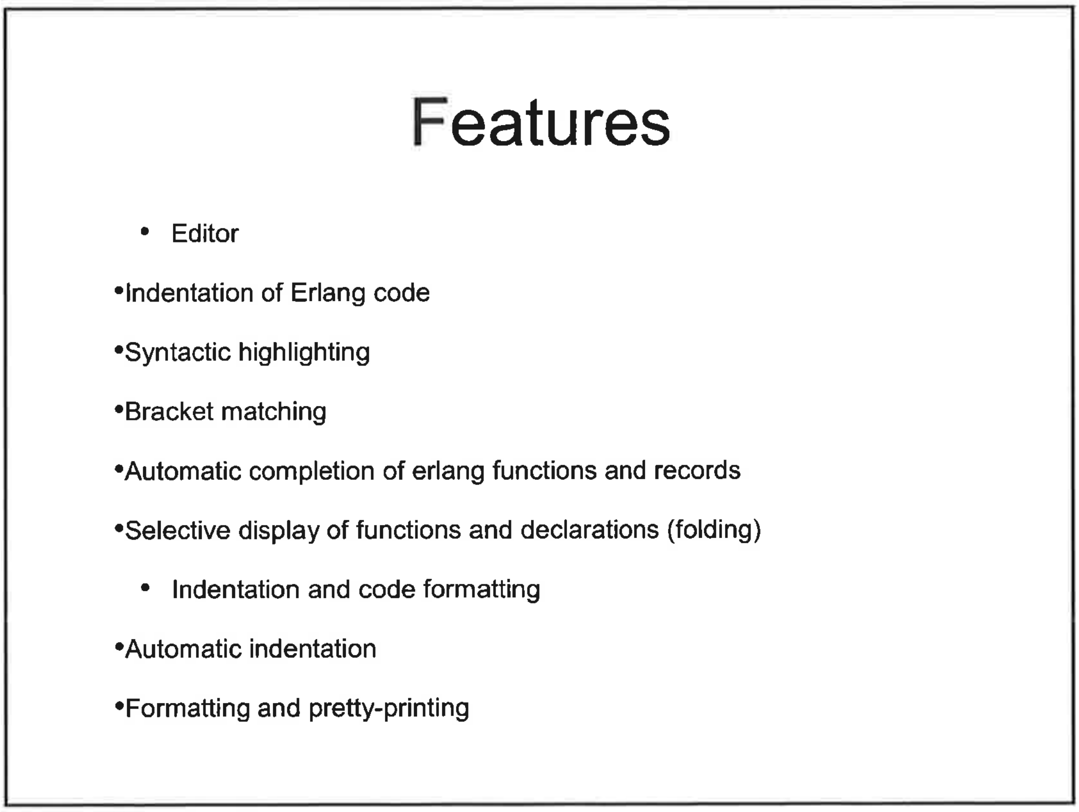
Architecture

- Erlang side provides services to Java side
  - RPC
  - event based
- Infrastructure to make communication as seamless as possible
  - implement Java interfaces in Erlang



ErlIDE and OTP

- Requires R11B-5 or later
- Special versions of jinterface, scanner, parser, syntax_tools, edoc, debugger



Features

- Editor

•Indentation of Erlang code

•Syntactic highlighting

•Bracket matching

•Automatic completion of erlang functions and records

•Selective display of functions and declarations (folding)

- Indentation and code formatting

•Automatic indentation

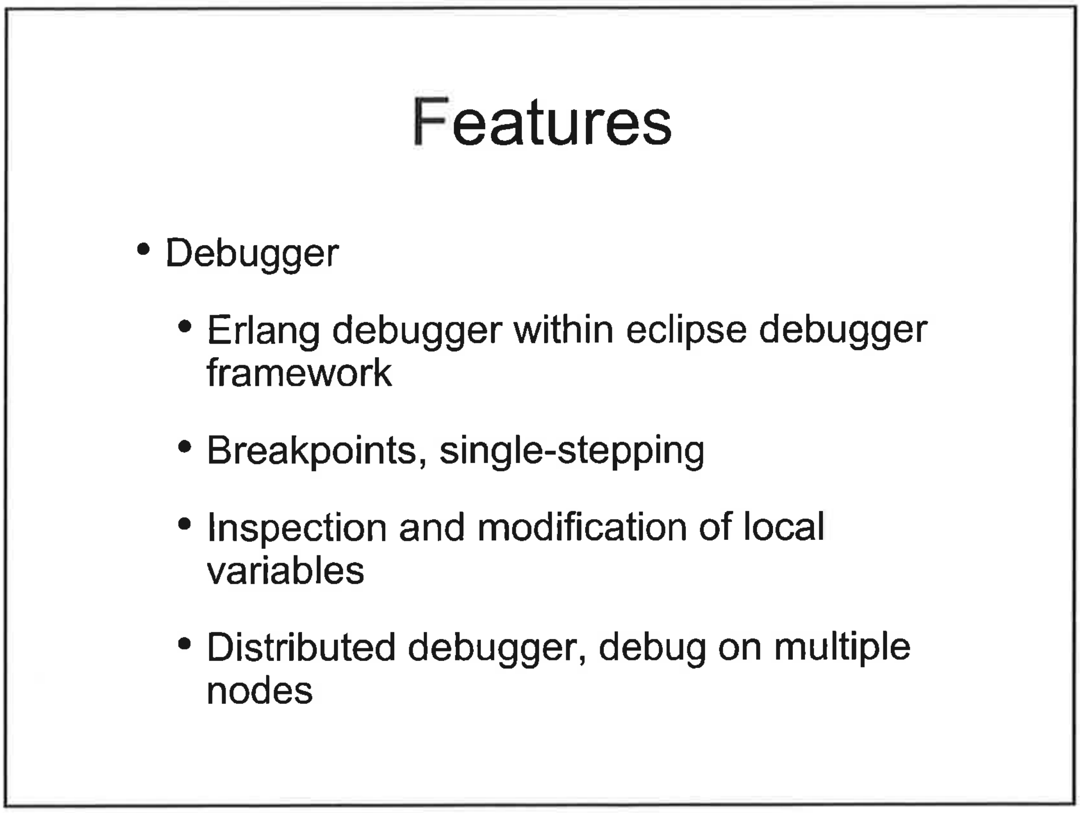
•Formatting and pretty-printing

Features

- Hover
  - OTP documentation shown for external calls
  - Show declaration of macros and records
- Outline
  - Code outline of the structure of an Erlang module
  - Filtering of Erlang functions and declarations
  - Quick outline for selecting Erlang function or declaration

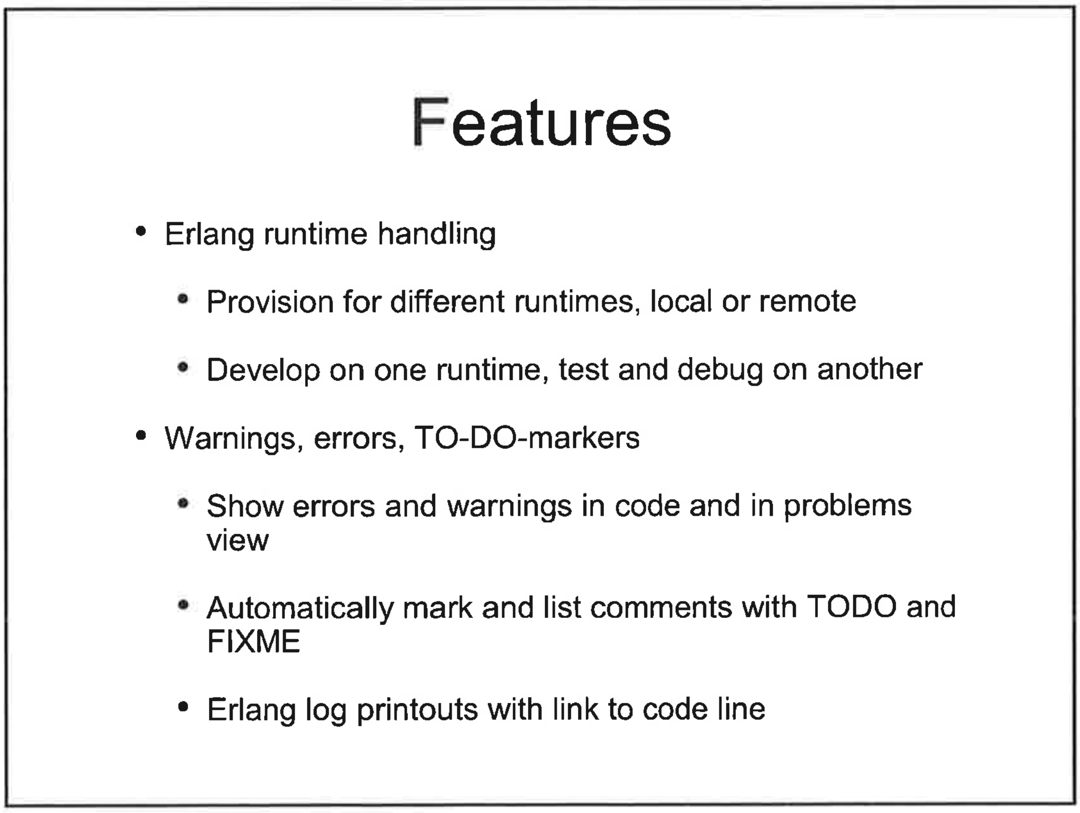
Features

- Navigation
  - Go to declaration of function, macro or record
- Project Import
  - Erlang-aware import of projects
- Creation wizards
  - Create Erlang projects
  - Create Erlang modules with code skeletons



Features

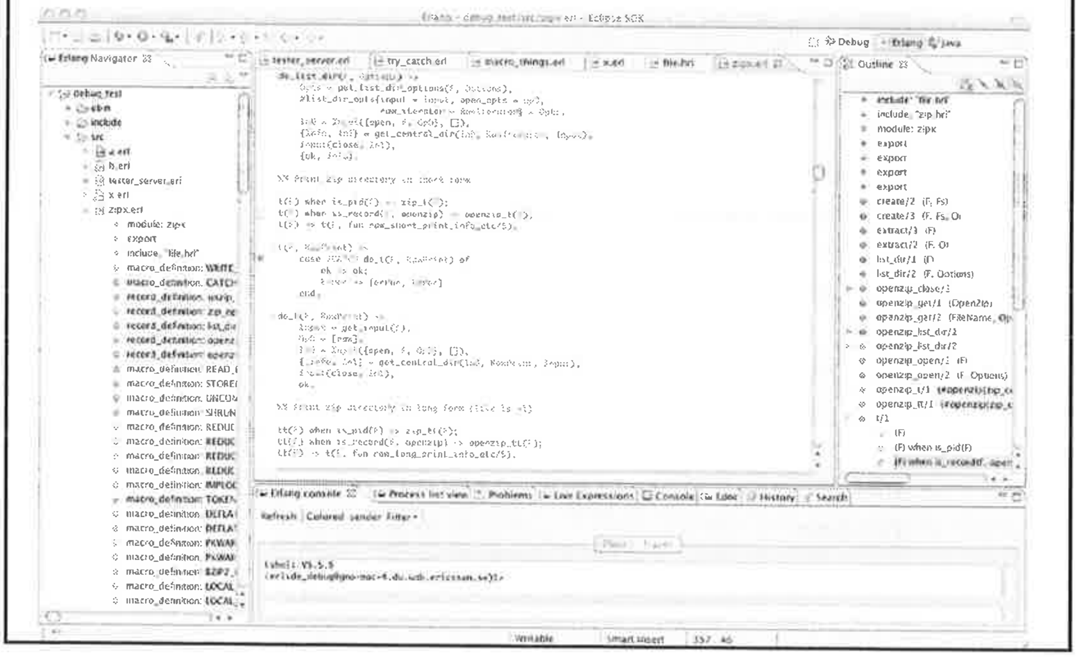
- Debugger
  - Erlang debugger within eclipse debugger framework
  - Breakpoints, single-stepping
  - Inspection and modification of local variables
  - Distributed debugger, debug on multiple nodes



Features

- Erlang runtime handling
  - Provision for different runtimes, local or remote
  - Develop on one runtime, test and debug on another
- Warnings, errors, TO-DO-markers
  - Show errors and warnings in code and in problems view
  - Automatically mark and list comments with TODO and FIXME
  - Erlang log printouts with link to code line

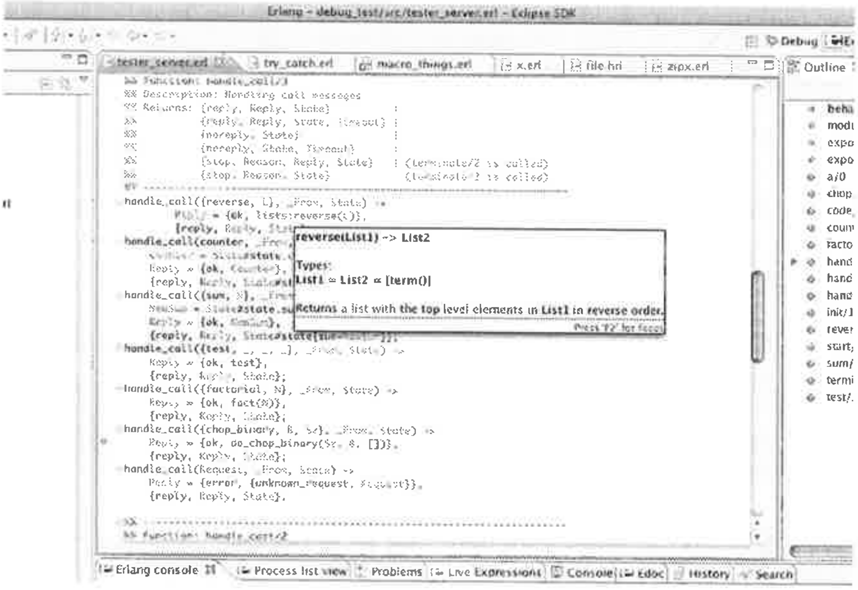
Demonstration



```
do_list_dir(F, Options) ->
    Opts = get_list_dir_options(F, Options),
    #list_dir_opts{input = Input, open_opts = OpO,
                   raw_iterator = RawIterator} = Opts,
    In0 = Input({open, F, OpO}, []),
    {Info, In1} = get_central_dir(In0, RawIterator, Input),
    Input(close, In1),
    {ok, Info}.

%% Print zip directory in short form

t(F) when is_pid(F) -> zip_t(F);
t(F) when is_record(F, openzip) -> openzip_t(F);
t(F) -> t(F, fun raw_short_print_info_etc/5).

t(F, RawPrint) ->
    case ?CATCH do_t(F, RawPrint) of
        ok -> ok;
        Error -> {error, Error}
    end.

do_t(F, RawPrint) ->
    Input = get_input(F),
    OpO = [raw],
    In0 = Input({open, F, OpO}, []),
    {_Info, In1} = get_central_dir(In0, RawPrint, Input),
    Input(close, In1),
    ok.

%% Print zip directory in long form (like ls -l)

tt(F) when is_pid(F) -> zip_tt(F);
tt(F) when is_record(F, openzip) -> openzip_tt(F);
tt(F) -> t(F, fun raw_long_print_info_etc/5).
```

```
Eshell V5.5.5
(erlide_debug@gno-mac-4.du.uab.ericsson.se)1>
```

Demonstration



```
%% Function: handle_call/3
%% Description: Handling call messages
%% Returns: {reply, Reply, State}          |
%%          {reply, Reply, State, Timeout} |
%%          {noreply, State}               |
%%          {noreply, State, Timeout}      |
%%          {stop, Reason, Reply, State}   | (terminate/2 is called)
%%          {stop, Reason, State}            (terminate/2 is called)
%%--------------------------------------------------------------------
handle_call({reverse, L}, _From, State) ->
        Reply = {ok, lists:reverse(L)},
        {reply, Reply, State};
handle_call(counter, _From,
    NewValue = State#state.c
    Reply = {ok, Counter},
    {reply, Reply, State#st
handle_call({sum, N}, _From
    NewSum = State#state.su
    Reply = {ok, NewSum},
    {reply, Reply, State#state{sum=NewSum}};
handle_call({test, _, _, _}, _From, State) ->
    Reply = {ok, test},
    {reply, Reply, State};
handle_call({factorial, N}, _From, State) ->
    Reply = {ok, fact(N)},
    {reply, Reply, State};
handle_call({chop_binary, B, Sz}, _From, State) ->
    Reply = {ok, do_chop_binary(Sz, B, [])},
    {reply, Reply, State};
handle_call(Request, _From, State) ->
    Reply = {error, {unknown_request, Request}},
    {reply, Reply, State}.

%%--------------------------------------------------------------------
%% Function: handle_cast/2
```

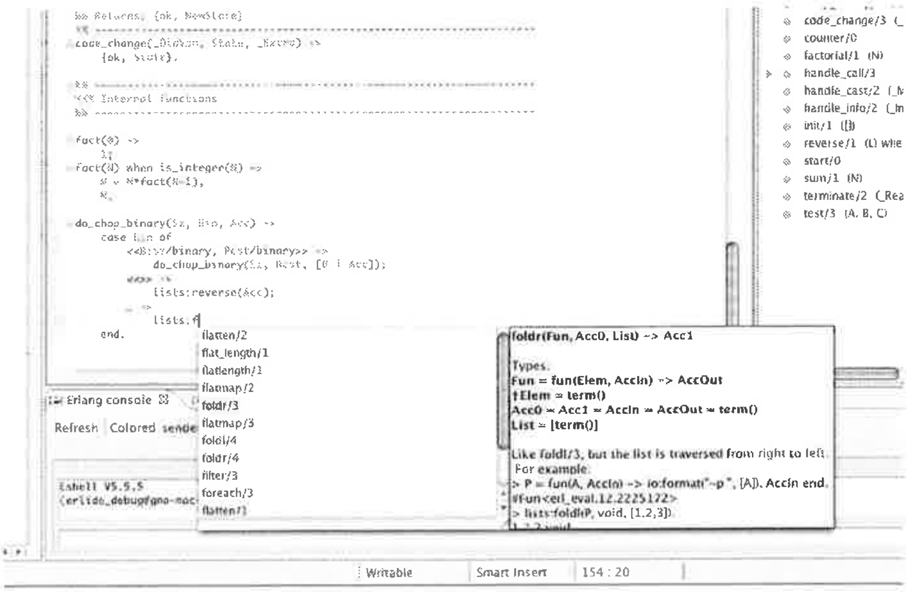
reverse(List1) -> List2

Types:
List1 = List2 = [term()]

Returns a list with the top level elements in **List1** in reverse order.

Press 'F2' for focus

Demonstration



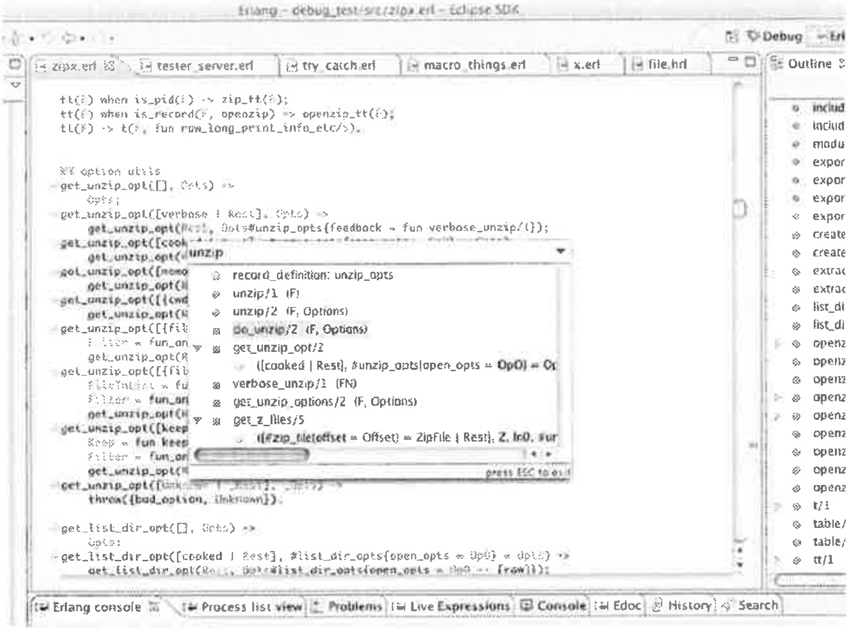
```
%% Returns: {ok, NewState}
%% --------------------------------------------------------------------
code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

%% --------------------------------------------------------------------
%%% Internal functions
%% --------------------------------------------------------------------

fact(0) ->
    1;
fact(N) when is_integer(N) ->
    X = N*fact(N-1),
    X.

do_chop_binary(Sz, Bin, Acc) ->
    case Bin of
        <<B:Sz/binary, Rest/binary>> ->
            do_chop_binary(Sz, Rest, [B | Acc]);
        <<>> ->
            lists:reverse(Acc);
        _ ->
            lists:f
    end.
```

```
flatten/2
flat_length/1
flatlength/1
flatmap/2
foldr/3
flatmap/3
foldl/4
foldr/4
filter/3
foreach/3
flatten/1
```

```
foldr(Fun, Acc0, List) -> Acc1

Types:
Fun = fun(Elem, AccIn) -> AccOut
†Elem = term()
Acc0 = Acc1 = AccIn = AccOut = term()
List = [term()]

Like foldl/3, but the list is traversed from right to left.
 For example:
> P = fun(A, AccIn) -> io:format("~p ", [A]), AccIn end.
#Fun<erl_eval.12.2225172>
> lists:foldl(P, void, [1,2,3]).
```

```
Eshell V5.5.5
(erlide_debug@gna-moc
```

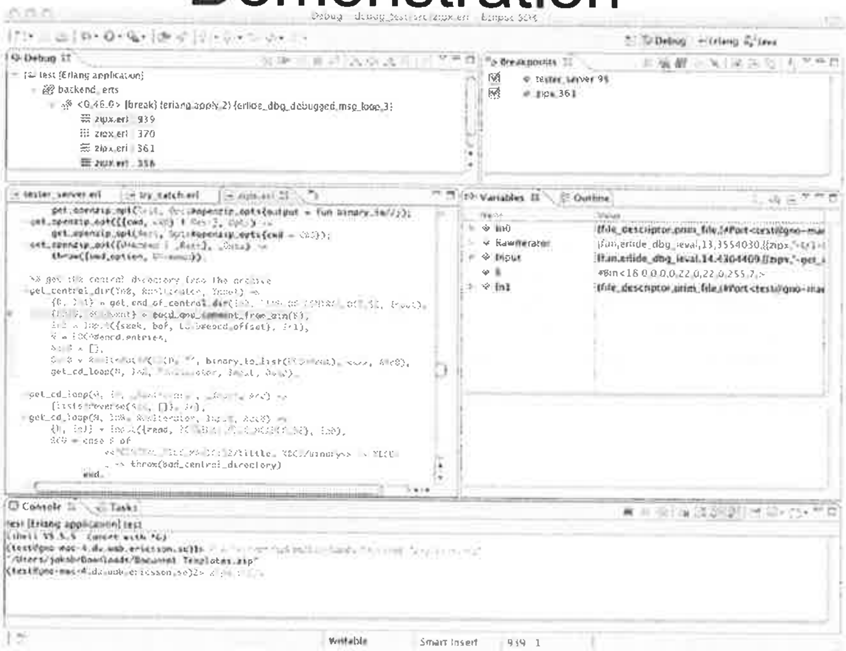
Demonstration



```
tt(P) when is_pid(P) -> zip_tt(P);
tt(P) when is_record(P, openzip) -> openzip_tt(P);
tt(P) -> t(P, fun raw_long_print_info_etc/5).


%% option utils
get_unzip_opt([], Opts) ->
    Opts;
get_unzip_opt([verbose | Rest], Opts) ->
    get_unzip_opt(Rest, Opts#unzip_opts{feedback = fun verbose_unzip/1});
get_unzip_opt([cook
    get_unzip_opt(
get_unzip_opt([memo
    get_unzip_opt(R
get_unzip_opt([{cwd
    get_unzip_opt(R
get_unzip_opt([{fil
    Filter = fun_on
    get_unzip_opt(R
get_unzip_opt([{fil
    FileInList = fu
    Filter = fun_on
    get_unzip_opt(R
get_unzip_opt([keep
    Keep = fun keep
    Filter = fun_on
    get_unzip_opt(R
get_unzip_opt([Unknown | Rest], _Opts) ->
    throw({bad_option, Unknown}).

get_list_dir_opt([], Opts) ->
    Opts;
get_list_dir_opt([cooked | Rest], #list_dir_opts{open_opts = OpO} = Opts) ->
    get_list_dir_opt(Rest, Opts#list_dir_opts{open_opts = OpO ++ [raw]});
```

```
unzip
record_definition: unzip_opts
unzip/1 (F)
unzip/2 (F, Options)
do_unzip/2 (F, Options)
get_unzip_opt/2
    ([cooked | Rest], #unzip_opts{open_opts = OpO} = Op
verbose_unzip/1 (FN)
get_unzip_options/2 (F, Options)
get_z_files/5
    ([#zip_file{offset = Offset} = ZipFile | Rest], Z, In0, #ur
```

Demonstration



```
test [Erlang application]
  backend_erts
    <0.46.0> [break] {erlang,apply,2} {erlide_dbg_debugged,msg_loop,3}
      zipx.erl 939
      zipx.erl 370
      zipx.erl 361
      zipx.erl 356
```

```
Breakpoints
☑ tester_server 95
☑ zipx 361
```

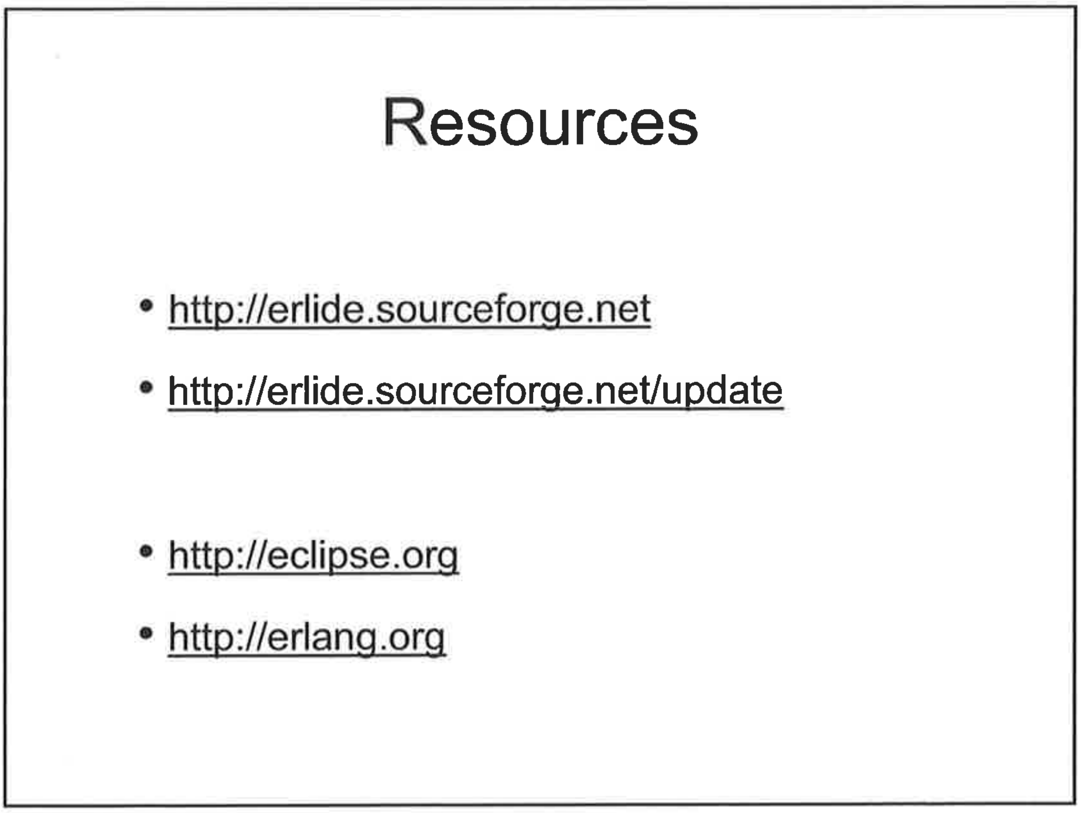
```
    get_openzip_opt([illegible], [illegible]#openzip_opts{output = fun binary_to[illegible]});
get_openzip_opt([{cwd, [illegible]} | [illegible]], [illegible]) ->
    get_openzip_opt([illegible], [illegible]#openzip_opts{cwd = [illegible]});
get_openzip_opt([[illegible] | [illegible]], [illegible]) ->
    throw({bad_option, [illegible]}).

%% get the central directory from the archive
get_central_dir([illegible], [illegible], [illegible]) ->
    {[illegible]} = get_end_of_central_dir([illegible], [illegible]),
    {[illegible], [illegible]} = eocd_and_comment_from_bin([illegible]),
    [illegible] = [illegible]({seek, bof, [illegible]_offset}, [illegible]),
    [illegible] = [illegible]#eocd.entries,
    [illegible] = [],
    [illegible] = [illegible](<<[illegible], binary_to_list([illegible]), [illegible]),
    get_cd_loop([illegible], [illegible], [illegible], [illegible], [illegible]).

get_cd_loop([illegible], [illegible], [illegible], [illegible], [illegible]) ->
    {lists:reverse([illegible], []), [illegible]};
get_cd_loop([illegible], [illegible], [illegible], [illegible], [illegible]) ->
    {[illegible], [illegible]} = [illegible]({read, [illegible]}, [illegible]),
    [illegible] = case [illegible] of
              <<[illegible]:32/little, [illegible]/binary>> -> [illegible]
              _ -> throw(bad_central_directory)
          end,
```

Name	Value
In0	{file_descriptor,prim_file,{#Port<test@gno-ma[illegible]
RawIterator	{fun,erlide_dbg_ieval,13,3554030,{{zipx,[illegible]
Input	{fun,erlide_dbg_ieval,14,4304409,{{zipx,"-get_[illegible]
B	#Bin<18,0,0,0,0,22,0,22,0,255,7,[illegible]>
In1	{file_descriptor,prim_file,{#Port<test@gno-ma[illegible]

```
test [Erlang application] test
Eshell V5.5.5  (abort with ^G)
(test@gno-mac-4.dx.uab.ericsson.se)1> [illegible]
"/Users/jakob/Downloads/Document Templates.zip"
(test@gno-mac-4.dx.uab.ericsson.se)2> [illegible]
```

Some future features

- Semantic highlighting in editor (function calls, variable usage, etc)
- Show documentation of user functions
- Erlang-aware search
- Refactoring
- xref-based indexing and searching
- Tracing viewer
- Syntax error correction
- What tools do <u>you</u> need?



Resources

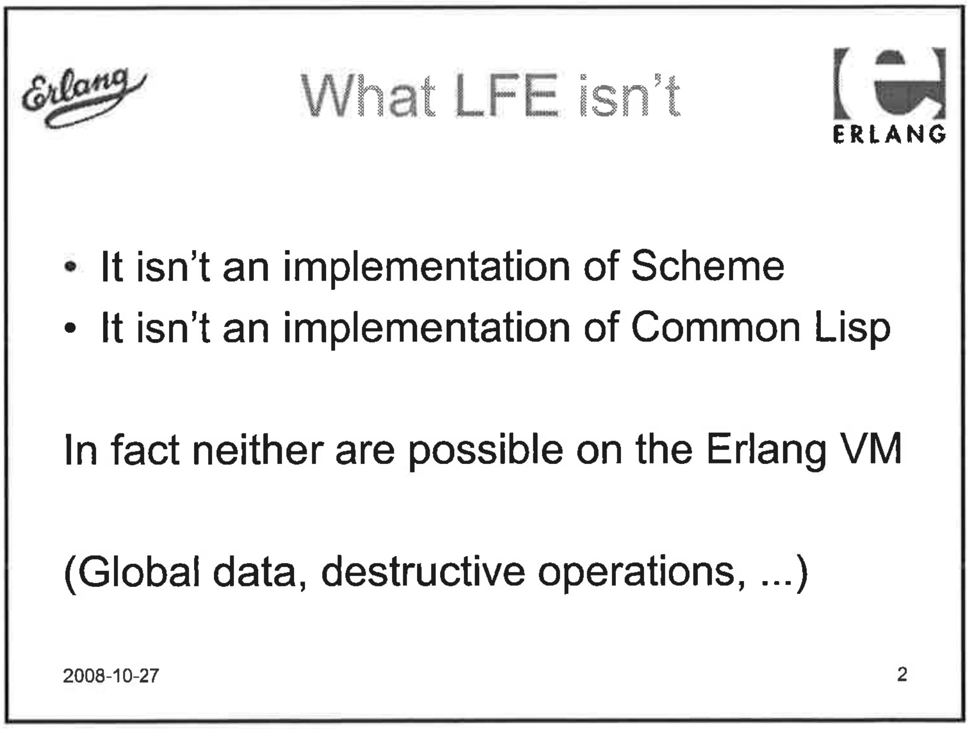
- http://erlide.sourceforge.net
- http://erlide.sourceforge.net/update

- http://eclipse.org
- http://erlang.org

Lisp Flavoured Erlang LFE

Adding a new flavour to Erlang

Robert Virding



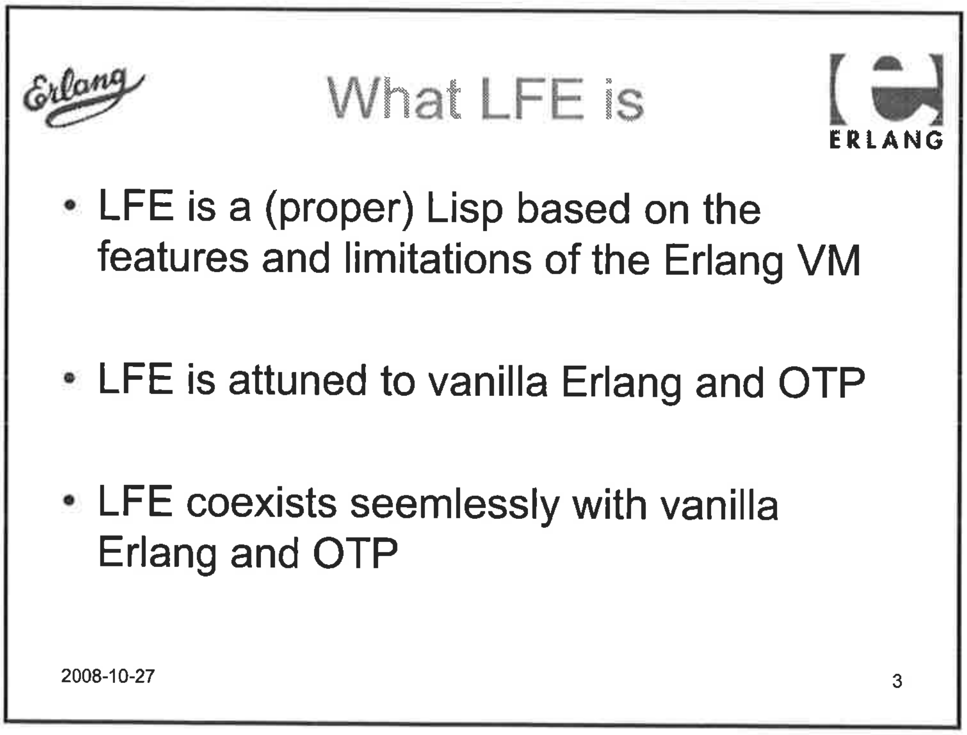
What LFE isn't

- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp

In fact neither are possible on the Erlang VM

(Global data, destructive operations, ...)

2008-10-27 2

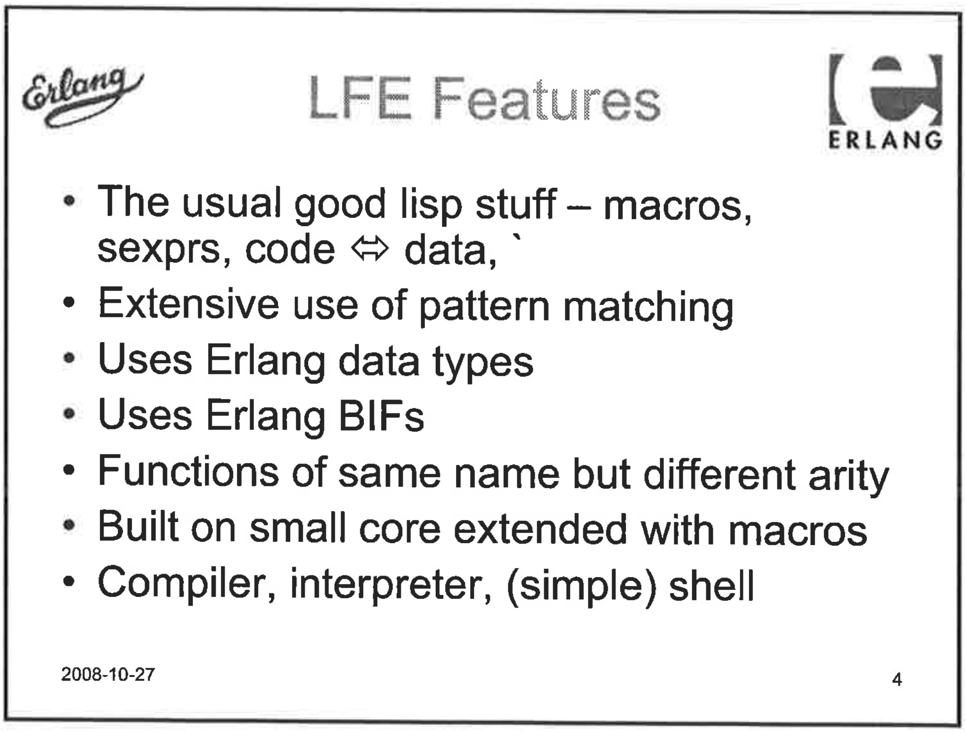


What LFE is

- LFE is a (proper) Lisp based on the features and limitations of the Erlang VM
- LFE is attuned to vanilla Erlang and OTP
- LFE coexists seemlessly with vanilla Erlang and OTP

2008-10-27

3

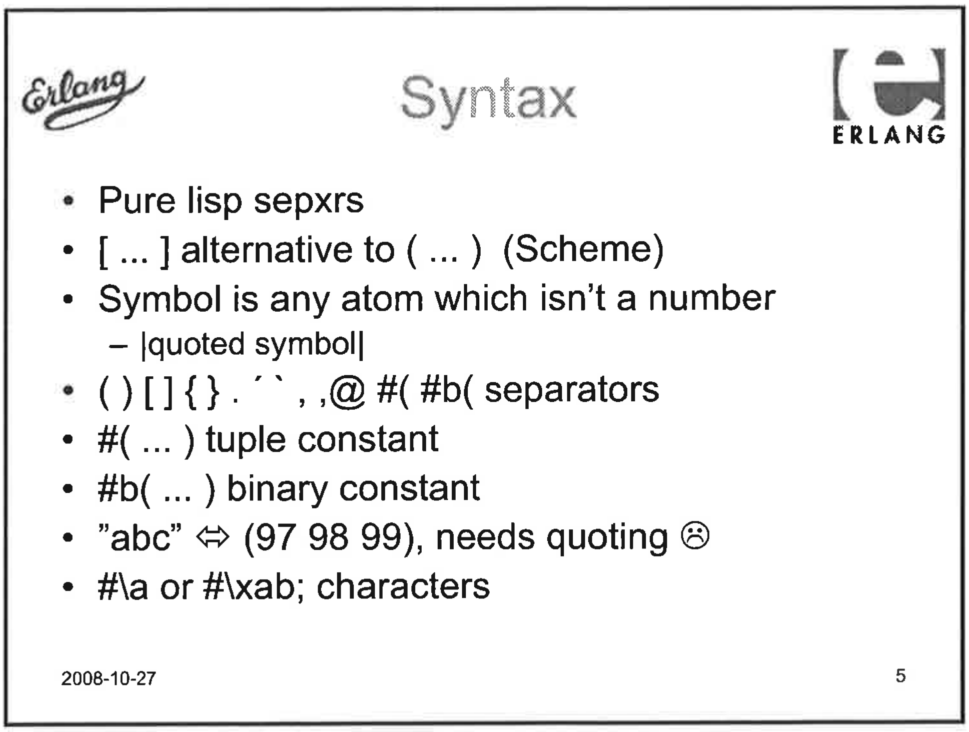


LFE Features

- The usual good lisp stuff – macros, sexprs, code ⇔ data, `
- Extensive use of pattern matching
- Uses Erlang data types
- Uses Erlang BIFs
- Functions of same name but different arity
- Built on small core extended with macros
- Compiler, interpreter, (simple) shell

2008-10-27

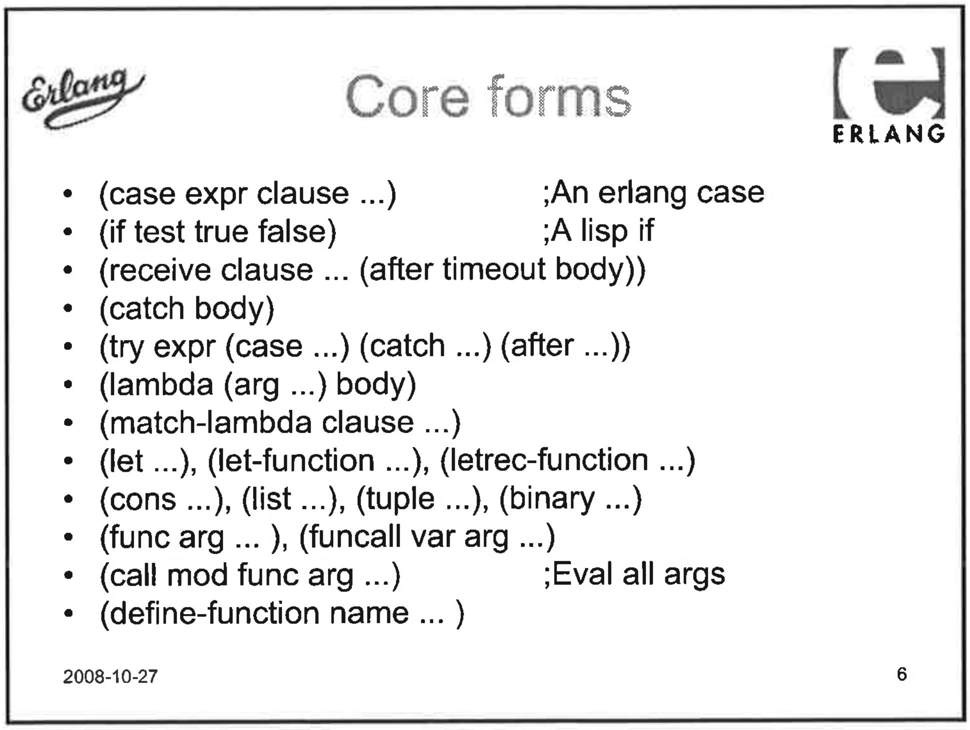
4



Syntax

- Pure lisp sepxrs
- [...] alternative to (...) (Scheme)
- Symbol is any atom which isn't a number
  - |quoted symbol|
- ( ) [ ] { } . ´ ` , ,@ #(#b(separators
- #(...) tuple constant
- #b(...) binary constant
- "abc" ⇔ (97 98 99), needs quoting ☹
- #\a or #\xab; characters

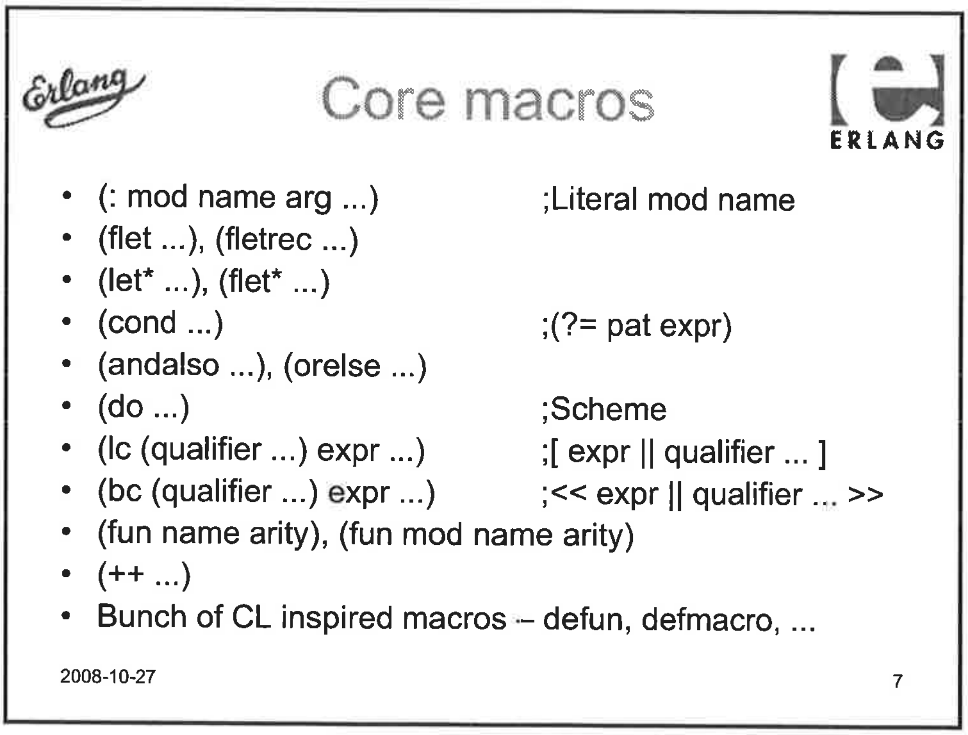
2008-10-27 5



Core forms

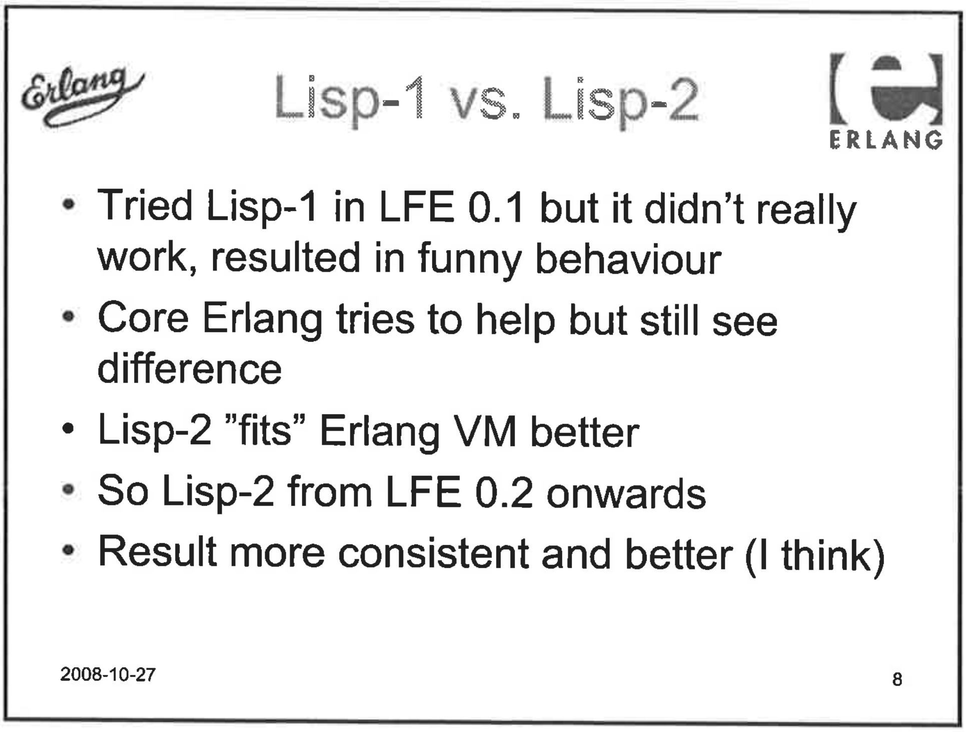
- (case expr clause ...) ;An erlang case
- (if test true false) ;A lisp if
- (receive clause ... (after timeout body))
- (catch body)
- (try expr (case ...) (catch ...) (after ...))
- (lambda (arg ...) body)
- (match-lambda clause ...)
- (let ...), (let-function ...), (letrec-function ...)
- (cons ...), (list ...), (tuple ...), (binary ...)
- (func arg ...), (funcall var arg ...)
- (call mod func arg ...) ;Eval all args
- (define-function name ...)

2008-10-27 6



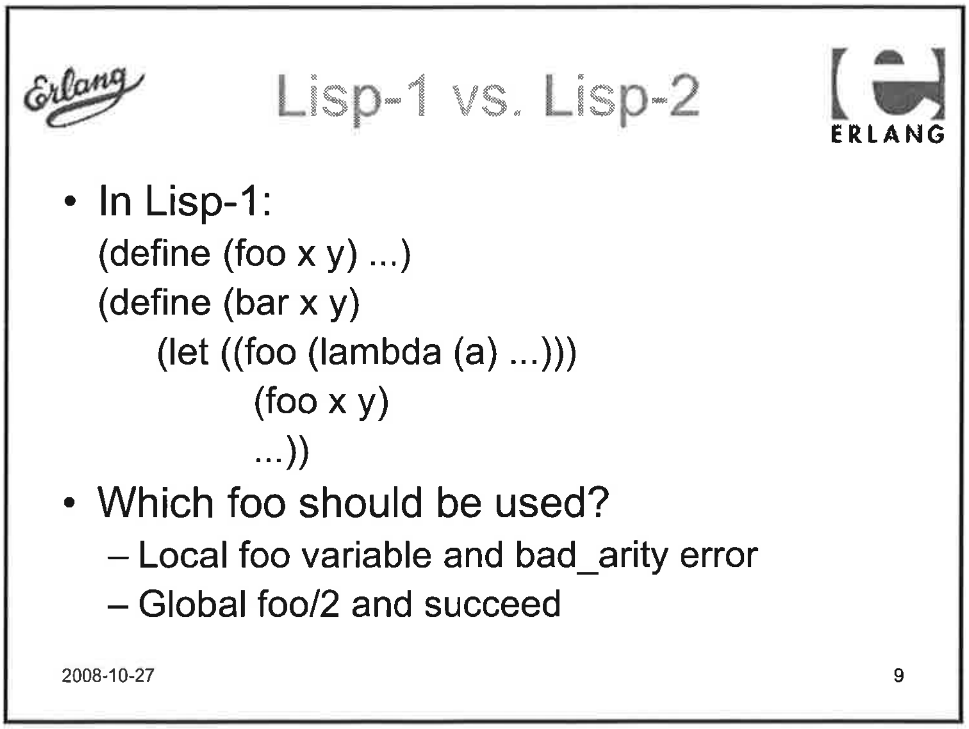
Core macros

- (: mod name arg ...) ;Literal mod name
- (flet ...), (fletrec ...)
- (let* ...), (flet* ...)
- (cond ...) ;(?= pat expr)
- (andalso ...), (orelse ...)
- (do ...) ;Scheme
- (lc (qualifier ...) expr ...) ;[expr || qualifier ...]
- (bc (qualifier ...) expr ...) ;<< expr || qualifier ... >>
- (fun name arity), (fun mod name arity)
- (++ ...)
- Bunch of CL inspired macros – defun, defmacro, ...



Lisp-1 vs. Lisp-2

- Tried Lisp-1 in LFE 0.1 but it didn't really work, resulted in funny behaviour
- Core Erlang tries to help but still see difference
- Lisp-2 ”fits” Erlang VM better
- So Lisp-2 from LFE 0.2 onwards
- Result more consistent and better (I think)

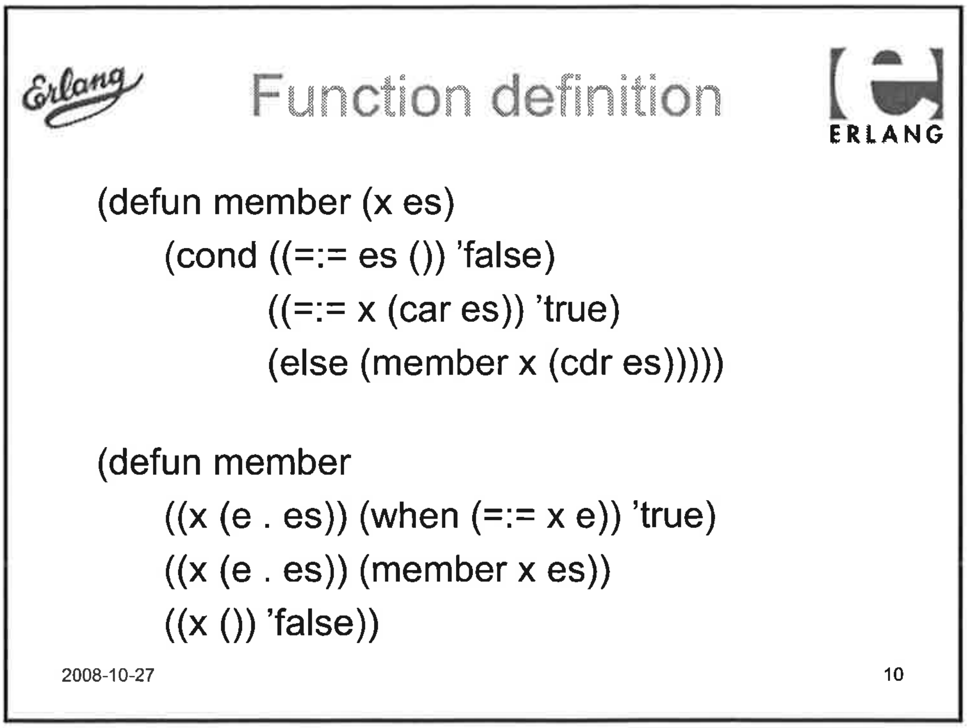


Lisp-1 vs. Lisp-2

- In Lisp-1:

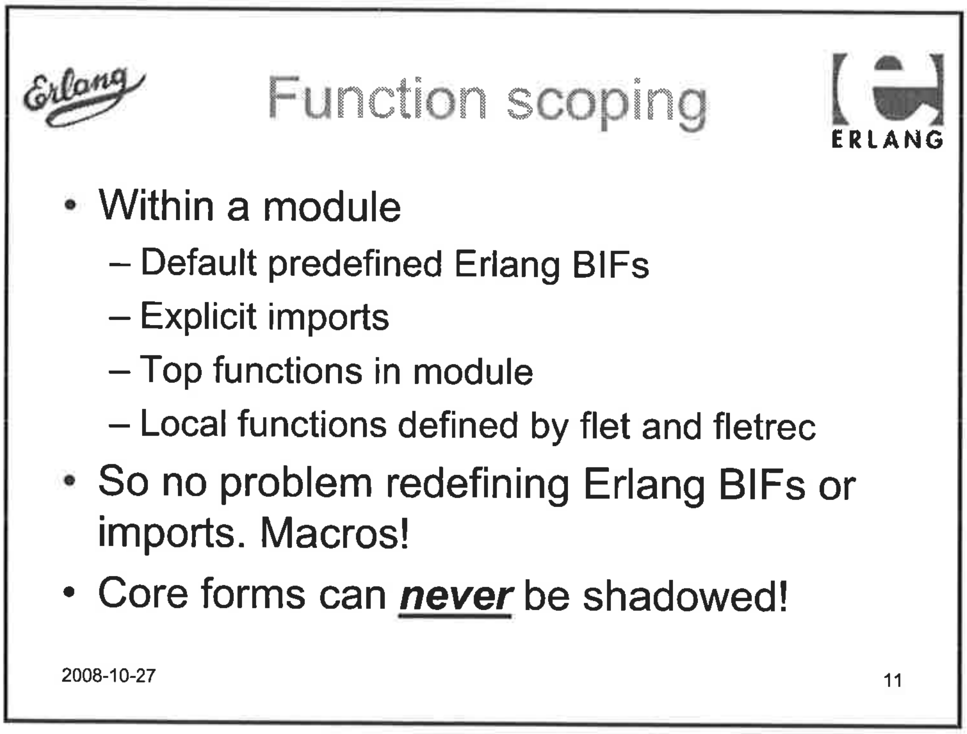
```
(define (foo x y) ...)
(define (bar x y)
     (let ((foo (lambda (a) ...)))
            (foo x y)
            ...))
```

- Which foo should be used?
  - Local foo variable and bad_arity error
  - Global foo/2 and succeed



Function definition

```
(defun member (x es)
     (cond ((=:= es ()) ’false)
             ((=:= x (car es)) ’true)
             (else (member x (cdr es)))))

(defun member
     ((x (e . es)) (when (=:= x e)) ’true)
     ((x (e . es)) (member x es))
     ((x ()) ’false))
```

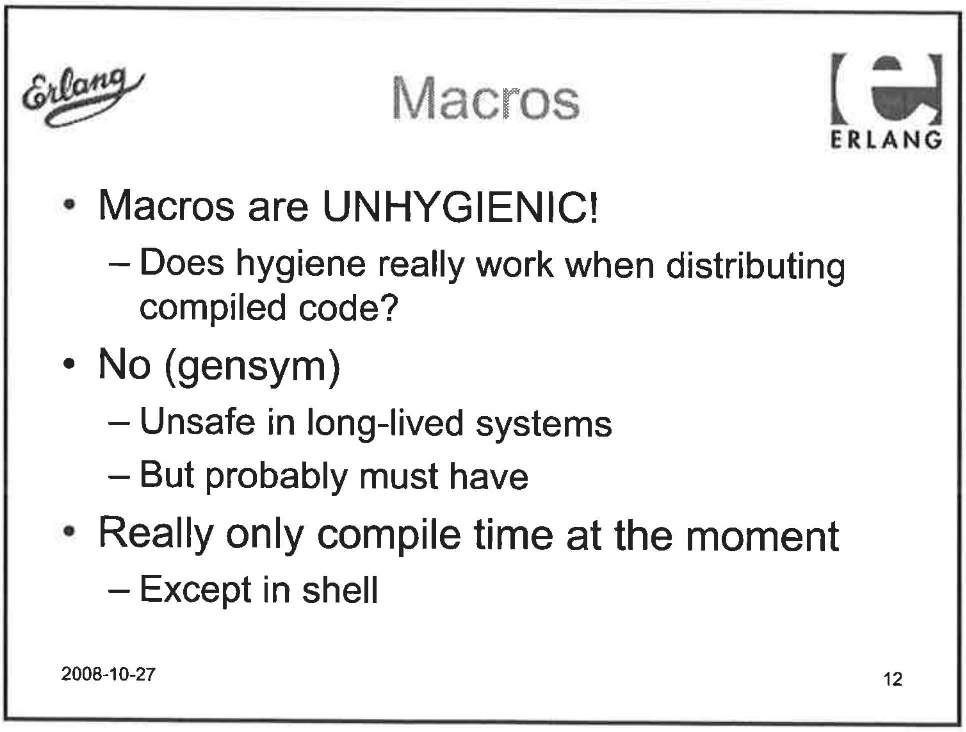


Function scoping

- Within a module
  - Default predefined Erlang BIFs
  - Explicit imports
  - Top functions in module
  - Local functions defined by flet and fletrec
- So no problem redefining Erlang BIFs or imports. Macros!
- Core forms can ***never*** be shadowed!

2008-10-27

11

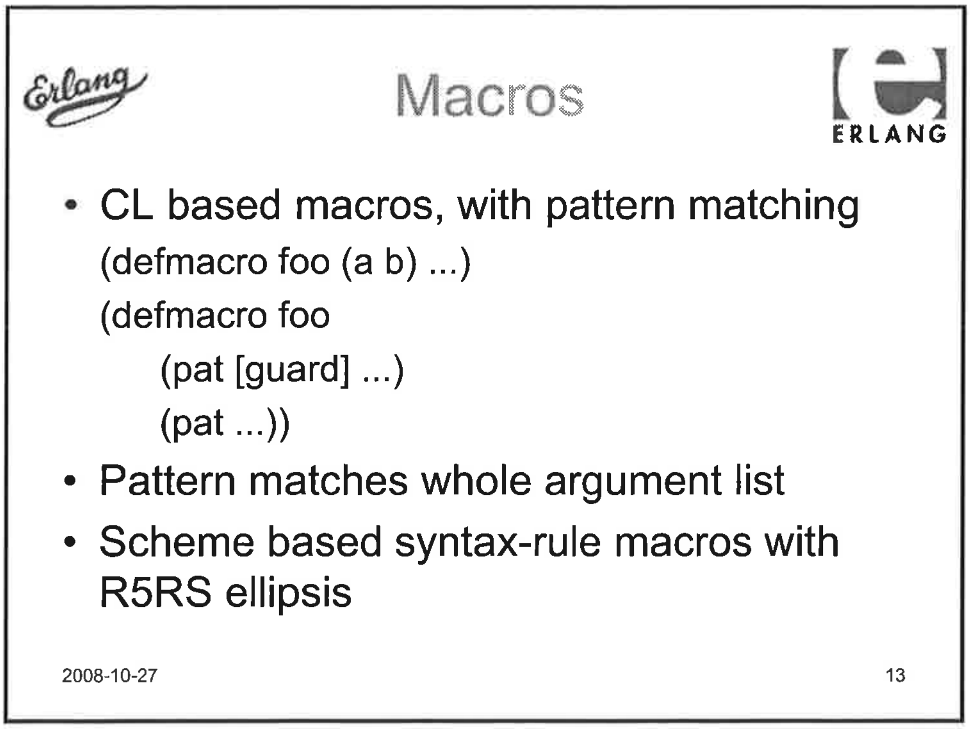


Macros

- Macros are UNHYGIENIC!
  - Does hygiene really work when distributing compiled code?
- No (gensym)
  - Unsafe in long-lived systems
  - But probably must have
- Really only compile time at the moment
  - Except in shell

2008-10-27

12



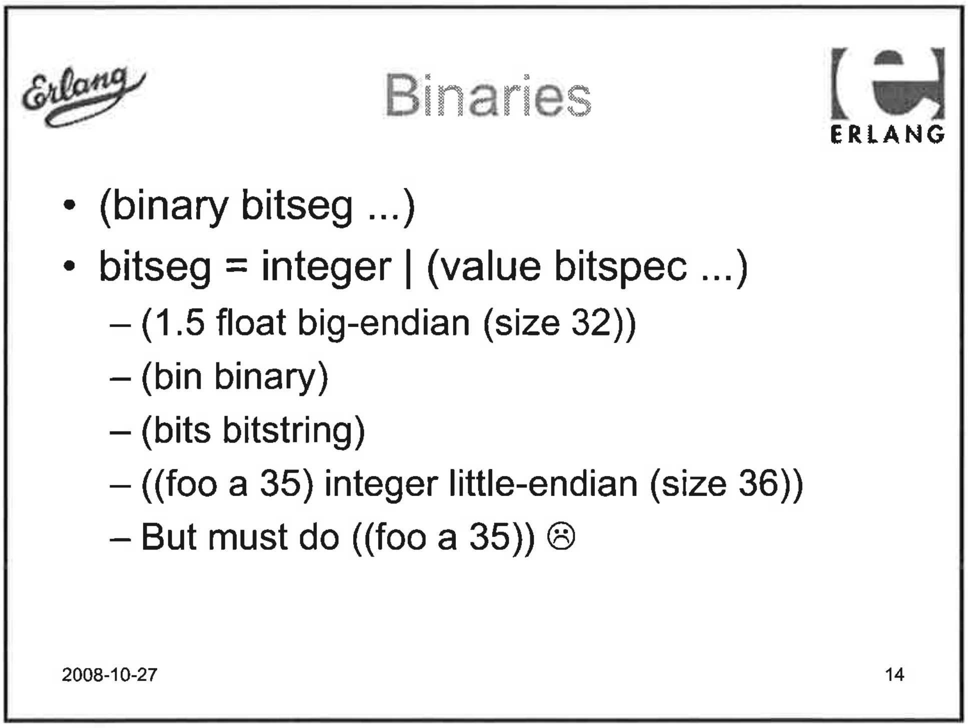
Macros

- CL based macros, with pattern matching

```
(defmacro foo (a b) ...)
(defmacro foo
    (pat [guard] ...)
    (pat ...))
```

- Pattern matches whole argument list
- Scheme based syntax-rule macros with R5RS ellipsis

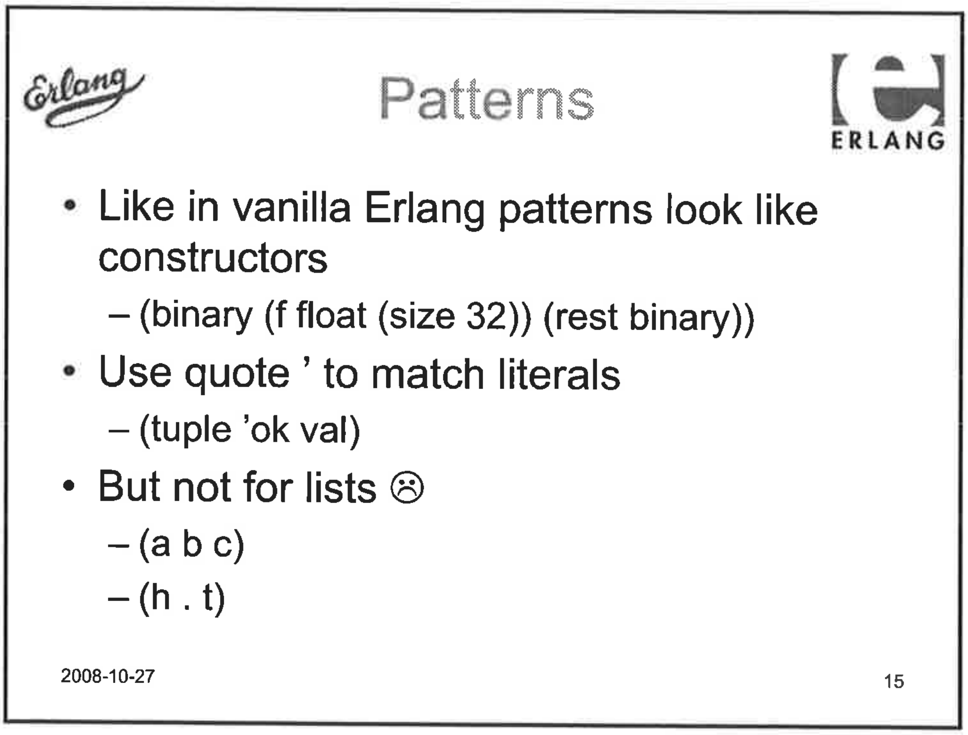
2008-10-27 13



Binaries

- (binary bitseg ...)
- bitseg = integer | (value bitspec ...)
  - (1.5 float big-endian (size 32))
  - (bin binary)
  - (bits bitstring)
  - ((foo a 35) integer little-endian (size 36))
  - But must do ((foo a 35)) ☹

2008-10-27 14

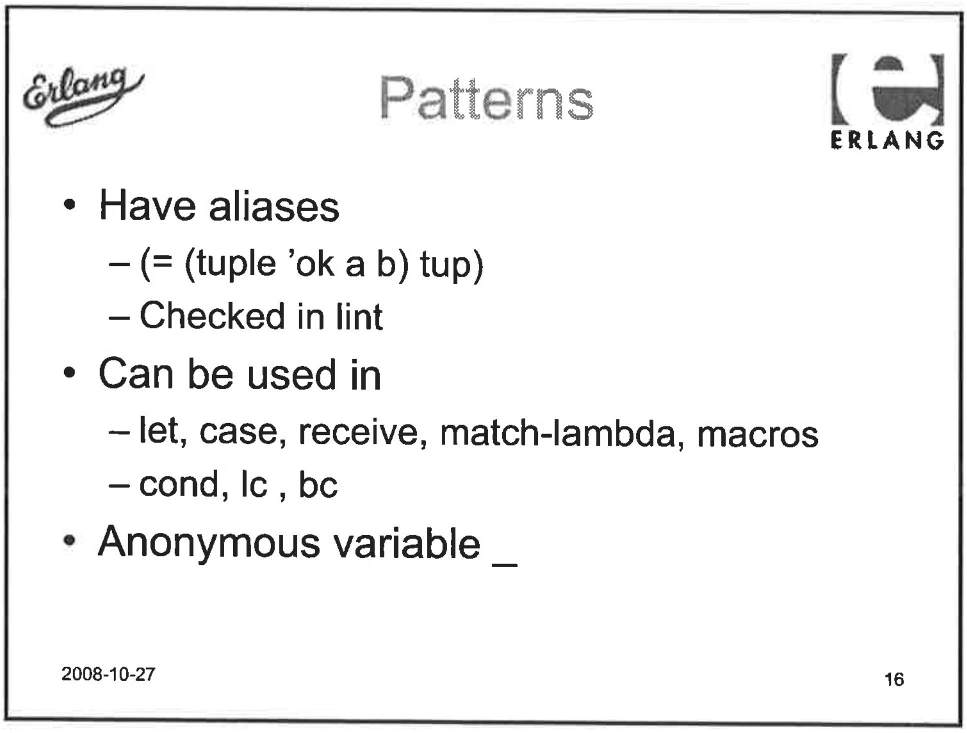


Patterns

- Like in vanilla Erlang patterns look like constructors
  - (binary (f float (size 32)) (rest binary))
- Use quote ’ to match literals
  - (tuple ’ok val)
- But not for lists ☹
  - (a b c)
  - (h . t)

2008-10-27

15

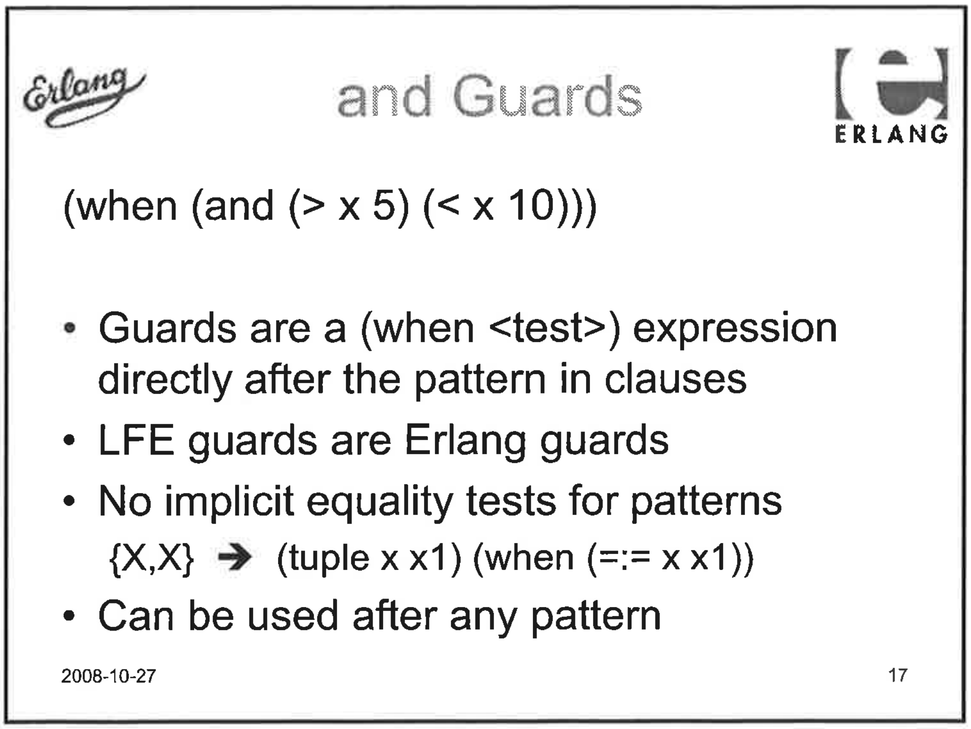


Patterns

- Have aliases
  - (= (tuple ’ok a b) tup)
  - Checked in lint
- Can be used in
  - let, case, receive, match-lambda, macros
  - cond, lc , bc
- Anonymous variable _

2008-10-27

16

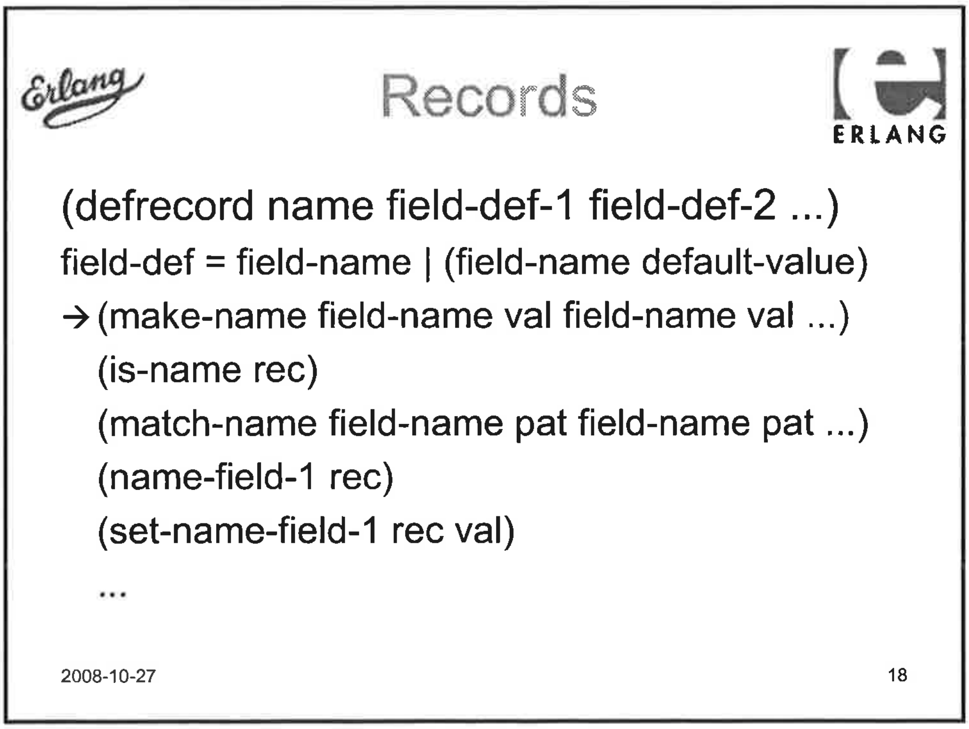


and Guards

(when (and (> x 5) (< x 10)))

- Guards are a (when <test>) expression directly after the pattern in clauses
- LFE guards are Erlang guards
- No implicit equality tests for patterns
  {X,X} ➔ (tuple x x1) (when (=:= x x1))
- Can be used after any pattern

2008-10-27 17



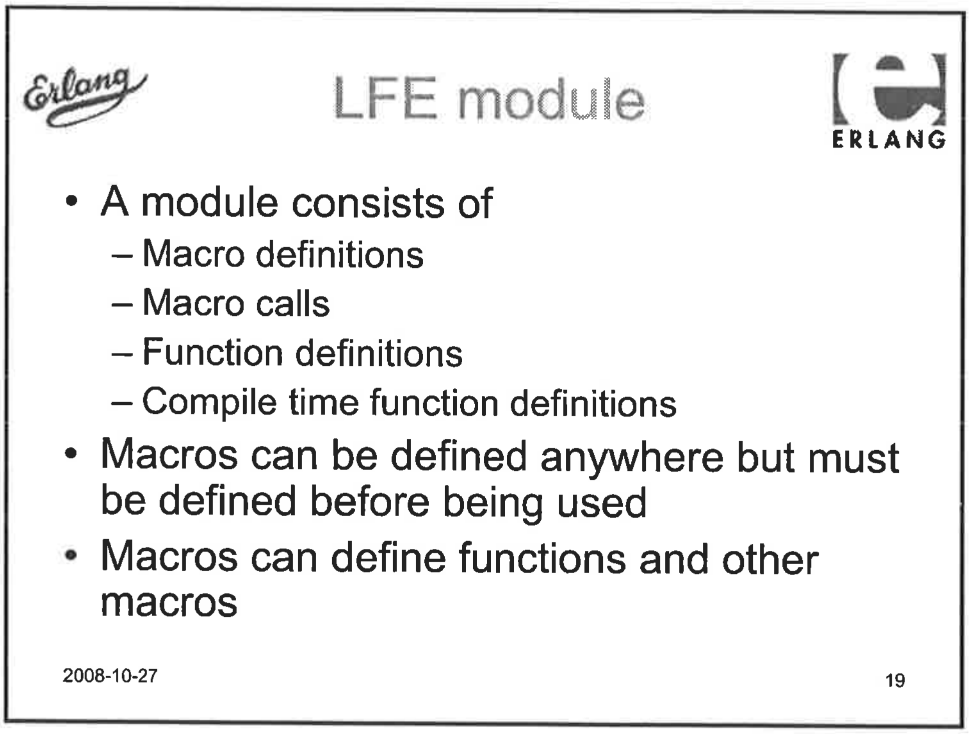
Records

(defrecord name field-def-1 field-def-2 ...)

field-def = field-name | (field-name default-value)

→ (make-name field-name val field-name val ...)
(is-name rec)
(match-name field-name pat field-name pat ...)
(name-field-1 rec)
(set-name-field-1 rec val)
...

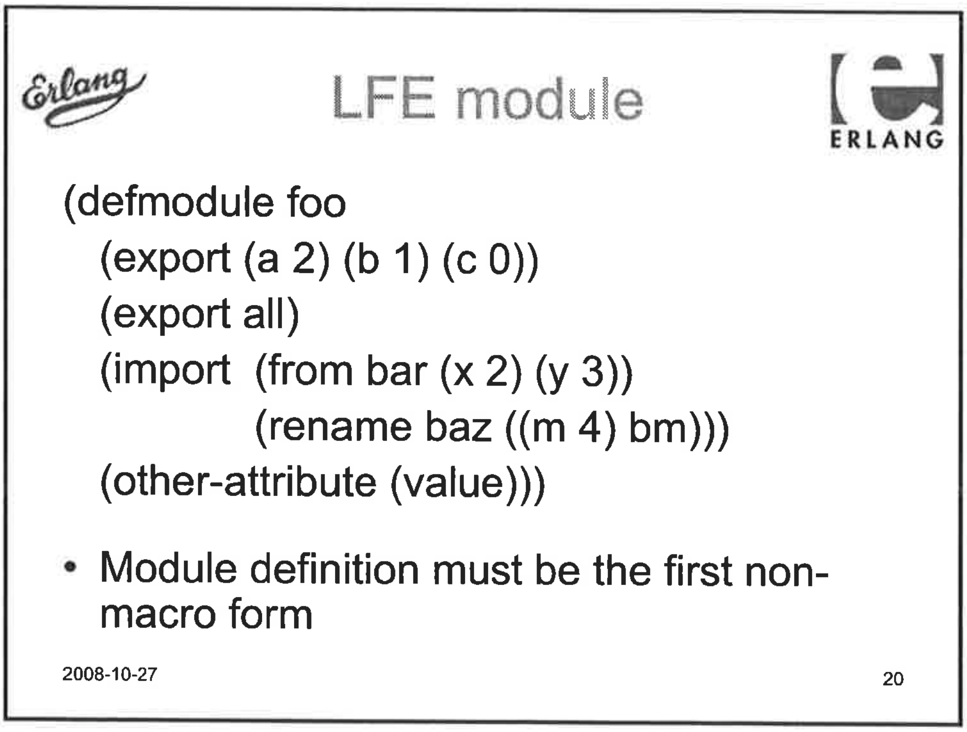
2008-10-27 18



LFE module

- A module consists of
  - Macro definitions
  - Macro calls
  - Function definitions
  - Compile time function definitions
- Macros can be defined anywhere but must be defined before being used
- Macros can define functions and other macros

2008-10-27 19

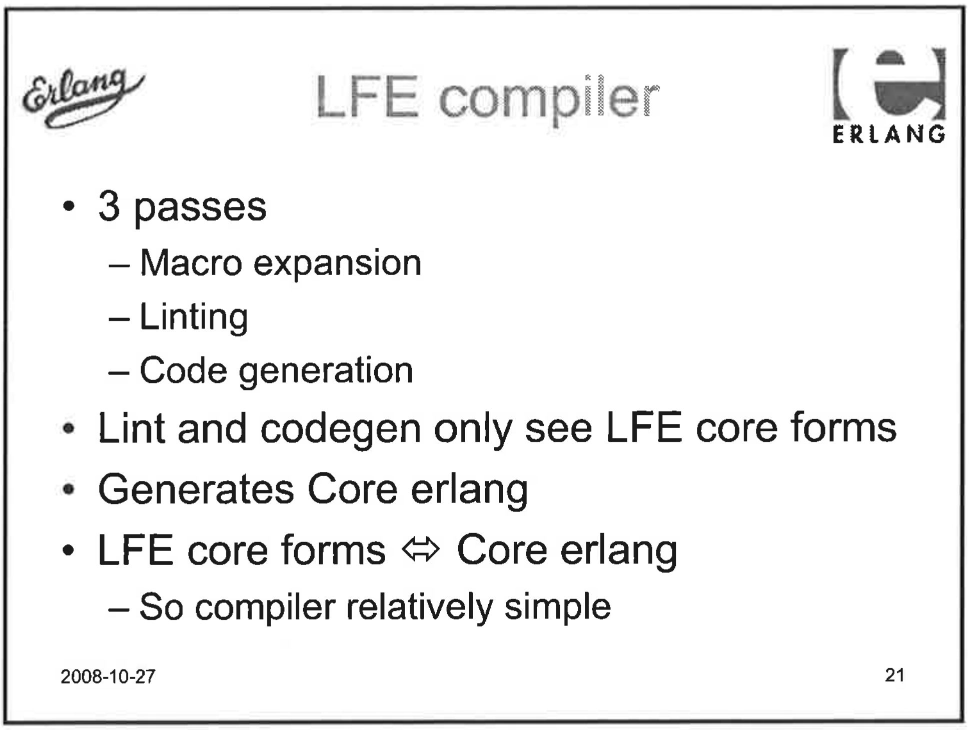


LFE module

```
(defmodule foo
  (export (a 2) (b 1) (c 0))
  (export all)
  (import  (from bar (x 2) (y 3))
           (rename baz ((m 4) bm)))
  (other-attribute (value)))
```

- Module definition must be the first non-macro form

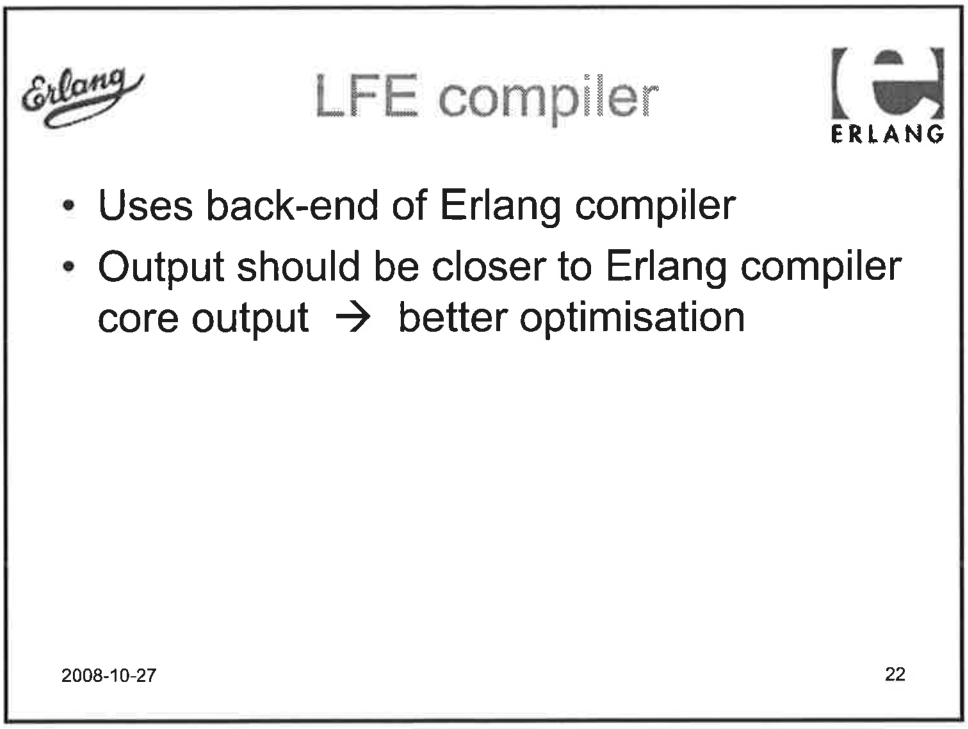
2008-10-27 20



LFE compiler

- 3 passes
  - Macro expansion
  - Linting
  - Code generation
- Lint and codegen only see LFE core forms
- Generates Core erlang
- LFE core forms ⇔ Core erlang
  - So compiler relatively simple

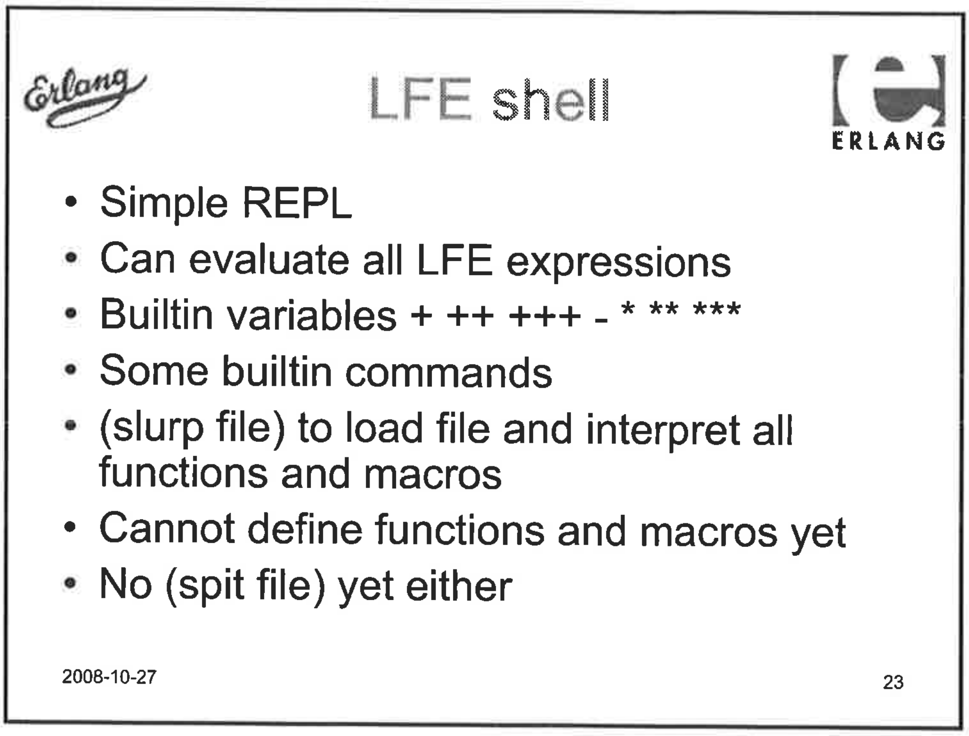
2008-10-27 21



LFE compiler

- Uses back-end of Erlang compiler
- Output should be closer to Erlang compiler core output → better optimisation

2008-10-27 22

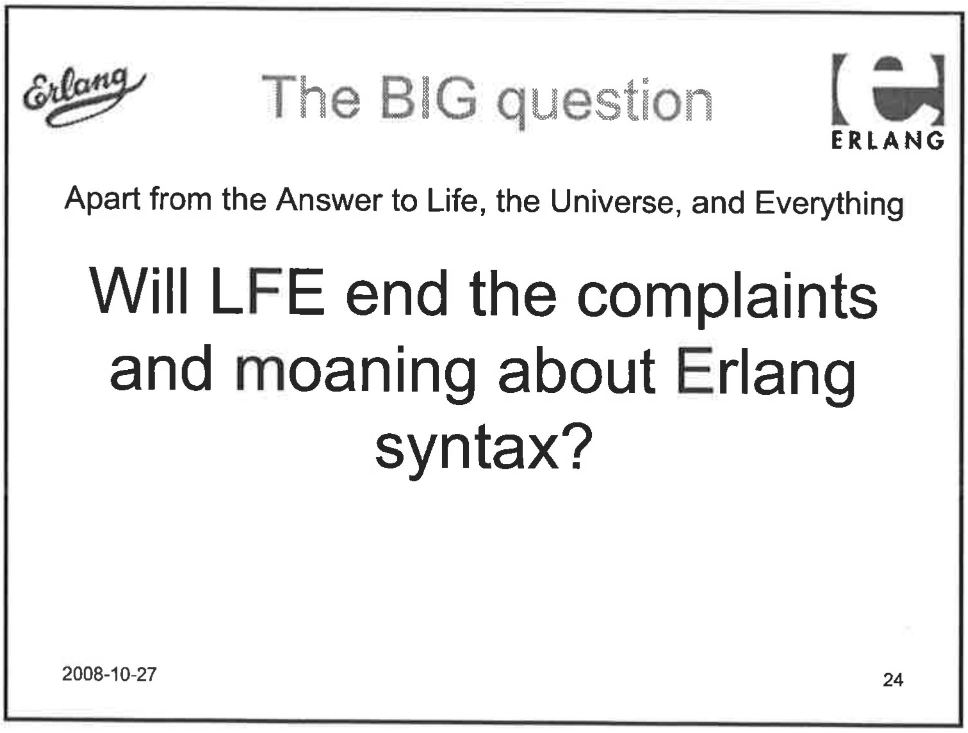


LFE shell

- Simple REPL
- Can evaluate all LFE expressions
- Builtin variables + ++ +++ - * ** ***
- Some builtin commands
- (slurp file) to load file and interpret all functions and macros
- Cannot define functions and macros yet
- No (spit file) yet either

2008-10-27

23



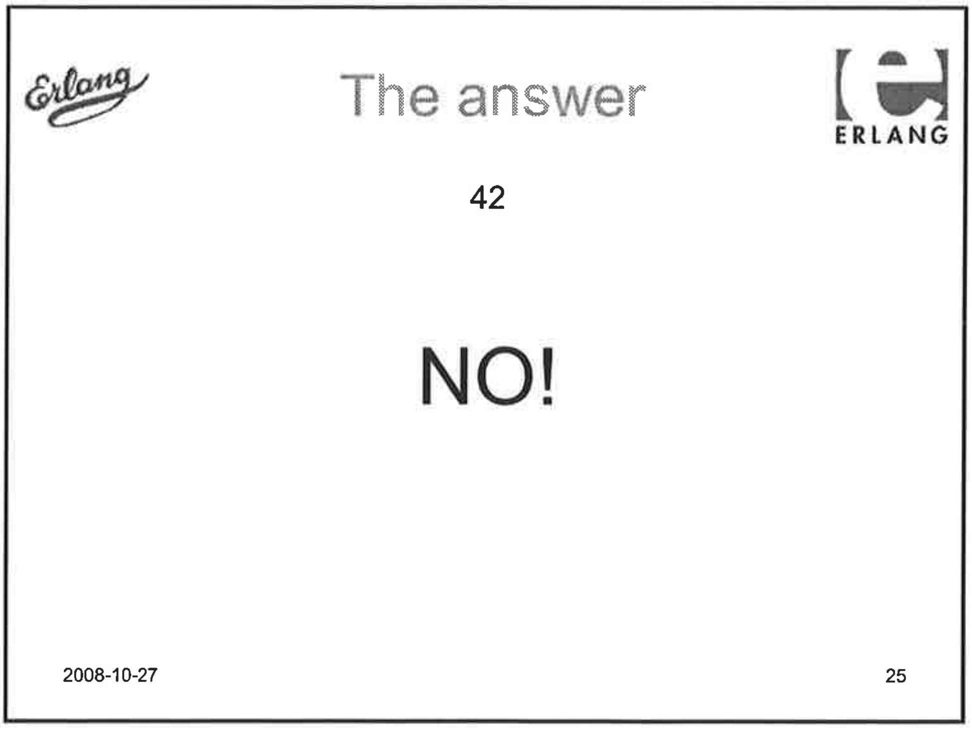
The BIG question

Apart from the Answer to Life, the Universe, and Everything

Will LFE end the complaints and moaning about Erlang syntax?

2008-10-27

24



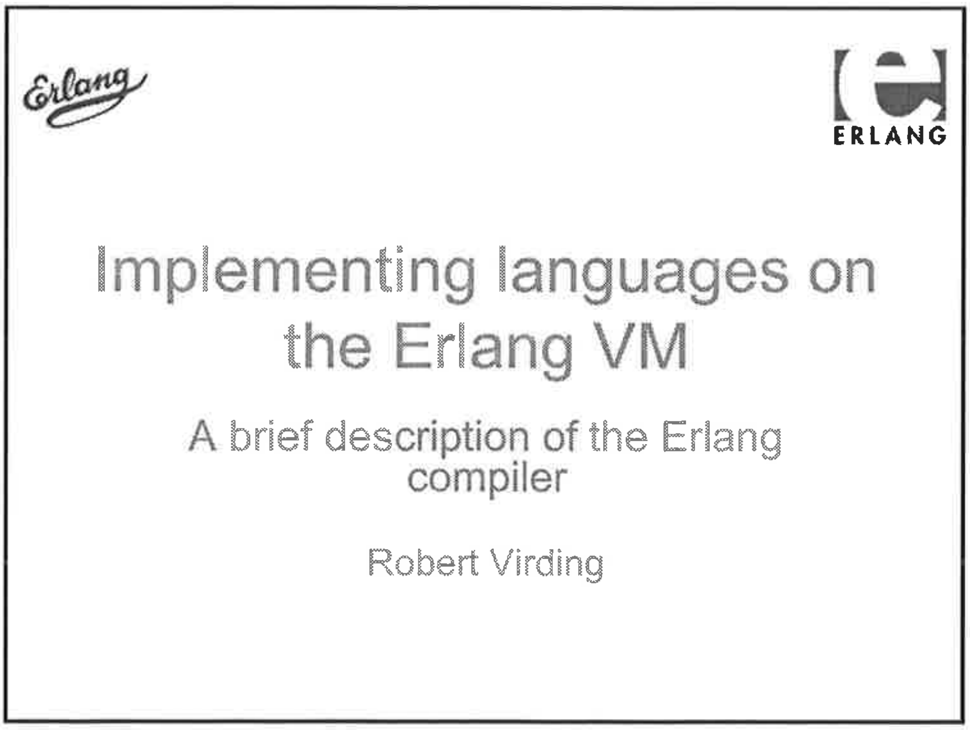
The answer

42

NO!

2008-10-27

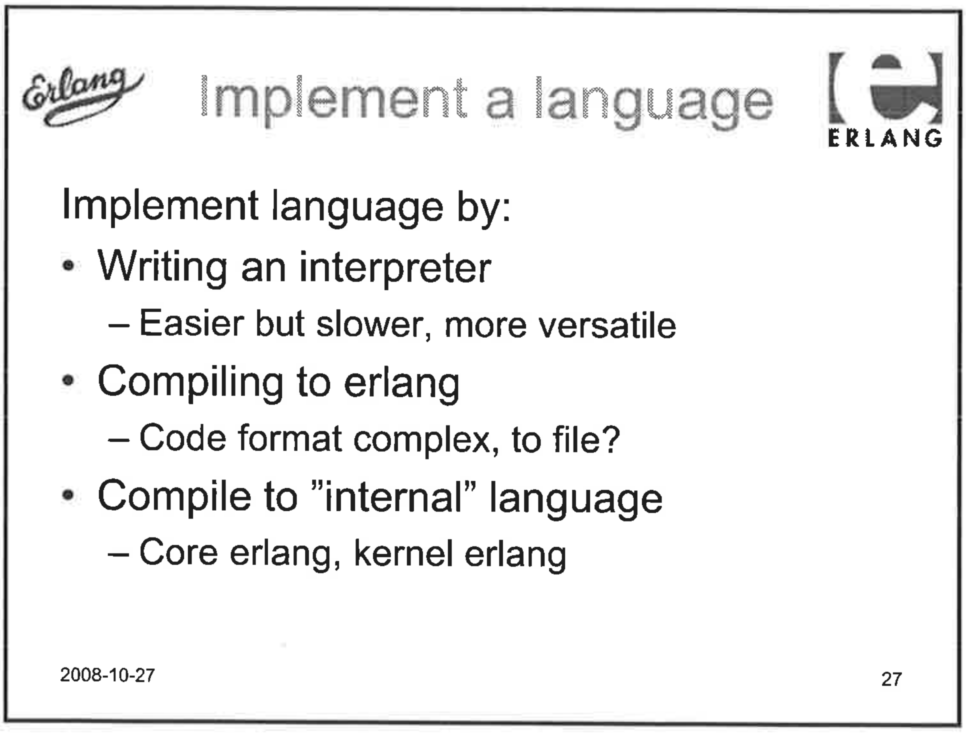
25



Implementing languages on the Erlang VM

A brief description of the Erlang compiler

Robert Virding



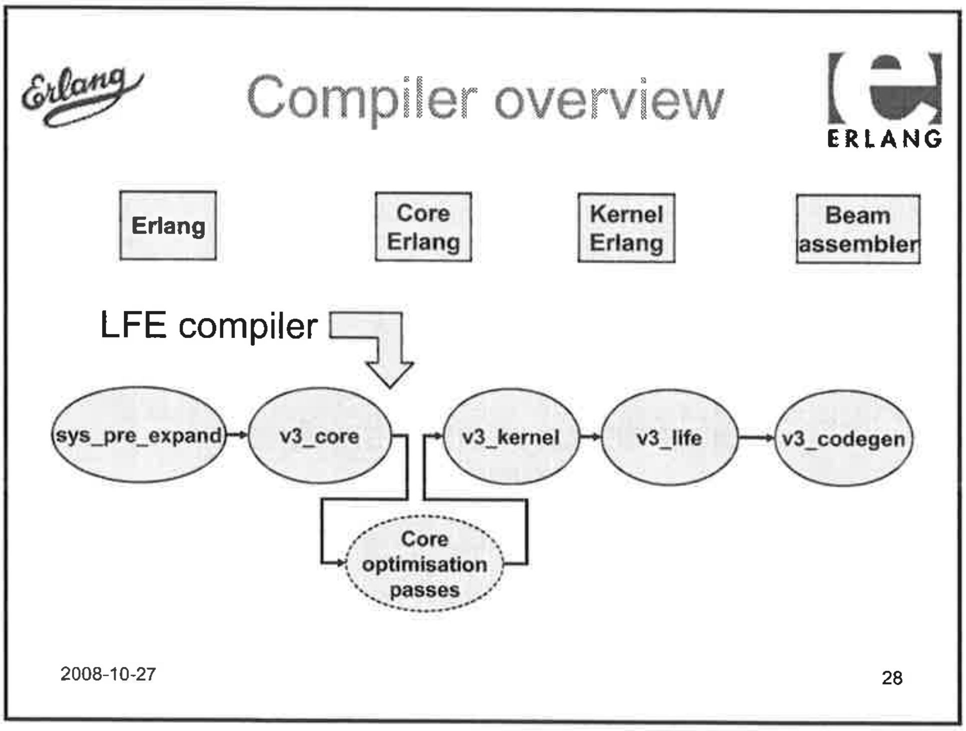
Implement a language

Implement language by:

- Writing an interpreter
  - Easier but slower, more versatile
- Compiling to erlang
  - Code format complex, to file?
- Compile to ”internal” language
  - Core erlang, kernel erlang

2008-10-27

27



Compiler overview

Erlang

Core Erlang

Kernel Erlang

Beam assembler

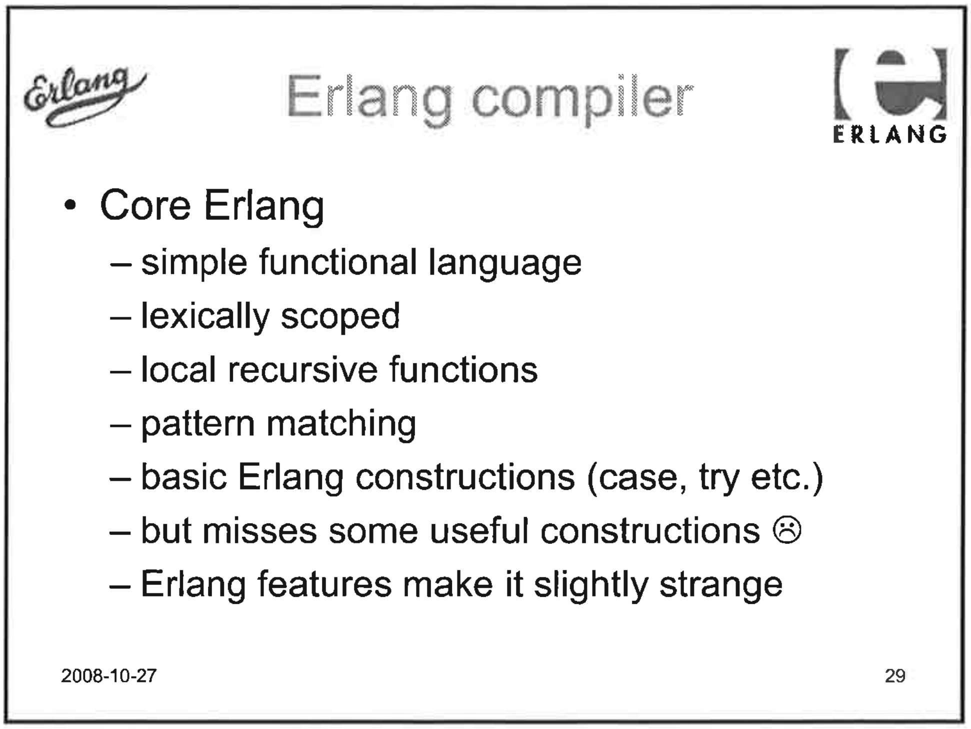
LFE compiler

sys_pre_expand → v3_core → v3_kernel → v3_life → v3_codegen

Core optimisation passes

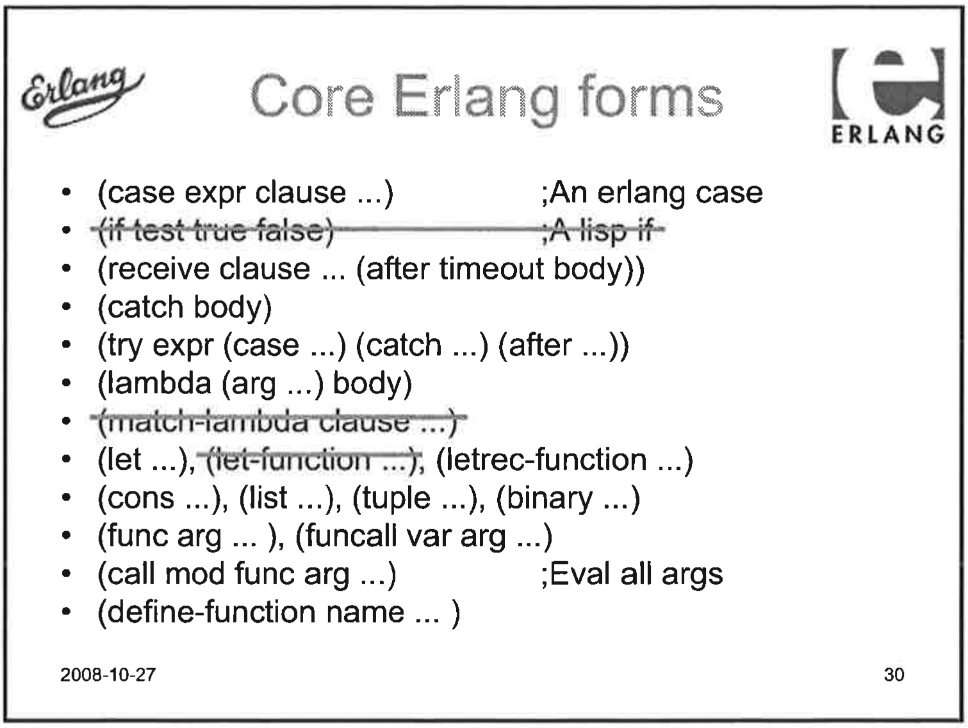
2008-10-27

28



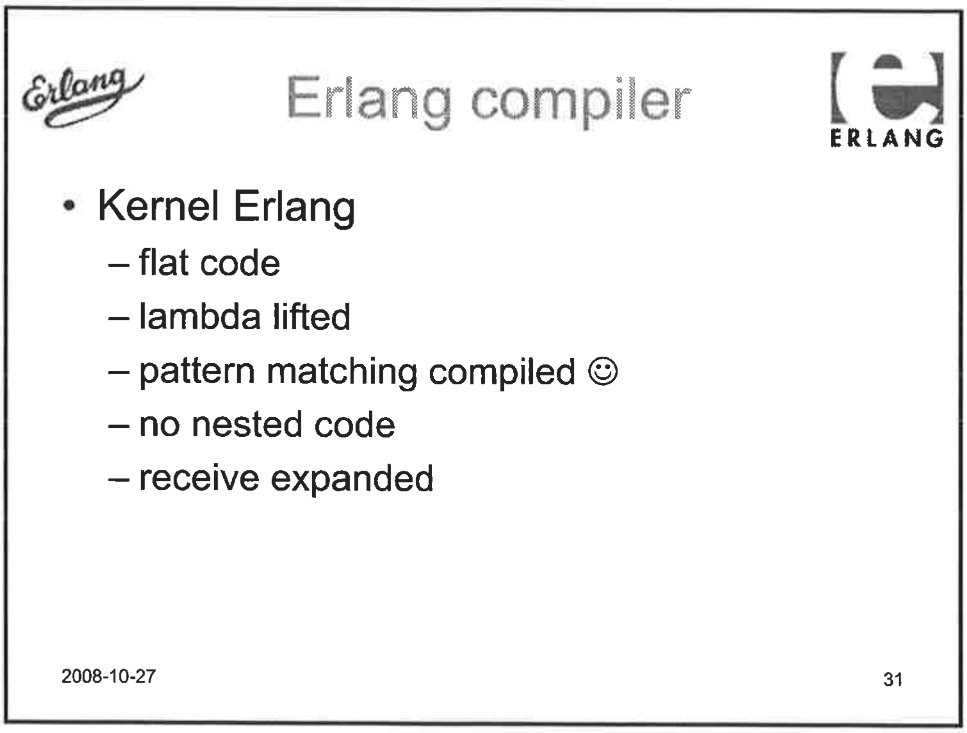
Erlang compiler

- Core Erlang
  - simple functional language
  - lexically scoped
  - local recursive functions
  - pattern matching
  - basic Erlang constructions (case, try etc.)
  - but misses some useful constructions ☹
  - Erlang features make it slightly strange



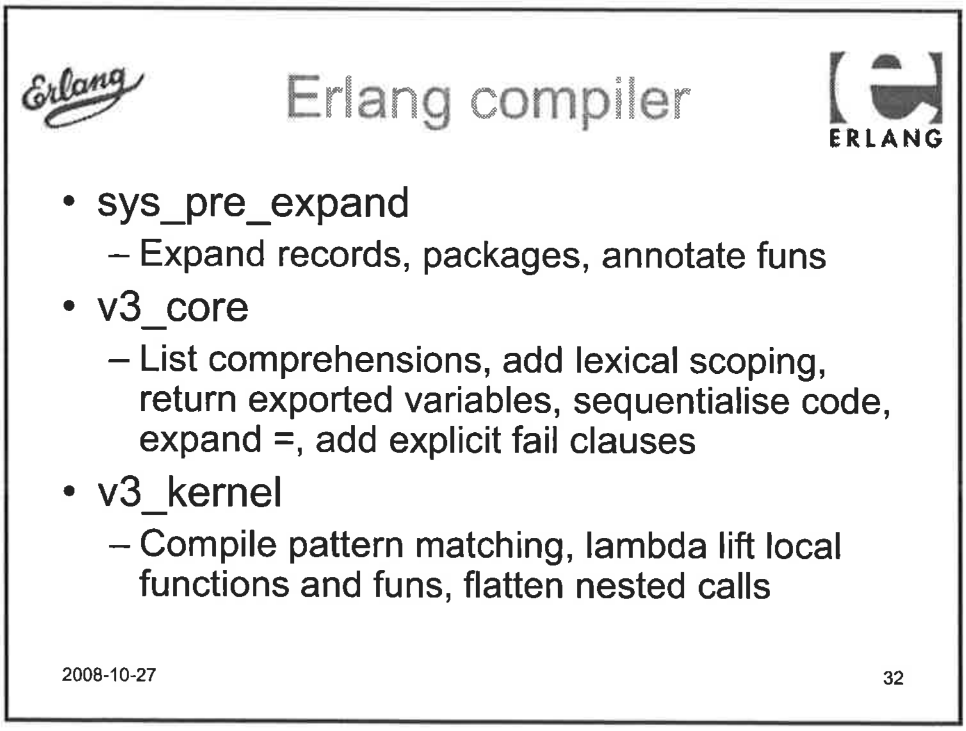
Core Erlang forms

- (case expr clause ...)   ;An erlang case
- ~~(if test true false)   ;A lisp if~~
- (receive clause ... (after timeout body))
- (catch body)
- (try expr (case ...) (catch ...) (after ...))
- (lambda (arg ...) body)
- ~~(match-lambda clause ...)~~
- (let ...), ~~(let-function ...)~~, (letrec-function ...)
- (cons ...), (list ...), (tuple ...), (binary ...)
- (func arg ...), (funcall var arg ...)
- (call mod func arg ...)   ;Eval all args
- (define-function name ...)



Erlang compiler

- Kernel Erlang
  - flat code
  - lambda lifted
  - pattern matching compiled ☺
  - no nested code
  - receive expanded



Erlang compiler

- sys_pre_expand
  - Expand records, packages, annotate funs
- v3_core
  - List comprehensions, add lexical scoping, return exported variables, sequentialise code, expand =, add explicit fail clauses
- v3_kernel
  - Compile pattern matching, lambda lift local functions and funs, flatten nested calls

Rev A

Inside the Erlang VM

with focus on SMP

Prepared by Kenneth Lundin, Ericsson AB
Presentation held at
Erlang User Conference,
Stockholm, November 13, 2008

1 Introduction

The history of support for SMP (Symmetrical Multi Processor) in Erlang started around 1997-1998 as a master thesis work by Pekka Hedqvist with Tony Rogvall (Ericsson Computer Science Lab) as supervisor.

The implementation was run on a Compaq with 4 Pentium Pro 200 Mhz CPU's (an impressive machine in those days) and showed a great potential for scalability with additional processors but suffered from bad IO performance.

The work with SMP did not continue at that time since it was so easy to increase performance by just upgrading the HW to the newest processor. There simply was no business case for it at the time.

The SMP work was restarted at 2005 and now as part of the ordinary development. The work was driven by the Erlang development team at Ericsson with participation and contributions from Tony Rogvall (then at Synapse) and the HiPE group at Uppsala University.

The strategy was (and still is):

- First, "make it work"
- Second, "measure" and find bottlenecks
- Third, "optimize" by removing bottlenecks

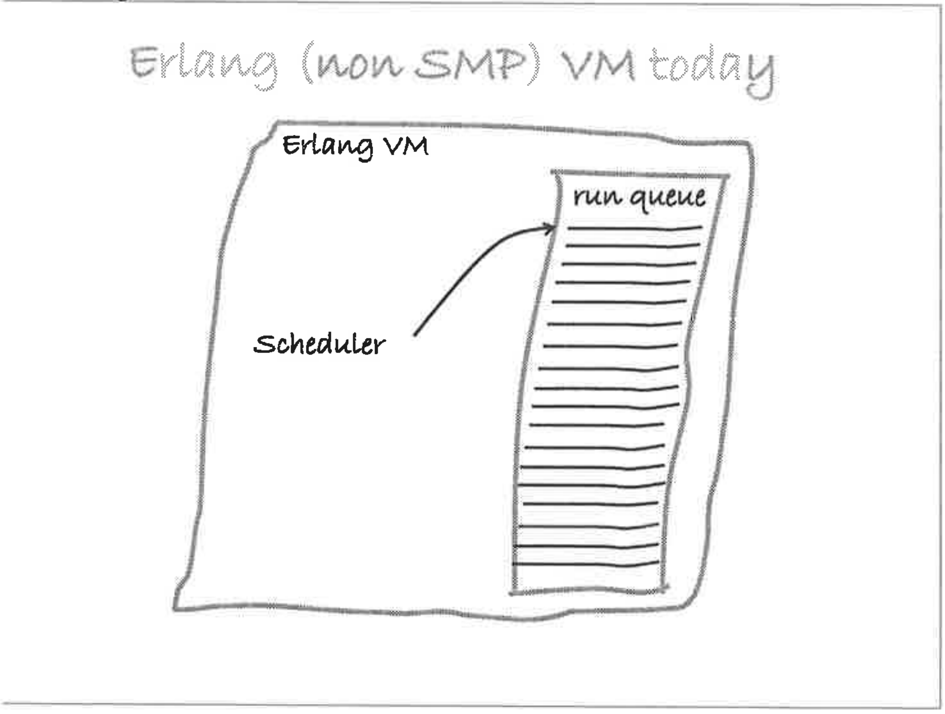
The first release of a stable runtime system with support for SMP came in OTP R11B in May 2006.

This ended the first cycle of the strategy and a new iteration with "measure", "optimize" and "make it work" started. Read more about it in the next pages.

2 How it works

2.1 Erlang VM with no SMP support

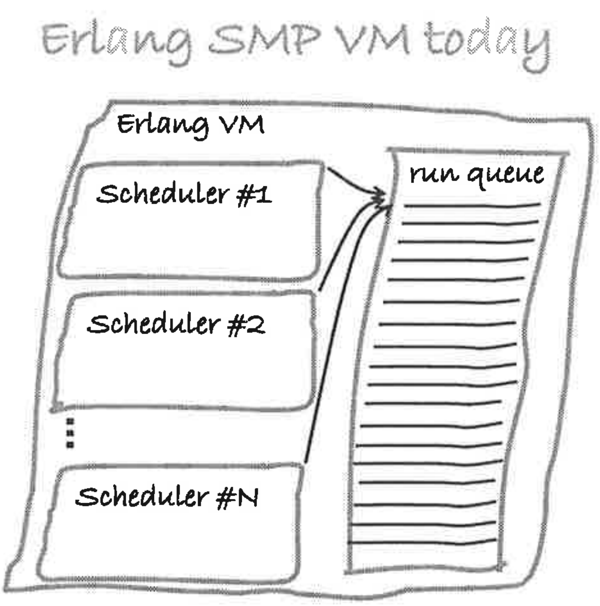
The Erlang VM without SMP support has 1 scheduler which runs in the main process thread. The scheduler picks run able Erlang processes and IO-jobs from the run-queue and there is no need to lock data structures since there is only one thread accessing them.



2.2 Erlang VM with SMP support (in R11B and R12B)

The Erlang VM with SMP support can have 1 to 1024 schedulers which are run in 1 thread each.

The schedulers pick run able Erlang processes and IO-jobs from one common run-queue. In the SMP VM all shared data structures are protected with locks, the run-queue is one example of a shared data structure protected with locks.



2.2.1 First release for use in Products, March 2007

Measurements from a real telecom product showed a 1.7 speed improvement between a single and a dual core system.

It should be noted that it took only about a week to port the telecom system to a new OTP release with SMP support, to a new Linux distribution and to a new incompatible CPU architecture, the Erlang code was not even recompiled.

It took a little longer to get the telecom system in product status, a few minor changes was needed in the Erlang code because Erlang processes now can run truly parallel which changes the timing and ordering of events which the old application code did not count for.

The performance improvements achieved on a dual core processor for a real telecom system where encouraging and after that several other telecom systems have also taken benefit from the SMP support in Erlang.

2.2.2 SMP in R12B

From OTP R12B the SMP version of the VM is automatically started as default if the OS reports more than 1 CPU (or Core) and with the same number of schedulers as CPU's or Cores.

You can see what was chosen at the first line of printout from the `erl` command. E.g.

```
Erlang (BEAM) emulator version 5.6.4 [source] [smp:4] .....
```

The `[smp:4]` above tells that the SMP VM is run and with 4 schedulers.

The default behaviour can be overridden with the "`-smp [enable|disable|auto]`" `auto` is default and to set the number of schedulers, if smp is set to `enable` or `auto` use "`+S Number`" where `Number` is the number of schedulers (1..1024)

Note ! It is normally nothing to gain from running with more schedulers than the number of CPU's or Cores.

Note2 ! On some operating systems the number of CPU's or Cores to be used by a process can be restricted with commands. For example on Linux the command "`taskset`" can be used for this. The Erlang VM will currently only detect number of available CPU's or Cores and will not take the mask set by "`taskset`" into account. Because of this it can happen and has happened that e.g. only 2 Cores are used even if the Erlang VM runs with 4 schedulers. It is the OS that limits this because it take the mask from "`taskset`" into account.

The schedulers in the Erlang VM are run on one OS-thread each and it is the OS that decides if the threads are executed on different Cores. Normally the OS will do this just fine and will also keep the thread on the same Core throughout the execution.

The Erlang processes will be run by different schedulers over time because they are picked from a common run-queue by the first scheduler that becomes available.

3 Performance and scalability

The SMP VM with only one scheduler is slightly slower (10%) than the non SMP VM. This is because the SMP VM need to use locks for all shared datastructures. But as long as there are no lock-conflicts the overhead caused by locking is not that high (it is the lock conflicts that takes time).

This explains why it in some cases can be more efficient to run several SMP VM's with one scheduler each instead on one SMP VM with several schedulers. Of course the running of several VM's require that the application can run in many parallel tasks which has no or very little communication with each other.

If a program scale well with the SMP VM over many cores depends very much on the characteristics of the program, some programs scale linearly up to 8 and even 16 cores while other programs barely scale at all even on 2 cores.

This might sound bad, but in practice many real programs scale well on the number of cores that are common on the market today, see below.

Real telecom products supporting a massive number if simultaneously ongoing "calls" represented as one or several Erlang processes per core have shown very good scalability on dual and quad core processors.

Note, that these products was written in the normal Erlang style long before the SMP VM and multi core processors where available and they could benefit from the Erlang SMP VM without changes and even without need to recompile the code.

4 Our strategy with SMP

Already from the beginning when we started implementation of the SMP VM we decided on the strategy:

"First make it work, then measure, then optimize".

We are still following this strategy consistently since the first stable working SMP VM that we released in May 2006 (R11B).

Another important part of the strategy is to hide the problems and awareness of SMP execution for the Erlang programmer. Erlang programs should be written as usual using processes for parallel tasks, the utilization of CPUs and cores should be handled by the Erlang VM. It must be easy and cost effective to utilize multicore and SMP HW with Erlang this is one of our absolute strengths compared to other programming languages.

There will be new BIF's for SMP related stuff but we try to avoid that as much as possible. We think it is preferable to configure SMP related things such as number of cores to use, which cores to use on the OS level and as parameters to the Erlang VM at startup.

The principle is that an Erlang program should run perfectly well on any system no matter what number of cores or processors there are.

5 Next steps with SMP and Erlang

There are more known things to improve and we address them one by one taking the one we think gives most performance per implementation effort first and so on.

We are now putting most focus on getting consistent better scaling on many cores (more than 4).

The SMP implementation is continually improved in order to get better performance and scalability. In each service release R12B-1, 2, 3, 4, 5 , ..., R13B-0, 1, …, R14B etc. you will find new optimizations.

5.1 Some known bottlenecks

Below some of the most significant bottlenecks that we know of are described, there are for sure more bottlenecks than this and we intend to address them one after one. It is worth noting that after removal of one bottleneck there might be new ones coming up and the already known ones may have got changed importance.

5.1.1 The common run-queue

The single common run-queue will become a dominant bottleneck when the number of CPU's or Cores increase.

This will be visible from 4 cores and upwards, but 4 cores will probably still give ok performance for many applications.

We are working on a solution with one run-queue per scheduler as the most important improvement right now. Read more about this later in the document.

5.1.2 Ets tables

Ets tables involves locking. Before R12B-4 there was 2 locks involved in every access to an ets-table, but in R12B-4 the locking of the meta-table is optimized to reduce the conflicts significantly (as mentioned earlier it is the conflicts that are expensive).

If many Erlang processes access the same table there will be a lot of lock conflicts causing bad performance especially if these processes spend a majority of their work accessing ets-tables.

The locking is on table-level not on record level. An obvious solution is to introduce more fine granular locking.

Note! that this will have impact on Mnesia as well since Mnesia is a heavy user of ets-tables.

5.1.3 Message passing

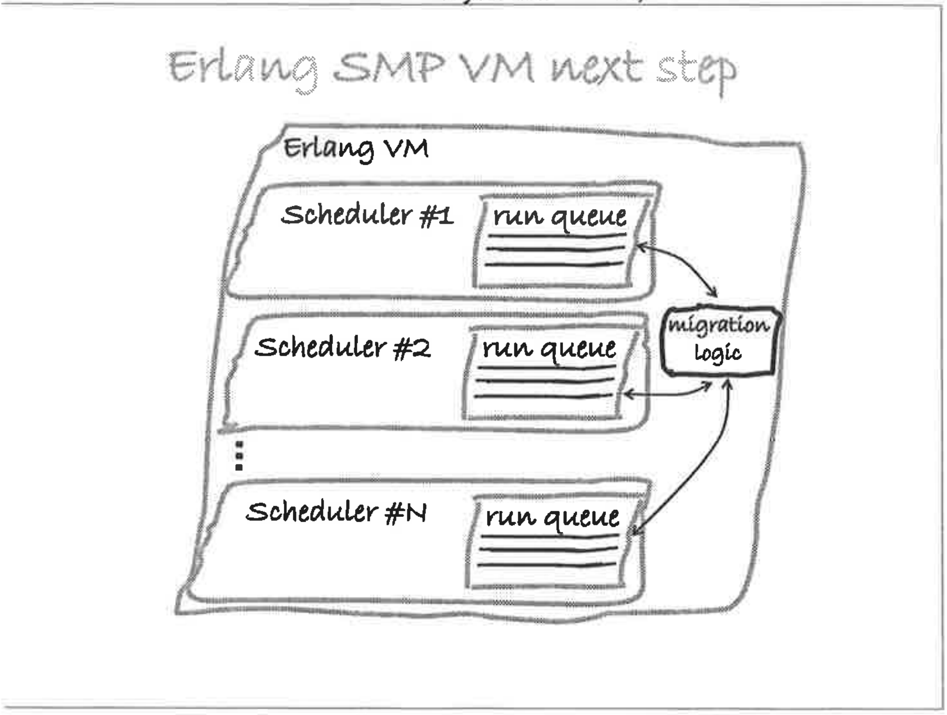
When many processes are sending messages to the same receiving process there will be a lot of lock conflicts. There are ways to optimize this by reducing the amount of work being done while having the lock.

5.1.4 A process can block the scheduler

If a process is blocked waiting to get a lock for example to access an ets-table the whole scheduler is blocked doing nothing until the lock is accuired and the process can continue it's execution. This can be improved by introducing what we call "*process level locking"* which means that if a process is blocked waiting to get a lock it will be scheduled out and the scheduler will schedule in the next process from the run-queue instead. We have already implemented and measured on this solution and concluded that it probably can be introduced when the separate run-queues are in place. We still need to verify that it does not degrade performance for certain special cases.

5.2 Separate run-queues per scheduler

The next big performance improvement regarding SMP support in the Erlang runtime system is the change from having one common run-queue to having a separate run-queue per scheduler. This change will decrease the number of lock conflicts dramatically for systems with many cores or processors. The improvement in performance will in many applications be significant already from 4 cores and will of course be even more noticeable in systems with 8, 16 or even more cores.



5.2.1 Migration logic

When there are separate run-queues per scheduler the problem is moved from the locking conflicts when accessing the run-queue to the migration logic which must be both efficient and reasonably fair.

The implementation we have so far will need a lot more benchmarking and fine tuning before it works optimally. It works roughly like this:

The maximum number of run able processes over all schedulers is measured approximately 4 times per second. This value divided by number of schedulers is then used to trigger migration of processes from one scheduler to another scheduler.

When a scheduler is about to schedule in a new process it will first check if its number of run able processes is above the max value described above and if it is it will migrate the process to another scheduler according to the migration path set up.

There are also 2 other occasions in addition to the "schedule in" of a new process when a process migration can occur:

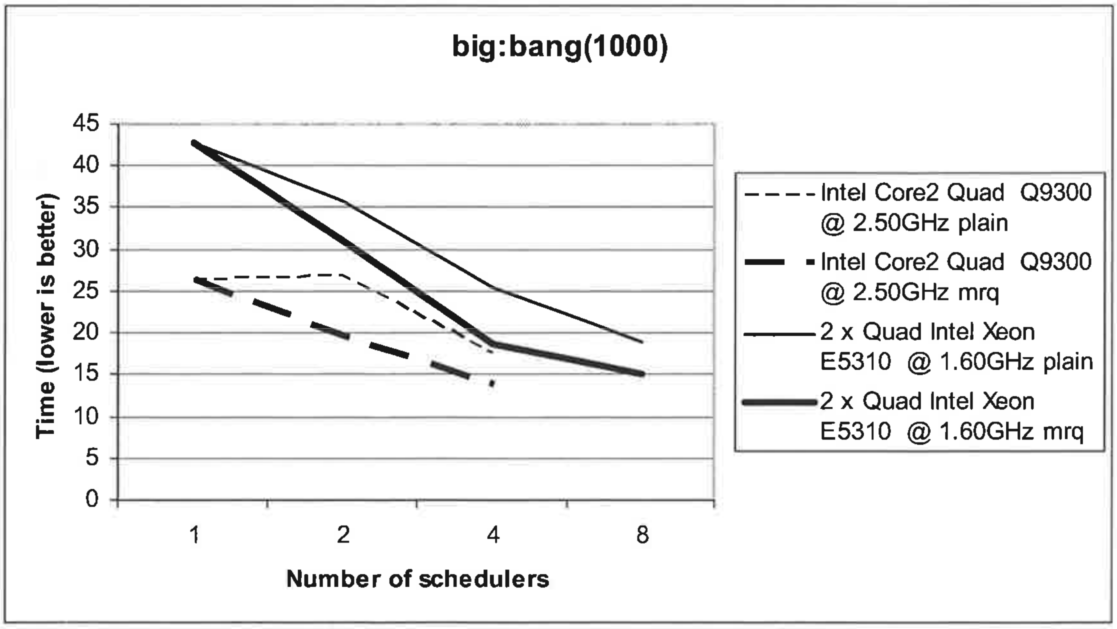
1. If a scheduler gets out of jobs it will steal jobs from other schedulers.
2. Underloaded schedulers will also steal jobs from heavily overloaded schedulers in their migration paths.

Below follows some measurements that show early indications of the improvements the system with separate run-queues per scheduler and the migration logic described above will give.

The graph below shows the results from running the benchmark "big bang" with 1, 2, 4, 8 schedulers on both the current system with one single run-queue and on the next to come system with multiple run-queues one per scheduler.

The benchmark spawns 1000 processes which all sends a 'ping' message to all other processes and answer with a 'pong' message for all 'ping' it receives.

The "fat" lines in the graph shows the multiple run-queue case and as can be seen the improvement is significant.



6 Frequently Asked Questions

6.1 Is there any difference in the .beam file depending on if it should run in a SMP or non SMP system?

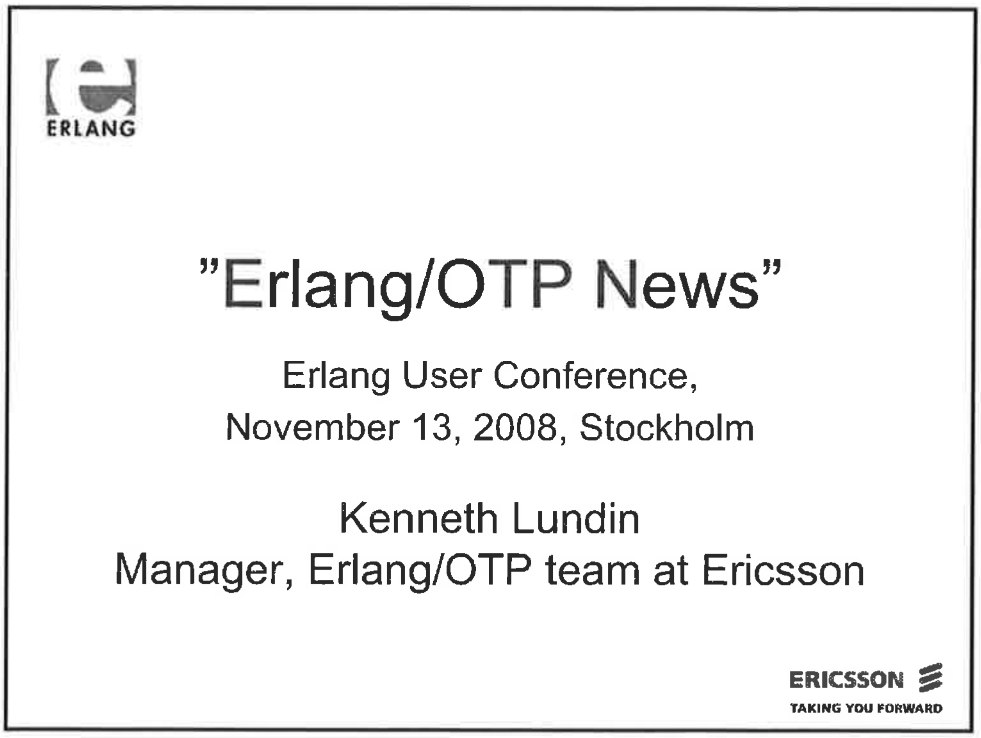
As long as the module is not compiled with "native" option with a HiPE enabled system the .beam files are the same and can be run in both SMP and non SMP systems.

6.2 Can an Erlang process be locked to a specific processor core?

An Erlang process can not be locked to a specific processor by the programmer and this is intentional. In a future release it might be possible to lock a scheduler to a specific core.

6.3 What is the relation between asynch threads and SMP?

The asynch thread pool has nothing with SMP todo. The asynch threads are only used by the file driver and by user written drivers that specifically uses the thread pool. The file driver uses this to avoid locking of the whole Erlang VM for a longer time period in case of a lengthy file operation. The asynch threads was introduced long before the SMP support in the VM and works for the non SMP VM as well. In fact the asynch threads are even more important for a non SMP system because without it a lengthy file operation will block the whole VM.

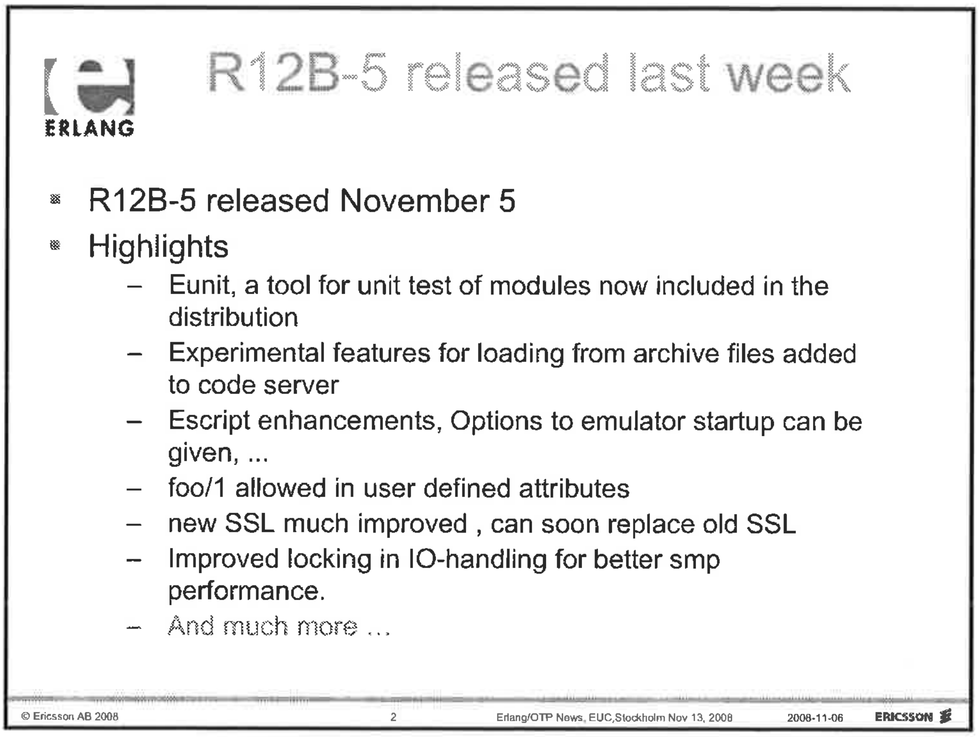


"Erlang/OTP News"

Erlang User Conference,
November 13, 2008, Stockholm

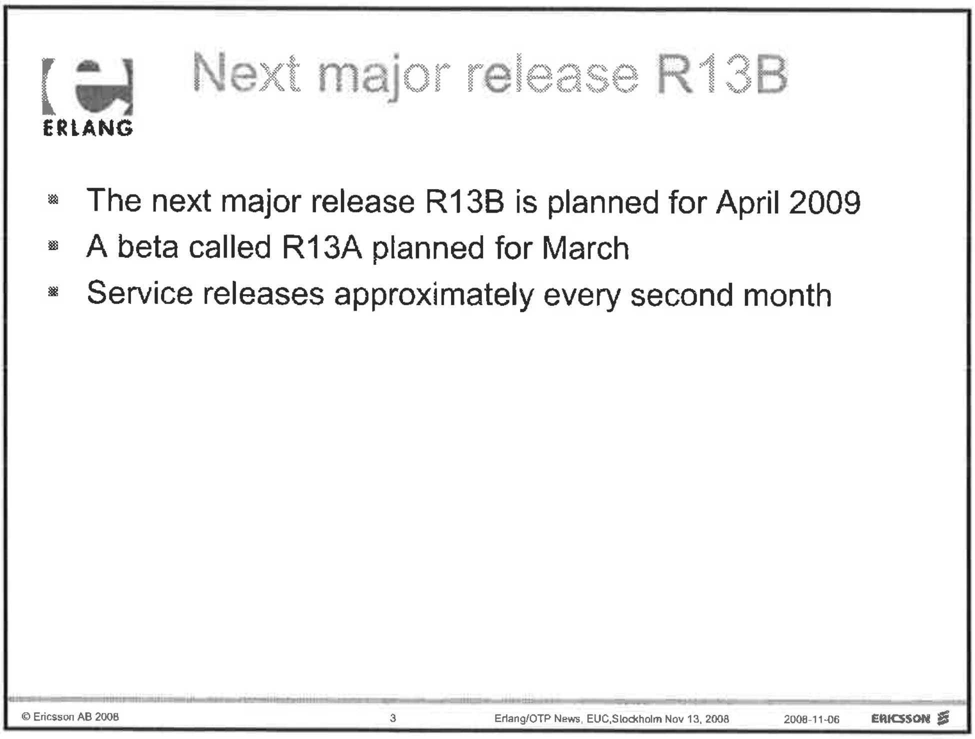
Kenneth Lundin
Manager, Erlang/OTP team at Ericsson

TAKING YOU FORWARD



R12B-5 released last week

- R12B-5 released November 5
- Highlights
  - Eunit, a tool for unit test of modules now included in the distribution
  - Experimental features for loading from archive files added to code server
  - Escript enhancements, Options to emulator startup can be given, ...
  - foo/1 allowed in user defined attributes
  - new SSL much improved , can soon replace old SSL
  - Improved locking in IO-handling for better smp performance.
  - And much more ...

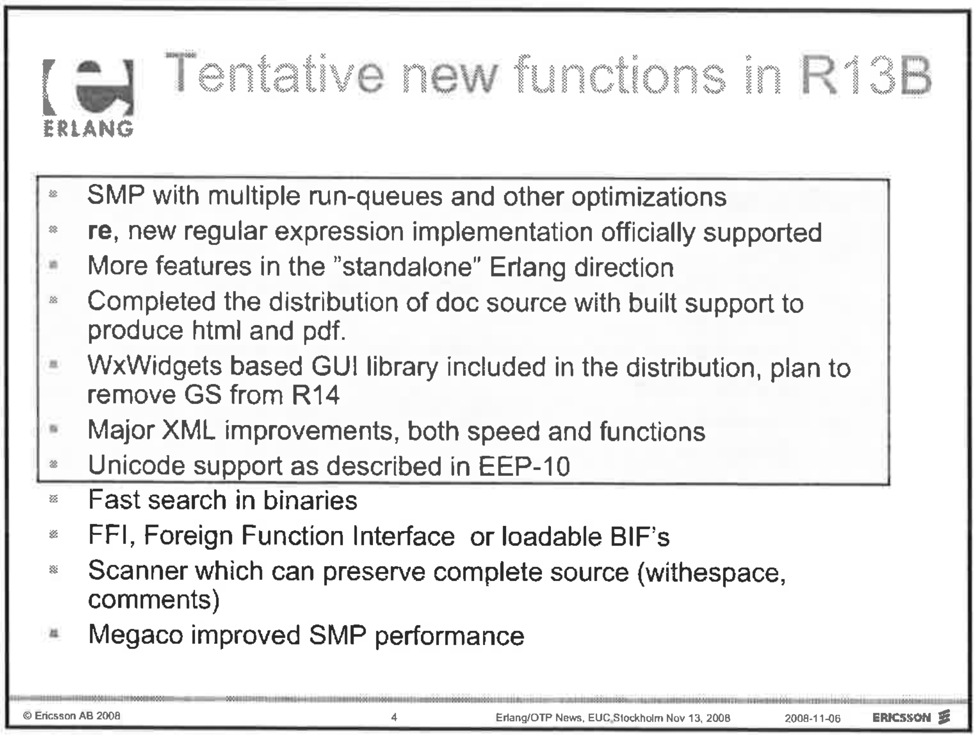


ERLANG

Next major release R13B

- The next major release R13B is planned for April 2009
- A beta called R13A planned for March
- Service releases approximately every second month

© Ericsson AB 2008 3 Erlang/OTP News, EUC,Stockholm Nov 13, 2008 2008-11-06 ERICSSON



ERLANG

Tentative new functions in R13B

- SMP with multiple run-queues and other optimizations
- **re**, new regular expression implementation officially supported
- More features in the ”standalone” Erlang direction
- Completed the distribution of doc source with built support to produce html and pdf.
- WxWidgets based GUI library included in the distribution, plan to remove GS from R14
- Major XML improvements, both speed and functions
- Unicode support as described in EEP-10
- Fast search in binaries
- FFI, Foreign Function Interface or loadable BIF’s
- Scanner which can preserve complete source (withespace, comments)
- Megaco improved SMP performance

© Ericsson AB 2008 4 Erlang/OTP News, EUC,Stockholm Nov 13, 2008 2008-11-06 ERICSSON

Erlang/OTP User Conference 2008

Speakers

Thomas	Arts	Quviq	Göteborg	Sweden	thomas.arts@quviq.com
Garry	Bulmer		Coventry	England	gbulmer@gmail.com
Jakob	Cederlund	Ericsson AB OTP	Stockholm	Sweden	
Francesco	Cesarini	Erlang Training & Consulting	London	England	francesco@erlang-consulting.com
Vlad	Dumitrescu	HiQ	Göteborg	Sweden	vladdu55@gmail.com
Zoltán	Horváth	Eötvös Loránd University	Budapest	Hungary	hz@inf.elte.hu
John	Hughes	Quviq	Göteborg	Sweden	john.hughes@quviq.com
Robert	Ippolito	Mochi Media	San Francisco	USA	bob@mochimedia.com
Kenneth	Lundin	Ericsson AB OTP	Stockholm	Sweden	kenneth.lundin@ericsson.com
Hans	Nilsson	Ericsson AB	Stockholm	Sweden	Hans.R.Nilsson@ericsson.com
Anders	Nygren	Telexpertise de Mexico		Mexico	anders.nygren@gmail.com
Kostis	Sagonas	University of Uppsala	Uppsala	Sweden	kostis@it.uu.se
Michal	Slaski	Erlang Training & Consulting	London	England	michal@erlang-consulting.com
Simon	Thompson	University of Kent	Canterbury	England	S.J.Thompson@kent.ac.uk
Robert	Virding		Stockholm	Sweden	rvirding@gmail.com

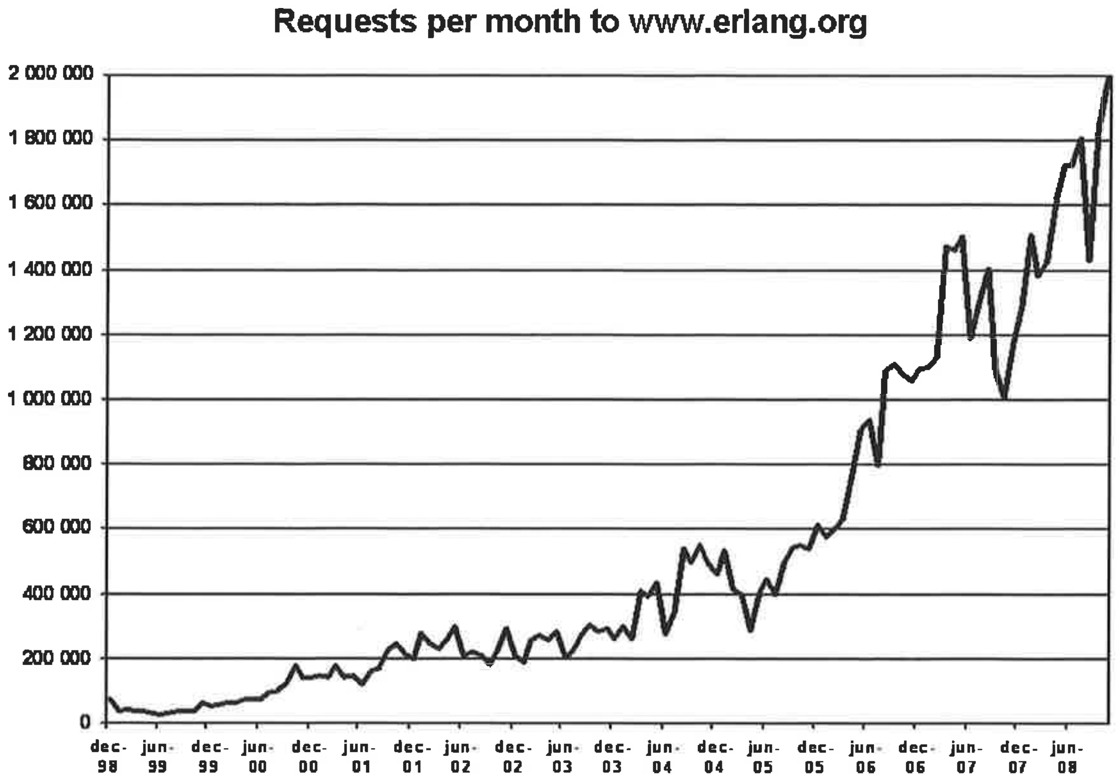
Participants

Tomas	Abrahamsson	Ericsson AB	Linköping	Sweden	tomas.abrahamsson@ericsson.com
Roberto	Aloi	Erlang Training & Consulting	London	England	roberto@erlang-consulting.com
Kristoffer	Andersson	Synapse Mobile Networks	Stockholm	Sweden	kristoffer.andersson@synap.se
Peter	Andersson	Ericsson AB OTP	Stockholm	Sweden	
Robert	Andersson	University of Uppsala	Uppsala	Sweden	robert.andersson.4801@student.uu.se
Ingela	Anderton-Andin	Ericsson AB OTP	Stockholm	Sweden	ingela@theheartofgold.org
Anna	Anesiadou-Hansen		Euskirchen	Germany	anna.a-hansen@web.de
Marcus	Arendt	Synapse Mobile Networks	Stockholm	Sweden	marcus@arendt.se
Joe	Armstrong	Ericsson AB	Stockholm	Sweden	joearms@gmail.com
Tuncer	Ayaz		Aachen	Germany	tuncer.ayaz@gmail.com
Henrik	Back	Mobile Arts AB	Uppsala	Sweden	Henrik.back@mobilearts.se
John-Olof	Bauner	Ericsson AB	Stockholm	Sweden	john-olof.bauner@ericsson.com
Per	Bergqvist	Synapse Mobile Networks	Stockholm	Sweden	per@synap.se
Xingdong	Bian	Erlang Training & Consulting	London	England	bian@erlang-consulting.com
Martin	Björklund	Tail-f Systems AB	Stockholm	Sweden	mbj@tail-f.com
Benjamin	Black	Joyent	Seattle	USA	bb@joyent.com
Ivan	Bodunov	Nokia Siemens Networks	Espoo	Finland	ivan.bodunov@gmail.com
Mikael	Bohlin	Itancan Consulting			mikael.bohlin@itancan.com
Hans	Bolinder	Ericsson AB OTP	Stockholm	Sweden	hans.bolinder@ericsson.com
Franc	Bozic	eXcenter d.o.o.	Radovljica	Slovenia	Franc@akcija.net
Mikael	Bylund	TeliaSonera Sverige AB	Uppsala	Sweden	mikael.bylund@teliasonera.com
Göran	Båge	Mobile Arts AB	Stockholm	Sweden	goran.bage@mobilearts.com
Larry	Canady	University of Uppsala	Uppsala	Sweden	laca1583@student.uu.se
Geoff	Cant	Process-One	Paris	France	nem@erlang.geek.nz
Richard	Carlsson	Kreditor	Stockholm	Sweden	Richard.Carlsson@Kreditor.se
Mats	Cronqvist	Kreditor	Stockholm	Sweden	Mats.Cronqvist@Kreditor.se
Graham	Crowe	Ericsson AB	Stockholm	Sweden	graham.crowe@ericsson.com
Björn-Egil	Dahlberg	Ericsson AB OTP	Stockholm	Sweden	
Anders	Dahlin	Dahlin Energy AB	Stockholm	Sweden	anders@dahlinenergy.se
Matthew	Dempsky	Mochi Media	San Francisco	USA	matthew@mochimedia.com
Bjarne	Däcker	CSLab	Stockholm	Sweden	bjarne@cs-lab.org
Niclas	Eklund	Ericsson AB OTP	Stockholm	Sweden	nick@erix.ericsson.se
Emad	El-Haraty	Mochi Media	San Francisco	USA	emad@mochimedia.com
Martin	Engström	Kreditor	Stockholm	Sweden	Martin.Engstrom@Kreditor.se
Sverker	Eriksson	Ericsson AB OTP	Stockholm	Sweden	
Maxim	Escurel	Kreditor	Stockholm	Sweden	Maxim.Escurel@Kreditor.se
Gerd	Flaig	Google	Zürich	Switzerland	gefla@google.com
Magnus	Fröberg	Kreditor	Stockholm	Sweden	Magnus.Froberg@Kreditor.se
Dan	Gudmundsson	Ericsson AB OTP	Stockholm	Sweden	dgud@erix.ericsson.se
Björn	Gustavsson	Ericsson AB OTP	Stockholm	Sweden	bjorn@erix.ericsson.se
Rainer	Hansen	Ericsson AB	Bonn	Germany	rainer.hansen@ericsson.com
Dale	Harvey	Hypernumbers	London	England	dale@hypernumbers.com

Andreas	Hasselberg	Kreditor	Stockholm	Sweden	Andreas.Hasselberg@Kreditor.se
Dragan	Havelka	Mobile Arts AB	Stockholm	Sweden	dragan.havelka@mobilearts.com
Per	Hedeland	Tail-f Systems AB	Stockholm	Sweden	per@tail-f.com
Pekka	Hedqvist	Optimobile	Stockholm	Sweden	Pekka.Hedqvist@OptiMobile.SE
Anders	Heimer	TietoEnator	Karlstad	Sweden	Anders.Heimer@tietoenator.com
Andreas	Hellström	TeliaSonera Sverige AB	Uppsala	Sweden	andreas.hellstrom@teliasonera.com
Magnus	Henoch	Erlang Training & Consulting	Göteborg	Sweden	magnus@erlang-consulting.com
Søren	Hilmer	wideTrail	Allingåbro	Denmark	sh@widetrail.dk
Sean	Hinde	Synapse Mobile Networks	Christchurch	England	sean@synap.se
Anders	Hjelm	Consoden AB	Uppsala	Sweden	Anders.Hjelm@consoden.se
Henrik	Hoffström	Ericsson AB		Sweden	henrik.hoffstrom@ericsson.com
Klas	Johansson	Ericsson AB	Linköping	Sweden	klas.johansson@ericsson.com
Mikael	Kardell	Kreditor	Stockholm	Sweden	Mikael.Kardell@Kreditor.se
Micke	Karlsson	Mikadako AB	Utö	Sweden	micke@mikadako.com
Mikael	Karlsson	Creado Systems	Stockholm	Sweden	mikael.karlsson@creado.com
Roland	Karlsson	Erlang Training & Consulting	Stockholm	Sweden	roland@erlang-consulting.com
Bengt	Kleberg	Ericsson AB	Stockholm	Sweden	bengt.kleberg@ericsson.com
Mikael	Laaksonen	Mobile Arts AB	Stockholm	Sweden	mikael.laaksonen@mobilearts.com
Huiqing	Li	University of Kent	Canterbury	England	H.Li@kent.ac.uk
Tobias	Lindahl	Kreditor	Stockholm	Sweden	Tobias.Lindahl@Kreditor.se
Adam	Lindberg	Erlang Training & Consulting	London	England	adam@erlang-consulting.com
Christopher	Lindbergh	Streamnow AB	Luleå	Sweden	christopher@scg.nu
Mikael	Lindmark	Kreditor	Stockholm	Sweden	Mikael.Lindmark@Kreditor.se
Mattias	Ljunggren	Synapse Mobile Networks	Stockholm	Sweden	mattias@synap.se
Peter	Lund	Synapse Mobile Networks	Stockholm	Sweden	peter.lund@synap.se
Peter Henry	Mander	T-Mobile	Hatfield	England	Peter.Mander@t-mobile.co.uk
Tomas	Mannerstedt	Combitech AB	Stockholm	Sweden	mannerstedt@gmail.com
Håkan	Mattsson	Ericsson AB OTP	Stockholm	Sweden	hakan@erix.ericsson.se
Sean	McEvoy	Erlang Training & Consulting	Stockholm	Sweden	sean@erlang-consulting.com
Peter	Mechlenborg	Mu	Aarhus	Denmark	peter@mu.dk
Hunter	Morris	Smarkets Ltd.	London	England	hunter.morris@smarkets.com
Chandru	Mullaparthi	T-Mobile	Hatfield	England	chandrashekhar.mullaparthi@gmail.com
Peter	Nagy	Ericsson AB	Budapest	Hungary	peter.nagy@ericsson.com
Daniel	Nibon	Mobile Arts AB	Stockholm	Sweden	daniel.nibon@mobilearts.com
Raimo	Niskanen	Ericsson AB OTP	Stockholm	Sweden	raimo@erix.ericsson.se
Linus	Nordberg	Net Insight	Stockholm	Sweden	linus@nordberg.se
Patrik	Nyblom	Ericsson AB OTP	Stockholm	Sweden	pan@erix.ericsson.se
Jan-Henry	Nyström	Erlang Training & Consulting	Uppsala	Sweden	jan@erlang-consulting.com
Kim	Olsson	Ericsson AB	Stockholm	Sweden	kim.xx.olsson@ericsson.com
Nicolae	Paladi	IT University of Göteborg	Göteborg	Sweden	n.paladi@gmail.com
Anders	Ramsell	TeliaSonera Sverige AB	Uppsala	Sweden	anders.ramsell@teliasonera.com
Mickaël	Rémond	Process-One	Paris	France	mickael.remond@process-one.net
Tony	Rogvall	Rogvall Invest AB	Gustavsberg	Sweden	tony@rogvall.se
Christophe	Romain	Process-One	Paris	France	christophe.romain@process-one.net
Mikael	Roseen	Kreditor	Stockholm	Sweden	Mikael.Roseen@Kreditor.se
Dan	Sahlin	Erlang Training & Consulting	Stockholm	Sweden	dan@erlang-consulting.com
Jan-Erik	Sankala	Streamnow AB	Luleå	Sweden	js@scg.nu
Andreas	Schumacher	Ericsson AB	Stockholm	Sweden	andreas.schumacher@ericsson.com
Rahul	Singh	Ericsson AB	Stockholm	Sweden	rahul.singh@ericsson.com
Håkan	Stenholm	Kreditor	Stockholm	Sweden	hokan@kreditor.se
Erik	Stenman	Kreditor	Stockholm	Sweden	Erik.Stenman@Kreditor.se
Ben	Stovold	Blue Tomato Ltd.	London	England	bstovold@gmail.com
Sebastian	Strollo	Tail-f Systems AB	Stockholm	Sweden	seb@tail-f.com
Göran	Stupalo	Ericsson AB OTP	Stockholm	Sweden	
Hans	Svensson	IT University of Göteborg	Göteborg	Sweden	hanssv@gmail.com
Gunnar	Sverredal	TeliaSonera Sverige AB	Uppsala	Sweden	gunnar.sverredal@teliasonera.com
Marcus	Taylor	Erlang Training & Consulting	London	England	marcus@erlang-consulting.com
Fredrik	Thulin	University of Stockholm	Stockholm	Sweden	ft@it.su.se
Robin	Thunell	Ericsson AB	Stockholm	Sweden	robin.xx.thunell@ericsson.com
Melinda	Tóth	Eötvös Loránd University	Budapest	Hungary	toth_m@inf.elte.hu
Zoltán Peter	Tóth	Ericsson AB	Budapest	Hungary	zoltan.peter.toth@ericsson.com
Michael	Truog	Nokia Siemens Networks	San Francisco	USA	michael.truog@nokia.com

Torbjörn	Törnkvist	Kreditor	Stockholm	Sweden	Torbjorn.Tornkvist@Kreditor.se
Marc	van Woerkom	Mayflower GmbH	München	Germany	mvanwoerkom@acm.org
Hasan	Veldstra	Hypernumbers	Edinburgh	Scotland	hasan@hypernumbers.com
Mats	Westin	Teligent	Stockholm	Sweden	mats.westin@gmail.com
Ulf	Wiger	Ericsson AB	Stockholm	Sweden	ulf@wiger.net
Claes	Wikström	Tail-f Systems AB	Stockholm	Sweden	klacke@hyber.org
Stefan	Willehadson	Synapse Mobile Networks	Stockholm	Sweden	stefanw@synap.se
Chris	Williams	Ericsson AB	Stockholm	Sweden	sailingareus@gmail.com
Dominic	Williams	Esmertec	Paris	France	dwilliams@esmertec.com
Patrik	Winroth	Synapse Mobile Networks	Stockholm	Sweden	patrik.winroth@synap.se
Hao	Zhang	Mobile Arts AB	Stockholm	Sweden	hao.zhang@mobilearts.com
Jimmy	Zhao	Ericsson AB	Uppsala	Sweden	Ming@zhao.nu
Jonas	Åman	Ericsson AB	Linköping	Sweden	jonas.aman@ericsson.com
Lennart	Öhman	Sjöland & Thyselius Telecom	Stockholm	Sweden	Lennart.Ohman@st.se
Lennart	Östman	Synapse Mobile Networks	Stockholm	Sweden	lennart.ostman@synap.se

2008-11-04


Requests per month to www.erlang.org
2 000 000
1 800 000
1 600 000
1 400 000
1 200 000
1 000 000
800 000
600 000
400 000
200 000
0
dec-98
jun-99
dec-99
jun-00
dec-00
jun-01
dec-01
jun-02
dec-02
jun-03
dec-03
jun-04
dec-04
jun-05
dec-05
jun-06
dec-06
jun-07
dec-07
jun-08