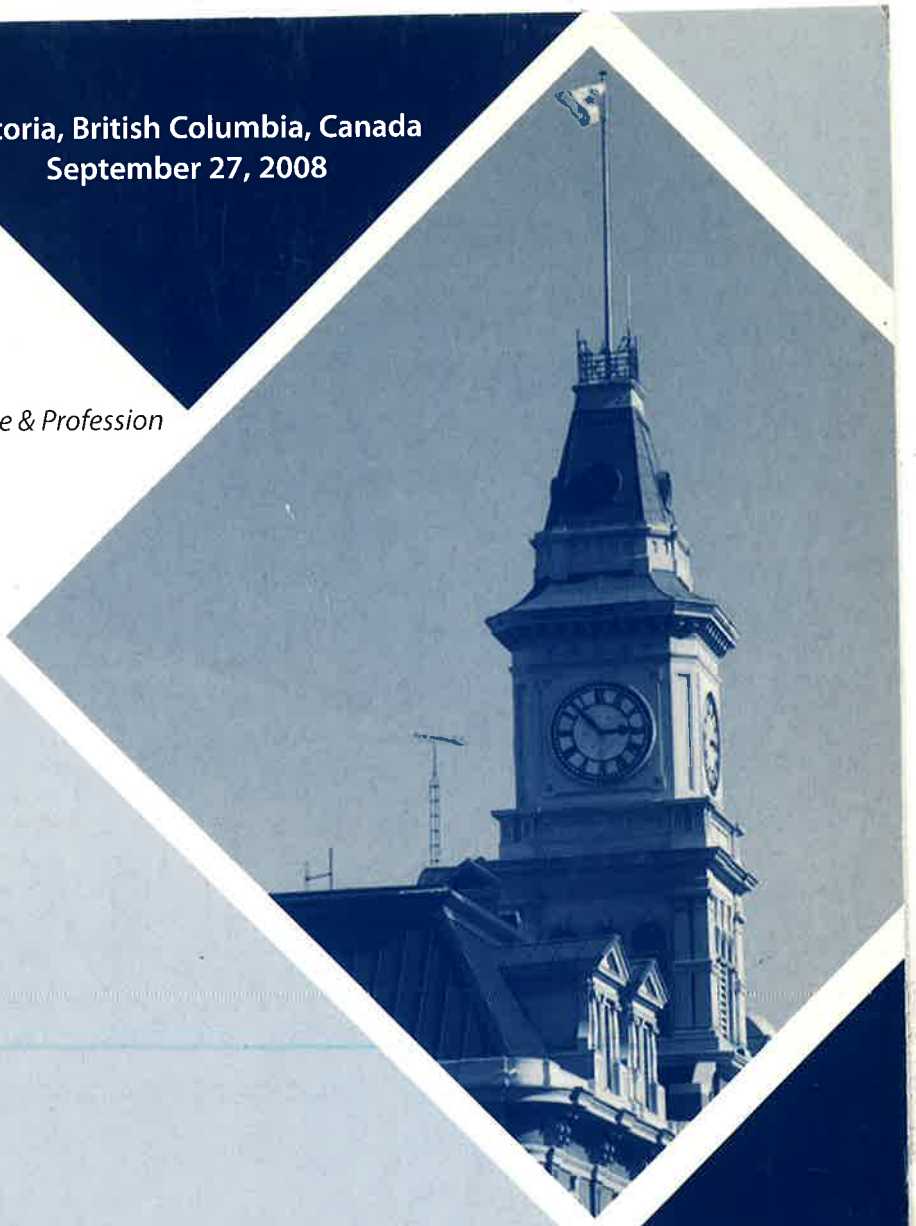
Victoria, British Columbia, Canada
September 27, 2008



Association for
Computing Machinery

Advancing Computing as a Science & Profession

Erlang'08

Proceedings of the 2008
SIGPLAN Erlang Workshop

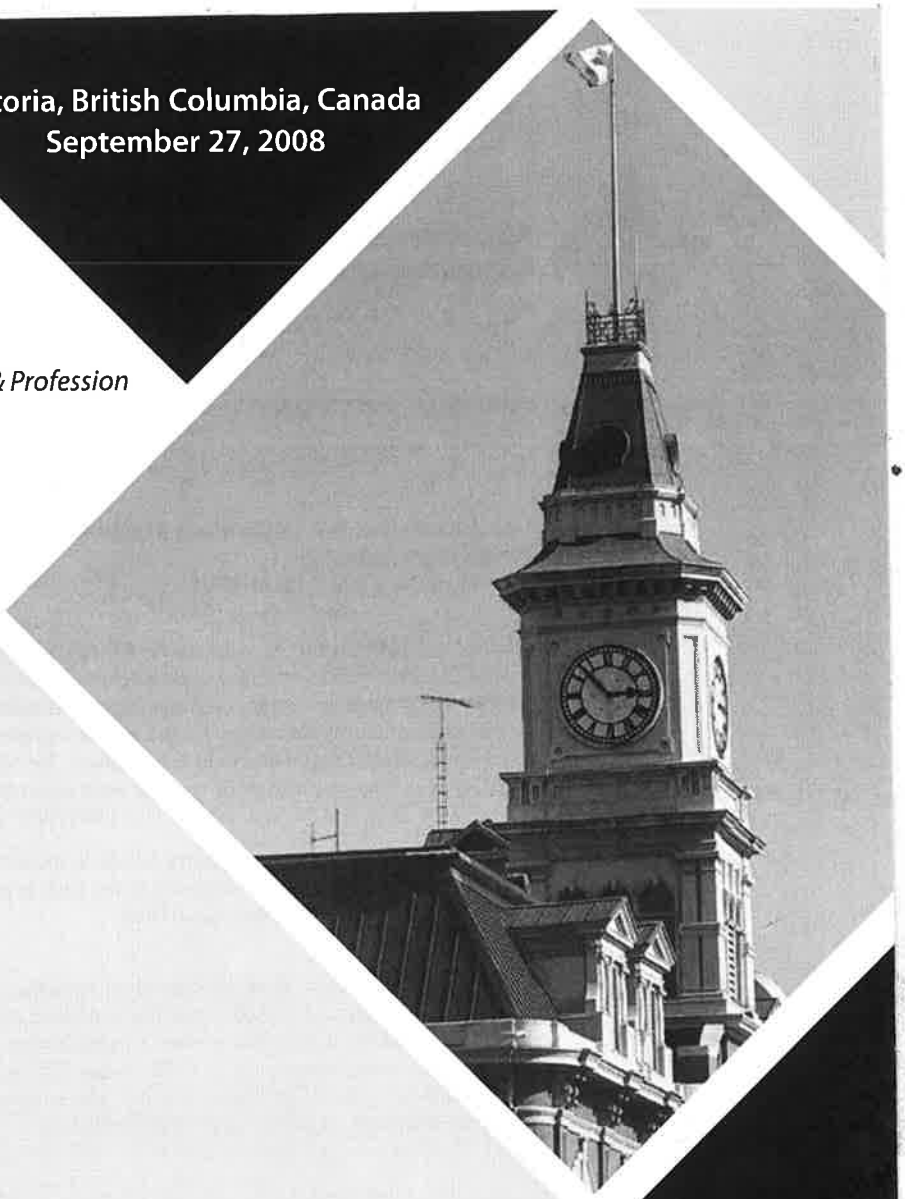
Sponsored by:

ACM SIGPLAN

Co-located with:

ICFP'08

Victoria, British Columbia, Canada
September 27, 2008



Association for
Computing Machinery

Advancing Computing as a Science & Profession

Erlang'08

Proceedings of the 2008
SIGPLAN Erlang Workshop

Sponsored by:

ACM SIGPLAN

Co-located with:

ICFP'08

Association for
Computing Machinery

Advancing Computing as a Science & Profession

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701

ISBN: 978-1-60558-065-4

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 11405
New York, NY 10286-1405

Phone: 1-800-342-6626
(US and Canada)
+1-212-626-0500
(all other countries)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org

ACM Order Number 555087

Printed in the USA

Foreword

It is our great pleasure to welcome you to the 7th *ACM SIGPLAN Erlang Workshop, Erlang'08*. This years workshop continues the tradition of being co-located with the annual International Conference on Functional Programming (ICFP), and being a forum for the presentation of research theory, implementation and applications of the Erlang programming language.

The program committee accepted 10 papers that cover a variety of topics, including language aspects, typing, refactoring, testing, high-performance computing and applications. The program committee also invited a keynote presentation on the future of Erlang.

We are very grateful to the program committee members, the reviewers, the authors and to the invited speaker, for the time and effort they devoted to provide such a high quality program. The papers were each carefully checked by two reviewers selected from among the most qualified available and then revised once more by the authors.

We would also like to thank Michael Sperber, this year's ICFP Workshop Chair for his support, and the ICFP local organizers for their hard work on local arrangements. Special thanks go to Bjarne Däcker for maintaining the Erlang'08 website, advertising the workshop and sharing experiences of previous Workshops. Finally we would like to thank ACM SIGPLAN for their continued support.

The workshop continues the tradition to include into the program a five minutes talks session to provide opportunities for all participants to introduce themselves and their Erlang interests.

We hope you find this program interesting and that the workshop will provide you with a valuable opportunity to share ideas with other researchers and Erlang practitioners from industry and academic institutions.

Tee Teoh
Erlang'08 General Chair
Canadian Bank Note, Ottawa, Canada

Zoltán Horváth
Erlang'08 Program Chair
Eötvös Loránd University, Budapest, Hungary

Table of Contents

Erlang 2008 Workshop Organization vi

Session 1: Testing

- **Testing Erlang Data Types with Quviq QuickCheck** 1
  Thomas Arts *(IT University of Gothenburg and Quviq AB)*,
  Laura M. Castro *(MADS Group - University of A Coruña)*,
  John Hughes *(Chalmers Sweden / Quviq AB)*
- **Early Fault Detection with Model-Based Testing** 9
  Jonas Boberg *(Erlang Training and Consulting Ltd.)*
- **Erlang Testing and Tools Survey** 21
  Tamás Nagy, Anikó Nagyné Víg *(Erlang Training and Consulting Ltd.)*

Session 2: Applications

- **A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol** 29
  Francesco Cesarini *(Erlang Training and Consulting)*,
  Viviana Pappalardo, Corrado Santoro *(University of Catania)*
- ***Scalaris:* Reliable Transactional P2P Key/Value Store** 41
  Thorsten Schütt, Florian Schintke, Alexander Reinefeld *(Zuse Institute Berlin and onScale solutions)*
- **High-Performance Technical Computing with Erlang** 49
  Alceste Scalas, Giovanni Casu, Piero Pili
  (Center for Advanced Studies, Research and Development in Sardinia)

Session 3: Typing and Refactoring

- **Refactoring with Wrangler, updated Data and process refactorings, and integration with Eclipse** 61
  Huiqing Li, Simon Thompson *(University of Kent)*,
  György Orosz, Melinda Tóth *(Eötvös Loránd University)*
- **Gradual Typing of Erlang Programs: A Wrangler Experience** 73
  Konstantinos Sagonas *(National Technical University of Athens)*, Daniel Luna *(Uppsala University)*
- **Refactoring Module Structure** 83
  László Lövei, Csaba Hoch, Hanna Köllő, Tamás Nagy, Anikó Nagyné Víg,
  Dániel Horpácsi, Róbert Kitlei, Roland Király *(Eötvös Loránd University)*

Author Index 90

Erlang 2008 Workshop Organization

General Chair: Tee Teoh *(Canadian Bank Note, Ottawa, Canada)*

Program Chair: Zoltán Horváth *(Eötvös Loránd University, Budapest, Hungary)*

Steering Committee Chair: Bjarne Däcker (*Erlang Training and Consulting, UK*)

Program Committee: Thomas Arts *(IT University, Göteborg, Sweden)*
Francesco Cesarini (*Erlang Training and Consulting, London, UK)*
Clara Benac Earle *(University Carlos III, Madrid, Spain)*
John Hughes *(Chalmers University of Technology, Göteborg, Sweden)*
Erik Stenman *(Kreditor, Stockholm, Sweden)*
Zoltán Theisz *(Ericsson, Ireland)*
Simon Thompson *(University of Kent, Canterbury, UK)*
Rex Page *(University of Oklahoma, USA)*

Additional reviewers: László Lövei *(Eötvös Loránd University, Budapest, Hungary)*
Csaba Hoch *(Eötvös Loránd University, Budapest, Hungary)*

Sponsor: 

Testing Erlang Data Types with Quviq QuickCheck

Thomas Arts
IT University of Gothenburg (Sweden) / Quviq AB
thomas.arts@ituniv.se

Laura M. Castro
MADS Group - University of A Coruña (Spain)
lcastro@udc.es

John Hughes
Chalmers (Sweden) / Quviq AB
john.hughes@chalmers.se

Abstract

When creating software, data types are the basic bricks. Most of the time a programmer will use data types defined in library modules, therefore being tested by many users over many years. But sometimes, the appropriate data type is unavailable in the libraries and has to be constructed from scratch. In this way, new basic bricks are created, and potentially used in many products in the future. It pays off to test such data types thoroughly.

This paper presents a structured methodology to follow when testing data types using Quviq QuickCheck, a tool for random testing against specifications. The validation process will be explained carefully, from the convenience of defining a model for the datatype to be tested, to a strategy for better shrinking of failing test cases, and including the benefits of working with symbolic representations.

The leading example in this paper is a data type implemented for a risk management information system, a commercial product developed in Erlang, that has been used on a daily basis for several years.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms verification

Keywords erlang, quickcheck, datatypes

1. Introduction

Software testing is an as necessary as delicate matter. In particular, designing good test sets is a non-trivial task. Unconscious assumptions about the functionality of the subject under test may leave important scenarios or possibilities out of the testing range/scope. That is why automatic test generation tools that provide random input can be helpful products.

As a successful example, Quviq QuickCheck has proven itself as useful tool for testing Erlang programs (Thomas Arts et al. 2006; Quviq). The first QuickCheck tool was invented by Claessen and Hughes (Koen Claessen and John Hughes 2000). The version of Quviq is implemented in Erlang and adapted to better fit industrial needs.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

In this paper we present a method for validating user-defined Erlang data types using Quviq QuickCheck[1]. As a case study, we use the risk management information system ARMISTICE (ARMISTICE). We selected one data type from this risk management information system as our leading example for this paper: the *decimal* data type, a data type for fractional numbers without writing a numerator and denominator. We found a surprising error in the implementation of this data type, of which symptoms had occurred before, but whose scarce bug reports were not well understood. Errors in a data type may occur in a very unexpected way at the user level. The data type was fixed and thoroughly tested with the described method, increasing the confidence that the present implementation is correct.

2. Motivation

ARMISTICE (Víctor M. Gulías et al. 2006, 2005) is an information system whose business logic has been developed using Erlang/OTP. Using a functional language such as Erlang was a key factor for success not only in implementing a software application to deal with such a complex business domain as insurance management (David Cabrero et al. 2003), but also in reaching an abstraction level at the definition of the system which makes it applicable to different business fields. This innovative risk management system (RMIS) is meant to be a tool for both the expert and the daily non-expert users. An advanced profile will use ARMISTICE to specify a set of resources and their relevant properties of interest, as well as the insurance policies contracted to protect those resources from the consequences of potentially harmful events, whichever these might be for each particular case. On the other hand, the system is of assistance also to the kind of user that, without any expert knowledge regarding coverages and warranties, has to deal with incident reports, accident claims, and file trackings. In this case, ARMISTICE helps by retrieving and isolating just the relevant information for each scenario, according to the provided contextual data, and thus, giving valuable support to actions and decisions.

This software system has been in production for a few years now, after being tested by regular users during the last stages of development. Such testing process is common in software development cycles, but it is hardly ever complete and exhaustive. The fact that an application has been running daily without major problems is just a weak empirical proof of correctness.

In order to provide a greater degree of confidence, and taking advantage of the application's core being implemented in Erlang, we decided to use QuickCheck to automatically generate random tests. As a good starting point, we chose ARMISTICE data types. Data types are the smallest logic element that can be tested in most software applications, and the components on which all other business objects are built upon.

[1] In this paper, QuickCheck refers to Quviq QuickCheck version 1.13.

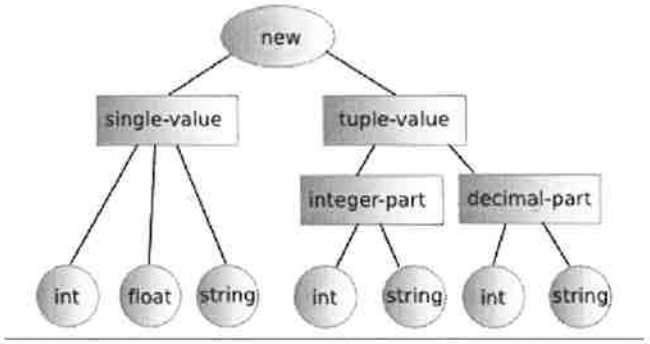


Figure 1. Decimal data type creation options

3. Testing Data Types

There are a number of data types implemented in ARMISTICE and some of them, such as *logico* for booleans and *entero* for integers, are very similar to the basic data types in Erlang. Other data types are built upon these basic types, for example a data type *currency* for representing amounts in different currencies.

In order to be able to have a uniform way of marshalling and unmarshalling values within the system, all ARMISTICE data types have the same structure: a value is represented by a record with the name of the value and wrapped in a tuple with *ok* as the first parameter. In case a data type operation results in an error, the value of the data type is represented by a tuple with first argument *error* and second argument an atom describing the cause of the error. Thus, instead of representing booleans by atoms *true* and *false*, they are represented by $\{ok,\#logico\{value = true\}\}$ and similar for *false*. A division by zero error with two *entero* values will not result in a crash, but in a return value $\{error,division\_by\_zero\}$. This way, even failing operations on the server side are detectable at the client side.

Communication between the ARMISTICE client, written in Java, and the ARMISTICE server, written in Erlang, uses an XML-RPC based protocol. The server first takes care of unmarshalling the messages to obtain Erlang terms, then invoking the corresponding business core service, and finally marshalling the results before replying to the client. For that reason, all data types have constructors to create a value from a string and similarly they all implement a function *to_string* to convert a value to a string. Furthermore, such operations are the basis of all communications with the client, so they are performed very frequently. This means that such conversions need to be fast.

All data types have been tested with the method[2] presented, motivated, and evaluated in this paper.

3.1 Decimal Data Type

We present the method for testing data types with QuickCheck by a leading example in the form of ARMISTICE's decimal data type. A *decimal* is used to represent values with some digits before and some after the decimal separator. It need not have the same range as floats, since it is used to represent sums of money. This data type is defined in a module called `decimal.erl` which exports a creator named `new` in four flavours. As displayed in Fig. 1, input to the function is either a single value or a two-element tuple (first component for the integer part, second for the decimal part). Values are either an integer, a float, or a string representation of one of the two. When providing a single string value, it can contain commas (thousands separator) and/or dot (decimal separator). The decimal separator cannot be used if the two-element tuple notation is used.

[2] The method has been developed by the company and has been fine tuned during this case study.

Other data type constructors provided include mathematical operators: sum, substraction, product, division, negation, absolute value, maximum and minimum; and relational operators, such as 'greater than', and 'less than or equal'.

Now, for testing the decimal data type, one needs a generator for this data type, creating random instances of it, and a property that represents what one likes the data type to fulfil. A relatively naïve approach to generate a *decimal* would be to define the following QuickCheck generator in which the function `new`[3] is applied to an arbitrary integer and an arbitrary positive integer.

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

Note that a decimal can be constructed in many ways, as explained by the different inputs `new` can take, but the temptation followed here is to keep the code for the generator small, since all possible decimals seem to be producible in this way. We will show later that giving in to this temptation results in a missed opportunity to catch an error.

One of the properties that one may like to check is that the sum operator is actually commutative,

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

QuickCheck is used to check this property with successful result, meaning that the specified property holds for tens of thousands of randomly generated test cases. So, `sum` seems indeed commutative. We now are faced with the questions: *which other properties do we add?* and *when do we have sufficiently many properties to cover testing of this data type?*

3.2 Model for data type

We know from the field of mathematics and formal methods (Floyd 1967; Hoare 1972) that creating a model of our data type could help in deciding whether we have created enough properties for the data type. We would like to show that each operation on *decimals* can be simulated by our model operations. Thus, we want an injection $[\![\circ]\!]$ from the decimal data type into our model, such that $\forall D_1, D_2 \in$ decimal

$$\begin{array}{rcl} [\![sum(D_1,D_2)]\!] & \equiv & [\![D_1]\!] + [\![D_1]\!] \\ [\![subs(D_1,D_2)]\!] & \equiv & [\![D_1]\!] - [\![D_1]\!] \\ [\![mult(D_1,D_2)]\!] & \equiv & [\![D_1]\!] * [\![D_1]\!] \\ [\![divs(D_1,D_2)]\!] & \equiv & [\![D_1]\!] \,/\, [\![D_1]\!] \\ [\![lt(D_1,D_2)]\!]_l & \equiv & [\![D_1]\!] < [\![D_1]\!] \\ & \vdots & \end{array} \tag{1}$$

In general, we may need to implement a model with all these operations. In this case, though, as our model we can use the standard Erlang implementation of floating point numbers (in itself built upon the C implementation that implements the IEEE 754-1985 standard (IEEE 1985)). We are lucky here, since this choice perfectly fits our *decimals* regardless some rounding issues we will discuss later on (see page 3), but in many situations one can implement a model that is simpler than the data type itself, for example by not caring about efficiency and leaving out optimisations.

We choose a simple injection, viz, mapping *decimals* to Erlang floating point numbers. In fact, this injection function was already present in the code under test:

[3] To enhance reading we simplified the operations in this paper. For instance, the function `new` is a local function that calls `decimal:new` and removes the `ok` tag from its result, and similarly for `sum`, `mult`, etc.

```
model(Decimal) ->
   decimal:get_value(Decimal).
```

Note that in the last equation shown above we use a different model, viz $[\![\circ]\!]_l$ for interpreting the result of the function `lt(D1,D2)`, since that result is a *logico*, not a *decimal*. The model for *logico* simply maps to Erlang booleans and the injection that we use is also already present in the corresponding module.

```
logico_model(Logico) ->
   logico:get_value(Logico).
```

QuickCheck properties to check whether our implementation is equivalent to the Erlang floating point implementation look like:

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

prop_lt() ->
   ?FORALL({D1, D2}, {decimal(), decimal()},
           logico_model(lt(D1, D2))
           == (model(D1) < model(D2))).
```

If we now create one such property for each operation defined in the data type, then by checking each of them for a large number of random inputs, we would gain confidence that we tested the data type operations sufficiently.

We start by checking the first property with QuickCheck, which immediately results in a failure.

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
....Failed! After 5 tests.
{[{decimal,1000000000000000}],
 [{decimal,11000000000000000}]}
false
```

After five tests, QuickCheck finds a counterexample against the equivalence between adding two `decimal` values and adding the corresponding two floats. However, the counterexample values reported back by QuickCheck are in their internal representation, which is hard to understand by anyone who is not familiar with the decimal data type implementation. This would be even worse for more complex data types. For a trained QuickCheck user, the values are at least surprising, since one expects rather small integer values and not values with 15 or more zeros. The fact that we actually failed in this test with values 1 and 1.1 is only directly obvious to the developer of the decimal data type.

We do not want to rely on the internal representation of a data type, for one reason because it may be hard for others than the developer of the module to understand, for another reason, because implementations of data types may change and we want tests to depend on them as little as we want implementations to depend on them. Moreover, the internal representation is only the final result of a computation constructing the data structure. We like to know which steps were performed in this construction, since they may reveal information about an observed failure. Therefore, we work with *symbolic* values instead of *real* values. This means that QuickCheck generates a symbolic representation of a *decimal*, which will be evaluated when needed (i.e., during testing). So, our generator is rewritten to:

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

Thus generating a symbolic call to `decimal:new(Tuple)`, i.e., creating a tuple with tag `call`, module, function name, and arguments, instead of actually performing the call.

Of course, we change properties accordingly and introduce in their definition the evaluation of such symbolic values, a standard QuickCheck function.

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

With these modifications, a similar failure is reported back by QuickCheck as:

```
> 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
```

Now it is much easier to see that the problem can be reproduced by using values 0.1 and 0.2.

Conform to the IEEE 754-1985 standard, Erlang float values present an unavoidable rounding error , demonstrated by typing the values in the shell:

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

In other words, since floats are represented as a list of bits, there is not always an exact representation of each *decimal*. Since our computations are performed on the *decimals* and the conversion to floats is only necessary to compare the results, we are satisfied with an approximate equality. Therefore, we define the equivalence relation == with respect to two maximum tolerance levels: an absolute error value (`ABS_ERROR`), which measures how different two floats are; and a relative error value (`REL_ERROR`), which takes into account not only the values themselves, but also their magnitudes (Dawson 2008). Note that we divide by the maximum of the absolute values of the two floats, ensuring that the maximum never is zero (unless they both are zero, in which case the absolute error value is used).

```
-define(ABS_ERROR, 1.0e-16).
-define(REL_ERROR, 1.0e-10).

equiv(F1,F2) ->
   if (abs(F1-F2) < ?ABS_ERROR) -> true;
      (abs(F1) > abs(F2)) ->
         abs( (F1-F2)/F1 ) < ?REL_ERROR;
      (abs(F1) < abs(F2)) ->
         abs( (F1-F2)/F2 ) < ?REL_ERROR
   end.
```

To set the reference error values, we use the knowledge that the decimal data type in ARMISTICE has 16 digits precision (hence, the value of `ABS_ERROR`) and agree to a 99.9999999999% accuracy (hence, the value of `REL_ERROR`). Just a minor change in the

QuickCheck specification is necessary to introduce the new equivalence function.

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

Finally, the property passes thousands of randomly generated test cases.

3.3 Generator to cover data structure

Now we need to introduce one such property for each operation on our abstract data type. Even though it might then seem that we completely tested the decimal data type, we recognise that this is not true. In the first place, and a simple code coverage can demonstrate this, we applied the function `new` to only one of its many types of input (cf. Fig. 1). In the second place, we may have missed to test part of the data structure. Operations on the data structure may actually invalidate an invariant. For example, imagine a data type *set* in which elements are stored in a sorted list, but a set obtained from the union of two sets may invalidate that invariant. Hence, not testing to delete an element from a set obtained form the union of two other sets is a missed opportunity to find an error. Note that code coverage will most likely **not** reveal that we missed testing to delete an element from a set created by the union of two sets, because we have one property testing all code involved in a deletion and one property testing all code involved in computing the union. Together, they cover all code involved in both, which is not the same as covering combinations of creating unions and deleting elements from the union.

Similarly, in our case study, we want to test, for example, multiplication of two *decimals* where each *decimal* itself is obtained by operations that may invalidate an invariant, e.g.

```
   mult(new("12,837.12"),
        sum(new(12), new({13,4}))).
```

To do so, we create a **recursive generator** to obtain arbitrary nesting of *decimals* as arguments of constructors for *decimals*. The depth of the recursion is determined by QuickCheck such that small values are tried first, slowly growing as long as no errors are detected. QuickCheck gives access to the parameter that controls the structural size of the generated test case via the `?SIZED` macro.

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

decimal(0) ->
   {call, decimal, new,
    [oneof([int(),
            real(),
            separator(decimal_string(),digits()),
            {oneof([int(), decimal_string()]),
             oneof([nat(), digits()])}
           ])
    ]};
decimal(Size) ->
   Smaller = decimal(Size div 2),
   oneof([
      decimal(0),
      {call, decimal, sum, [Smaller, Smaller]},
      {call, decimal, mult, [Smaller, Smaller]}
   ]).
```

Note that so far we only consider the `sum` and `mult` operators to create data structures. We will soon see how to add the other operators. Note also that we use `Size div 2` for smaller decimals. This is based on the fact that the smaller decimals have (at most) two subtrees and we want to keep the number of nodes in the tree roughly determined by the size parameter.

Additional code is necessary to specify how the strings in the input may look like. The generator for an arbitrary number of digits is again recursively defined.

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

digits(0) ->
   [digit()];
digits(Size) ->
   Smaller = digits(Size-1),
   oneof([digits(0), [digit()|Smaller]]).

digit() ->
   choose($0, $9).
```

We can use these digits to generate string representations of decimals, either as a long list of digits, or as such a list grouped by 3 digits at a time and commas inbetween.

```
decimal_string() ->
   signed(oneof([digits(), groups(3)])).

signed(G) ->
   ?LET(S, G, oneof([S, "+"++S, "-"++S]))

separator(G1, G2) ->
   oneof([G1,
          ?LET({S1, S2}, {G1, G2}, S1++"."++S2)]).

%% groups of N digits
groups(N) ->
   ?SIZED(Size, groups(N, Size div N)).

groups(G, 0) ->
   digits(N-1);
groups(G, Size) ->
   Smaller = groups(N, Size-1),
   oneof([
      groups(N, 0),
      ?LET([S1, S2],
           [Smaller, vector(N, digit())],
           S1++","++S2)
   ]).
```

With this recursive generator, symbolic calls can be generated that cover, in principle, the whole data structure. For example, in a QuickCheck sample, the following value was generated:

```
{call,decimal,sum,
  [{call,decimal,sum,
     [{call,decimal,mult,
        [{call,decimal,new,[{11,"4003351"}]},
         {call,decimal,new,["-930764"]}]},
      {call,decimal,new,[-2.35986]}]},
   {call,decimal,new,[1.64783]}]}
```

Of course, we need to add the other operators as well, but first we use QuickCheck to check our previously defined property for the sum operator, which successfully passes thousands of tests.

Now we add a similar property for multiplication, which fails after only a few generated tests.

```
6> eqc:quickcheck(decimal_eqc:prop_mult()).
...............Failed! After 16 tests.
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["+1"]},
       {call,decimal_eqc,new,[2.36314e+4]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[-5]},
       {call,decimal_eqc,new,[-9.61993e+5]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["74.4"]},
       {call,decimal_eqc,new,[{"-6,179","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["47"]},
       {call,decimal_eqc,new,["-467,725.079"]}]}]}}
Shrinking.............................(31 times)
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["+0"]},
       {call,decimal_eqc,new,[0.00000e+0]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[1]},
       {call,decimal_eqc,new,[10.1400]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["00.4"]},
       {call,decimal_eqc,new,[{"-0,000","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["40"]},
       {call,decimal_eqc,new,["-000,000.078"]}]}]}}
false
```

As we can see from this example, the failing test case contains a fairly large expression. The shrinking procedure reduces the test case quite a lot, but there are still a number of terms in there that a human tester would reduce further. For example the sign could be removed from `{call,decimal,new,["+0"]}`, but even better one could try to remove the whole term. Another reduction could be used for the string `"-000,000.078"`, where six zeros could be reduced to one, at least if the value is important and not the actual structure of the string.

3.4 Improved shrinking of recursive data types

The reason why the previous terms are not shrunk up to the extent we would like they were, lays in the definition of our generators. When we nest `LET` macros, QuickCheck first shrinks on the outermost level and when that is no longer possible, the generator defined inside the `LET` is shrunk[4].

QuickCheck offers a macro `SHRINK` with which one can define one's own shrinking rules. The macro has two arguments, the first being a generator, the second a list of generators. The generators in that list are added as shrinking alternatives to the generator in the first argument. These shrinking alternatives are applied before the built-in shrinking. For example, `SHRINK` could be used to ensure that the `signed` generator always tries to shrink to a representation without a plus or minus symbol.

```
signed(G) ->
   ?LET(S, G,
      ?SHRINK(oneof([S, "+"++S, "-"++S]), [S])).
```

Thus, whenever a test case fails, QuickCheck will first re-test by removing the sign. The original `signed` generator would not do so, since it would only shrink within the alternative chosen by the `oneof` generator.

Note that the order in which we mix `LET` and `SHRINK` above is important. Should we have written

```
signed(G) ->
   ?SHRINK(
      ?LET(S, G,
         oneof([S, "+"++S, "-"++S])), [G])).
```

then shrinking would choose a sequence of digits without a sign, but the sequence would have nothing in common with the original sequence. That is not what we want in this case.

In fact, we often want to know the generated value in order to decide which shrinking steps to add, thus, often having first a `LET` and then a `SHRINK` macro. Therefore, Quviq has added a `LETSHRINK` macro to QuickCheck which combines the two earlier mentioned macros. As arguments we provide a list of bindings and a list of generators, resulting in adding those bindings as shrinking alternatives. In that way, the above `signed` generator can be simplified to:

```
signed(G) ->
   ?LETSHRINK([S], [G],
              oneof([S, "+"++S, "-"++S])).
```

where `S` is automatically added as a shrinking alternative.

The `?LETSHRINK` macro can also be used to reduce the number of groups present in the failing test case `"-467,725.079"`, for example, which shrinks to `"-000,000.078"` instead of also removing the group of three zeros. We define that each smaller generator in the recursively defined generator is automatically added to the shrinking alternatives[5].

```
groups(G, Size) ->
   Smaller = groups(N, Size-1),
   oneof([
      groups(N, 0),
      ?LETSHRINK([S1, S2],
                 [Smaller, vector(N, digit())],
                 S1++","++S2)
   ]).
```

With this shrinking rule added the string `"-000,000.078"` will shrink to `"-000.078"`. Of course, we would still like to shrink that further, which means that we need to look at the generator for digits.

```
digits(Size) ->
   Smaller = digits(Size-1),
   oneof([digits(0),
          ?LETSHRINK([Digits], [Smaller],
                     [digit()|Digits])
   ]).
```

Which will now enable to shrink to `"-0.078"` as we can see from re-checking the property on the same example with all above shrinking alternatives added.

[4] The `choose` generator built-in shrinking strategy always defaults to the first element in the argument list, while shrinking of `int`, `real` or `nat` tends towards zero.

[5] Strictly speaking, a group should start with a number of digits less than or equal to three. However, we add the possibility to have a group with exactly three digits. We add this as our second shrinking alternative, whereas the first is a flexible number of digits.

```
Shrinking.....................(35 times)
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["0"]},
       {call,decimal_eqc,new,[0.00000e+0]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[1]},
       {call,decimal_eqc,new,[10.1400]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["0.4"]},
       {call,decimal_eqc,new,[{"-0","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["40"]},
       {call,decimal_eqc,new,["-0.078"]}]}]}}
false
```

With these rules for shrinking added, we increase the simplicity of our failing test case. However, the term structure is still unchanged, although we know that adding zero to a number would result in that same number. We can shrink the structure by once more using the `LETSHRINK` macro, now in the definition of the recursive generator for decimals.

```
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]})
   ]).
```

In addition to simplifying the actual term, we also want to be able to see the difference between the value created by the implementation and the value computed in our model. We can do so by adding a `WHENFAIL` macro to the property, which only evaluates its first argument if the second argument is false.

```
prop_mult() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
      begin
         D1 = eval(SD1),
         D2 = eval(SD2),
         Model = model(D1) * model(D2),
         Real = model(mult(D1, D2)),
         ?WHENFAIL(
            io:format("Real ~p\nModel ~p\n",
                      [Real, Model]),
            equiv(Real, Model))
      end).
```

Checking this re-defined property on the same example, shows a difference in outcome by a factor 10. Running QuickCheck a few additional times always shows the same factor 10 difference.

```
Shrinking.....................(51 times)
{{call,decimal_eqc,new,[10.1400]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,new,["0.4"]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["47"]},
       {call,decimal_eqc,new,["-0.078"]}]}]}}
Real -331.172
Model -33.1172
false
```

The difference was rather quickly identified as an error in the `decimal` module implementation: the carrier was incorrectly propagated. The problem arose when values were rounded to ignore the least significant digits, which are not to be stored, as previously said. In such cases, a rounding operation was considered for the last decimal digit to be stored, but no carrier was taken into account to be propagated to the left. Instead of that, if the last decimal was to be modified (rounded) and this digit turned out to be a 9, then the 9 was erroneously replaced by a 10. For instance, when rounding a large number like 123.456**789**987654 on six significant digits, we should obtain 123.456**790**. But, instead, the erroneous code was replacing it by 123.4567**10**. Since the internal representation of `decimals` is a sequence of digits with a fixed number of decimals (six in this example), this longer sequence is then interpreted as: 1234.567810.

Strangely enough, this rounding error had been in the code for several years without being found a problem. However, after detecting it, the developer could actually relate it to some unsolved, obscure error reports from the customer.

After correcting the code, the properties for addition and multiplication operators passed thousands of generated test cases.

3.5 Well defined generators

We now add the additional operations substraction and division to the generator and create properties for them similar to those for addition and multiplication. This takes hardly any effort after having the framework in place.

```
decimal(Size) ->
   Smaller = decimal(Size-1),
   oneof([
      decimal(0),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, sum, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, mult, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, subs, [SD1, SD2]}),
      ?LETSHRINK(
         [SD1, SD2], [Smaller, Smaller],
         {call, decimal, divs, [SD1, SD2]})
   ]).
```

We check the properties with QuickCheck and now the property for the addition fails again! It does because it crashes in the evaluation of a generated value. In other words, we generated a symbolic value that does not correspond to a real value.

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
............Failed!
After 13 tests.
Shrinking....(4 times)
Reason:
{'EXIT',{{not_ok,{error,decimal_error}},
         [{common_lib,ok,1},
          {decimal_eqc,'-prop_subs/0-fun-0-',1},
          {eqc,'-forall/2-fun-4-',2},
  ...]}}
{{call,decimal,divs,
   [{call,decimal,new,[{0,[]}]},
    {call,decimal,new,["0"]}]},
```

```
 {call,decimal,new,[0]}}
false
```

Indeed, shrinking helps us in determining the problem: we divide by zero when we create our decimal. Evaluating this symbolic value returns a different wrapper than `ok`, viz, `not_ok` and hence the function taking the wrapper away creates a *badmatch* exception.

Actually, we do want to test that division by zero gives us an expected error value. That is, precisely, done in the property for division, since there we test for each *decimal* value that a division in the model results in the same value as a division of *decimal* values. Division by zero in the model should equally well generate an exception.

```
prop_divs() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
      begin
         D1 = eval(SD1),
         D2 = eval(SD2),
         case model(D2) == 0.0 of
            true ->
               {'EXIT', _} =
                  (catch model(D1)/model(D2)),
               is_error(divs(D1, D2));
            false ->
               equiv(model(divs(D1, D2)),
                     model(D1) / model(D2))
         end
      end).
```

With such a property, we already check that division by zero is an exception case. What we want is to avoid generating these exception cases, instead we only want to generate well defined symbolic values, meaning that such symbolic value does not raise an exception when evaluated. We use a simple, generally applicable concept for doing so. We define a generator `defined` which evaluates the symbolic value and catches a potential exception; the symbolic value is only defined if no exception occurs. We count on the fact that the majority of the symbolic values will not raise an exception when we evaluate them and we introduce a generator `well_defined` to keep generating values until we find a defined one.

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

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

We use these generators in our QuickCheck specification for *decimals* by replacing the generator for decimal by

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

One may consider this to be cheating. After all, we could have an error in our code that makes an operation crash, even though we do not want that to happen. If we filter the generators and never produce such faulty value, how well do we test? Here it is important to remember that we check **each** operation in a property. Thus, if there is any value for which the addition fails, then that value is checked in the property for addition. The only case we never check is whether the `new` operation crashes on a specific input. Therefore, we add one additional property, checking that generating base values always succeeds. In addition we also check that applying the model function to an obtained *decimal* does not raise an exception.

```
prop_new() ->
   ?FORALL(SD,decimal(0),
           is_float(model(eval(SD)))).
```

Finally we succeed in passing hundred thousand tests for each property in our specification. Therewith we have thoroughly tested the *decimal* data type.

A last property remained untested, which is the actual motivation for introducing the *decimal* data type in the software. The data type is used to ease marshalling from and to strings to enable communication via a web-service-like interface. We want to verify that, for each and every `decimal` data structure we generate in any possible way, converting it to a string and then performing the reverse operation (call `decimal:new` with that same string) is idempotent.

```
prop_decimal_string() ->
   ?FORALL(SD, decimal(),
      begin
         D = eval(SD),
         decimal:new(decimal:to_string(D)) == D
      end).
```

With this last property also successfully checked, we conclude the testing of the *decimal* implementation.

4. Conclusions

The contribution of this paper is the introduction of a methodology to test data types, consisting in four steps:

1. Define a model for the data type: the goal is checking whether the data type implementation is equivalent to the model, thus holding equivalent properties.
2. Create as many model-equivalence checking properties as operations in the data type. Work with symbolic values instead of real values, to keep independent of internal representation of the data type.
3. Write data type generators. Make sure they cover all the data structure, i.e., define recursive generators that include all operations producing values of the data type as a result. Mind that generated values are well-defined, placing exception cases testing just at relevant properties.
4. If needed, define your own shrinking preferences to help reaching the least significant counterexample.

With this methodology, QuickCheck can be used in a more structured way, giving more confidence in the result of all QuickCheck tests. The methodology has been applied to Erlang data types, but the approach should as well be applicable to data types defined in another language, such as C and Java, provided we can interface Erlang with such language. We plan on investigating that as part of our continued research.

Even though one may expect data types to be rather simple pieces of software, we have shown that failures can be detected in software considered very stable and used for several years. We have applied the developed methodology to other data type implementations as well, for example, the *entero* and *logico* data types in the same risk management information system, and the *queue* data type in the Erlang standard library. This showed that the method is generally applicable and basically a recipe to follow.

Using either QuickCheck or another test case generation tool is not a trivial step, though. A too naïve approach can convice

ourselves we have tested enough when we are actually missing a lot or even not really proving anything relevant. In this paper we have explained, step by step, an exhaustive procedure that can be easily followed. By describing the process thoroughly we have tried to justify each stage of the process and illustrate possible problems (wrong approaches, obscurity of dealing with real values and their internals, failure to test all possibilities, unsatisfactory shrinking) and how to overcome them (model definition, symbolic values, recursive generators, self-defined shrinking rules).

As said before, we plan to extend this methodology to non-Erlang data types in the near future, taking advantage of already existing intercommunication alternatives between Erlang and other technologies such as C or Java. We are also working on a broader methodology that will consider how to check not just a relatively simple issue as data types, but a whole application business logic. We aim to do so focusing on layered client/server applications, and independently from the user interface and persistent storage.

Acknowledgments

We thank Víctor M. Gulías from the University of A Coruña for his support and for creating the possibility for this collaboration. This research was partly sponsored by two grants: FARMHANDS (MEyC TIN2005-08986 and XUGA PGIDIT06PXIC105164PN) and EU FP7 Collaborative project *ProTest*, grant number 215868.

References

ARMISTICE. Armistice. http://www.madsgroup.org/armistice/, 2002.

David Cabrero, Carlos Abalde, Carlos Varela, and Laura M. Castro. Armistice: An experience developing management software with erlang. In *Principles, Logics, and Implementations of High-Level Programming Languages*, Uppsala, Sweden, August 2003.

Bruce Dawson. Comparing floating point numbers. http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm, 2008.

R.W. Floyd. Assigning meaning to programs. In *Proc. of Symposia in Appl. Math.* American Mathematical Society, 1967.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

IEEE. Standard for binary floating-point arithmetic. http://grouper.ieee.org/groups/754/, 1985.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

Quviq. Quviq. http://www.quviq.com, 2008.

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*, New York, NY, USA, 2006. ACM Press.

Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. A new risk management approach deployed over a client/server distributed functional architecture. In *18th International Conference on Systems Engineering*, pages 370–375, University of Nevada, Las Vegas (USA), August 2005. IEEE Computer Society. http://www.icseng.info.

Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. Formalisation of a functional risk management system. In *8th International Conference on Enterprise Information Systems*, pages 516–519, Paphos (Cyprus), May 2006. INSTICC Press. http://www.iceis.org.

Early Fault Detection with Model-Based Testing

Jonas Boberg

Erlang Training and Consulting Ltd.
29 London Fruit & Wool Exchange
Brushfield Street
London, E1 6EU, UK
jonas@erlang-consulting.com

Abstract

Current and future trends for software include increasingly complex requirements on interaction between systems. As a result, the difficulty of system testing increases. Model-based testing is a test technique where test cases are generated from a model of the system. In this study we explore model-based testing on the system-level, starting from early development. We apply model-based testing to a subsystem of a message gateway product in order to improve early fault detection. The results are compared to another subsystem that is tested with hand-crafted test cases.

Based on our experiences, we present a set of challenges and recommendations for system-level, model-based testing. Our results indicate that model-based testing, starting from early development, significantly increases the number of faults detected during system testing.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Verification

Keywords Model-based testing, system testing

1. Introduction

The cost of finding and fixing faults in software typically rises as the development project progresses into a new phase. Faults that are found after the system has been delivered to the customer are many times more expensive to track down and correct than if found during an earlier phase [1]. Current and future trends for software include increasingly complex requirements on interaction between systems [2]. The increased complexity means that a system may have potentially infinite combinations of inputs and resulting outputs. It is difficult to get satisfactory coverage of such a system with hand-crafted manual or automatic test cases [3].

Model-based testing is a test technique where test cases are generated from a model of the system. There are model-based testing tools that can automate the generation of test cases from a behavioral model, including *test oracles* that can determine whether the system under test behaved correctly at the execution of the test case [4]. Test cases generated from a model have been shown to give a high coverage of system interaction points, given that the generation is carefully guided [5].

Intensive research on model-based testing has been conducted, and the feasibility of the approach has been demonstrated. Still, few conducted studies focus on early-fault detection and the application of the technique on specific test levels. Industrial adoption of model-based testing remains low [6]. Although this is partially due to technical limitations, process-related issues remain a large concern. The model-based testing practice must be integrated into current software processes [6]. Limited understanding of the benefits model-based testing delivers at different levels of testing, and the associated challenges of its application in real world projects, is therefore an obstacle to adoption of the technique.

Finding problems faced in industrial software development, and finding solutions that developers will embrace, is an often listed basis for successful technology transfer [7]. We have conducted a pre-study of an ongoing system development project at Erlang Training and Consulting (ETC). The project develops a message gateway product with two subsystems, an E-mail gateway and an Instant Messaging (IM) gateway. The gateway is implemented using the Erlang/OTP platform. Both subsystems essentially interconnect networks that use different communication protocols, by performing the required mapping of protocol messages. The E-mail gateway allows a client to access multiple types of e-mail servers using a single communication protocol. The IM gateway provides mobile devices an interface to multiple instant messaging protocols.

The pre-study indicates that faults which should have been found during system testing of the IM gateway subsystem, have repeatedly been left undetected until customer acceptance testing. Several negative consequences as a result of this fault-slip through have been observed:

- Reproducing and locating the source of the fault requires more effort as the customer anomaly reports often are on a high level.
- The developing organization and customers' confidence in the system passing the acceptance test exit criteria is reduced.
- Additional effort has to be spent on building and deploying new release candidates as faults are found and fixed.

In this study, we show that applying model-based testing to the E-mail gateway system, starting from early development, significantly increased the number of faults found during system testing.

1.1 Purpose

The purpose of this mixed methods study is to better understand how model-based testing can be used as a system-level test technique, starting from early development. This will be done by converging both quantitative and qualitative data. *Faults-slip-through* will be used to measure the relationship between using model-

Erlang'08, September 27, 2008, Victoria, BC, Canada.

based testing as a system-level test technique from early development, and the number of faults that should have been detected during system testing but are left undetected until customer acceptance testing.

As described above, the message gateway, developed at ETC (the research site), has two subsystems. In this study, model-based testing will be used for system-level testing of the E-mail gateway subsystem. The fault-slip-through measurements for this subsystem will be compared to the measurements of the baseline subsystem – the IM gateway. The IM gateway is tested with a combination of manual and automated test cases that are hand-crafted without the support of a model. There are many different kinds of testing. This study will focus on black box functionality testing (both positive and negative). At the same time, the managers' and developers' perception of the impact of the model-based testing will be explored using observations and qualitative interviews.

1.2 Approach

Studies of software process improvement suggest that regardless of whether a quality initiative is technical or organizational, the human factor should be considered, because of the potential barriers to change [8]. ETC has not used model-based testing as a system-level test technique before. Model-based testing is not just a matter of generating tests, and executing them to detect defects. It involves several other activities, such as creation of the system model, analyzing the output, reporting defects and generating reports. Another important activity is regression testing, which is often cited as the most important benefit of test automation [9]. The model-based testing practice must therefore be integrated into the project's test process [6]. This integration can meet resistance as existing local practices may directly conflict with the model-based testing technique [10]. In view of this, this study will be conducted in the form of a *software process improvement* initiative.

1.3 Overview

The rest of this paper is organized as follows. Section 2 gives an overview of relevant concepts, summarizes related studies and presents our research questions and hypothesis. Section 3 describes the method used in the study, including how we collected and analysed our data. Section 4 describes the two subsystems under study. Section 5 gives examples of how the system under test was modeled, describes encountered challenges and presents a set of recommendations for model-based testing on the system-level. Section 6 describes the actual results, in terms of changes in fault-slip-through. Section 7 discusses the results and presents issues in the conducted comparisons. Finally, we conclude and suggest questions for further research in section 8.

2. Related Research

This section has four parts. The first part gives an overview of model-based testing. The second part presents an overview of studies that relate directly to this one – model-based testing as a fault-detection practice in industry, and its impact on the development process. The third part presents selected measurements of early fault detection. Finally, the fourth part specifies the research questions and hypothesis of this study.

2.1 Model-based testing

The term model-based testing is commonly used for a wide variety of test generation techniques. In this article, *model-based testing* is a test technique, by which test cases are generated from a *behavior model* of the system under test [10]. Furthermore, we constrain ourselves to the generation of test cases that include a *test oracle* [10], which can assign a pass/fail verdict to each executed test.

Model-based testing typically involves the following steps: [4, 10, 11]

1. Building an abstract model of the behavior of the system under test. The model captures a subset of the system requirements.
2. Definition of test selection criteria. The criteria defines what test cases to generate from the model.
3. Validating the model. This is typically done by sampling abstract test cases from the model and analyzing them. This step is performed to detect major errors in the model that may even hinder generation of test cases.
4. Generating abstract tests from the model, using the defined test selection criteria. At this stage, the generated test cases are expressed in terms of the abstractions used by the model.
5. Transforming (concretize) the abstract test cases into executable test cases.
6. Executing the test cases. At execution time, an adaptor component transforms the output of the system to the abstraction of the model.
7. Assigning of a pass/fail verdict to executed test case.
8. Analyzing the the execution result.

The remainder of this section gives an overview of the variations within the model-based testing practice.

2.1.1 Model types

The behavior of a system can be described using a variety of different model types. Common model-types, such as finite state machines, extended finite state machines, state charts, markov chains and temporal logic are widely described in literature.

2.1.2 Model abstraction

The model must be validated against the system requirements, which may be specified at any level of formality. This implies that the model must be more abstract than the system under test. If it were not, validating the model would require as much effort as validating the system. At the same time, details of the system that are not modeled, cannot be verified using the model [12]. An overview of abstractions that can be applied in the creation of the model is provided in [13] and [12].

2.1.3 Model notation

In principle, all notations with formal semantics can be used as a basis for model-based testing. Examples of commonly used notations are formal specification languages (such as Z), tool vendor specific languages, general-purpose programming languages, the Unified Modeling Language (UML) and domain specific languages. [4]

2.1.4 Concretizing abstract test cases

The approach to concretization of the test cases depends on the nature of the model abstraction. When only the system input data is abstracted by the model, an *adaptor* component (sometimes called *driver* [12]) is typically used. The adaptor adapts the input part of the test case to the format accepted by the system under test [11]. This adaption may also be delayed to test execution time. If the test case is on a higher level of abstraction a template is used to derive a concrete test case. The template adds additional semantics making the test case executable [4].

2.1.5 Online and offline testing

The online testing technique generates the steps of a test case from the model in lock-step with executing them. This generation

technique handles the non-determinism that arises in the testing of reactive and concurrent systems [14]. With off-line test generation, a complete test case is generated before execution. This has other practical advantages, for example the test case can be executed repeatedly (regression testing). It also allows for analyzing and simplification of the test case before it is executed.

2.1.6 Available tools

Generating test cases from high-level specifications is not a recent idea. In 1986 Hayes [15] showed how to systematically derive tests of abstract data structures from a formal specification. At that time, however, the generation and execution of test cases was performed manually. Today, there is a growing number of tools available that automate many of the steps involved in model-based testing. Utting and Legeard [4] provides a comprehensive overview of model-based testing tools.

2.1.7 Quviq QuickCheck

QuickCheck, developed by Quviq AB, is a testing tool for guided random and model-based testing. QuickCheck can simplify a failed test case to a minimal failing test case [16], thereby reducing the problem of deducing the cause of failure for complex test cases. A minimal test case is a test case where every part of the system input is significant in reproducing the failure.

QuickCheck provides a framework for modelling the system under test using an Abstract State Machine. A model is built using Erlang, a general-purpose programming language [17], with support of the library provided with QuickCheck.

Applicability A case study, where Ericsson's Media Proxy was tested using QuickCheck, indicates that the tool can be applied to testing communication protocols. The study also found that QuickCheck potentially reduces the required investment compared to hand-crafted test cases [16]. The system that was tested with QuickCheck in the study had already been pre-release tested by the development team. There are no studies on using QuickCheck earlier in the development.

2.2 Model-based testing in industry

This section outlines and evaluates prior studies on model-based testing as a fault detection practice in industrial projects. Only studies where the test case generation and execution steps were automated are included.

AGEDIS (Automated Generation and Execution of Test Suites in Distributed Component-based Software) was a three-year research project on the automation of software testing funded by the European Union. Five case studies were conducted. These studies focused on applying model-based testing methods and tools to test problems in industrial settings. The studies were conducted at France Telecom, Intrasoft and IBM. The findings were that modeling increased the understanding of the system under test and was found to be an effective way to analyze complex requirements. It was also found that when a requirement changed, adapting the model and regenerating the test cases required less effort compared to updating manually constructed test cases. [18]

Artho et al. [19] presents a case study that applied model-based testing to the controller component of NASAs K9 planetary rover. The modeling language and test framework used was based on a discrete temporal logic. The technique was found promising and located a fault in the controller. The model was developed, and the testing conducted after implementation of the full system was finished.

Dalal et al. [10] reports on obstacles of introducing model-based testing into test organizations. Four case-studies of four large-scale projects are presented. One finding was that model-based tests were sometimes seen as mysterious. This was because the *objectives* of each test case are not as clearly defined as in a typical manually crafted test case. The study suggests that the projects' test process, including test strategies and planning must be adapted, so that the model-based testing is well integrated. Establishing the infrastructure for running and logging the massive amount of generated test cases requires effort. Also, modeling was found to improve the specification and the understanding of the system, which is in line with the findings of the AGEDIS studies [18].

Dalal et al. suggests that defects in the model can be minimized by ensuring traceability from the requirements to parts of the model. This allows fault analysis to faster determine whether the requirements or the implementation is incorrect. The suggestion originates in that more than half of the failed tests related to defects in the model, rather than the system under test. Other studies indicate similar model defect ratios (see Pretschner et al. [12], Blackburn et al. [3]). The following limitations in the case studies can be identified: Only individual components were tested. Test oracles were not generated from the model, but added manually.

Dalal et al. also identifies the following questions for future work:

- What are the challenges of applying model-based testing during different phases of testing?
- What benefits will model-based testing deliver at different phases of testing? [10]

Pretschner et al. [12] investigates whether the model-based testing approach pays off in terms of quality and cost. Quality of model-based test cases are compared to traditional, hand-crafted test cases. The system under test is a network controller for a automotive infotainment system. The findings of the study were that model-based testing did not detect more faults in the implementation than hand-crafted test cases. Also, no correlation between severity of errors and types of tests were found. On the other hand, the model-based tests resulted in the detection of significantly more requirement defects. The study indicates that tests were executed after the system was completely implemented. Therefore, the benefit of test case re-generation were not seen in the study. Also, the length of the test cases was restricted, as a failing test case was inspected manually, and the execution time was long due to hardware limitations. The study acknowledges these deficiencies and points out that the economics of using model-based testing based on behavioral models is not yet understood, in particular whether the life-cycle spanning updating of the model is efficient.

Blackburn et al. [3] discusses the specific skills and practices that are needed to incorporate model-based testing into an organization's test process. The presented material is based on learnings from working with model-based testing in companies and projects during multiple years. The article suggests that an incrementally developed model detects inconsistencies and missing details in requirements early in the development process. Also, objective measurements are pointed out as important to make the effects of the model-based testing visible.

Evaluation Summary Based on evaluation above, it can be concluded that existing research on model-based testing for early-fault detection is lacking. In conducted studies, model-based testing has typically not been an integrated part of the development process. Also, most studies apply model-based testing on the component level, or to a limited part of the system. Few studies focus on the the application of the technique on the system-test level.

2.3 Measuring early fault detection

The central concept underlying this study is that the cost of finding and fixing faults in software rises as the development of the

software progress into a new phase [9]. Reducing the number of defects that are left undetected until customer acceptance testing is a type of improvement work – improvement of the test process. In improvement work, measurements are important in order to know whether you are actually improving. A common test process metric is the number of defects found on different test levels[9]. An important criteria for selecting a performance measure is that it relates closely to the issue under study. As shown by Damm, most measurements cannot acknowledge that not all faults are effectively found on the test level that they were introduced on. Instead, Damm proposes the use of a method called Faults-Slip-Through (FST) [9].

2.3.1 Faults-Slip-Through

When applying the FST method, faults are classified according to the phase in which they *should* have been found. The method has the following steps [9]:

1. Determine which defects should be found on which test level. This is part of the test strategy. It is not already documented by the organization, that has to be done first.
2. For each reported fault, find the test level the defect was found on, and which test level it should have been found on, according to the documented test strategy (see 1)
3. For each test level, summarize the number of faults that should have been found earlier, per test level.

Phase Input Quality Phase Input Quality – the fault-slip-through ratio, can be calculated from the absolute FST data as follows [9]:

$$\text{Phase Input Quality}(PIQ) = \frac{SF}{PF} \cdot 100$$

where *SF* is the number of faults found on test level X that slipped from earlier test levels and *PF* the total number of faults found on test level X.

This formula calculates the percent of faults found at a test level that should have been found earlier. Damm et al. observed a relationship between the number of faults found and the PIQ; if the PIQ for the test level is high, and many faults are found on that test level, this indicates that the test strategy compliance of the earlier test level is low [9]. It is important to analyze the PIQ in the context of the absolute number of found faults. For example, the PIQ of a test level could be high, although only a few faults were found on that test level. Few faults found typically indicates that none or little improvement is needed.

2.3.2 ODC trigger analysis

One way to classify faults is by software triggers. A trigger is a type of stimulus that activates a fault into a failure. A method of such classification is ODC trigger analysis[20]. In ODC trigger analysis, each fault is assigned to an activator category. For example, faults that go into the *Test Sequencing* category are such that require execution of a sequence of functions for the fault to surface. On the other hand, faults that go into the *Interaction* category require the execution of a sequence of functions with varying parameters [20]. ODC trigger analysis can be used to evaluate the test process by identifying what types of faults are found on a specific test level. In combination with the fault-slip-through, it can be used to evaluate what activities in the test process are in need of improvement, and the success of such improvements [9].

2.4 Research Questions and Hypothesis

The central question asked in this study is:

> How can model-based testing be applied at the system-level to enable early fault-detection and increased confidence in the system?

Based on the evaluation of related work in section 2.2, and the challenges identified, the following sub-questions will be addressed by this study:

Q1. How can model-based testing be used to reduce the number of faults that are left undetected until customer acceptance testing?

Q2. What are the challenges of applying model-based testing at the system-level?

The introduction of automated system tests with a high level of interaction coverage should decrease the number of faults that are left undetected until customer acceptance testing. This expectation constitutes the hypothesis for this study:

H1. Applying model-based testing on the system level will decrease the *fault-slip-through* from system testing to customer acceptance testing.

3. Methods

The study was conducted using the action research method [21]. This was motivated by the practice oriented nature of the study, and the author's involvement in both practice and research. Action research is cyclic. Each cycle typically includes planning, acting, observing, and reflecting.[21]

The studied development project used a development process based upon the DSDM framework (The DSDM framework is further discussed in DSDM, Business Focused Development [22]). The length of each time-box was approximately four weeks. The customer conducted an acceptance test on each system release.

The study covers two releases of the messaging gateway. Figure 1 shows the timeline of the study. Due to confidentiality reasons, we cannot state the actual release names. We denote these releases as Release X and Release X+1, respectively. The second release of the E-mail gateway subsystem was developed during two shorter time-boxes, with an interim release, which we call Release X+0.5. No acceptance test was conducted on the interim release. We present the results for this release separately.

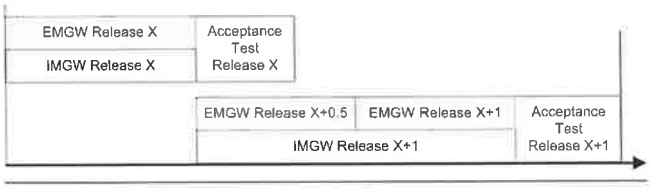


Figure 1. Study timeline

3.1 Research cycles

Each action-research cycle corresponded to a time-box in the E-mail gateway project. The following actions were conducted during each of the three time-boxes:

Planning A set of functional requirements to add to the system model was selected. The selection was based on the given priority of requirements for the current time-box. In addition the developers were asked to prioritize the modeling of the requirements. The latter was done to allow for early testing of features that the developers delivered for system testing (the features due for system testing in the time-box were not all delivered at the same time).

Acting The selected functionality was modeled. A set of abstractions was applied in the process of modeling the system. The abstractions selected were based upon the project's test strategy (what aspects to test on the system level) and trade-offs including difficulty of modeling and ease of validation of the model.

As soon as features were released by the developers for system testing, the model was used for test generation and execution. Detected anomalies were reported in the projects issue tracking system. After analyzing an anomaly, a fault was typically found to be present in either the model or in the system. As faults were corrected, the tests were re-executed.

Observing After completion of the time-box, the results of the testing were observed and analyzed.

Reflecting The developers and test manager were involved in reflecting on the results of the model-based testing and suggesting improvements for the next time-box.

3.2 Site of study

The site of study was Erlang Training and Consulting (ETC). Part of ETC's core business is to develop distributed fault tolerant systems utilizing the Erlang/OTP platform, most of which are network-intensive. Erlang/OTP includes the general purpose programming language Erlang, which has built-in support for concurrency, fault-tolerance, and a set of libraries for application development [17].

3.2.1 Experience of model-based testing

Prior to this study, model-based testing was used by some developers for testing on the unit level. The tool used was QuickCheck (see section 2.1.7). The majority of the developers of ETC had undergone training in use of the tool.

3.3 Researcher's role

The author's role in this study, except from data collection and analysis, was to introduce model-based testing as a technique for system testing in the project. The author also constructed the model of the system, and executed the tests generated from the model. The risk of bias inherent due to the author's involvement and interventions is acknowledged.

3.4 Data collection and analysis

This section presents the applied data collection and analysis procedures. Both quantitative and qualitative data was collected in this study. Found faults were analyzed and the fault-slip-through to acceptance testing measured. The impact of the model-based testing was verified through qualitative interviews with the test manager and developers. In addition, the experiences of system-level modeling, test execution were logged.

3.4.1 Fault Analysis

Faults found during system testing and acceptance testing were measured at the end of each time-box for each of the two subsystems. Data on found faults was collected from the organization issue tracking system. Both the internal organization and the customer reports detected anomalies into this system. The reports include the details of the anomaly, the reporter and the date of the report. This data was sufficient for the FST measurement. Each anomaly report was analyzed according to the following criteria:

1. The anomaly has been confirmed to have been caused by a fault in one of the two subsystems
2. The fault related to a functional aspect of the system

Note that (1) also implies that faults in a shared component of the two subsystems were filtered out. Reports that fulfilled these two criteria were used as input for the fault-slip-through measurement (see section 3.4.2) and classified through ODC fault analysis (see section 3.4.3).

The detected faults for the E-mail gateway system were further analyzed according to whether they were found due to execution of a manually crafted test case, or a test case generated from the model. This distinction was useful to evaluate the model-based testing at the end of each time-box. For example, what faults did the manually crafted test-cases detect, that the ones generated from the model did not? The evaluation was used as input to the improvement of the model for the next time-box.

In case of duplicate anomaly reports for a fault in the system, both reports were used to classify the fault, but only one fault was counted. No faults had an anomaly report from both a manually crafted and a generated test case.

3.4.2 Fault-slip-through

The fault-slip-through to system testing and acceptance testing was measured at the end of each time-box, for each of the two subsystems. The Phase Input Quality for the two test levels was then derived from the fault-slip-through data.

Definition of test levels The FST measurement requires a definition of what defects should be found on which test level (see section 2.3.1). This definition was created by means of open-ended interviews with the test manager, and four developers. In the interview, the subject was first asked to identify the test levels of the test process. For each identified test level, the subject was then asked to describe the type of defects that should be found on that level. At the end of the interview, the subject was presented ten anomaly reports, selected from the issue tracking system. The anomaly reports were randomly selected from both development projects, with the constraint that they were all reported during the last two time-boxes, and that the reported anomaly had been found to be caused by a fault in the system. For each report, the subject was asked to classify on which test level the fault should have been found. This second step was performed to validate the answers in the interview.

Defined test strategy We formalized the test strategy based on analysis of the interview results. The test strategy defined the following test levels: unit testing, integration testing, external integration testing, system testing and acceptance testing. The definition of the test strategy was verified with the interview subjects.

3.4.3 ODC trigger analysis

Faults were classified using ODC trigger analysis. For about 80% of the reported anomalies, the information present in the anomaly report was sufficient for classification. In the rest of the case, the involved project members were consulted. As recommended by Damm et al [9], we iteratively developed the classification scheme during fault analysis. The scheme used is shown in table 1.

Category	Description
Coverage	Execution of a single function
Sequencing	Execution of a sequence of functions
Interaction	Execution of a sequence of functions and multiple parameters interacting with each other
Variation	As Interaction, but including invalid parameters (negative testing)
Fault tolerance	Recovering from faults and fail-over scenarios
Concurrency	Faults that only occur due to concurrent interaction with the system
Configuration	Faults related to specific configurations

Table 1. ODC Trigger Classification Scheme

The E-mail gateway and IM-gateway requires the client to connect and login before any other function can be used. Faults that were triggered by the execution of a single function, after connection and login, were therefore classified to the *Coverage* category. An exception was made for faults where the parameters to the login function affected whether the fault was triggered or not.

3.4.4 Qualitative interviews

We conducted qualitative interviews with the test manager and four developers. The purpose of the interviews was to explore the perceived impact of the model-based system testing. The test manager was involved in the testing of both subsystems, while two of the developers worked primarily on the IM gateway and two on the E-mail gateway.

Time and location The subjects were planned to be interviewed at two instances. Once after the end of the second research cycle, with the intent to get detailed in-process feedback on the improvement initiative. In addition, once before the acceptance test of the last studied system release (Release X+1). Due to time constraints, we conducted only the first set of interviews. The interviews were conducted at the research site.

Interview outline The interviews were conducted using the interview guide approach and had two parts. In the first part, the subjects were asked about their confidence in the testing of the respective subsystem and their confidence that the acceptance test exit criteria would be fulfilled without extending the acceptance test phase due to detected faults. Second, they were asked to elaborate on the factors involved in their level of confidence. In the second part, the subjects were directly asked about their perceived impact of the model-based system testing.

3.4.5 Observations

Observations of team and customer meetings, and e-mail correspondence between the customer and the developing organization provided additional data on the confidence in the system and feedback on the improvement initiative.

To assist in the data collection, a field log was used to record observations. The field log was also used to document experiences on the system modeling, test execution, and reporting of the test results.

3.5 Model-based testing tool

The tool used in this study was Quviq QuickCheck (see section 2.1.7). QuickCheck uses Erlang as a specification language. This means that there was in-house competence in the specification language used to model the system under test. Also, the tool had been used and proven at the site of study (see section 3.2). It was not within the scope of this study to compare different tools for model-based testing. Therefore, the selection of a tool that has seen successful use in the development environment allowed the study to stay within its focus area.

3.6 Verification

The concurrent triangulation strategy[23] was used to verify the findings of the study. The fault-slip-through measurements were compared to the impact of the model-based testing, as perceived by the interview subjects.

4. System under study

The E-mail gateway (EMGW) provides e-mail clients a uniform interface to message store servers that use a variety of access protocols. Supported servers include those that use the Post Office Protocol version 3 (POP3), the Internet Message Access Protocol version 4rev1 (IMAP4rev1) and the Mobile Services Protocol (MSP). The client access a message store, through the gateway, using a subset of the IMAP4rev1 protocol. The gateway also supports the IMAP IDLE extension. The extension enables the client to be notified as messages arrive to a mailbox, without having to poll the server.

The Instant Messaging Gateway (IMGW) uses the Wireless Village Client-Server protocol to provide mobile clients access to multiple instant messaging protocols.

Both of these subsystems were developed using the Erlang/OTP platform. They share large parts of the architecture, and a set of core components, originally developed for the IMGW. Most of the implementation was conducted in Erlang, while some parts were done in C. All development was conducted in a GNU Linux/OpenSuse environment. The project team consisted of 9 persons on full time. Some of the developers worked solely on one of the subsystem, while some were involved in both systems.

4.1 The IMAP4 protocol

The IMAP4rev1 is specified by a Request For Comments (RFC) and is ratified as an *internet standard* by the Internet Engineering Task Force (IETF) [24]. The 108 page long specification defines a set of commands that the client can send to the server, how each command is to be interpreted by the server and the responses that may be returned. The IMAP4rev1 protocol builds on the Multipurpose Internet Mail Extensions (MIME), defined by a range of RFCs [25] which specifies the format of e-mails.

An important part of the protocol is that the connection can be in different states (see figure 2).

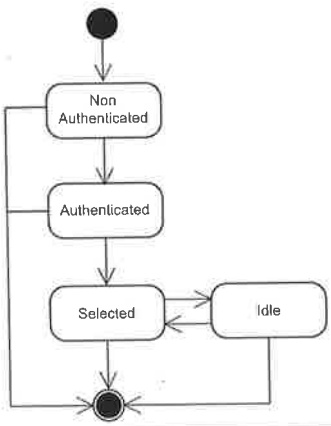


Figure 2. IMAP connection states

Only a subset of the commands are valid in each state. The connection starts in the `Non Authenticated` State. A successful login command results in a transition to the `Authenticated` state. The client can now select a mailbox to work with. In this state, commands that affect the messages in the mailbox can be executed. In addition, the client can issue an *idle* command, causing a transition to the `Idle` state, in which the client is notified about changes to the mailbox. The connection can be terminated from any state, either by the client sending a logout command, or by the connection being closed (due to a client or server error).

Although only a subset of the IMAP4 protocol is implemented by the EMGW, there is still a wide range of variations that the system must be able to handle. The subset of the protocol that the system must handle is defined by the internal ETC system requirement specification document. This document and the relevant RFCs were the main sources of information when developing the model of the system.

4.2 The Wireless Village protocol

The Wireless Village Client-Server protocol is part of a set of specifications for mobile instant messaging. The protocol uses the HTTP1.1 protocol as a bearer and operates over TCP/IP. The Extensible Markup Language (XML) is used to exchange data between the client and server. [26] The core of the protocol is specified by a set of five documents, in total 220 pages, not including the HTTP1.1 and XML specifications.

5. Modeling and Testing Challenges

We constructed a QuickCheck Abstract State Machine (ASM) model of the EMGW system, which was subsequently used to generate and execute test cases. This section gives examples of how the system was modeled, and describes encountered challenges in the modeling and test execution.

5.1 QuickCheck Abstract State Machines

A QuickCheck ASM is specified by a state, a callback module and a set of command generators which are used by the callback module. The ASM is normally used in a QuickCheck property which is then tested. QuickCheck properties are further discussed in Testing Telecoms Software with Quviq QuickCheck [16].

A generated test case is a list of symbolic commands, on the form:

```
{set,{var,1},{call,Module,Function,Arguments}}
```

each of which represents the execution of an *external function*. ASM external functions allow the modeling of non-determinism in the system environment, see Sequential Abstract-State Machines Capture Sequential Algorithms [27] for more information on this topic. During test case generation, QuickCheck represents the results of a symbolic command by a symbolic variable (`{var 1}` in the example above), that can be used as part of subsequent commands. During test case execution the symbolic variables are replaced by the actual value provided by the environment.

QuickCheck implements a subset of the ASM execution theory (described by Gurevich [28]). A limitation is that there is no built in support for separating parts of the model. That is, separating different concerns of the system, in order to avoid a monolithic model that is difficult to validate and maintain. At the same time, it is relatively easy for the modeler to create nested state machines, as the model is specified using a general purpose programming language. This is a technique that was used extensively for the EMGW model.

5.2 The EMGW state

Only a subset of the IMAP commands are valid in each protocol state. The state of each client connection must be modeled to be able to constrain the test case generation to mostly positive test cases. In addition, the state of each mailbox must be modeled, to be able to generate valid commands that affect the messages.

The EMGW state was modeled by an Erlang record:

```
-record(state, {clients=[], accounts}.
```

where `clients` is a list of client connections, and `accounts` a list of e-mail accounts. The following Erlang record models each client connection:

```
-record(client, {connection, user_id, imap_state,
                 selected=undefined, idle=false}).
```

`connection` is a reference to the client connection in the adaptor component, `user_id` an abstraction of the login credentials, that is also used to identify the account. `imap_state` is the current state of the IMAP connection, `selected` is the name of the selected mailbox and `idle` specifies whether the connection is in the `idle` state.

5.3 Adaptor

One of the first tasks was to develop the *adaptor* (see section 2.1.4). The adaptor is an abstract concept. In practice, the adaptor may be split into multiple components. For the EMGW, we used two adaptor components:

- Adaptor for the IMAP protocol
- Adaptor for sending e-mail messages to an account, to be able to test the message related IMAP commands.

We recognized that the IMAP protocol abstractions would have to change as additional system features were incrementally added to the model. With the design principle "encapsulate what varies", the IMAP protocol adaptor was constructed in two parts:

- An IMAP client that enables communication with the EMGW using a functional Erlang interface. The client formats the IMAP commands and parses the server response into an Erlang term representation.
- An interface to the aforementioned client that maps from and to the abstraction of the model

In this way, changes to the model abstractions led to isolated changes in the second part of the adaptor. Also, the IMAP client was not specific to the model abstractions, and could later be reused by the project team for system load testing.

5.4 Generating IMAP commands

IMAP commands were generated in the form of calls to the adaptor. As a running example, this and the following section will use the IMAP *select* command.

```
select_cmd(Client, State) ->
   {call, imap_adaptor, select,
          [Client#client.connection,
           mailbox_name(Client,State)]}.
```

Here, a `select` command is generated, given a client and the current state. A random mailbox name is generated by the `mailbox_name` generator, which is specified as follows:

```
mailbox_name(Client, State) ->
   ?LET(Mailbox,
        oneof(account(Client#client.user_id, State))
          #account.mailboxes),
        Mailbox#mailbox.name).
```

The clients account is looked up from the state, and a random mailbox is picked, whose name is returned by the generator.

5.5 ASM callback functions

A QuickCheck ASM callback module specifies a `precondition`, `next_state` and `postcondition` function. The callback functions for the `select` command were specified as follows.

The `precondition` determines whether to include a symbolic command in a test case, given the current state.

```
precondition(State, {call, _, select, [CPid,_]}) ->
   min_state(client(CPid,State), ?AUTH)
```

Here, `min_state` ensures that the client is at least in the `Authenticated` state for the command to be included in the test sequence.

The `next_state` function updates the state, given the executed command, its result, and the state the command was executed in.

```
next_state(State, _Result,
           {call, _, select,
                    [CPid, MailboxName]}) ->
   Client = client(CPid, State),
   update(Client#client{imap_state=?SELECTED,
                        selected=MailboxName},
          State);
```

The `next_state` function specifies that the client connection transitions to the `Selected` state, and the selected mailbox name is updated when the `select` command is executed.

The `postcondition` evaluates the result of a command, given the state the command was executed in. A test case is assigned a *failed* verdict if a postcondition fails.

```
postcondition(State, {call, _, select,
                      [CPid,_]}, Result) ->
   is_status(Result, ?OK_RESP);
```

Here, `is_status` checks that the server responds to the select command with the `OK` response.

5.6 Challenges and lessons learned

This section presents a set of encountered challenges and recommendations for system-level modeling and test execution.

Develop the model iteratively Iteratively developing the model is seen as crucial. There is a high investment in creating the model. Using it from early development is seen to give higher returns as the partial model can be used to test the system in early development. Early modeling also allowed the modeler to gradually build up the high level of domain knowledge required. In addition, we found that the modeling practice contributed to the understanding of the system requirements, as validation of the model and the analysis of detected faults led to the discovery of unspecified behavior.

Finding abstractions Finding abstractions that allow the model to be more abstract than the system under test can be difficult. We found that this can be remedied by using multiple layers of abstraction, and a combination of multiple model types. We used a Backus-Naur form (BNF) grammar to generate a parser for the IMAP protocol, which was used in the IMAP client part of the adaptor. The parser threw an exception for any malformed system output. In practice this means that part of the verification was performed by the adaptor, but allowed for a simpler ASM model.

Test techniques are complementary Manually crafted test cases can test samples of complex behavior, without having to create a complete model of the behavior. We found that the techniques of model-based testing and manually crafting test cases are complementary. For example, the system fail-over scenarios, that make sure that service is maintained in the event of a failure were seen as overly complex to model and were instead tested using a set of hand-crafted test cases. A useful technique, that can be applied when a system feature is overly complex to model completely, is to only model the feature partially, and limit the command generators to test cases that the model is valid under.

Put effort into the adaptor Developing an adaptor for a system-level model requires considerable effort. On unit and component level, the interface under test is typically less complex. Testing the IMAP protocol required the development of an IMAP client, and applying of multiple layers of abstractions. The total effort involved in developing the adaptor exceeded 40 working hours. On the other hand, we found that a well designed adaptor can simplify the modeling, as described above. Parts of a layered adaptor can also potentially be re-used in other parts of the project.

Model validation by testing The complexity of a system-level model means that the modeling has to start as soon as the requirements for the current time-box have been established. On the other hand, the validation of the model, aside from code review and sampling, cannot start until the system is ready for system testing. We experienced that validation of the model had to continue throughout the full system-test phase. We also found that faults in the system sometimes not only hid other faults in the system, but also faults in the model. As faults in the system were repaired, the next test runs often found faults in the model that were previously undetected. We recommend that a project uses short time-boxes, with small increments in both the implementation and model. This allows for shorter feedback-cycles which eases the model validation.

Reliance on external components System level tests rely on a large number of external components, many of which cannot be be replaced by a dummy (stubbed). For the EMGW system, the test environment included a POP3 and IMAP server which was used through the gateway. During system testing, we found two faults in this server that hindered further testing, and could not be worked around. Tracking down the problem and patching the server took two working days. There is a high risk of failures in external components as the model generated test-cases tests intricate scenarios. We recommend that the suppliers of external components are carefully selected, and that good contacts are maintained with the suppliers, in case of failures.

Execution of partial tests When a failure in the system is detected by the model-based testing, testing cannot be continued since the same failure is likely to occur again. It should be possible to continue testing, in the presence of minor faults in the system. QuickCheck does not provide any support for constraining test case generation or disable parts of the model, in order to allow for this. Instead we had to perform temporary changes to the model. In the absence of tool support, we recommend that such changes are tracked. We used a commenting convention to mark temporarily changed model parts.

6. Results

6.1 Detected faults

The following sections presents the faults found in the studied releases. We present faults detected during system testing and acceptance testing of two releases of the messaging gateway, Release X and Release X+1. We also present the faults detected in the interim release, Release X+0.5 that only included changes in the E-mail gateway.

6.1.1 Release X

Table 2 and 3 shows the fault-slip-through to system testing and acceptance testing for the two subsystems.

Found/Belonging	System Test	Acceptance test
Unit Test	1 (0)	1
Integration Test	1 (1)	0
External Integration Test	0 (0)	0
System Test	9 (6)	3
Acceptance Test	0 (0)	2
Total found/test level	**11** (7)	**6**

Table 2. FST Release X – Email Gateway

As can be seen in table 2, 11 faults were found in the E-mail gateway during system testing. The numbers in parenthesis show

how many of the faults were detected by the model-based tests. 7 of the 11 faults were detected by the model-based tests. Of the 4 faults found with the hand-crafted test cases, 3 could have been found with the model-based tests (but were found by a hand-crafted test case before that), one had a trigger classified to the category *Fault tolerance*. 6 faults were found during acceptance testing. Of the 6 faults found during acceptance testing, only two should have been found on that test level.

Found/Belonging	System Test	Acceptance test
Unit Test	6	4
Integration Test	0	0
External Integration Test	0	0
System Test	8	10
Acceptance Test	0	4
Total found/test level	**14**	**18**

Table 3. FST Release X – IM Gateway

Although more faults were found in the IM Gateway during system testing (see table 3), significantly more faults were also found during acceptance testing. Compared to the E-mail gateway, a larger percentage of the faults should have been detected already during unit testing.

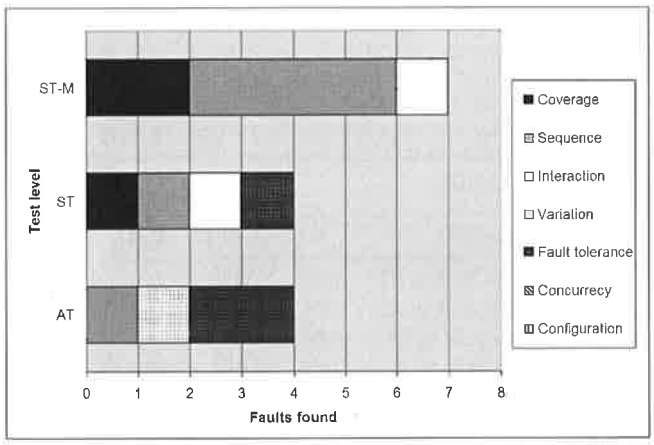


Figure 3. EMGW Release X, ODC trigger distribution by test level

ODC Trigger Analysis Figure 3 shows the ODC trigger distribution for the faults found during system and acceptance testing of Release X of the E-mail gateway. The bar labeled *ST-M* shows the distribution of the faults found by the model-based tests, while the bar labeled *ST* shows the distribution of the faults found by the manually-crafted tests. *AT* shows the distribution of the faults that slipped through to acceptance testing. A majority of the system test triggers are in the sequence category.

As can be seen in figure 4, a large proportion of the IM Gateway fault triggers are in the interaction category, for both test levels. The number of faults with a sequence trigger are about the same as for the E-mail gateway. It is also notable that four of the faults found during acceptance testing have triggers in the coverage category, as each of these faults could have been found earlier by a test case invoking only a single function (with a specific set of parameters).

6.1.2 Release X+0.5

The E-mail gateway had an interim delivery halfway to release X+1. No acceptance testing was conducted on this delivery. Table

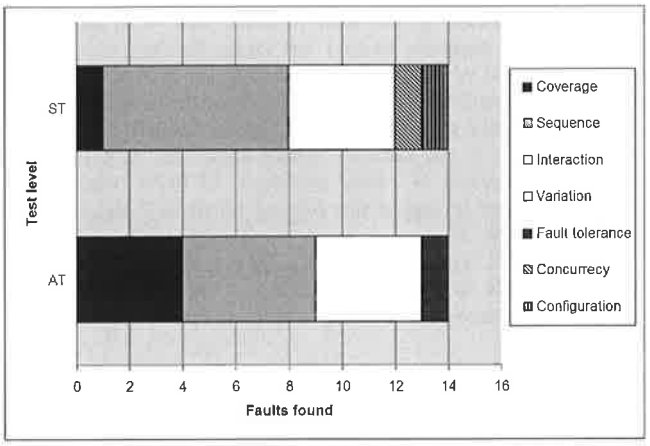


Figure 4. IMGW Release X, ODC trigger distribution by test level

4 shows the number of faults found during system testing. The numbers in parenthesis show how many of the faults were detected by the model-based tests. In total, 17 faults were found, of which 11 were found with the model-based tests. The 6 faults found with the hand-crafted test cases and not by the model-based tests were further analyzed, with the following findings:

- Four of the faults related to the server sending the wrong error code in response to an invalid request. Although the error conditions were part of the system model, the specific error code for invalid requests were not. They were subsequently added to the model.
- Two of the faults were classified to the trigger category *Fault Tolerance*. Such faults were not to be found with the model-based tests, according to the test strategy.

Found/Belonging	System Test
Unit Test	4 (4)
Integration Test	0 (0)
External Integration Test	0 (0)
System Test	13 (7)
Acceptance Test	0 (0)
Total found/test level	**17** (11)

Table 4. FST Release X+0.5 – E-mail Gateway

6.1.3 Release X+1

Table 5 and 6 shows the fault-slip-through for the two subsystems.

Found/Belonging	System Test	Acceptance test
Unit Test	24 (21)	3
Integration Test	1 (1)	0
External Integration Test	0 (0)	0
System Test	39 (26)	4
Acceptance Test	0 (0)	10
Total found/test level	**64** (48)	**17**

Table 5. FST Release X+1 – E-mail Gateway

As can be seen in table 5, 64 faults were detected during system testing of the E-mail Gateway. 48 of these faults were detected by

the model-based testing. The table indicates that 16 of the faults were found by manually crafted test cases. In fact, 9 of these faults were found while manually analysing the system input and output traces, resulting from the model-based tests. It is notable that 24 of the faults should have been detected already during unit testing. Analysis of the anomaly reports show that most of these are related to parsing of e-mail messages. 17 faults were found during acceptance testing of this release. Of these 7 should have been found earlier.

Only 14 faults were detected in the IM Gateway during system testing (see table 6). The same number of faults were detected during acceptance testing. A majority of the faults should have been found earlier.

Found/Belonging	System Test	Acceptance test
Unit Test	1	4
Integration Test	1	0
External Integration Test	0	1
System Test	12	7
Acceptance Test	0	2
Total found/test level	**14**	**14**

Table 6. FST Release X+1 – IM Gateway

ODC Trigger Analysis Figure 5 shows the ODC trigger distribution for the faults found during system and acceptance testing of Release X+1 of the E-mail gateway. Most of the triggers are in the sequence category, but a significant number of the faults found by the model-based testing are in the coverage category. Most of the faults with a coverage trigger are those that slipped through from unit testing, and are related to parsing. The model-based tests detected six faults with interaction triggers.

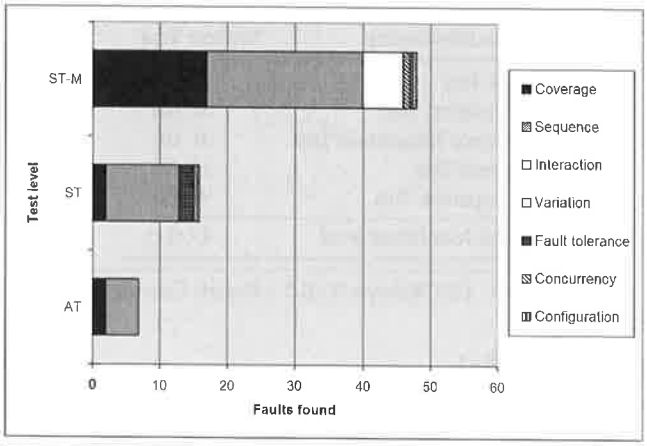


Figure 5. EMGW Release X+1, ODC trigger distribution by test level

Figure 6 shows the ODC trigger distribution for the IM gateway. The trigger distribution is similar to that of Release X of the subsystem. Again, a large proportion of the triggers are in the interaction category.

6.1.4 Summary – Phase Input Quality

By calculating the Phase Input Quality (PIQ) from the FST measurements (as described in section 2.3.1), we can compare the fault-slip-through data of the two subsystems. As relatively few faults were detected for some test levels, the statistical power of the PIQ

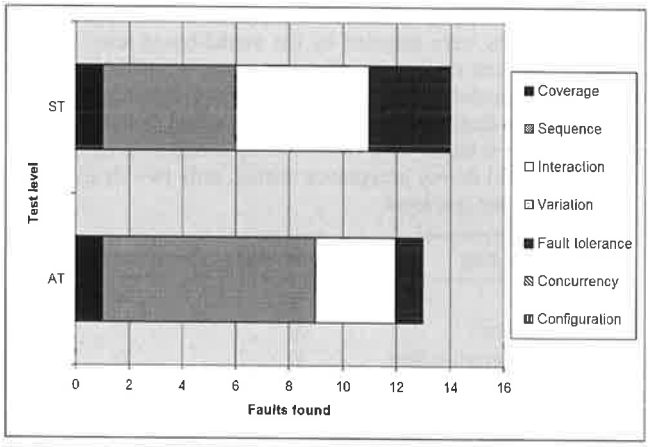


Figure 6. IMGW Release X+1, ODC trigger distribution by test level

is low. The comparisons should therefore only be reviewed in context of the absolute number of faults.

As can be seen in table 6.1.4, the E-mail gateway system test PIQ was 18% for Release X, the first studied release, and increased for subsequent releases. For the IM gateway the system test PIQ has instead decreased from 79% to 14%, although this latter value has a very low statistical power due to the few number of faults found during system testing. Although 17 faults were found during the acceptance testing of release X+1 of the E-mail gateway, a majority of these faults were not slip-throughs. This gives a acceptance test PIQ of 41%. This can be compared to the IM gateway, were a majority of the faults (86%) found during acceptance testing should have been found earlier.

Release/FST	X	X+0.5	X+1
EMGW FST to ST	18%	24%	39 %
EMGW FST to AT	67%	–	41%
IMGW FST to ST	43%	–	14%
IMGW FST to AT	78%	–	86%

Table 7. PIQ - Comparison between releases of subsystems

6.2 Perceived impact and feedback

This section summarizes the results of the first set of interviews, conducted at the end of the second research cycle.

IM gateway The three subjects involved in the development and testing of the IM gateway generally expressed a low confidence in release X of this subsystem. The fact that system testing was not completely finished as the acceptance testing started was stated as a factor in this low confidence level. Two subjects also stated that they perceived the system testing as not sufficiently covering all feature interaction points, with the risk that new features might cause yet undetected side effects in other parts of the system.

E-mail gateway The three subjects involved in development and testing of the E-mail gateway expressed a high level of confidence in release X and release X+0.5 of this subsystem. All of the subjects stated that improved testing on all levels contributed to this level of confidence. The three subjects (who had also experience with the development and testing of the IM gateway) pointed out improved unit testing and the model-based system testing as contributing activities. One of the subjects thought that the results of the model-based testing was clear to internal organization. The two other

subjects thought that only the capability of detecting defects was clear, but that visibility in terms of executed tests and system coverage needed improvement. All three subjects stated that they perceived the results as unclear to the external organization.

7. Discussion

A high number of faults were found in both subsystems during the acceptance test of release X. Over half of these defects should have been found during system testing. We can thus not measure any significant difference in fault-slip-through between the two subsystems for this release. A high number of faults were detected by the model-based tests during system testing of Release X+0.5 and Release X+1. Of the faults detected in the E-mail gateway during the acceptance testing of release X+1, most could not have been detected on previous test levels. There is a significant difference compared to the IM gateway, where most of the faults detected during acceptance testing should have been found earlier. The finding that our improvement initiative did not see effect until Release X+1 is in line with prior studies that suggest that successful implementation of improvements requires multiple iterations [29].

7.1 Complexity of protocol

The difference in complexity of the two subsystems external protocols are likely to have influenced the ODC trigger analysis results. In release X, the number of detected faults with a interaction trigger were significantly higher for the IM gateway, while few faults with the interaction trigger have been found in the E-mail gateway overall. The IM Gateway uses the Wireless village protocol (see section 4.2), which we perceive as being more complex than the IMAP protocol (see section 4.1), in terms of the number of interacting request and response parameters.

7.2 Stability of IM gateway

Significantly less faults have been found during the system test of the IM gateway in release X+1. Differences in the types of features added to the two subsystems is likely to have influenced these figures. Multiple new components were developed for the E-mail gateway for this release. The new IM gateway features on the other hand, were mostly implemented by the extension of existing components that have been thoroughly tested in previous releases. Notwithstanding the higher stability of the IM gateway, more faults in this subsystem slipped through to acceptance testing, compared to the E-mail gateway.

7.3 Fault-slip-through to system testing

We can see that the fault-slip-thorough to system testing was significantly lower in the E-mail gateway, compared to the IM-gateway, in release X. We attribute this to improvements in unit testing, attributed to another improvement initiative (see section 7.4, Internal validity). In subsequent releases of the E-mail gateway, more faults have slipped through to system testing. One plausible reason for this is that as the system testing improves, the developers perform less unit testing. This has been indicated by two of the developers, who during development of release X+1 stated that they would like to start system testing early, due to the model-based tests' potential of finding defects.

7.4 Validity Threats

When conducting an empirical study in industry, the subject cannot be controlled as in a laboratory research experiment. The following validity threats may therefore be of concern in this study.

Internal validity A possible threat to the internal validity of this study is another process improvement initiative that was executed at the research site during the duration of this study. The objective of the other initiative is to increase the quality of the releases delivered to the customer. It might therefore be of concern as to whether any observed changes in fault-slip-through can be attributed to this model-based testing initiative. This threat is mitigated by two facts. First, this other initiative is project wide (it affects both subsystems), secondly the fact that the model-based test cases have found a majority of the faults in the E-mail gateway.

Another validity threat is whether other factors than the model-based testing have had an impact on an eventual reduction of fault-slip-through to acceptance testing. The interviews conducted with the test manager and developers increase the validity, as they show that the internal organization is in agreement about the impact of the model-based testing.

External validity In this study, the results are overall not fully generalizable since they are dependent on the studied project using certain processes and tools. Nevertheless, the results should be generalizable within similar contexts.

A threat to the generalizability of the results of this study is the fact that the tested system is developed in Erlang, which is also the specification language used with the model-based testing tool. Concerns that the results are not generalizable to projects where the implementation and specification languages differ may therefore be raised. However, the interfaces of the system, that the system tests interacted with, are internet standard protocols, layered over TCP/IP. While Erlang mechanisms were taken advantage of to issue test specific commands (for example to restart components to ensure a consistent state at the start of each test case), the effort to implement these test specific interfaces in another environment is seen as negligible, compared to the total effort required for the model-based testing.

8. Conclusions and future work

This study set out to contribute to the understanding of system-level model-based testing as a test technique for early fault detection. Our experiences of modeling and test execution are generally in line with those reported by prior studies. We contribute further to the understanding of system-level model-based testing by presenting a set of challenges and recommendations specific to this test level.

A substantial initial investment is required to integrate the model-based testing into the test process. As the management and customers are highly dependent on measurable results and progress reports from system testing, introducing model-based tests on this level requires considerable planning and effort.

The test results for the two subsystems shows that, for the subsystem tested with the model-based tests, significantly less faults slipped through from system testing to customer acceptance testing. This supports our hypothesis H1, that model-based testing would decrease our fault-slip-through from system testing to customer acceptance testing.

During the study, we observed that the customers' confidence in the system is dependent on the availability of test reports. The customer (external organization) requires these reports to see that the system has been sufficiently tested. As identified by prior studies, it is also important to make the results of the model-based testing visible within the internal organization, to get commitment to the technique. We identify the following question for further research:

- How can reports on the coverage and results of model-based tests be presented to the internal and external organizations?

Acknowledgments

Part of this work has been carried out during my time at the IT University of Göteborg. The work has been sponsored by Erlang Training and Consulting Ltd and Quviq AB. I would like to thank my supervisor Thomas Arts for his help and support during the project.

References

[1] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–147, January 2001.

[2] Barry Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.

[3] Mark Blackburn, Robert Busser, and Aaron Nauman. Why model-based test automation is different and what you should know to get started. In *International Conference on Practical Software Quality*. Software Productivity Consortium, NFP, 2004.

[4] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[5] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, 13(1):25–53, March 1997.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Adrian M. Colyer. From research to reward: Challenges in technology transfer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 569–576, New York, NY, USA, 2000. ACM.

[8] Mahmood Niazi, David Wilson, and Didar Zowghi. A maturity model for the implementation of software process improvement: an empirical study. *Journal of Systems and Software*, 74(2):155–172, 2005.

[9] Lars-Ola Damm. *Early and Cost-Effective Software Fault Detection - Measurement and Implementation in an Industrial Setting*. PhD thesis, Blekinge Institute of Technology, Department of Systems and Software Engineering, 2007.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[11] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Working Papers 2006. Department of Computer Science, The University of Waikato (New Zealand), April 2006.

[12] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM.

[13] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. In *Electronic Notes in Theoretical Computer Science. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 116, pages 59–71, Los Alamitos, CA, USA, January 2005. Elsevier Science Publishers Ltd.

[14] Colin Campbell, Margus Veanes, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer, technical report MSR-TR-2005-59. Microsoft Research, May 2005.

[15] I J Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, 1986.

[16] 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.

[17] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[18] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In *1st European Conference on Model Driven Software Engineering*. AGEDIS, December 2003.

[19] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

[20] Ram Chillarege and Kathryn A. Bassin. Software triggers as a function of time - odc on field faults. *DCCA-5: Fifth IFIP Working Conference on Dependable Computing for Critical Applications*, September 1995.

[21] Wesley Vernon. An introductory guide to putting action research into practice. *PodiatryNow*, February 2007.

[22] Jeniffer Stapleton. *DSDM, Business Focused Development, Second Edition*. Pearson Education, 2003.

[23] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications Inc., 2003.

[24] Network Working Group. Request for comments 3501 - internet message access protocol - version 4rev1. The Internet Engineering Task Force, http://www.ietf.org/rfc/rfc3501.txt, March 2003.

[25] Network Working Group. Request for comments 2045 - multipurpose internet mail extensions (mime) part one: Format of internet message bodies. The Internet Engineering Task Force, http://www.ietf.org/rfc/rfc2045.txt, November 1996.

[26] Wireless Village. Wv client-server protocol v1.1. Open Mobile Alliance Ltd, 2002.

[27] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.

[28] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[29] Anna Borjesson and Lars Mathiassen. Successful process implementation. *IEEE Software*, 21(4):36–44, 2004.

Erlang Testing and Tools Survey *

Tamás Nagy and Anikó Nagyné Víg
Erlang Training and Consulting Ltd, London, United Kingdom
{aniko, tamas}@erlang-consulting.com

Abstract

As the commercial usage of Erlang increases, so does the need for mature development and testing tools. This paper aims to evaluate the available tools with their shortcomings, strengths and commercial usability compared to common practices in other languages.

To identify the needs of Erlang developers in this area we published an online survey advertising it in various media. The results of this survey and additional research in this field is presented. Through the comparison of tools and the requirements of the developers the paper identifies paths for future development.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

General Terms Verification, Management

Keywords Erlang, Test Tools, Market analysis

1. Introduction

The aim of this paper is to conduct research on the tool chains used in Erlang software development projects.

- To identify the strengths and weaknesses of the existing practices.
- To point out missing components of a healthy workflow.

The focus is on property and model based testing and test driven development. Furthermore, it aims to create guidelines for further development by identifying the common patterns in this area.

We published an online survey about Erlang tools. This had been advertised on mailing lists and separate emails had been sent out to selected Erlang oriented people. We have got ~200 responses from developers, managers, testers and research engineers.

The research uses the results of the online survey aimed at Erlang developers, but takes into account tools which are currently not used by the Erlang community but have similar functionality to existing ones. This is to provide a cross reference to common practices in other communities.

The next section describes the research methods for collecting information from Erlang users. In Section 3, non-Erlang testing tools are explored to provide a basis for comparison and determining the functionalities we need (gaps) from Erlang tools. Section 4 focuses on tools used by the Erlang community, using the survey's results to identify the most widely used tools and applications which could be used to create an integrated framework for development. Section 5 rates the applicability of these tools in developing commercial products.

* Supported by EU FP7 Collaborative project ProTest, grant number 215868

Erlang'08 September 27, 2008, Victoria, BC, Canada.

2. Research method

We published an online survey in order to gather data about the usage and spread of Erlang tools and applications in the community. We advertised this via the Erlang mailing list [2], which has around 1000 registered users, and on separate smaller Erlang related mailing lists. For example the trapexit [1] site's mailing list, and the ProTest project's [3] mailing list. This survey was open for everyone, and anonymous submission was accepted as well.

There was a separate survey, with the same questions, open to a selected 200 people who were known to have a background in Erlang but not necessarily in software development.

Overall, for the two surveys, 200 responses were received. The submitters were questioned about their role within their organization. From the answers it is clearly visible that 40-45% were developers and the remaining 55-60% almost equally proportioned between the managers, testers, researchers and students.

2.1 Structure of the survey

The survey contained 20 questions covering four different topics. The questions can be found in the Appendix A. The topics covered were:

1. The development environment of the Erlang users.
2. How widely are the currently available Erlang tools known and used. The participants had the possibility to raise specific problems, if any occured, which put off the adoptation of the tools.
3. The submitter's role within the organization and background in Erlang.
4. Identify common processes which could be helped with tool support.

3. Testing tools

There is a lot of research focused on automated testing with many successful industrial case studies justifying these techniques having a place in a commercial setting.

These tools cater to different purposes. Because of this, it is difficult to show everything in one paper. The aim is to highlight the unique features available in these tools, providing a bridge to Erlang.

3.1 Model based testing

In brief, model based testing means that test cases are derived from a model that describes some aspects of the system being tested.

The Microsoft Spec Explorer [Silva et al.(2008)] is a model based testing tool developed in .NET. There are two specification languages one can use to create the model of a system:

- Spec# - which is an extended version of C# with a possibility to specify pre- and postconditions
- AsmL [Barnett et al.(2003)] - which is an abstract state machine language

One of its features is the possibility to visualise the model of the system. Because the models of large systems tend to be very complex, the visualisation makes it possible to merge states of the system into hyperstates. The idea of hyperstates is merging states together, creating a layered state space, it is possible to de-clutter the model. This technique makes it possible to unfold only those parts of the model that one is interested in.

The tool has a wide range of possibilities to influence the test runs. For example with parameter selection, method restriction and state filtering.

It can be used for offline and on-line testing. Offline means that pregenerated tests are run to test the system. On-line or on-the-fly testing means that the test is generated dynamically as it is running. With online test generation the reproducibility might be lost, but because the test generation can take into account the actual responses of the system, timing and performance also influence the result.

The Spec Explorer has excellent features, but from a commercial point of view it is not applicable because it is only available for non-commercial purposes. There are commercially available model- based testing tools with similar functionalities. For example, Conformiq Qtronic [Huima(2007)] makes it possible to run offline and on-the-fly tests as well.

There are tools which address model based testing through a different approach. They use strictly typed domain specific language to specify the system under test. One of these tools is HOTTest [Sinha et al.(2006)]. Using this approach, assuming that all the domain-specific requirements are available and the model was created, it is possible to automatically generate a test oracle. Furthermore, it is possible to extract domain specific invariants to create additional test cases from the model.

Sinha claims in [Sinha et al.(2006)] that for database systems this technique was the most effective to capture domain-specific requirements either explicitly stated in the documentation/specification or implicit. Implicit requirements are more considered to the writer of the specification, and as a result they are not written down but still part of the model.

There is another group of model based testing tools, closely related to the ones using domain specific languages to specify the model of a system. These testing tools use the UML specification of the system as a model to test against. The difference is that the domain specific languages are, as the name suggests, designed for a specific problem, whereas UML is a general-purpose modeling language. Commercially these tools have wider acceptance, because usually there is already a UML model available to base the testing on. Both Conformiq Qtronic [Huima(2007)], Rhapsody TestConductor and TnT [Hartmann et al.(2000)] support testing against UML models. These tools usually provide a graphical interface allowing users to create the model of the system. With complex models however these can become complicated [Sinha et al.(2006)], because of their number of states. There are limitations in graphical systems of how many states can be handled effectively.

3.2 Automated unit testing

Even though unit testing is less time consuming and less complex than system testing, automating parts of the process can help improve quality.

JUnit [Riehle(2008)] and JTest together create a framework for Java which can automatically generate test cases for unit testing. There is functionality to add your own tests to the previously automatically generated ones. In C/C++/C# NTest provides similar functionality by providing automated testing of classes. What makes these tools highly valuable for the developer is that these tests can be run after every change in the code without major effort.

3.3 Integrated Frameworks

There are directions in the industry/research which aim to create a common framework that provides a well defined interface for different kinds of model based testing tools.

One of these is the AGEDIS [Hartman et al.(2004)] framework. The user has to specify three different things for the testing:

- the behavioral model of the system
- the test execution directives which describe the testing architecture of the system under test (SUT)
- the test generation directives which describe the strategies to be employed in testing the SUT.

Then the system provides three different API for model testing tools to hook into the framework. Using these interfaces the test generation and oracle checks can be specialised without the need of changing the user input.

4. Erlang Tools

In Erlang there are many tools and applications useful during the development and testing process. In this chapter we try to give a short overview with an introduction to the most commonly used tools. The usage and awareness of each tool can be seen in Figure 1.

4.1 Testing tools

Almost every developer uses some kind of test method, either with an existing test environment or with "home grown" functions and applications. From the ratio of the used testing tools it is clear that there is no freely accessible, widely used, flexible and stable tool. Most developers use proprietary tools or manually written test functions.

The available set provides different levels and methods for testing from unit testing to system testing. We will show how well are they adapted to the general requirements to keep the testing methods easy and quick for integration into the development cycle.

4.1.1 EUnit

EUnit [Carlsson and Rémond(2006)] is an open source light weight test server for Erlang. It tries to transfer the main ideas of the existing unit testing frameworks from other languages like SmallTalk (SUnit), Java (JUnit) [Riehle(2008)] applied to Erlang. The main goal is to provide a system where writing test cases is easy, the test cycle is fast and the results of the tests are presented tersely.

During unit testing independent parts of the program are tested separately. The units can be functions, modules, processes or even applications. EUnit allows the developer not only to test functions with assertions, but even to write test generator functions with the support of built-in macros. Test generators provide a representation of tests to be able to run with EUnit.

An important feature is the ability to disable the testing by defining special macros during the compilation or by adding them to the source file.

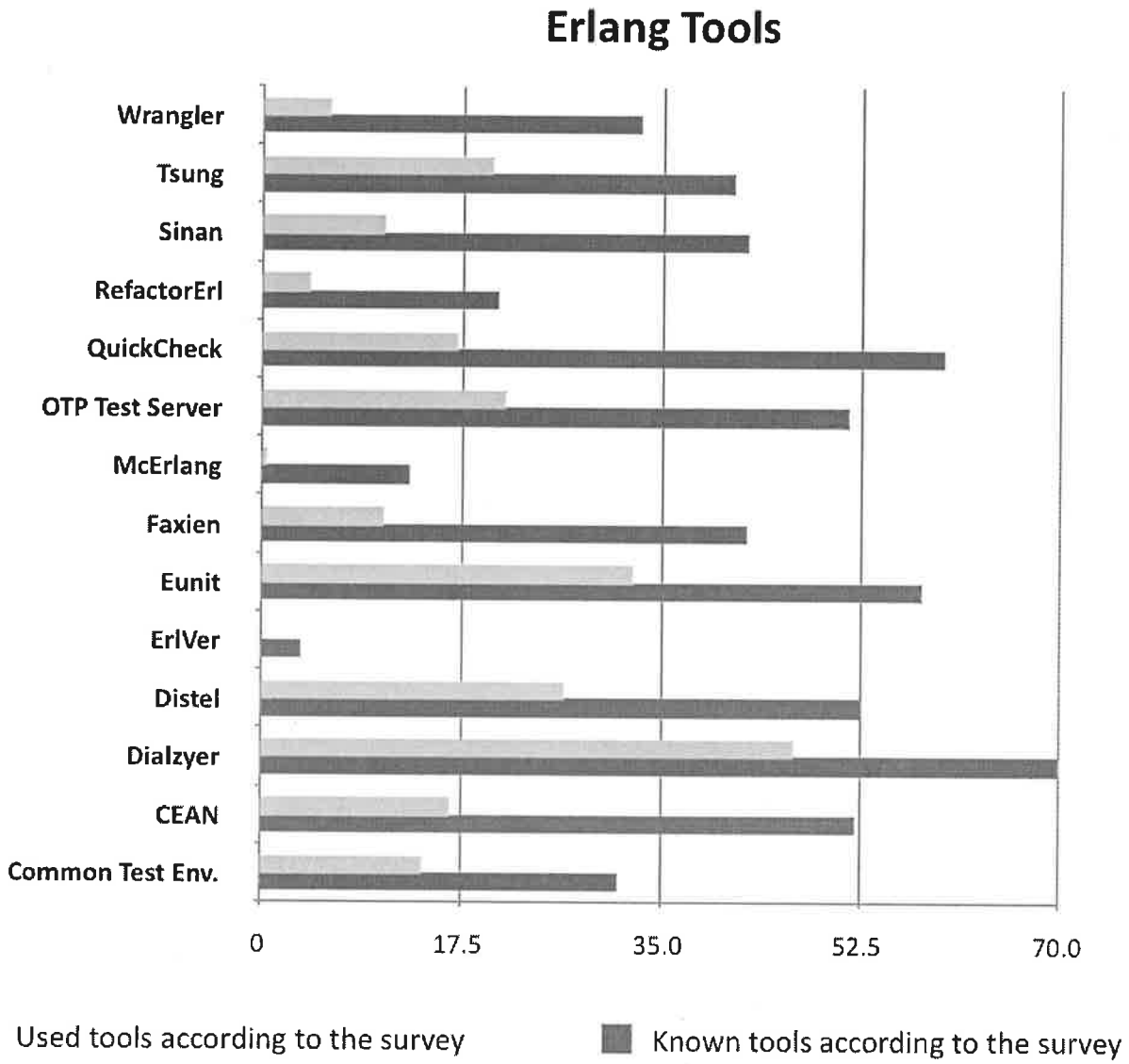


Figure 1. Popularity of Erlang tools among the community(%)

4.1.2 OTP Test Server and Common Test Environment

The OTP (Open Telecom Platform) developers designed a test suite execution environment based on Erlang to support regular automated testing. OTP Test Server [Wiger et al.(2002)] is a portable test server for application testing. The suites can be run on local or remote targets, the progress is logged and the result can be viewed in HTML pages.

Common Test [Blom and Jonsson(2003)] is a framework based on the OTP Test Server application. It is suitable for both black-box and white-box testing. Black-box testing can be done on any type of target system, not just ones implemented in Erlang, as long as the testing can be executed through standard Operation and Maintenance (O&M) interfaces. It provides code coverage analysis by integrating the OTP Cover Tool. According to the survey, the set up of the test environment is complex, suffers lack of documentation and its strict regulation about name conventions makes it hard to use and adopt.

4.2 QuickCheck

QuickCheck [Arts et al.(2006)] is a commercial tool for property and model based testing. It tests running code against formal specification. It can be used at different levels of testing from unit testing to system testing. The test case generation is random but controllable. One of the strongest points of the system is that it has a built in automated test case simplification, called shrinking, to support and facilitate the error.

The system is really efficient for testing and finding problems in complex systems at an early stage of development, especially for testing code against formal specifications. A model can be built according to the specification, and the system can be tested against separate aspects of the built model.

The users would like to have supervision with statistics on what has actually been tested. A dashboard reporting code coverage and executed test cases would rate it user friendly. Another aspect of

QuickCheck is that it has a steep learning curve, even for an Erlang developer.

4.3 Tools for supporting the development cycle

In this chapter we collected projects with different functionalities, which later can be used in an integrated test framework.

4.3.1 Refactoring tools

Refactoring means improving the quality of code through rewriting without changing its external behaviour. The most common program transformations in Erlang are:

- renaming variables, functions, records, modules
- converting data structures (tuples to records)
- function argument operations
- detecting and resolving duplicate code parts

The existing tools are already integrated into the most common editors (in Figure 2). They are based on static analysis of the code and therefore can safely work on big code basis. They can however not support all possibilities of the language; OTP specific behaviours and some of the dynamic function calls are under development, but will not be available for some time.

Wrangler Wrangler [Li and Thompson(2008)] supports both Emacs (with Distel) and Eclipse. The refactoring palette contains seven transformation and two search options.

RefactorErl RefactorErl [Lövei et al.(2007)] has six refactor steps. It stores the analysed source code in a database and provides an interface which can be a base for more applications. It's installation is easy as an installer is provided for Windows, while on Unix systems it is distributed as a source package.

4.3.2 Analysis and checking tools

Dialyzer [Lindahl and Sagonas(2004)] is a static analysis tool for detecting discrepancies in the source or byte code files automatically. Typical errors detected by the tool include

- obvious type errors
- redundant tests
- unsafe virtual machine byte codes
- unreachable code parts

McErlang [Fredlund and Svensson(2007)] is a model checker for verifying distributed Erlang programs. It has excellent support for the most difficult characteristic of the language, namely general process communication, node semantics, OTP component libraries, fault detection and tolerance through process linking.

4.3.3 Automated build tools and package management

Faxien [6] is a package management which helps to find and install or publish OTP applications.

Sinan [5] is a build system for OTP projects. It not only compiles and builds OTP applications but provides a framework for building the documentations, runs dialyzer, checks unit tests, produces output reports and handles all application dependencies.

4.3.4 Load tools

Tsung [4] is an open source multi-protocol distributed load and stress testing tool. Different kinds of servers can be stressed by using the loader, stimulating thousands of virtual users concurrently connecting from several client machines. It is used to test the scalability and performance of the TCP/IP based client-server applications.

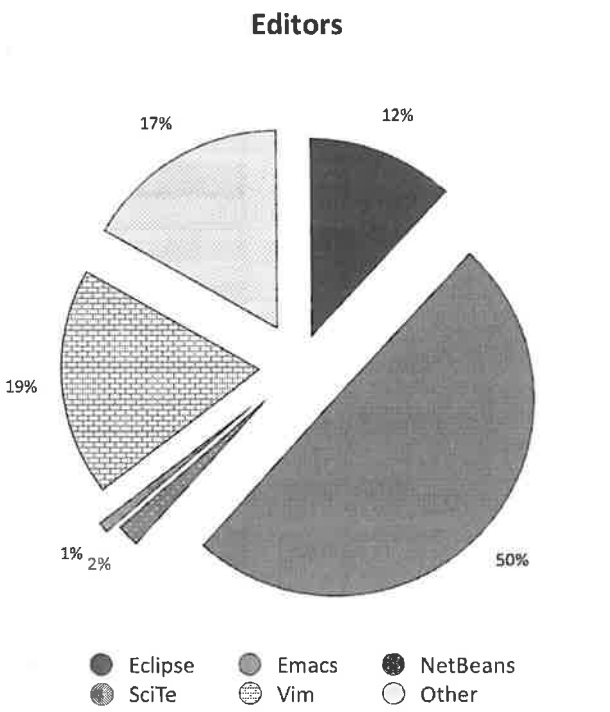


Figure 2. Editor usage among Erlang users.

4.4 Usability of the tools

One of the keys to the popularity of a tool or toolset is user friendliness. The interface should be easy to use, and if it is possible well integrated to the development environment. The layer of the integration can be on the following levels:

1. editors with plugins
2. OS commandline
3. developing software interface (Erlang shell)

It is important to design the interface(s) of the tool, according to the common development environments. Graphs showing the usage of operating systems and editors are in Figure 2 and 3.

According to the survey result the most common editors are platform independent. It is an important result, since the usage of operating systems is quite balanced between the different distributions of BSD, Linux, Macintosh and Windows. All tough Linux distributions are the majority.

The Erlang users' favourite editor is the Emacs with different plugins. The most widely used plugins in descending order are:

- Erlang mode in emacs: included in the Erlang distribution, supports syntax highlight, indentations, comments, function header commands, skeleton templates and compiling. The other Emacs plugins are extensions of this plugin.
- Distel connects to a live Erlang node. It is more than a simple extension of the erlang mode. It can handle dynamic tags, provides the advantages of shell options to the editor such as auto completion of module and function names, process and documentation viewer, a debugger and profiler frontend.
- Eflymake runs a syntax check in the background and highlights the erroneous codeparts

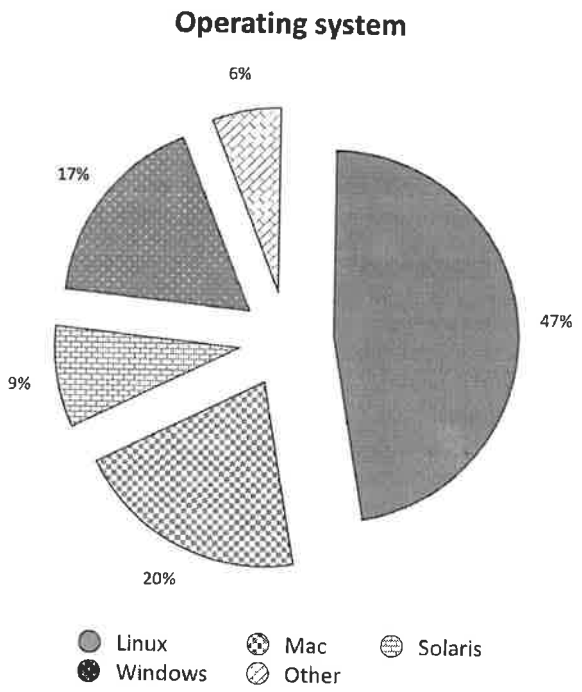


Figure 3. OS usage among Erlang users.

- Wrangler and RefactorErl (see in Section 4.3.1) are refactor plugins built in to the Erlang mode, they can be used with Distel
- Esense or ErlangSense provides features similar to IntelliSense or CodeSense in other editors: completes module names, exported functions, record names, record fields, macros, and shows popup documentation for the above elements.

Most developers use plugins, modes with their editors.

- Eclipse with Erlide is widely used, but some of the programmers found it heavy weighted when compared to Emacs and its Erlang mode. They found it hard to install, and not well tested.
- Vim has syntax highlighting, indentation, and many "home grown" scripts for different functionalities.
- Most popular editors after Emacs and Eclipse are operating system specific with different levels of Erlang support, mostly with syntax highlighting:
  - Textmate with Erlang Bundle
  - Notepad++
  - KDevelop
  - Kate
  - Gedit

There is a need for more complex development helping functionalities such as function argument pop ups, auto complete, function description notes, auto formatting, a good build environment and a debugger.

4.5 Open problems

We asked the survey participiants to name problems which make the tools cumbersome to use:

- "Documentation is not enough, and needs some examples"
- "bad documentation"
- "Generally most tools lack the documentation to use them properly without digging into the source code."
- "Generally a lack of tutorials, examples of usage and best pratices"
- "Haven't used them enough to comment but cursory opinion seems to indicate that test suites need better management facilities"
- "[any application] is not complete"
- "It is difficult to stub my other modules that are being tested"
- "Unstable and not complete"
- "Most of them are not very thought through. Most of them are badly layered are not extensible in other ways. Most of them organically grown quick hacks without structure"
- "Too much work to set them up"
- "Some of them need a lot of configurations to run, and you need to do it manually"
- "I tried to use dialyzer but it isn't easy to use"
- "It would be great if eclipse plug-ins worked so that I could get the same benefits I get in Java: function argument pop ups, auto complete, function description notes, auto formatting, good build environment and debugger"

The missing applications/functionalities are, identified by the submitters:

- "Specialized Traffic Generators"
- "Create and Interactive protocol tester"
- "It would be nice to be able to somehow generate, at least partially, mock modules to isolate modules under unit test."
- "Continuous integration and hooks for svn, cvs and git"
- "Improve [various testing applications]"
- "'A standard harness that runs: a compile check, EUnit tests, Common Tests, the Dializer'

According to these survey results, we would like to highlight the general shortcomings and weaknesses of almost every Erlang tool and project:

1. lack of documentation
2. missing examples and tutorials
3. not completed, tested when published
4. not well designed: badly layered, not extensible, no structure
5. doubts about sustainability
6. hard to install and use especially for non Erlang users
7. sometimes too much configuration is needed and has to be done manually

Even if there are useful tools widely used by the Erlang community, the survey reports that it is still missing functionality and integrated frameworks for:

- The ability to test efficiently on different levels, stubbing functions cause big problems and a lot of effort. At the moment there is no available stubbing tool. No tool can "generate at least partially mock modules to isolate modules under unit test".
- The missing aspect of the testing tools is a high quality display of the results from different interfaces with statistics and graphs.
- Even though there are available load testers for different protocols there is no freely available interactive protocol tester,

framework for testing web applications through http or specialized traffic generators.

- A missing area is continuous integration; hook towards version control systems (svn, cvs and git), integrated into a general framework.
- Existing frameworks do not contain enough functionality. There is a need for a standard harness that runs compile check, unit tests, system tests, xref, dialyzer and tsung performance tests.

5. Applicability of tools in developing commercial products

According to the survey results the testing phases of Erlang projects are poorly supported. Most of the developers use proprietary or manual solutions for unit and system level testing.

From a commercial point of view, the feature set of one tool is less important than other factors namely:

- ease of use
- support
- good documentation
- examples
- ease of comprehending the results

5.1 Ease of use

This covers the ease of initial configuration to non-cryptic error messages related to user interaction. Again the survey showed that the initial configuration is considered hard, manual and error prone work. A lot of test systems tend to return cryptic error messages. Most of these problems could be solved by integrating the tools into one framework, an Integrated Development Environment (IDE). Since this would cut down the configuration needed, the tools would work in the same environment providing similar behaviour.

5.2 Support

Problems with the toolset being used for testing can result in a major loss of development time. It is important to be able to resolve the issue efficiently as soon as possible. This was clearly visible from the survey, because 30% of users would pay for support and training.

5.3 Good documentation

If there is little documentation for a tool the possibility of commercial adoption is low. This was also evident in the survey. Missing or bad documentation was the top problem which came to light with the existing tools.

5.4 Examples

It helps the early adopters if there are available hands on exercises, use cases and examples. If the early impressions are positive the acceptance rate of the tools will be higher. From the survey it is clearly visible that there is a need for training courses.

5.5 Ease of comprehending the results

If it is really hard to interpret the results of a tool it will never pass the evaluation phase. From the survey it seems there is a need for online and offline solutions where more is better. The online means an immediate result as the testing progresses. Offline means a configurable report generation functionality, including browsable webpages, a dashboard or pdf reports.

6. Conclusion

Our research clearly indicates that the tools available for Erlang, commercially or otherwise, have shortcomings in many fields, although they address relevant issues.

One of these shortcomings, for most of the tools in the survey, is the lack of tutorials and examples alongside vague documentation. Even if these tools are useful, addressing existing problems should be a priority of the community. This is clearly visible from the fact that 30% of the survey participants are willing to pay for consultancy on some tools together with a wide adaptation of QuickCheck.

The other field that limits widespread use is lack of automated build and test of software. Even though there are tools solving this problem, they are not integrated with each other, and their output is very diverse. The survey shows that most people yearn after an IDE where building, testing and reporting the results are integrated in the overall workflow.

Although it has already been mentioned, the need for a reporting functionality cannot be emphasised enough. One of the most important functionality is the ability to report. The more structured information available, the better. For example in the model-based testing case (see in Section 4.2) it is really hard to get information about coverage and what has been actually tested.

Many of the mentioned test tools can be used in the development of non-Erlang projects. They are however, rarely used by companies who do not have a history of using Erlang. As Erlang gets wider acceptance and is used to test non-Erlang projects, it is important to tackle issues concerning user interaction, as they will affect the evaluation and adoption of Erlang.

A. Appendix: The questions of the survey

1. Which editor do you use for Erlang development?
   (a) Eclipse
   (b) Emacs
   (c) NetBeans
   (d) SciTe
   (e) Vim
   (f) Other
2. Do you use special Erlang plugin/mode with your editor (Erlide,ErlyBird,Distel,Emacs mode, etc.)?
3. On what platform is the development done?
   (a) Linux/Unix: Debian, Radhat, Suse, Ubuntu, Other
   (b) Macintosh: Tiger, Leopard, Other
   (c) Solaris
   (d) Windows: XP, ME, Vista, Other
   (e) Other
4. Have you ever heard of the following Erlang tools?
   (a) CommonTest Environment
   (b) CEAN
   (c) Dialyzer
   (d) Distel
   (e) ErlVer
   (f) Etomcrl
   (g) EUnit
   (h) Faxien
   (i) McErlang
   (j) OTP Test Server
   (k) Quickcheck
   (l) RefactorErl
   (m) Sinan
   (n) Tsung
   (o) Wrangler
   (p) None of the above
   (q) Other
5. Have you used the following Erlang tools?
   The option list is the same as in the previous question.
6. What Erlang open source applications have you used?
   (a) CouchDB
   (b) Ejabberd
   (c) Erlinda
   (d) ErlSDB
   (e) ErlSoap
   (f) Erlsom
   (g) Erlyweb
   (h) OSERL
   (i) Yaws
   (j) Mochiweb
   (k) Tsung
7. Would it help if we generate automated tests for these open source applications?
8. Do the tools you are using have any shortcomings which make them harder to use?
9. Are there any other tools you tried but decided not to proceed with? What were their shortcomings?(Unstable, Not completed, Not resolving my problem, etc).
10. Do you have automated builds and test suites in place? If so, what system are you using?
11. What methods are you using to test your system?
    (a) Test driven development
    (b) Develop-test iteration
    (c) Black box testing
    (d) Gray box testing
    (e) White box testing
12. What testing tools are you using?
    (a) Common Test
    (b) EUnit
    (c) Manually written functions
    (d) OTP Test Server
    (e) QuickCheck
    (f) Other
13. Would you be interested in trying out the test tools which will result from the protest project?
14. Do you have time and resources to try new tools?
15. Would you prefer training courses - with hands on exercises - for these new tools?
16. Would you consider short term consultancies?
17. Would you be willing to pay for these courses and consultancies?
18. Imagine that you have three wishes, what Erlang testing applications would you like to: create or improve already existing ones?
19. Thank you for filling out our survey! If we have managed to pique your interest please choose from the following options
    (a) Do you want a summary of this survey when completed?
    (b) Do you want to join the protest project announcement list?
    (c) Do you want us to contact you when the first tools are released?
20. Please give us your contact details
21. What is your role in the organisation?
    (a) Manager
    (b) Project manager
    (c) Software developer
    (d) Tester
    (e) Researcher
    (f) Other

References

[Arts et al.(2006)] 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. ISBN 1-59593-490-1. doi: http://doi.acm.org/10.1145/1159789.1159792.

[Barnett et al.(2003)] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-based testing with asml.net. In *1st European Conference on Model-Driven Software Engineering*, December 2003.

[Blom and Jonsson(2003)] Johan Blom and Bengt Jonsson. Automated test generation for industrial erlang applications. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, New York, NY, USA, 2003. ACM. ISBN 1-58113-772-9. doi: http://doi.acm.org/10.1145/940880.940882.

[Carlsson and Rémond(2006)] Richard Carlsson and Mickaël Rémond. Eunit: a lightweight unit testing framework for erlang. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. doi: http://doi.acm.org/10.1145/1159789.1159791.

[Fredlund and Svensson(2007)] Lars-AAke Fredlund and Hans Svensson. Mcerlang: a model checker for a distributed functional programming language. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 125–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: http://doi.acm.org/10.1145/1291151.1291171.

[Hartman et al.(2004)] A. Hartman and K. Nagin. The agedis tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1013886.1007529.

[Hartmann et al.(2000)] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. Uml-based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM. ISBN 1-58113-266-2. doi: http://doi.acm.org/10.1145/347324.348872.

[Huima(2007)] Antti Huima. Implementing conformiq qtronic. In *TestCom/FATES*, pages 1–12, 2007.

[Li and Thompson(2008)] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 199–203, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-977-7. doi: http://doi.acm.org/10.1145/1328408.1328437.

[Lindahl and Sagonas(2004)] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

[Lövei et al.(2007)] László Lövei, Zoltán Horváth, Tamás Kozsik, and Roland Király. Introducing records by refactoring. In *Proceedings of the 2007 SIGPLAN Erlang Workshop*, pages 18–28, Freiburg, Germany, Oct 2007.

[Riehle(2008)] Dirk Riehle. Junit 3.8 documented using collaborations. *SIGSOFT Softw. Eng. Notes*, 33(2):1–28, 2008. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1350802.1350812.

[Silva et al.(2008)] José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based user interface testing with spec explorer and concurtasktrees. *Electron. Notes Theor. Comput. Sci.*, 208:77–93, 2008.
ISSN 1571-0661.
doi: http://dx.doi.org/10.1016/j.entcs.2008.03.108.

[Sinha et al.(2006)] Avik Sinha and Carol Smidts. Hottest: A model-based test design technique for enhanced testing of domain-specific applications. *ACM Trans. Softw. Eng. Methodol.*, 15(3):242–278, 2006. ISSN 1049-331X.
doi: http://doi.acm.org/10.1145/1151695.1151697.

[Wiger et al.(2002)] Ulf Wiger, Gösta Ask, and Kent Boortz. World-class product certification using erlang. In *ERLANG '02: Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 24–33, New York, NY, USA, 2002. ACM. ISBN 1-58113-592-0. doi: http://doi.acm.org/10.1145/592849.592853.

[1] Trapexit http://www.trapexit.org Jun 2008

[2] Erlang http://www.erlang.org Jun 2008

[3] ProTest, property based testing http://www.protest-project.eu Jun 2008

[4] Tsung http://tsung.erlang-projects.org Jun 2008

[5] Sinan http://code.google.com/p/sinan Jun 2008

[6] Faxien http://code.google.com/p/faxien Jun 2008

A Comparative Evaluation of Imperative and Functional Implementations of the IMAP Protocol

Francesco Cesarini
Erlang Training and Consulting
416 Fruit & Wool Exchange
Brushfield Street, London E1 6EL,
England
francesco@erlang-consulting.com

Viviana Pappalardo
University of Catania
Dept. of Computer and Telecomm.
Engineering
Viale A. Doria, 6
95125 - Catania, Italy
viviana.pappalardo84@alice.it

Corrado Santoro
University of Catania
Dept. of Mathematics and Informatics
Viale A. Doria, 6
95125 - Catania, Italy
santoro@dmi.unict.it

Abstract

This paper describes a comparative analysis of several implementations of the IMAP4 client-side protocol, written in Erlang, C#, Java, Python and Ruby. The aim is basically to understand whether Erlang is able to fit the requirements of such a kind of applications, and also to study some parameters to evaluate the suitability of a language for the development of certain type of programs. We analysed five different libraries, comparing their characteristics through some software metrics: *number of source lines of code*, *memory consumption*, *performances (execution time)* and *functionality of primitives*. We describe pros and cons of each library and we conclude on the suitability of Erlang as a language for the implementation of protocol- and string-intensive TCP/IP-based applications.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms Algorithms, Performance, Design, Standardization, Languages.

Keywords Erlang, IMAP, Imperative Languages, Functional Languages, Comparative Evaluation

1. Introduction

The choice of programming language and platform to be used in a software system is an issue faced at the start of every project. In the majority of cases, this choice will be influenced by the skills of the programmers and the software development policies of the company. This is quite reasonable from the prospective of the decision makers, but it should not solely rest on these factors alone. This decision should be based on other aspects such as the evaluation of how a language and platform best fits the problem domain. However, even if a particular language could be very appropriate for an application, sensibly reducing the development time while increasing performance, the lack of programmer knowledge results in an exclusion, advantaging other choices better known to the developers.

When several choices of programming languages and platforms are available, debates over the advantages and drawbacks of one paradigm or language or platform with respect to different ones arise, compromising between an objective evaluations of features and personal taste. The issue is that it is not so easy to derive metrics that allow programmers to objectively understand the appropriateness of a language for the development of certain types of applications [13, 15]. When these metrics are available, applying them to a project–either not started or completed–is not an easy task. Best practices often consider an a-posteriori evaluation of an already developed application, aiming at understanding if certain parts of the system (or the whole), which have been particularly hard and time consuming to implement or do not perform could have be realized better using a different technology. Such an evaluation implies an analysis of (*i*) the effort that was needed to develop the application, (*ii*) the difficulties encountered in the development process (due to lack of language constructs or library functions), (*iii*) the complexity of the developed program which leads to a more error-prone application requiring more effort in debugging and testing, (*iv*) the performances of the whole system.

With these aspects in mind, this paper aims at providing a comparative evaluation of different implementations of client libraries for the IMAP email protocol [4]. The starting point was an Erlang project requiring the implementation of an IMAP client. In order to understand the appropriateness of Erlang in similar applications, we performed a series of tests comparing the Erlang solution with some other implementations written in different programming languages, ranging from compiled/imperative, such as Java and C#, to scripting/imperative ones such as Python and Ruby. The aim is to evaluate some parameters like *performances*, *capability to meet user requirements* and *effort* needed to develop the library using that language. This is achieved by both using metrics detailed in the following Sections of the paper and performing a critic analysis of the code and the characteristics of the provided primitives.

The paper is structured as follows. Section 2 provides an overview of related work. Section 3 gives a brief description of the IMAP protocol, showing the basic issues that have to be faced in the development of a client-side library. Section 4 illustrates the basic software architecture of an IMAP client library. Section 5 presents the metrics used to perform the analysis. Section 6 illustrates the implementations we analysed, highlighting their characteristics, and Section 7 summarises the results of the evaluation of them. Section 8 completes the paper with our conclusions.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

2. Related Work

Any experienced Erlang developer will confirm that Erlang programs they have written consist of four to ten times less code than their counterparts in C, C++ and Java. This has been an urban legend among the Erlang programmers at Ericsson long before Erlang was released as open source. Almost a decade after its release, however, there still is very little scientific evidence to back up these claims. Examples such as pivot sort using list comprehensions or a distributed remote procedure call server example are used to argue the case. Indeed, they might prove a point when compared to other languages, but when looking at Erlang, there is a necessity to benchmark whole systems, not code snippets.

Similar arguments should be made when looking at performance. As a result, existing language shootouts of sequential code, though valid, do not provide the whole picture. This is enforced by Bogdan Hausman's just in time C compiler [9], which was part of the early versions of the beam emulator, and through the compilation of Erlang to native code in the High Performance Erlang project [11]. In both research projects, a notable increase in the execution of sequential code was achieved. But when complete production systems were benchmarked, the results were more modest. Erlang systems consist of more than the sequential computation of Fibonacci or factorial sequences. They are complex, distributed massively concurrent systems.

In 1989, an Erlang prototype named ACS/Dunder implemented 10% of the features of the MD110 private automatic branch exchange (PABX). Its purpose was to validate Erlang as a language for programming the next generation of telecom applications, providing the productivity comparisons between Erlang and PLEX, the proprietary Ericsson language originally used to develop the MD110. Though the results of the ACS/Dunder project were never made public, Joe Armstrong in 2007 revealed that, depending on the implemented feature, an improvement in design efficiency of a factor three to twenty-two [3] was achieved. These figures were hotly debated at the time, and depending on whether you believed in Erlang or not, were considered highly controversial. As a result of this controversy, the final results were downgraded from an average of eight to three. Quoting Joe Armstrong, "The factor three was totally arbitrary, chosen to be sufficiently high to be impressive and sufficiently low to be believable".

Ulf Wiger, in his 2001 paper "Four-Fold Increase In Productivity and Quality" [18] states that comparisons of projects within Ericsson using C, C++, Erlang, Java and PLEX project yield a similar line per hour productivity and a similar bug density per line of code. What differed between the projects was the final source code volume. Comparisons of C++ projects which were rewritten in Erlang resulted in four to ten times less code, concluding that using Erlang equated to a four to ten-fold increase in productivity and quality. Wiger, however, states that though these numbers would not hold up to a scientific scrutiny, they provide a consistent pattern with experiences of non-Ericsson projects.

The first study to provide scientific evidence and back up the findings by the ACS/Dunder project and Wiger was the comparison of C++ and Erlang for Telecoms Software. Run as a collaboration between Herriot Watt University and Motorola Labs, the research project consisted in refactoring two C++ applications which were in production at Motorola to Erlang [12]. Comparisons were made on the Performance, Robustness, Productivity and impact on programming language constructs. The conclusions were a 70 – 85% reduction in code for the Erlang based system. The code reduction was explained by the fact that 27% of the C++ code consisted of defensive programming, 11% of memory management and 23% of high level communication, all features which in Erlang are part of the semantics of the language or implemented in the OTP libraries. One of the applications completely rewritten in Erlang resulted in a 300% increase in throughput, but that can be argued was a result of Erlang and its light weight concurrency model being the right tool for the job. The Dispatch application in question had lots of concurrency, short messages and little in terms of heavy processing and number crunching. The C++ version had been written with resilience in mind, not performance. Resilience comes almost for free in Erlang. Other conclusions from the project were that robustness and scalability were higher in Erlang while maintenance costs were lower.

3. Overview of the IMAP Protocol

IMAP stands for Internet Message Access Protocol. It allows client programs to access and handle electronic mails stored on remote mail servers. It was proposed by Mark Crispin as an alternative to the POP3 protocol. The current version, IMAP4 revision 1 (IMAP4rev1), is defined in RFC3501 [4]. It supports communication either over a TCP/IP connection using port 143, or over a SSL connection using port 993. The most important advantage of IMAP4 over POP3 is that emails can be stored on the server, allowing any client to access them from any location.

An IMAP4rev1 session is based on reliable client/server network connection over which a request-reply model is run. Client/server interactions start with an initial greeting message from the server and consist of a client request, followed by optional data sent by the server[1] and terminated by a result response. Client and server transmit strings terminated by CR+LF character sequence. Each request sent by a client consists of a *Tag*, a unique alphanumeric prefix for each message used to match client request to server reply, and a *Command*. If the command from the client does not require the server to send additional data[2], the server replies immediately with a *tagged* response message, including an indication of the outcome of the interaction (e.g. success or failure). An example of a client/server interaction (a LOGIN command) showing the tagged response is reported below:

```
Client: A001 LOGIN username password
Server: A001 OK LOGIN Ok
```

When data needs to be sent before the response message, the server reply can be split into one or more lines according to the size of the messages themselves. These responses are called *untagged*, they are prefixed not by the *tag* but with characters '*' or '+'. Below is an example of untagged responses returned as the result of the SELECT command:

```
Client: A002 SELECT INBOX
Server: * FLAGS (Junk NonJunk ...
Server: * OK [PERMANENTFLAGS (Junk NonJunk ...
Server: * 4270 EXISTS
Server: * 0 RECENT
Server: * OK [UIDVALIDITY 1186814135] Ok
Server: * OK [MYRIGHTS "acdilrsw"] ACL
Server: A002 OK [READ-WRITE] Ok
```

A client/server session is represented by the finite-state machine depicted in Figure 1. Most commands are available only in certain states, so if a command is sent by the client when in an inappropriate state, an error message is returned.

The initial state, "*server greeting*", is reached after connection. In this state, the server sends a message containing a welcome text and, in some cases, the server's capabilities.

[1] It depends on the command to be executed.

[2] This happens, for example, in LOGIN and LOGOUT commands.

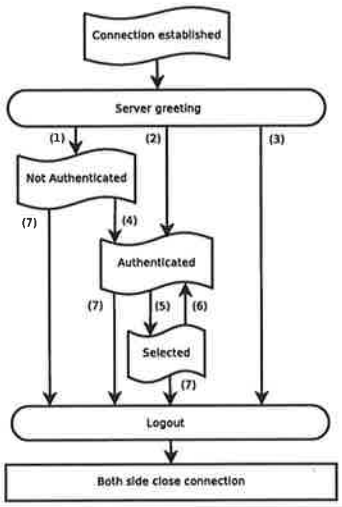


Figure 1. IMAP4rev1 State Diagram

Following this message, the client enters the *Not Authenticated* state (transition 1) unless the connection is pre-authenticated.[3]. In the *Not Authenticated* state, most commands are rejected, because the client has to supply its credentials in order to start its session. This is performed by a proper LOGIN command, if successful (transition 4), the client reaches the *Authenticated* state and must use the SELECT command to choose which mailbox to access. This command contains the name of the mailbox to be manipulated, allowing the client to enter the *selected* state (transition 5). When activities are over, the LOGOUT command terminates the session (transition 7) and closes the connection. In general the *logout* state is entered following the LOGOUT command from the client; however, if a protocol error is detected during the session or a session timeout elapses, the server is allowed to unilaterally trigger the logout and close the connection.

A basic IMAP4 session is characterized by the initial sequence of commands LOGIN and SELECT, followed by commands for manipulating the messages and the mailboxes such as (but not limited to) FETCH, LIST, STORE, COPY and finally terminated by a LOGOUT. FETCH is the important command of the IMAP4 protocol, as it is used to retrieve data items such as header fields, text or attachments of one or more messages. The basic syntax requires the inclusion of additional parameters, expressed by means of proper keywords and used to select specific portions of the message(s). As instance, keyword BODYSTRUCTURE (or simply BODY) allows a client to get information on the structure of a message and its various components. Possible component information might cover attachments, text, their size, encoding character set and the MIME type, to mention but a few. The keyword BODY[*section*] retrieves one or more of the specified sections of the whole body; sections can be HEADER, HEADER.FIELDS, MIME, TEXT, etc.; keyword ENVELOPE returns the structure of the envelope of an email message containing the fields date, subject, from, sender, reply-to, to, cc, bcc, in-reply-to, and message-id.

Together with the data to be retrieved, the FETCH command requires an identifier of the involved message(s). To this aim, IMAP4rev1 defines two type of numeric message identifiers: the *message number* (or *message sequence number*), reflects the position of the message in the given mailbox, while the *unique identifier (UID)* is a number assigned to the message when it is placed in the mailbox (delivered from a sender or as a result of the COPY command). There is a basic difference between these identifiers: while UID is *unique* and its value does not change during the life span of the message, the sequence number can vary by modifying the position of the message in the mailbox; this situation can happen when messages are deleted or inserted.

The response to a FETCH command contains data organized as LISP-like nested lists and enclosed between round parentheses, "(" and ")". At a first glance, this response has a simple structure, but it can get complex when nesting occurs as a result of the message forwarding or the inclusion of an RFC822 form [17]. The structure of a message contains a header, a body section and one or more attachments, unless it is a MIME-IMB [6] message consisting of several sections.

The following example[4] of the FETCH BODYSTRUCTURE is the typical nested structure of the FETCH reply message, where each level of nesting is indicated by a pair of parentheses. The structure of the message reported in this reply is composed of two parts; the text of the mail, written in both plain ASCII and HTML, and an attached binary file named "es.rar", encoded in BASE64. The "boundary" element is a delimiter between the text and the attachment.

```
* 3421 FETCH (BODYSTRUCTURE
 ((
   ("TEXT" "PLAIN" ("charset" "iso-8859-1")
    NIL NIL "quoted-printable" 22 2)
   ("TEXT" "HTML" ("charset" "iso-8859-1")
    NIL NIL "quoted-printable" 393 19)
   "ALTERNATIVE"
   ("boundary"
    "- - _=_NextPart_002_01C72136.F960F3A9")
  )
  ("APPLICATION" "OCTET-STREAM" ("name" "es.rar")
   NIL "es.rar" "base64" 261814 NIL
   ("attachment" ("filename" "es.rar"))
  ) "MIXED"
 ("boundary"
  "- -_=_NextPart_001_01C72136.F960F3A9")
 )
)
```

When the reply contains the headers or the text of an email, complexity increases as the nested structure can encapsulate bare, unformatted ASCII or binary data such as email headers, text, attachments, and other included emails. In this case, the presence of the octets stream is indicated by enclosing its size in curly brackets.

Another commonly used IMAP command is LIST, allowing a client to look into a mailbox hierarchy displaying all folders, relationships and attributes. The syntax of the command requires the *reference* and *mailbox_name* parameters. As mailboxes are hierarchically organized, the *reference* denotes where in the hierarchy the inspection should be started. The *mailbox_name* denotes which mailbox to search in. Its notation allows wild cards, denoted by the special characters '*' or '%'. The reply to the LIST command returns a set of untagged responses, one for each folder of the hierarchy. They contain the folder name, its path, the hierarchy delimiter and folder attributes[5].

[3] Pre-authenticated conditions (transition 2) are indicated by a proper capability in the greeting message. Pre-authentication is in general performed by a check on the client's IP address or other peculiar mechanisms.

[4] Line breaks and indentations have been added by authors in order to make the example clearer; real server's reply is not formatted as shown, even if it is contained in several lines.

[5] According to [4], attributes are chosen among the following values: `\Noinferiors`, `\Noselect`, `\Marked`, `\Unmarked`, `\HasNoChildren`, `\HasChildren`.

3.1 Implementation Issues

As the reader can observe from the explanation of the IMAP protocol, the client/server interactions and the data transport are straight forward to implement, as data exchange occurs by encoding messages in ASCII strings terminated by the CRLF character and sending/receiving them through a socket. The problem with IMAP is in the application layer, as the grammar of the server response is very articulated and complex, putting a requirement on the client to be able to parse and interpret all replies. As a result, most of the implementation effort of an IMAP client is in the parsing of the IMAP server responses. In particular, as has been illustrated above, parsing the FETCH response can be challenging because of the variety of information included in the structure of the message and included attachments. The response to a "FETCH BODYSTRUCTURE" command is one of the most complex to parse, as it can contain many levels of nesting. The response to a "FETCH BODY[section]" command can also present some difficulties in the transcoding activity, as the client has to translate text or attachments contents from the reply encoding (such as BASE64) to an encoding handled by a client program (such as UTF-8 or binary). Finally, as the grammar described in standards [4, 6] also permits user-defined fields or attribute values, another implementation difficulty is making the parser module generic and flexible so as to cater for these situations.

4. Architecture of an IMAP Client Library

This Section gives a brief overview of the software architecture of an IMAP client. According to the specifications provided in the standard [4], we can consider an IMAP client library as composed of the following software layers:

1. **Communication layer**, handling socket connectivity.
2. **Low-level IMAP protocol handler** which is responsible to manage request/reply communication properly adding the *tag* to the request and differentiating *tagged* and *untagged* responses.
3. **IMAP interpreter**, whose task is to parse server replies, transforming them into native types of the target language used in the client implementation.
4. **IMAP FSM**, which handles the finite-state machine of the IMAP protocol (see Figure 1), managing state transitions and ensuring that sent commands are valid in the current state.
5. **IMAP high-level interface**, which implements the various commands of the IMAP protocol by directly using the services provided by the low-level IMAP protocol handler and the IMAP interpreter.

This architecture depicted in Figure 2 is generic and consistent with all of the client implementations we analyzed. According to the specific functionality of the library, however, one or more of the layers could have been omitted depending on the functionality of the client. As an example, a library which does not parse server replies, returning them directly to the user will not have an *IMAP Interpreter* layer; similarly, if the handling of the FSM protocol is not support by the client side, the *IMAP FSM* layer will be not present.

As will be argued in Section 7.1, the presence (and lack) of layers will impact the software writing process of the IMAP client and influence the estimation of the effort involved. Indeed, we can consider that the lower layers such as the *Communication*, *Low-level IMAP protocol handler*, *IMAP interpreter* and *IMAP FSM* (when present) have to be developed first, requiring a big initial effort, while implementing the various commands can be considered quite simple; they can be done incrementally, one at a time, directly exploiting the services provided by the lower layers.

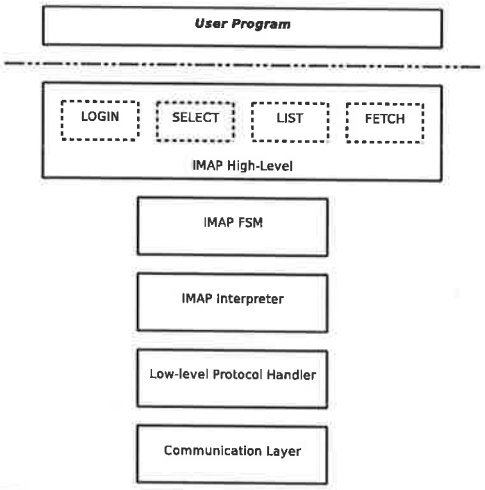


Figure 2. Software Architecture of an IMAP Client Library

```
...
def fetch(set, attr)
  return fetch_internal("FETCH", set, attr)
end

def fetch_internal(cmd, set, attr)
  if attr.instance_of?(String)
    attr = RawData.new(attr)
  end
  synchronize do
    @responses.delete("FETCH")
    send_command(cmd, MessageSet.new(set), attr)
    return @responses.delete("FETCH")
  end
end

def store(set, attr, flags)
  return store_internal("STORE", set, attr, flags)
end

def store_internal(cmd, set, attr, flags)
  if attr.instance_of?(String)
     attr = RawData.new(attr)
  end
  synchronize do
    @responses.delete("FETCH")
    send_command(cmd, MessageSet.new(set), attr, flags)
    return @responses.delete("FETCH")
  end
end
...
```

Figure 3. Code Snippet of the Ruby IMAP Library

The listing in Figure 3, which is the code for the FETCH and STORE commands of the Ruby IMAP library clarifies this aspect. Adding another command involves replicating the basic structure of the code and replacing the command string to handle the new functionality.

5. Evaluation Criteria

As argued in Section 1, providing objective parameters to measure software quality is not an easy task. The literature reports some indicators [5, 12] and, even if some developers and researchers still do not agree on their objectiveness, they should be considered adequate and provide a good degree of confidence [15]. According to this literature, we selected the following parameters:

- *Number of source lines of code* (SLOC);
- *Functionality of primitives*
- *Amount of memory required*
- *Execution time/throughput*

5.1 Source Lines of Code

This is obtained by counting the numbers of lines of all the source files composing the library and removing blank lines and comments. This parameter should provide an indication of the *effort* required to develop the application: roughly speaking, the more the SLOC the more the time required to write the software and to test it. However, as argued in [10], SLOC is dependent of at least three factors: *programmer's skills*, *programming language* and *program functionality*.

Indeed, it is quite simple to understand that sources written by different programmers, with different programming abilities, could sensibly differ in the number of SLOC; in general, if all programs exhibit the same functionalities, the more skilled the programmer the less the number of lines written. In any case, given that most of the implementations compared in this paper are part of the library of the language we tested, we can assume that they have been written by programmers with a very good knowledge of the language itself and therefore they can be considered "good software"; so, as for this point of view, it makes sense to compare the number of SLOC of these implementations.

Another factor that affects the number of SLOC is the programming language employed. Obviously, the presence of certain language constructs has a strong impact on the lines of program used for a certain functionality. For example, extracting the first element of a string is obtained in Erlang by means of a simple one-line statement: `[H|T] = String.` Moreover, if this extraction is performed in the declaration of a function clause, as is often the case, the code becomes even more compact, as the line will contain the function declaration, check a condition (that the string is not empty) and perform a double assignment. In Python or Java, the extraction of the first element requires two lines of code, and additional lines when we intend to also include the "string not empty" check:

Python	Java
`H = String[0]` `T = String[1:]`	`H = string.charAt(0);` `T = string.substring(1);`

`if String != "" :`
`  H = String[0]`
`  T = String[1:]`
	`if (string.length() != 0) {`
	`  H = string.charAt(0);`
	`  T = string.substring(1);`
	`}`

This dependence of SLOC upon the language used is not only fundamental for our study, but also quite desirable as we want to understand the effort needed to develop the IMAP client using different languages. The presence of certain constructs that require less lines of code implies less time needed to write the software and to test it.

Another important aspect that influences the number of SLOC is the functionality the software exhibits; indeed two different implementations of an application which at first glance might appear the same but in reality differ in functionality, cannot be compared in terms of SLOC, as the feature rich implementation will presumably have a higher number of SLOC. This is the reason why, in evaluating the software, the SLOC parameter is weighted by means of the so-called *Function Point Analysis* (FPA) [8]; this is a technique that allows developers to analyze a software implementation and derive a parameter, called *number of function points*, providing an objective evaluation of the functionalities of the analyzed software. Given this measure, weighting the number of SLOC with respect to the number of function points should give a more precise estimate of the required development effort required.

However, performing FPA is not an easy task, above all when dealing with a library. According to [7], FPA seems more appropriate for a complete application or software system rather than a single component. For this reason, in the analysis of SLOC, we used a different approach with the aim of obtaining the same objective evaluation. Indeed, as argued in Section 4, by analyzing the implementations given with their source code (i.e. Python, Ruby, C# and Erlang), we noted their software architectures were very similar to one another. In particular, once the blocks for the low-level protocol, request assembly and reply parsing had been developed, adding a new functionality (i.e. support for another IMAP command) only involved writing a function (or method) that used the primitives provided by these basic blocks. From such a characteristic, we can derive that comparing the number of SLOC of two implementations, even if they support different sets of commands, can be performed by simply analyzing the parts of the software which provide the *same* functionalities, namely the low-level protocol, request assembly, reply parsing and the common commands in the different implementations.

5.2 Functionality of primitives

As argued before, Function Point Analysis is not easy to perform and also not appropriate for a library. On the other hand, in a comparative analysis as the one described in this paper, an understanding of the specific functionalities of a given library should be mandatory. The aim should be to try to understand the *quality* of an implementation in terms of its capability to provide a complete, transparent and flexible support of the IMAP protocol.

Checking the completeness of a solution is quite simple, since it implies to analyse if the library is able to support all or a subset of the commands and functions of the IMAP4 standard. As for transparency and flexibility, an analysis of the software architecture of the solution and the interface of the various primitives (i.e. signatures of function or methods) is needed, in order to understand if a good level of abstraction is provided. Indeed, many commands of the IMAP4 standard require additional parameters which should be passed to the function (or method) implementing that functionality. Even if they are then sent as string in the protocol messages, these parameters may vary in type and semantics, therefore a well-written library should treat them according to their real meaning. As an example, a function implementing the FETCH command could require the additional parameters to be passed as either different arguments or a complete string. In the latter case, there is a lack of a proper abstraction level, since the programmer has to manually create the string to be passed: in other words, to use the library the programmer has to possess a certain knowledge of protocol messages, an aspect which is in contrast to the common rules of software engineering that instead require library/software modules to *hide* specific low-level (protocol) details by providing a high-level flexible and uniform interface.

A similar argumentation can be given for function/method response values, which should reflect the outcome and the reply of the command implemented. If a reply is already interpreted by the library and provided as a structured data type of the language (primitive or derived), the programmer can directly use it without writing additional parsing code. Once again, this is an indicator of a proper encapsulation and abstraction level of the library, characteristics not featured by e.g. an implementation returning raw protocol replies, since, in this case, an additional programming effort is required to properly understand and use them.

5.3 Memory Consumption

Memory consumption is a performance parameter that expresses the memory usage of an application using the IMAP library belonging to a certain language. We developed a simple test program that logs in and fetches a bunch of messages. During the execution of the test program on a Linux OS, we collected information about memory usage by checking the status contained in the "`/proc`" file system, in particular, by looking at *total program size*, *code size* and *data size*.

5.4 Execution time/throughput

The second performance parameter we evaluated was the execution time of certain IMAP commands. As reported in [4], most of the commands imply the execution of a specific activity and reply with a simple success/fail response. Replies from commands such as SELECT, FETCH or LIST, however, are more articulated and probably require complex parsing of the result. For this reason, in order to evaluate the interaction throughput featured by each specific command of the various implementations, we let the client perform both simple and complex commands. Simple commands provide an indication of the performances of the low-level request/reply protocol, while complex commands can be used to measure the performances of the parser when it is present in the tested implementation. In detail, the commands we used for performance evaluation are LOGIN, SELECT, FETCH and LIST.

6. Evaluated Solutions

In this Section the IMAP libraries which constitute part of this study will be described. All of them have been found in the Internet. Some of them are free while others are commercial products. Furthermore, among the free libraries, some are provided with the source code while other are available only in compiled format, so analysis based on SLOC, described in Section 5.1 will not be applicable. The aim of all of the solutions studied is to provide IMAP4 client support to developers. Each library provides only some of the features described in the IMAP standard [4], but they all include the most important commands. These libraries represents different approaches to the solution of the IMAP4 interfacing problem, therefore, to analyze and study their characteristics provides their respective advantages and disadvantages. Each library supports developers in different ways, with varying functionality, different parameters and return values. The following subsections will provide an in-depth description of the characteristic of each of analyzed solutions.

6.1 Java

Even if there are many IMAP solutions for Java, we selected the JavaMail package [2], as it is the library officially released by Sun Microsystems. This package provides a platform-independent and multiprotocol framework which can be used to implement email and messaging applications.

The philosophy behind JavaMail is interesting, as it allows client programs to download message(s) from the remote IMAP server and cache them locally, bringing it closer to POP3 than the other IMAP4 approaches. This package is multiprotocol, in the sense that it allows a transparent access to both IMAP and POP3 servers. A client program using this library must principally, create a `Session` object, useful to retrieve the proper `Store`; the latter determines the correct connection mechanism between client and server depending on the specific protocol required (IMAP, POP3, SMTP). Indeed the `Store` object models a message store and its access protocol. The `Store` object has a `connect` method to establish the connection to the server, and by means of the IMAP "LOGIN" command, supports a simple authentication mechanism.

Once successfully authenticated, the client can access a mailbox by obtaining a `Folder` object from the `Store`. By means of the `Folder` object, the client can open a mailbox in read/write or read-only mode and retrieve data invoking the `getMessages` method. This method returns a vector of `Message` objects which model an email message and provide a set of "`getXXX`" methods[6] to retrieve message data, such as text, flags, sender's and recipient's addresses, etc. In detail, from the point of view of the socket streaming, each time a "`getXXX`" method is invoked, a proper interaction with the server is started to obtain the desired data item; such an information is then cached in the `Message` object in order to make it readily available if it is further needed. Such a data retrieving policy could be considered a drawback, as for performances, since it causes a new protocol interaction with server each time a not yet retrieved message attribute is needed. Indeed, this drawback can be overcome by using a "fetch profile", i.e. the programmer can personalize data recovering by preparing a `FetchProfile` object and then use the `Folder.fetch` method; the latter retrieves, using a single transaction for each message, all data items requested, making them available without needing any further interaction with the server.

The following code is a brief example of what a Java program using the JavaMail library program has to do to retrieve email messages.

```
import javax.mail.*;
...
// Get a Session object
Session session =
    Session.getInstance(properties, null);
// Get a Store object
Store store = session.getStore(protocol);
store.connect(host, port, user, password);
// Open the Folder
Folder folder = store.getDefaultFolder();
folder = folder.getFolder(mailbox_name);
// try to open the folder in read/write mode
folder.open(Folder.READ_WRITE);
Message [] msgs = folder.getMessages();
// Create a fetch profile
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
fp.add(FetchProfile.Item.FLAGS);
// fetch messages using the profile
folder.fetch(msgs, fp);
...
```

Even if the policy of JavaMail is not the proper way to fetch messages with respect to IMAP4 protocol, its performances are good enough as reported in Section 7.

6.2 C#

The C# solution we analyzed is the "ImapLibrary", available at `http://www.codeproject.com/KB/IP/imaplibrary.aspx` and provided with the source code. This library exploits the dotNet platform, using the framework's native packages such as the socket library and the XML data handling packages. This solution has peculiar return values in some object's methods; in some cases, the output of the parsing activity is identified by an XML file, where each node of the XML tree represents a portion or a section of an email message. The library contains three main source files:

- *ImapBase.cs*; it defines the base class to handle low-level client/server communication, data transmission and reception to/from the socket channel.
- *Imap.cs*; it implements the IMAP4 protocol; it is in charge of preparing client queries, sending the equivalent IMAP4 command and parsing server's response. The `Imap` class defined in this source file is derived from the `ImapBase` class.
- *ImapException.cs*; it defines some exceptions to manage internal errors and protocol faults.

The basic entity of the library is the `Imap` class, which defines a set of methods, each executing a specific command of the IMAP

[6] For example `getFrom`, `getFlags`, `getAllRecipients`, `getReplyTo`, etc.

protocol. In detail, the `Login` method allows a client to perform both a connection with the server and user authentication. This method verifies if the server is connected and if the client is already logged in. If so, it does not send another login command and choosing the correct policy to manage this condition. After a login, the client is requested to select a mailbox through the `SelectFolder` method, requiring a string as a mailbox name. Subsequently, to obtain message data, the client can use the following three methods:

- `FetchPartHeader`. It retrieves the header of the email message or an encapsulated part (i.e. an attachment) and, in particular, implements the IMAP commands "FETCH BODY [HEADER]" and "FETCH BODY [section.MIME]". It requires the UID of the message to be retrieved, the section, and returns (by reference) an `ArrayList` object; each element of this object is a string containing one of the lines of the header of the retrieved message.
- `FetchPartBody`. It retrieves the body (i.e. text or attachment data) of a part of a message by using the IMAP command "BODY[section]". The programmer has to specify the UID of the message and the section number; the return value is a string which contains the requested body data.
- `FetchMessage`. This method retrieves all the parts of a message and produces, as output, an XML file, properly formatted, contained all message data. The method requires the UID of the message and an `XmlWriter` object to be used for output writing.

The code below illustrates a snippet of a simple client program that uses the Imap library.

```
...
Imap oImap = new Imap();
oImap.Login(host, user, password);
oImap.SelectFolder(mailbox_name);
...
XmlTextWriter oXmlWriter =
              new XmlTextWriter(sFileName,
              System.Text.Encoding.UTF8);
oXmlWriter.Formatting = Formatting.Indented;
oXmlWriter.WriteStartDocument(true);
oXmlWriter.WriteStartElement("Message");
oXmlWriter.WriteAttributeString("UID", sUid);
oImap.FetchMessage(1234, oXmlWriter, true);
oXmlWriter.WriteEndElement();
oXmlWriter.WriteEndDocument();
...
```

As the description and the code above highlight, the library performs a partial parsing of the reply as for the methods `FetchPartHeader` and `FetchPartBody`, and a full parsing in the method `FetchMessage`, but, in the latter method, the output is not directly usable by a client program: instead, the programmer is requested to re-interpreter the XML file in order to obtain the information s/he needs.

6.3 Python

Python [16] offers an IMAP client module delivered with its standard library called "imaplib". As with all modules of the Python library, imaplib is free and provided with its source code.

The basic component of the library is the `IMAP4` class, which defines methods for all the commands of the IMAP standard, providing a 1:1 mapping. All of the library methods have similar interfaces; all of them require a string which represents the parameter(s) for the command to be sent to the server, therefore the library creates the IMAP command by simply appending the string received as argument and adding the *Tag*. Similarly, the reply is returned by the method "as is", without any parsing, which if required, must be done by the client. Indeed, the return value of all methods is a tuple with two elements: the first element is the *tagged* response, representing the outcome of the command, while the second element is the *untagged* response containing the data sent back by the server.

An interesting capability of this library is the authentication mechanism, which is provided through the "AUTHENTICATE" command. It does not use clear text, relying on a more flexible and secure authentication model described in [14].

A python client program which wants to connect to an IMAP4 server must create an `IMAP4` object and use it to invoke the `login` or `authenticate` method. Once the connection is established, the client can select the folder to examine and call the `fetch` method to pick up message(s) and associated data. This method requires two parameters: the first parameter is the message sequence number, expressed as an integer, or a range of messages, provided as a string in the form ``first message number:last message number''; the second parameter is a string representing the complete argument to be appended to the "FETCH" command. The code below shows a basic client program in python using the imaplib:

```
import imaplib
...
imap = imaplib.IMAP4(host)
result, data = imap.login(username, password)
# The default mailbox is 'inbox'
result, data = imap.select()
# Let's get the text and the UID of the
# 10th message
result, msg_data =
  imap.fetch(10, '(UID BODY[TEXT])')
# Let's get the 'From' and 'Subject' fields of
# the messages from 1 to 5
result, msg_data =
  imap.fetch('1:5',
         '(body[header.fields (from subject)])')
```

The library provides additional features since offers other commands, such as APPEND, CREATE, DELETE, to modify the selected mailbox, or UID, FLAGS, LIST, to retrieve information about message or mailbox characteristics.

6.4 Ruby

Ruby [1], like python, is an interpreted (scripting) language. Its basic library offers an IMAP module called `net/imap` which comes as part of the language distribution. This library is free and the code is accessible to the developers.

The library provides a class `Net::IMAP` whose interface is similar to the module provided for the Python language. However, unlike the IMAP Python implementation, the Ruby library performs a parsing activity and returns the server response to the client program by means of proper language types which can be directly used. The code below illustrates a simple ruby client program:

```
require "net/imap.rb"
...
imap = Net::IMAP.new(host)
imap.login(username, password)
imap.select(mailbox_name)
info = imap.fetch(20,
       "BODY[HEADER.FIELDS (FROM SUBJECT)]")
message = imap.fetch(10..15, "BODY[HEADER]")
```

As the example illustrates, the client program must create a `Net::IMAP` object, specifying the server address as parameter, and then invoke the "login" or "authenticate" method. Subsequently, the client can select a mailbox (`select` method) and manipulate it

through the proper commands defined in the library. As for message retrieving, the `fetch` method needs two parameters resembling the fetch primitive of the python implementation, even if the reply is properly parsed. Indeed, the `fetch` method returns server's reply as a native Ruby type, i.e. a structured type composed of two fields: `seqno` and `attr`, the former is the message sequence number while the latter is a hash table in which each item is a couple {*key, value*}; here the *key* is one of the arguments specified with the "FETCH" command (e.g. "FLAGS", "BODYSTRUCTURE", "UID", etc.) and *value* is the associated data. This data is generally returned as is, without any parsing, unless one of the following arguments are passed: "BODYSTRUCTURE", "BODY[TEXT]", "BODY[section]", "ENVELOPE". In such cases, the *value* field may be one of the following defined types:

- `Net::IMAP::BodyTypeBasic`, which represents the structure of the body of a simple message;
- `Net::IMAP::BodyTypeMultipart`, which represents a message composed of more than one part, that is a text and one or more attachments;
- `Net::IMAP::BodyTypeText`, that contains the text of a message;
- `Net::IMAP::BodyTypeMessage`, which represents a message of the type MESSAGE/RFC822, i.e. a message encapsulated in another message, as it happens in the case of forwarding an email.

There structured data types are very useful, as, by properly navigating their fields, a programmer can directly access all the information regarding the fetched messages. A proper exception is finally raised if the command does not succeed due to a protocol error or a bad parameter.

6.5 Erlang

The client library we implemented in Erlang as part of this study represents an interface between a front-end and an IMAP4 server. It handles both TCP and SSL client/server connections. The library is based on OTP and works as a stand-alone Erlang application. It manages the most important IMAP4 command such as LOGIN, SELECT, FETCH, LIST, IDLE performing an analysis of server responses in order to create an Erlang representation of email data. Indeed, the library parses server response, verifies it correctness according to IMAP4 standard, and produces an Erlang-native result value.

The library is mainly composed by three modules: `im_client`, the interface between the front-end and the IMAP4 server, it implements all the IMAP4 commands; `scanning`, which performs a lexical analysis of the reply recognizing and carrying out tokens; `parsing`, that instead parses the output of the previous module by generating an Erlang-native term for the server response.

The basic module of the library is `im_client`, which also provides the interface functions to execute IMAP commands. The module is a `gen_server` and the process' status data hold, together with information such as the socket connection, the state of the finite-state machine of the protocol (see Section 6.5) in order to perform correct sending and handling of commands.

The interface provided by `im_client` is simple and quite similar to that of other described libraries. Starting the activities implies to activate the `gen_server` process managing the protocol through the `start_link` function, which takes two parameters, mail server address and connection type ("tcp" or "ssl"). Subsequently, the client can invoke the `login` function to perform authentication; it uses the basic authentication mechanism is based on username and password, and, like all other libraries (excluding the Python implementation) currently does not support the AUTHENTICATE IMAP4 mechanism. The next step is mailbox selection, which is performed through function `select`; after this, messages can be retrieved using the `fetch` function, which requires two parameters, the message or the list of messages to be retrieved and the additional arguments for the FETCH command. A flexible way to specify such arguments has been implemented, indeed messages to retrieve can be specified by providing a list of their message numbers (or UIDs); moreover, if a range of messages is required, it is possible to specify, as an element of the list, a tuple in the form `{first,last}`. As for the arguments of the FETCH command, they are provided as a list of strings, such as `[''flags'', ''bodystructure'', ''size'']`, which are then suitably interpreted by the library in order to prepare the complete FETCH command. Therefore, these parameters are not barely concatenated with "FETCH", as it happens for example in Ruby and Python, but they express the precise information that a programmer wants to obtain from a message; for example, to retrieve the sender, the recipient, the subject and the text of an email, it suffices to pass the term `[{''body'', 1, 1}, ''from'', ''to'', ''subject'']`: the library is able to interpret the provided information, prepare the proper command, which in this case is "(BODY.PEEK[1] BODY[HEADER.FIELDS (FROM TO SUBJECT)])", and suitably parsing the received response.

All of the IMAP protocol functions of the `im_client` module return a tuple containing two elements: {`ResultValue, Parsed Reply`}, the former is an atom representing the server response to the command[7], while the latter is the server's reply data, as provided by the parser modules. In functions `select`, `fetch` and `list`, this second element of the return value is a "proplist"[8] ({*key,data*}), in which the *key* is a symbol with expresses a message data item and *data* is the associated value. As an example, the function call, `im_client:fetch ([5,6], [''bodystructure'', ''date''])`, with generates the IMAP command "FETCH 5:6 (BODYSTRUCTURE BODY[HEADER.FIELDS (DATE)])", replies (on success) the following data:

```
{ok,
  [[{"seq_no",5},
    {"bodystructure","text/plain","7bit","24","834"},
    {"date",{{2007,3,6},{23,2,52}}}],
   [{"seq_no",6},
    {"bodystructure","text/plain","7bit","20","577"},
    {"date",{{2007,3,6},{17,9,0}}}]]}
```

The piece of code below illustrates the use of this library:

```
...
im_client:start_link (Mailserver_name, ssl),
{ok, _} = im_client:login (Username, Password),
{ok, MBoxInfo} = im_client:select ("INBOX"),
{ok, MsgSet1} =
    im_client:fetch( [{1,3}, 10],
                     [{"body", 1, 1}, "from",
                      "to", "subject"]),
{ok, MsgSet2} =
    im_client:fetch( [{15, 20}],
                     ["bodystructure"]),
...
```

The Erlang library also offers some interesting features. One of them is the ability to check the connection and re-instantiate it, if needed; indeed the library avoids some protocol errors such as the "brutal closure connection", which occurs when a login is not performed within a timeout after establishing the connection or when, due to a long period of inactivity, the server unilaterally

[7] It can assume the value 'ok', 'not' or 'bad'.

[8] In particular, for the `fetch` command, it is a list of "proplists".

decides to abort the connection. To this aim, before sending each command, the library verifies if the user is just authenticated and the connection has been set up. Another interesting feature is the ability to support multiple client commands, since their sequencing is then handled internally. Finally, since the library is based on OTP concepts, it can be run as an OTP application embedded in a more complex Erlang system, running in the same memory space as the application using it.

7. Results

This Section compares software productivity measures of all IMAP libraries evaluated in this work. The software metrics used for this comparison are the followings:

- *SLOC*, which measures software size and exactly the number of logical software lines;
- *Functionality of primitives*, which analyses the functionalities of primitives provided by the IMAP libraries;
- *Amount of memory*, which measures the space cost of the library;
- *Execution time*, which measures the time required by different libraries to perform protocol activities.

The following subsections will report all results that have been collect during this work.

7.1 SLOC

The SLOC is a software metric that provides a measure of the effort needed to create an IMAP library in a specific programming language. As discussed in Section 5, libraries analyzed in this paper present different characteristic as they do not provide the same number of functionalities, in terms of both IMAP primitives supported and parsing of server responses. For these reasons, we decided to consider the pieces of code, of the various libraries, which implement (more or less) common IMAP primitives.

Language	*Lines*
Erlang	1,189
Python	472
Ruby	1,612
C#	1,089
Java	n/a

Table 1. SLOC

Table 1 summaries the lines of code calculated; the highest number of SLOC is that of Ruby implementation, while the shortest library is the Python one. Erlang and C# feature a similar number of SLOC, while, for JavaMail, this measure cannot be performed since the source code is not released.

The small SLOC number featured by Python is justified by the complete absence of any parser. With respect to the architecture illustrated in Section 4, the Python library only possesses the *Communication Layer*, the *Low-level Protocol Handler* and the *IMAP High-Level Layer*.

On the other hand, Erlang and Ruby implementations support a *full* parsing of servers replies, and as a result present a higher number of SLOC. The parser in the Ruby library is however more complex than the Erlang one due to the definition of data types and an intensive use of complex regular expressions. The lines devoted in Ruby to reply analysis were 1048 in contrast to the 818 from the Erlang implementation. The reduction in Erlang is a result of making use of function clause matches in order to directly identify reply tokens and perform the appropriate data extraction. This aspect not only improves performances (see Subsection 7.4), but also the comprehension and maintainability of the source code.

Finally, the C# implementation presents a number of SLOC comparable to that of Erlang. This is quite interesting even if C# does not provide full parsing of the server replies. It does not parse the (often complex) result of the SELECT command, something Erlang does in full. It does not support the LIST command and only some parts of the FETCH command, both of which Erlang also fully supports.

7.2 Functionality of primitives

In this Subsection, we will describe the functionalities offered by the various IMAP libraries comparing and contrasting their characteristics. In particular, we will focus on the primitives "LOGIN", "SELECT/EXAMINE", "FETCH" and "LIST". Table 2 summarizes and compares the results of this analysis.

7.2.1 Login

The first functionality we deal with is authentication; to this aim, each library has method or function that performs the IMAP LOGIN command. All of the implementations require the "user_name" and "password" to authenticate client to the server. The Python and Erlang implementations return a result value indicating the success or the failure of the login command; other libraries raise an exception in the case of failed authentication.

7.2.2 Select

The second functionality we analyzed is the selection of the mailbox. The client program can access the desired mailbox by means of the SELECT or EXAMINE command: the former opens the mailbox in read/write mode, the latter in read-only mode. With the exception of JavaMail, all the libraries feature a proper SELECT/EXAMINE function or method. The Java implementation, however, uses a different approach as it provides an `open` method in the `Folder` object which represents the mailbox. This method requires a parameter that indicates how to open the specified mailbox (READ_ONLY or READ_WRITE). Once a folder/mailbox is open, its mode cannot be changed, e.g. from read-only to read-write or vice versa: the `Folder` object must be "closed" and then re-opened with the new mode. This policy is due to the class hierarchy and model used by JavaMail and could make the client more complex: indeed, even if the IMAP4 protocol manages a SELECT command followed by the EXAMINE command (to change the mode to read-only), the Java library does not manage this condition and must close the selected folder and re-open it.

As for the reply to the SELECT/EXAMINE command, C# does not perform any parsing while Java does it, but manages the resulting information internally.

Python returns the total number of messages in the mailbox (which is given by the EXISTS keyword in one of the untagged responses of the server) while Ruby uses a different approach; indeed, the reply to the `select` method is bare text (with no parsing) of the server response[9], but the parsing is internally handled: the `responses` attribute of the `Net::IMAP` object is a dictionary whose keys "EXISTS" and "RECENT" are the number of total and new messages of the mailbox, returned as a result of the SELECT command.

The Erlang library provides the most complete parsing of the SELECT command; the provided `select/1` function returns the tuple `{result_value, parsed_response}`: the former parameter is the outcome of the command (success or failure) and the latter

[9] In particular, the return value is the Ruby structure `Net::IMAP::TaggedResponse`, which contains other two structs, `Net::IMAP::ResponseText` and `Net::IMAP::ResponseCode`.

Command/Primitive	*Erlang*	*Python*	*Ruby*	*C#*	*Java*
LOGIN	Yes	Yes	Yes	Yes	Yes
SELECT	Full parsing	No parsing	Partial parsing	No parsing	Internal w/full parsing
FETCH	Parametric Query Flexible expression of ranges Full parsing	Parameters as string Ranges as string No parsing	Parameters as string Range or single messages Partial parsing	Only some queries supported Single messages Partial parsing	Encapsulated Queries Single messages Full parsing
LIST	Full parsing	No parsing	Full parsing	n/a	Full parsing Supported in current context

Table 2. Comparison of Functionality of Primitives

element is a "proplist", i.e. a list of tuples of the form {`keyword, value`}. In this proplist, the keyword is an Erlang atom representing the status information of the mailbox such as `flags`, `exists`, `recent` etc., while the second tuple element is the related value[10].

7.2.3 Fetch

The FETCH command is treated by Python and Ruby libraries in a similar way. As introduced in Sections 6.3 and 6.4, the method provided requires two parameters: a sequence number or a range and the string command to send to the server. The Ruby implementation parses the server response and generates a proper result value by means of Ruby structured type. The Python library, one the other hand, only parses of the tagged response, interpreting the outcome of the command, returning the untagged response "as-is".

The Java library provides a `fetch` method with the `Folder` class that downloads locally remote messages; once fetching is complete, messages are internally cached so new requests stored locally will not result in a query to the server. Even if the cache allows the client to pick up data quickly, it is not able to send a user-defined fetch command using this method. Moreover, it can retrieve only the fetch attributes provided defined in the FetchProfile class.

The C# implementation has three fetch methods that retrieve different section of a message:

- `FetchPartBody`, which retrieves the body text of a message, the text is returned to the client as string;
- `FetchPartHeader`, which picks up the header of a message and returns its fields within an ArrayList;
- `FetchMessage`, which generates an XML file containing the field of header and body structure reply.

Therefore, the type of the reply and the support for parsing depend on the method called; this is quite unusual because according to good software design, methods of the same class with similar functionalities should behave similarly.

The Erlang library interface for the fetch primitive is strongly based on native data types. As detailed in Section 6.5, the programmer can specify both single messages and message ranges within the single query, and the data items to be retrieved by means of proper list of pre-defined symbols. The reply organizes parsed data items in a "proplist", so it is quite easy, for a programmer, to pick the requested item by means of a simple `proplists:get_value/2` function call.

7.2.4 List

The "LIST" command aims at examining the hierarchy of the folder/mailboxes. It is similarly supported by all libraries with the exception of the C# implementation, which does not provide this functionality.

[10] It can be either a single value (for instance the value of the EXISTS untagged response is only a number representing the total number of message in the mailbox), or a list of values if the untagged response contains many value, as the FLAGS untagged response.

The JavaMail package provides a `list` method in the `Folder` class; therefore a client that has opened a mailbox can investigate only its context.

The methods/functions provided by Python, Ruby and Erlang have the same signature of the list function and require two arguments; the *reference name*, which represents the context in which to investigate, and the *mailbox name*, with possible wildcards. The main difference is in the return value. The Python library does not perform any parsing and returns the (untagged) response as a string for further analysis. The Ruby implementation instead parses the response by returning an array of `Net::IMAP::MailboxList` structures whose fields represent the attributes of a folder, returned by the command. The Erlang implementation carries out, as result of a "LIST" query, the tuple {`result value, list`}; the second element is a list containing other lists; each inner list represents a folder that matches the LIST pattern and its items are tuples of the form {`keyword, value`}[11] carrying the attributes associated with this folder. An example of the return value `list` function of the Erlang implementation is reported below:

```
{ok,[[{name,"mailbox_name_A"},
      {separator,"."},
      {noinferiors,false},
      {noselect,false},
      {marked,false},
      {unmarked,false}],
     [{name,"mailbox_name_B"},
      {separator,"."},
      {noinferiors,false},
      {noselect,false},
      {marked,false},
      {unmarked,false}]]}
```

7.2.5 Discussion

The comparative outcome of the analysis of the functionality of primitives is reported in Table 2. The Table highlights that the most featured and transparent implementations are those in Erlang and Java, while the most feature-poor implementation is the Python one. Erlang and Java solutions, however, differ in some aspects of the software architecture because, while the Erlang implementation provides a direct interface to the IMAP commands, JavaMail is based on an object hierarchy with abstractions such as `Folder` and `Message`; from this point of view, JavaMail supports a higher level interface but undoubtedly introduces an overhead.

7.3 Amount of memory required

This metric evaluates the amount of memory required by an application using the various IMAP libraries. The results provide the memory imprint computed for the various IMAP clients at run-time while they are using these libraries. The second column in Table 3 contains the total memory used by the program, the third one represents the memory taken up by the text code and the last one contains

[11] It is a "proplist" again.

the amount of memory used by the stack and heap of the program to keep its data[12].

Language	*Total*	*Code*	*Data/Stack*
Erlang	8,056	2,868	4,656
Python	6,748	4,400	1,684
Ruby	14,942	4,332	10,386
C#	18,800	11,148	2,748
Java	212,852	14,428	190,060

Table 3. Quantity of Memory Space Used (values are in KB)

It should be noted that this measure mainly reflects the usage of the virtual machine and the library, since all the tested languages are interpreted. As reader can deduce, the highest amount of memory is required by the Java client program, which needs 212 MB of main memory; indeed, since the code is only 14 MB large, the most of the space is devoted to the class library which is loaded and compiled (by the JIT) at the startup of the JVM.

The lighter platform, from the point of view of the total size, is Python: the VM size is only 4 MB and also the library is quite small.

Erlang provides very interesting performances: its virtual machine is very small and is the lighter among the tested languages as it occupies only 2 MB. The size of data space (4.5 MB) is due to the Erlang/OTP runtime which is activated at the startup of the system.

7.4 Execution time/throughput

This Subsection compares the execution times of some primitives of the IMAP implementations described in this paper. The aim is to evaluate the performance of each library in order to compare and contrast the throughput. These results will allow us to get a sense of which programming languages and which architecture best meets the productivity requirements and performance of a distributed system. In general, performances are determined by communication and process management, therefore, for each library, we measure the time of the network interaction. Network interaction is the time required to send and receive protocol commands together with the time taken to parse the server result.

All the tests have been conducted using the same mail server and the same set of email messages. The testing platform (the client) is a 2 GHz Intel Centrino PC, equipped with 1 GB of RAM and Ubuntu Linux 7.10 (kernel 2.6.22). Client and server are connected to the same fast-Ethernet LAN. The runtime systems used for the languages tested are reported in Table 4.

Language	*Platform/Release*
Erlang	R11B-5 (erts 5.5.5)
Python	2.5
Ruby	1.8
C#	Mono JIT 1.2.4
Java	Sun's JDK 1.6

Table 4. Platform/Runtime Systems for the Libraries Tested

Table 5 reports the execution times obtained for each command on all the IMAP libraries. Unluckily, the Java and C# libraries do not provide exactly the same functionality offered by other IMAP libraries; indeed, JavaMail and ImapLibrary have a different policy to retrieve data when compared with the other implementations. Furthermore, they present particular primitives which perform a composite IMAP4 command which cannot be mapped to any of the other commands generated by other libraries. For instance, the C# library offers the `fetchMessage` method to send the IMAP4 "FETCH BODYSTRUCTURE" and "FETCH BODY[HEADER]" commands, which produces an XML file as the outcome of its parsing activity. In addition, the high-level interface of the C# implementation does not offer primitives to directly perform a basic IMAP4 command; only JavaMail allows developers to send a user-defined command overriding the `doCommand` method of the `IMAPFolder` class. This last function executes the user-defined IMAP4 command and returns an array of `Response` objects which contain the ASCII server response. As a result, this method does not perform any parsing activity.

As discussed in the previous Section, the JavaMail package offers a large range of classes and methods to perform IMAP4 protocol activity. The retrieval of data and handling of the mailbox are encapsulated within a Java object, hiding the whole mechanism from the developers. Even if this policy simplifies the client program, it made it hard to compute or compare the execution time of some of the primitive, resulting the "not-available" for some measures.

Comparing the execution times reported in Table 5, the reader can notice that the Ruby implementation provides the worst performances and also the C# solution does not present good execution times. At first sight, the Python library seems the best; we should remind that it does not perform any parsing activity, so the reported measures are (more or less) the times of the network/socket communication.

As for the Erlang and Java solutions, as the Table shows, they feature comparable performances for the commands providing the same functionalities (network transaction + parsing), which are reported in columns 1, 2, 3, 4, 7 and 8: with the exception of "FETCH BODY[TEXT]", the Erlang solution executes in less time with respect to the Java library[13]. This is an interesting result, since we should consider that the Java program is executed in native code (it is compiled by the JIT), while, in Erlang, execution is mainly interpreted. This result is important as it confirms the ability of Erlang to fit the requirements of a distributed application, not only in terms of *distribution* and *fault tolerance* (handled by the native mechanisms of OTP), but also for the performance aspects concerned.

8. Conclusions

In this paper, we reported the results of a comparative analysis of five client-side IMAP protocol libraries implemented in different programming languages: Python, Ruby, C#, Java and Erlang. The aim is to evaluate the performances of Erlang in order to get a sense of the ability of this programming languages and IMAP implementation architecture to meet requirements of productivity and performance for a distributed system. We selected different parameters to perform our comparison: number of *Source Lines of Code (SLOC)*, which provides an assessment of effort needed; *functionality of primitives*, which highlights the way in which IMAP protocol is supported and the quality of the interface to be then used by the developer; *amount of memory*, which measures the space cost of the application; *execution time*, which measures the time required to perform certain basic and critical IMAP activities.

From the analysis, we can conclude that the Erlang library can deliver the requirements of functionality and performance for a distributed system. From the latter point of view, if we consider

[12] `Total Size` also includes read-only data and other segments of memory space, this is the reason why this value is greater than the sum of `Code` and `Data/Stack` columns.

[13] The Erlang time in row 6 can be compared to the Java time in row 7 since these two commands are comparable in terms of both network overhead and reply structure.

#	*Command/Primitive*	*Erlang*	*Python*	*Ruby*	*C#*	*Java*
1	LOGIN	28.6	45.7*	26.1	32.7	30.7
2	SELECT	2.5	2.1*	44.3	13.2	4.5
3	FETCH BODY[TEXT]	43.0	41.1*	80.2	40.0	40.2
4	FETCH BODY[HEADER]	2.5	0.2*	44.5	5.3	2.9
5	FETCH BODY[HEADER.FIELDS]	2.3	0.2*	43.0	n/a	1.3*
6	FETCH BODYSTRUCTURE	1.8	0.2*	80.2	n/a	1.2*
7	FETCH (ENVELOPE SIZE BODY.PEEK[HEADER.FIELDS] ...)	n/a	n/a	n/a	n/a	1.9
8	LIST	0.7	0.9*	52.1	n/a	6.3

* = command executed without any parsing of the response

Table 5. Execution Times of Tested Commands (values are in milliseconds)

the implementations with similar functionalities (i.e. structured parsing of the responses), the execution times of the Erlang solution are high and comparable to those of Java, while the worst results are featured by the Ruby implementation. And even if the best results have been calculated for Python, it should be noted that its library lacks in-depth parsing activity and transparency of function signature. These results confirm that Erlang has significant benefits not only for the rapid production of robust distributed system, but also for the achievement of desired performances without high memory cost.

9. Acknowledgments

This work was a collaboration between Erlang Training and Consulting, UK, and the University of Catania, Italy.

References

[1] http://www.ruby-lang.org/en/.

[2] http://java.sun.com/products/javamail/downloads/index.html, 22Oct. 2007.

[3] J. Armstrong. A History of Erlang. In *Proceeding of History Of Programming Languages*, 2007.

[4] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. *ARPANET Request for Comment No. 3501*, 2003.

[5] T. DeMarco. Yourdon Press, New York, NY, USA, 1982.

[6] N. Freed and N. Borenstein. MIME (Multipurpose Internet Mail Extensions) Part One: Format of Internet Message Bodies. *ARPANET Request for Comment No. 2045*, 1996.

[7] H. D. Garmus D. *Function Point Analysis. Measurement practise for successful software projects*. ADDISON WESLEY, Nov. 2000.

[8] F. I. F. P. U. Group. *Function Points Counting Practices Manual (version 4.1.1), http://www.ifpug.org/, WWW*. 2000.

[9] B. Hausman. Turbo Erlang: Approaching The Speed of C. In *Proceedings of the Implementations of Logic Programming Systems Conference*, 1993.

[10] D. Hubbard. The IT Measurement Inversion. *CIO Enterprise Magazine*, 1999.

[11] E. Johansson. HiPE: A High Performance Erlang System. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2000.

[12] D. King, H. Nyström, and P. Trinder. Comparing C++ and Erlang for Motorola Telecoms Software. In *Proceedings of the International Erlang User Conference*, 2006.

[13] T. J. McCabe. A Complexity Measure. *IEEE Transaction on Software Engineering*, 2(4), 1976.

[14] J. Myers. Simple Authentication and Security Layer (SASL). *ARPANET Request for Comment No. 2222*, 1997.

[15] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[16] Python Software Foundation. http://www.python.org.

[17] QUALCOMM Incorporated. Internet Message Format. *ARPANET Request for Comment No. 2822*, 2001.

[18] U. Wiger. Four-fold Increase in Productivity and Quality. In *Proceedings of the FEmSYS, Deployment on Distributed Architectures*, 2001.

Scalaris: Reliable Transactional P2P Key/Value Store

Web 2.0 Hosting with Erlang and Java

Thorsten Schütt Florian Schintke Alexander Reinefeld

Zuse Institute Berlin and onScale solutions
schuett@zib.de, schintke@zib.de, reinefeld@zib.de

Abstract

We present *Scalaris*, an Erlang implementation of a distributed key/value store. It uses, on top of a structured overlay network, replication for data availability and majority based distributed transactions for data consistency. In combination, this implements the ACID properties on a scalable structured overlay.

By directly mapping the keys to the overlay without hashing, arbitrary key-ranges can be assigned to nodes, thereby allowing a better load-balancing than would be possible with traditional DHTs. Consequently, Scalaris can be tuned for fast data access by taking, e.g. the nodes' geographic location or the regional popularity of certain keys into account. This improves Scalaris' lookup speed in datacenter or cloud computing environments.

Scalaris is implemented in Erlang. We describe the Erlang software architecture, including the transactional Java interface to access Scalaris.

Additionally, we present a generic design pattern to implement a responsive server in Erlang that serializes update operations on a common state, while concurrently performing fast asynchronous read requests on the same state.

As a proof-of-concept we implemented a simplified Wikipedia frontend and attached it to the Scalaris data store backend. Wikipedia is a challenging application. It requires—besides thousands of concurrent read requests per seconds—serialized, consistent write operations. For Wikipedia's category and backlink pages, keys must be consistently changed within transactions. We discuss how these features are implemented in Scalaris and show its performance.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed databases; C.2.4 [*Distributed Systems*]: Distributed applications; D.2.11 [*Software architectures*]: Patterns; E.1 [*Data structures*]: Distributed data structures

General Terms Algorithms, Design, Languages, Management, Reliability

Keywords Wikipedia, Peer-to-Peer, transactions, key/value store

Erlang'08, September 27, 2008, Victoria, BC, Canada.

1. Introduction

Global e-commerce platforms require highly concurrent access to distributed data. Millions of read operations must be served within milliseconds even though there are concurrent write accesses. Enterprises like Amazon, eBay, Myspace, YouTube, or Google solve this problems by operating tens or hundreds of thousands of servers in distributed datacenters. At this scale, components fail continuously and it is difficult to maintain a consistent state while hiding failures from the application.

Peer-to-peer protocols provide self-management among peers, but they are mostly limited to write-once/read-many data sharing. To extend them beyond the typical file sharing, the support of consistent replication and fast transactions is an important yet missing feature.

We present *Scalaris*, a scalable, distributed key/value store. Scalaris is built on a structured overlay network and uses a distributed transaction protocol, both of them implemented in Erlang with an application interface to Java. To prove our concept, we implemented a simple Wikipedia clone on Scalaris which performs several thousand transactions per second on just a few servers.

In this paper, we give details on the design and implementation of Scalaris. We highlight Erlang specific topics and illustrate algorithm details with code samples. Talks on Scalaris were given at the IEEE International Scalable Computing Challenge 2008[1], the Google Scalability Conference 2008 [15] and the Erlang eXchange 2008.

The paper is organized as follows. After a brief review of related work we describe the overall system architecture and then discuss implementation aspects in Section 4. In Section 5, we present a generic design pattern of a responsive, stateful server, which is used in Scalaris. We then present our example application, a distributed Wikipedia clone in Section 6 and we end with a conclusion.

2. Related Work

Scalable, transactional data stores are of key interest to the community and hence there exists a wide variety of related work. Amazon's key/value store Dynamo [3] and its commercial counterpart SimpleDB which is used in the S3 service, are similar to our work, because they are also based on a scalable P2P substrate. But in contrast to Scalaris, they implement only eventual consistency rather than strong consistency. Moreover, Dynamo does not support transactions over multiple items.

The work of Baldoni *et al.* [2] focuses on algorithms for the creation of dynamic quorums in P2P overlays—an issue that is of particular relevance for the transaction layer in Scalaris. They show that in P2P systems the quorum acquisition time and the message latency are more important than the quorum size, which has been

[1] Scalaris won the 1^{st} price at SCALE 2008, www.ieeetcsc.org/scale2008

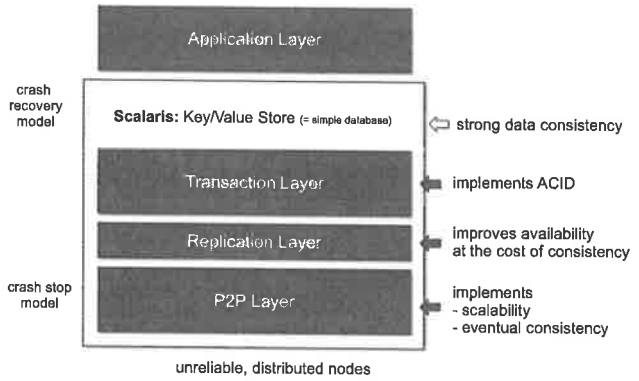


Figure 1. Scalaris system architecture.

traditionally used as a performance metric in distributed systems. This is in line with our results showing that an increasing replication degree r only marginally affects the access time, because the replicas residing in the $\lceil (r+1)/2 \rceil$ fastest nodes take part in the consensus process.

Masud *et al.* [10] also discuss database transactions on structured overlays, but with a focus on the consistent execution of transactions in the presence of failing nodes. They argue that executing transactions over the acquaintances of peers speeds up the transaction time and success rate. Scalaris has a similar concept, but here the peer 'acquaintances' are realized by the load balancer.

With Cassandra [8] and Megastore [4], Facebook and Google recently presented two databases based on the P2P paradigm. Megastore extends Bigtable with support of transactions and multiple indices. Cassandra is more similar to Dynamo as it also provides eventual consistency.

3. System Architecture

Scalaris is a distributed key/value store based on a structured P2P overlay that supports consistent writes. The system comprises three layers (Fig. 1):

- At the bottom, a structured overlay network with logarithmic routing performance builds the basis for the key/value store. In contrast to many other DHTs, our overlay stores the keys in lexicographical order, hence efficient range queries are possible.
- The middle layer implements replication and ACID properties (atomicity, concurrency, isolation, durability) for concurrent write operations. It uses a Paxos consensus protocol [9] which is integrated into the overlay protocol to ensure low communication overhead.
- The top layer hosts the application, a distributed key/value store. This layer can be used as a scalable, fault-tolerant backend for online services for shopping, banking, data sharing, online gaming, or social networks.

Fig. 1 illustrates the three layers. The following sections describe them in more detail.

3.1 P2P Overlay

At the bottom layer, the structured overlay protocol Chord$^{\#}$ [13, 14] is used for storing and retrieving key/value pairs in nodes (peers) that are arranged in a virtual ring. In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to Chord [17], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table cross an exponentially increasing number of nodes in the ring.

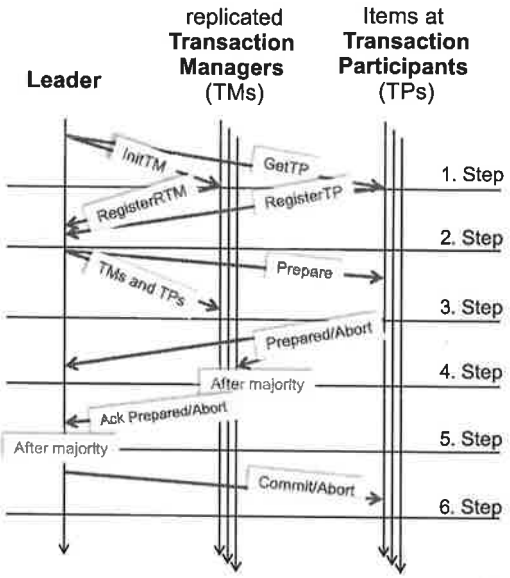


Figure 2. Adapted Paxos used in Scalaris.

Chord$^{\#}$ uses the following algorithm for computing the fingers in the routing table (the infix operator $x \ . \ y$ retrieves y from the routing table of a node x):

$$finger_i = \begin{cases} successor & : i = 0 \\ finger_{i-1} \ . \ finger_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node listed in its $(i-1)^{th}$ finger to which node his $(i-1)^{th}$ finger refers to. In general, the fingers in level i are set to the fingers' neighbors in the next lower level $i-1$. At the lowest level, the fingers point to the direct successors. The resulting structure is similar to a skiplist, but the fingers are computed deterministically without any probabilistic component.

Compared to Chord, Chord$^{\#}$ does the routing in the *node space* rather than the *key space*. This finger placement has two advantages over that of Chord: First, it works with any type of keys as long as a total order over the keys is defined, and second, finger updates are cheaper, because they require just one hop instead of a full search (as in Chord). A proof of Chord$^{\#}$'s logarithmic routing performance can be found in [13].

3.2 Replication and Transaction Layer

The scheme described so far provides scalable access to distributed key/value pairs. To additionally tolerate node failures, we replicate all key/value pairs over r nodes using symmetric replication [5]. Read and write operations are performed on a majority of the replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes.

Each item is assigned a version number. Read operations select the item with the highest version number from a majority of the replicas. Thus a single read operation accesses $\lceil (r+1)/2 \rceil$ nodes, which is done in parallel.

Write operations are done with an adapted Paxos atomic commit protocol [11]. In contrast to the 3-Phase-Commit protocol (3PC) used in distributed database systems, the adapted Paxos is non-blocking, because it employs a group of *acceptors* rather than a

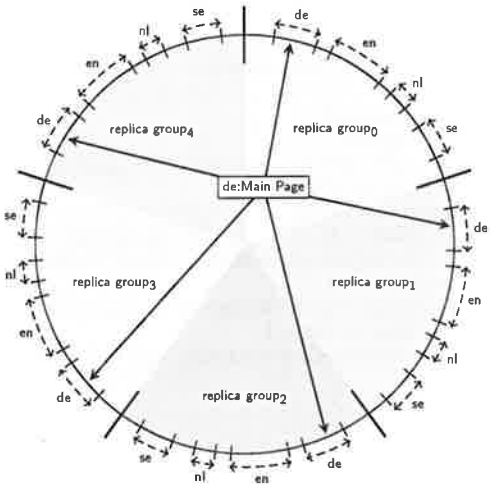


Figure 3. Symmetric replication and multi-datacenter scenario. By assigning the majority of the 'de'-, 'nl'-, and 'se'-replicas to nodes in Europe, latencies can be reduced.

single transaction manager. We select those nodes as acceptors that are responsible for symmetric replication of the transaction manager. The group of acceptors is determined by the transaction manager just before the prepare request is sent to the transaction participants (Fig. 2). This gives a pseudo static group of transaction participants at validation time, which is contacted in parallel.

Write operations and transactions need three phases, including the phase to determine the nodes that participate in the atomic commit. For details see [11, 16].

In Scalaris, the adapted Paxos protocol serves two purposes: First it ensures that all replicas of a *single* key are updated consistently, and second it is used for implementing transactions over *multiple* keys, thereby realizing the ACID properties (atomicity, concurrency, isolation, durability).

3.3 Deployment in Global Datacenters

While we also tested Scalaris on globally distributed servers using PlanetLab[2], its deployment in globally distributed datacenters is more relevant for international service providers. In such scenarios, the latency between the peers is roughly the same and the peers are in general more reliable.

When deploying Scalaris in multi-datacenter environments, a single structured overlay will span over all datacenters. The location of replicas will influence the access latency and thereby the response time perceived by the user. As Chord# supports explicit load-balancing, it can—besides adapting to e.g. heterogeneous hardware and item popularity—place the replicas in specific centers. A majority of replicas of German Wiki pages, for example, should be placed in European datacenters to reduce the access latency for German users.

Scalaris uses symmetric replication [5]. Here, a key 'de:Main Page' is stored in five different locations in the ring (see Fig. 3). The locations are determined by prefixing the key with '0', '1', ..., '5'. So the key of the third replica is '2de:Main Page' and the third replicas of all German articles will populate a consecutive part of the ring. By influencing the load-balancing strategy we can guarantee this segment to be always hosted in a particular datacenter.

[2] http://www.planet-lab.org

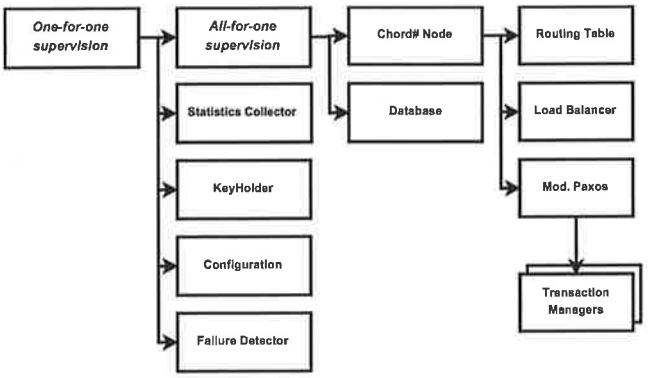


Figure 4. Supervisor tree of a Scalaris node. Each box represents one process.

4. Erlang Implementation

The *actor model* [7] is a popular model for designing and implementing parallel or distributed algorithms. It is often used in the literature [6] to describe and to reason about distributed algorithms. Chord# and the transaction algorithms described above were also developed according to this model. The basic primitives in this model are actors and messages. Every actor has a state, can send messages, act upon messages and spawn new actors.

These primitives can be easily mapped to Erlang processes and messages. The close relationship between the theoretical model and the programming language allows a smooth transition from the theoretical model to prototypes and eventually to a complete system.

Our Erlang implementation of Scalaris comprises many components. It has a total of 11,000 lines of code: 7,000 for the P2P layer with replication and basic system infrastructure, 2,700 lines for the transaction layer, and 1,300 lines for the Wikipedia infrastructure.

4.1 Components and Supervisor Tree

Scalaris is a distributed algorithm. Each peer runs a number of processes as shown in Fig. 4:

Failure Detector supervises other peers and sends a crash message when a node failure is detected.

Configuration provides access to the configuration file and maintains parameter changes made at runtime.

Key Holder stores the identifier of the node in the overlay.

Statistics Collector collects statistics and forwards them to central statistic servers.

Chord# Node performs all important functions of the node. It maintains, among other things, the successor list and the routing table.

Database stores the key/value pairs of this node. The current implementation uses an in-memory dictionary, but disk store based on DETS or Mnesia could also be used.

The processes are organized in a supervisor tree as illustrated in Fig. 4. The first four processes are supervised by a *one-for-one supervisor* [1]: When a slave crashes, it is restarted by the supervisor. The right-most processes (Chord# Node and Database) are supervised by an *all-for-one supervisor* which restarts *all* slaves when a single slave crashed. In Scalaris, when either of the Chord# Node or the Database process fails, the other is explicitly killed and both are restarted to ensure consistency.

4.2 Naming Processes

In Erlang, there are two ways of sending messages to processes: by process id or by addressing the name registered as an atom. This scheme provides a flat name space. We implemented a hierarchical name space for processes.

As described in Sec. 4.1, each Chord# node comprises a group of processes. Within this group, we address processes by name. For example, the failure detector can be addressed as failure_detector.

Running several Chord# nodes within one Erlang Virtual Machine (VM) would lead to name clashes. Hence, we implemented a hierarchical process name space where each Chord# node forms a 'process group'. As a side-effect, we can traverse the naming hierarchy to provide monitoring information grouped by Chord# nodes.

For this naming scheme, every process stores its group id in its own process dictionary. At startup time, processes announce their name and process identifier to a dictionary inside the VM, which is handled by a separate process in the VM. It can be queried to find processes by name or by traversing the process hierarchy. Additionally, most Chord# processes support the {'$gen_cast', {debug_info, Requestor}} message, which allows processes to provide custom monitoring information to the web interface.

4.3 WAN Deployment

Erlang provides the 'distributed mode' for small and medium deployments with limited security requirements. This makes it easy to port the application from an Erlang VM to a cluster. In large deployments, however, the network traffic caused by the management tasks within the VM dominates the overall traffic.

In our code, we replaced the '!' operator and the self() function by cs_send:send() resp. cs_send:this(). At compile time we can configure the cs_send module to use the Erlang distributed mode or our own transport layer using TCP/IP, which will be based on the Erlang SSL library in the future.

This approach also allows us to separate the application logic from the transport layer. Hence, NAT traversal schemes and firewall-aware communication can be implemented without the need to change Chord# code.

4.4 Transaction Interface

Transactions are executed in two phases, the read phase and the commit phase. The read phase goes through all operations of the transaction and keeps the result of each operation in the transaction log. During this phase, the state of the system remains unchanged. In the commit phase, the recorded effects are applied to the database when the ACID properties are not violated.

Read phase. For the read phase, we use a lambda expression which describes the individual operations to be performed in the transaction (see Alg. 4.1). The mentioned transaction log is passed through all calls to the transaction API and updated accordingly. Passing a function to the transaction framework allows us to easily re-execute a transaction after a failure due to concurrency.

Commit phase. The commit phase is started by calling do_transaction (see last line in Alg. 4.1). The transaction is executed asynchronously. The function spawns a new process and returns immediately. The ProcessId which is passed will be notified of the outcome of the transaction. The SuccessFun resp. FailureFun are applied to the result of the transaction before the result is sent back. For the Scalaris implementation, we use the two functions to include transaction numbers into the status messages when a process has several outstanding transactions.

We use the Jinterface package to enable Java programs to perform transactions. The transaction log is managed by the Java program. On a commit the complete log is passed to Erlang and the do_transaction function. Note that transaction descriptions in Java are usually more compact because error handling is done using exceptions (see Alg. 4.2) while in Erlang, the error handling is done in the actual code.

Algorithm 4.1 Incrementing the key *Increment* inside a transaction

```
run_test_increment(State, Source_PID)->
   % the transaction
   TFun = fun(TransLog) ->
      Key = "Increment",
      {Result, TransLog1} = transaction_api:read(Key, TransLog),
      {Result2, TransLog2} =
         if Result == fail ->
               Value = 1,                  % new key
               transaction_api:write(Key, Value, TransLog);
            true ->
               {value, Val} = Result,      % existing key
               Value = Val + 1,
               transaction_api:write(Key, Value, TransLog1)
         end,
      % error handling
      if Result2 == ok ->
         {{ok, Value}, TransLog2};
         true -> {{fail, abort}, TransLog2}
      end
   end,
   SuccessFun = fun(X) -> {success, X} end,
   FailureFun =
     fun(Reason)-> {failure, "test increment failed", Reason} end,

   % trigger transaction
   transaction:do_transaction(State, TFun, SuccessFun,
                              FailureFun, Source_PID).
```

Algorithm 4.2 Java Transactions

```
// new Transaction object
Transaction transaction = new Transaction();
// start new transaction
transaction.start();

//read account A
int accountA =
   new Integer(transaction.read("accountA")).intValue();
//read account B
int accountB =
   new Integer(transaction.read("accountB")).intValue();

//remove 100$ from accountA
transaction.write("accountA",
   new Integer(accountA - 100).toString());
//add 100$ to account B
transaction.write("accountB",
   new Integer(accountB + 100).toString());

transaction.commit();
```

5. Responsive, Stateful Server in Erlang

In distributed server software, slow write operations often block faster reads. Alg. 5.1 shows a generic server architecture (design pattern) that manages reads and writes on a shared state separately. This is done in such a way that read requests can be immediately answered even though a concurrent write operation still blocks the process. Two processes manage the shared state: a public asyn-

Algorithm 5.1 Responsive, stateful server

```
-module(account).
-export([start/0,syncloop/2,slowbalance/2]).

newAccount() -> 0.
start() -> spawn(fun() ->
      Account = newAccount(),
      SyncLoopPid = spawn(account, syncloop, [self(), Account]),
      asyncloop(SyncLoopPid, Account)
    end).

% all requests have to be send to the asyncloop
% read from State via spawns, if its a slow read
% forward writes to the syncloop
asyncloop(SyncLoopPid, State) ->
  receive
  {updatestate, StateNew} ->
    % for better consistency make a join for all spawned
    %   slow reads here
    % for better security, only allow the syncloop
    %   process to update the state
    asyncloop(SyncLoopPid, StateNew);
  {balance, Pid} ->
    Pid ! State,
    asyncloop(SyncLoopPid, State);
  {slowbalance, Pid} ->
    spawn(account, slowbalance, [State, Pid]),
    asyncloop(SyncLoopPid, State);
  % all other messages go to the synchronous loop
  Message ->
      SyncLoopPid ! Message,
      asyncloop(SyncLoopPid, State)
  end.

% internally use a syncloop to serialize all State changes
syncloop(AsyncLoopPid, State) ->
  receive
  {credit, Amount} ->
    NewState = State + Amount,
    AsyncLoopPid ! {updatestate, NewState},
    syncloop(AsyncLoopPid, NewState);
  {draw, Amount} ->
      NewState = State - draw(Amount),
      AsyncLoopPid ! {updatestate, NewState},
      syncloop(AsyncLoopPid, NewState);
    _ ->
      syncloop(AsyncLoopPid, State)
  end.

% functions, that take some time to be executed
slowbalance(State, Pid) ->
  receive
  after 60000 ->
    Pid ! State
  end.

draw(Amount) ->
  receive
  % the bank still works with your money for 10 seconds
  after 10000 ->
    Amount
  end.
```

chronous receive loop asyncloop that performs the reads and forwards the write requests to a private synchronous receive loop syncloop. By this means, write requests are serialized and there is a local atomic point in time when the state changes.

Slow reads may still deliver outdated state. This can be overcome by waiting for all outstanding reads to be completed before changing the state in the asyncloop (not depicted in the algorithm).

Example. Alg. 5.1 shows the processing of states for a bank account. The server provides two read requests (balance and slowbalance) and two write requests (credit and draw) for managing an account. Clients send all their requests to the asyncloop. The server is started by calling account:start(). This spawns a process, which first initializes the account with zero, spawns the syncloop with a reference to itself, and finally executes the asyncloop.

On a balance or slowbalance request to the asyncloop, the account balance is returned to the requesting process from the current state. In case of slowbalance the state is given to a spawned process, which is then executed concurrently in the background. In practice, this spawning should be used when some calculations or other time consuming tasks must be executed on the state before the request can be answered. This way, other requests can be performed by the server concurrently. Here, the corresponding function slowbalance just waits 60 seconds before delivering the result.

In addition, the asyncloop handles updatestate requests as discussed below. All other messages are forwarded to the syncloop.

The syncloop handles the write requests credit and draw. All other messages are ignored and dropped. The syncloop must not spawn processes to calculate state changes, as all state manipulation must be serial to ensure consistency. Here, the draw takes 10 seconds to be performed (the bank uses this time to work with your money). This time has to be consumed synchronously. In practice this could be a time consuming calculation which is necessary to determine the new state. After having calculated the new state, syncloop sends the state with an updatestate request to the asyncloop and works on the new state by itself.

When the asyncloop receives an updatestate message from the syncloop it takes over the new state from the message. This is the atomic point in time when the write request becomes active, as all future requests will operate on this new state.

This leads to a relaxed consistency in the server that is sufficient for updating the routing tables and successor lists. Here, relaxed consistency does not harm, because these tables are subject to churn and will be periodically updated with unreliable link information anyway. If a stronger consistency model is needed, the transaction mechanism of the Erlang Mnesia database package could be used.

6. Use Case: Wikipedia

To demonstrate Scalaris' performance, we chose Wikipedia, the 'free encyclopedia, that anyone can edit', as a challenging test application. In contrast to the public Wikipedia, which is operated on three clusters in Tampa, Amsterdam, and Seoul, our Erlang implementation can be deployed on worldwide distributed servers. We ran it in two installations, one on PlanetLab and one on a local cluster.

The public Wikipedia uses PHP to render the Wikitext to HTML and stores the content and page history in MySQL databases. Instead of using a relational database, we map the Wikipedia content to our Scalaris key/value store [12]. We use the following mappings, using prefixes in the keys to avoid name clashes:

	key	value
page content	title	list of Wikitext for all versions
backlinks	title	list of titles
categories	category name	list of titles

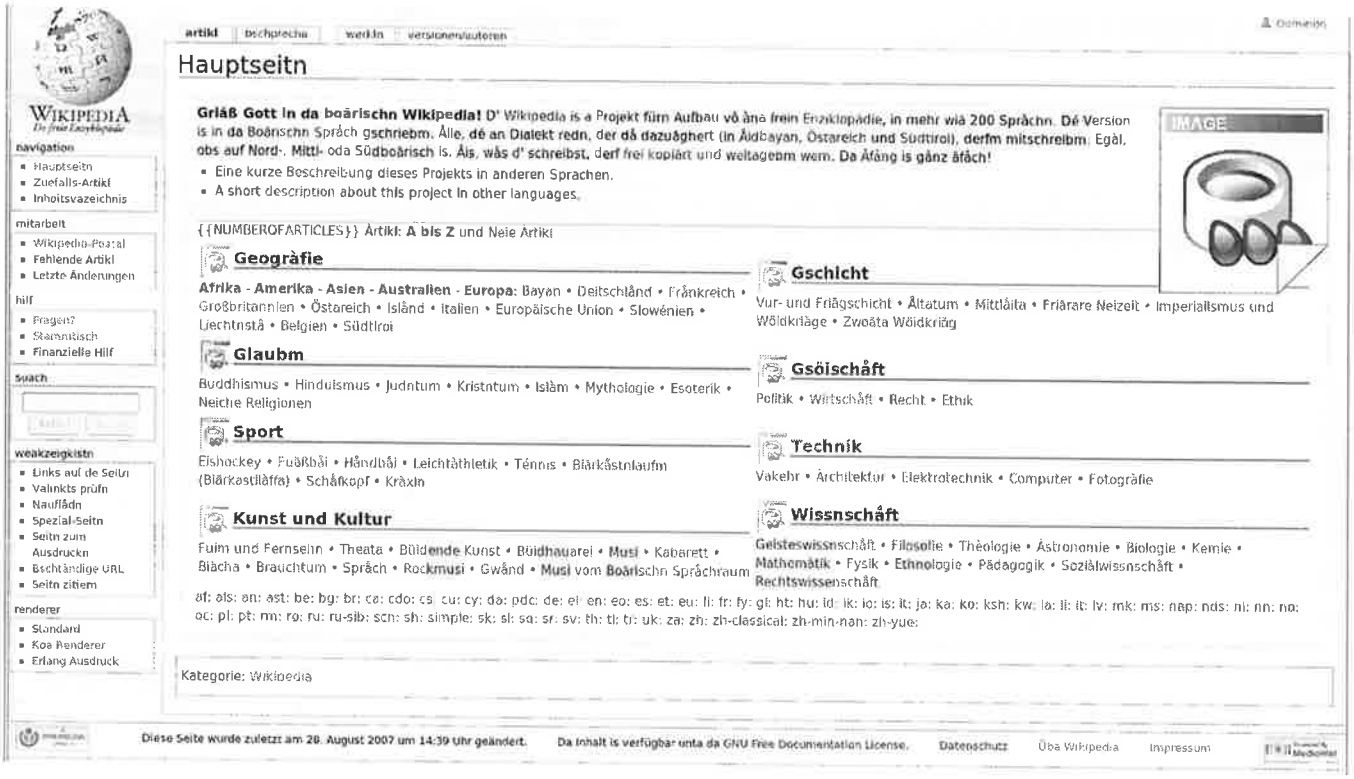


Figure 6. Screenshot of the Bavarian Wikipedia on Scalaris. Images are not included in the dump.

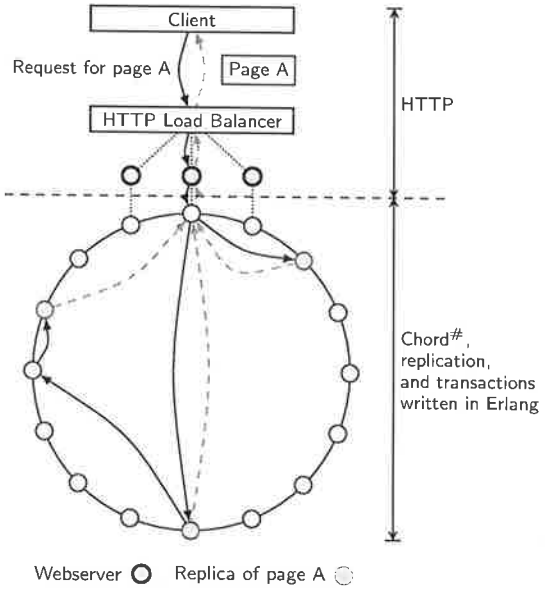


Figure 5. Wikipedia on Scalaris.

The page rendering of the Wikitext is done in Java in the web servers (see Fig. 5) running jetty. Here, we modified the Wikitext renderer of the *plog4u* project for our purposes.

Using this data layout, users may view pages by typing the URL, they can navigate to other pages via hyperlinks, they can edit pages and view the history of changes, and create new pages (see the screenshot in Fig. 7). Since the Wikipedia dumps do not include images, we render a proxy image at the corresponding positions instead. Moreover, we do not maintain a full text index and therefore full text search is not supported by our implementation. This could easily be performed by external crawling and search indexing mechanisms.

When modifying a page, a transaction over all replicas of the responsible keys is created and executed. The transaction includes the page itself, all backlink pages for inserted and deleted links, and all category pages for inserted and deleted categories.

Performance. Our Erlang implementation serves 2,500 transactions per second with just 16 servers. This is better than the public Wikipedia, which serves a total of 45,000 requests per second, of which only 2,000 hit the backend of approx. 200 servers. For the experiments, we used a HTTP load balancer (haproxy) to distribute the requests over all participating servers. The load generator (siege) requested randomly selected pages from the load balancer.

7. Conclusion

We presented *Scalaris*, a distributed key/value store based on the Chord# structured overlay with symmetric data replication and a transaction layer implementing ACID properties. With Wikipedia as a demonstrator application we showed that Scalaris provides the desired scalability and efficiency.

Our implementation greatly benefited from the use of Erlang/OTP. It provides a set of useful libraries and operating procedures for building reliable distributed applications. As a result, the code is more concise than C or Java code.

Additionally, we presented an Erlang pattern that implements responsive, stateful services by overlapping fast reads with concur-

rent synchronous (slower) write operations. This framework did not only prove useful in our key/value store, but it can be used in many other Erlang implementations.

We believe that Scalaris could be of great value for suppliers of online services such as Amazon, eBay, Myspace, YouTube, or Google. Today, global service providers face the challenge of ensuring consistent data access for millions of customers in a 24/7 mode. In such environments, system crashes, software faults and heavy load imbalances are the norm rather than exceptions. Here, it is a challenging task to maintain a consistent view on data and services while hiding failures from the application.

Our P2P approach with replication and ACID provides a dependable and scalable alternative to standard database technology, albeit with a reduced data model. Each additional peer contributes additional main memory to the system, hence the combined memory capacity resembles that of current (large) SAN storage systems. If this is not sufficient, Scalaris can be easily modified to write its data onto disk. For backup purposes, our ACID implementation allows to take consistent snapshots of all data items during runtime.

Apart from distributed transactional data management, Scalaris can also be used for building scalable, hierarchical pub/sub services, reliable resource selection in dynamic systems, or internet chat services.

Acknowledgments

Many thanks to Joe Armstrong for commenting on our responsive server code and to Nico Kruber for implementing the Java transaction interface and adapting the Wiki renderer. This work was partly funded by the EU project Selfman under grant IST-34084 and the EU project XtreemOS under grant IST-33576.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[2] R. Baldoni, L. Querzoni, A. Virgillito, R. Jiménez-Peris, and M. Patiño-Martínez. *Dynamic Quorums for DHT-based P2P Networks.* NCA, pp. 91–100, 2005.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[4] JJ Furman, J. S. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *SIGMOD 2008*, Jun. 2008.

[5] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *3rd Intl. Workshop on Databases, Information Systems and P2P Computing*, 2005.

[6] R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag 2006.

[7] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.

[8] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. *SIGMOD 2008*, Jun. 2008.

[9] L. Lamport. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.

[10] M. M. Masud and I. Kiringa. *Maintaining consistency in a failure-prone P2P database network during transaction processing.* Proceedings of the 2008 International Workshop on Data management in peer-to-peer systems, pp. 27–34, 2008.

[11] M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

[12] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Oct. 2007.

[13] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.

[14] T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.

[16] T.M. Shafaat, M. Moser, A. Ghodsi, S. Haridi, T. Schütt, and A. Reinefeld. Key-Based Consistency and Availability in Structured Overlay Networks. Third Intl. ICST Conference on Scalable Information Systems, June 2008.

[17] I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001.

High-performance Technical Computing with Erlang

Alceste Scalas Giovanni Casu Piero Pili

CRS4
Center for Advanced Studies, Research and Development in Sardinia
Polaris Scientific and Technological Park, Building 1, Pula (Cagliari – Italy)
{alceste,giocasu,piero}@crs4.it

Abstract

High-performance Technical Computing (HPTC) is a branch of HPC (High-performance Computing) that deals with scientific applications, such as physics simulations. Due to its numerical nature, it has been traditionally based on low-level or mathematically-oriented languages (C, C++, Fortran), extended with libraries that implement remote execution and inter-process communication (like MPI and PVM).

But those libraries just provide what Erlang does out-of-the-box: networking, process distribution, concurrency, interprocess communication and fault tolerance. So, is it possible to use Erlang as a foundation for developing HPTC applications?

This paper shows our experiences in using Erlang for distributed number-crunching systems. We introduce two extensions: a simple and efficient foreign function interface (FFI), and an Erlang binding for numerical libraries. We use them as a basis for developing a simple mathematically-oriented programming language (in the style of Matlab™) compiled into Core Erlang. These tools are later used for creating a HPTC framework (based on message-passing) and an IDE for distributed applications.

The results of this research and development show that Erlang/OTP can be used as a platform for developing large and scalable numerical applications.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming; G.4 [*Mathematical Software*]: Efficiency

General Terms Design, Languages, Measurement, Performance

Keywords Erlang, HPC, numerical applications

1. Introduction

With *High-performance Technical Computing (HPTC)* we refer to the use of parallel machines, or clusters of interconnected computers, for executing massive scientific and numerical applications (like physical simulations), possibly under real-time requirements. Today, clusters assembled with PC-class hardware are the most common HPTC solution, due to their low cost and increasing computing power.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

Distributed scientific applications are usually developed with low-level or numerically-oriented languages (such as C, C++, Fortran, etc.) that have no native concept of parallel execution and interprocess communication. These languages are thus extended with libraries like *MPI (Message Passing Interface)* [20, 21] or *PVM (Parallel Virtual Machine)* [26] that implement communication primitives and allow to spawn and monitor remote processes.

In the context of a CRS4 research project, we have been requested to build a framework for real-time HPTC, that should be used by physicists and engineers. Our idea was to use Erlang/OTP as a foundation for building distributed numerical applications, thus exploiting its parallel nature and networking capabilities. In other words, we wanted to replace the most common HPTC structural "building blocks" (MPI, PVM, etc) with Erlang/OTP, in order to:

1. rely on what Erlang/OTP does natively (parallelism, fault tolerance, etc.);
2. use a high-level and concurrency-oriented programming language and platform for creating and extending our HPTC framework, instead of building it on a lower-level ground using C/C++/Fortran.

We also had to take into account that our target users do not know Erlang, but are accustomed to numerical programming languages like Matlab™[27].

The choice of the Erlang path led us to implement several novel extensions and applications:

- a Foreign Function Interface for Erlang/OTP (section 3). We include benchmarks and code samples showing its advantages over the traditional Erlang linked-in driver interface;
- a *BLAS (Basic Linear Algebra Subprograms)* binding for Erlang/OTP (section 4), that guarantees native-speed numerical computation, and allows numerical data (i.e. matrices and vectors) to be easily managed using standard Erlang constructs;
- an imperative, Matlab™-style language for numerical computation, called Matlang (section 5), compiled into Core Erlang. It reduces the amount of code needed for numerical applications, and allows Erlang-unaware physicists and engineers to exploit the features of our Erlang-based HPTC platform;
- a model for distributed numerical applications in Erlang, and a framework (called FLOW) based on that model (section 6);
- an IDE, called ClusterL, for building applications based on the FLOW framework (section 7). It offers a visual approach (reflecting tools like Simulink™ [28] or LabVIEW™ [22]) for assembling distributed numerical applications, and its design allows to handle large projects.

We also made some benchmarks in order to measure the performance of our HPTC architecture in respect to the "classical" C/MPI/BLAS combination, obtaining very good results (section 8).

2. Real-time HPTC

The development of HPTC applications with real-time requirements must face four main issues related to the distribution of computation over a network of computers:

cluster assembly the clustering solution chosen for HPTC should implement some sort of distributed virtual machine, that allows processes to communicate in a network-transparent way. Interprocess communication may be implemented with message passing or (network-transparent) shared memory, with all the issues related to the "share-nothing vs. share-all" (i.e. "processes vs. threads") approaches;

data copying the amount of data being copied in memory at runtime should be minimized, thus reducing latencies and resources usage. When adopting the message passing IPC abstraction, messages should be preferably sent as references to shared memory buffers — but it also means that side effects of each process/thread must be strictly controlled, in order to avoid memory corruptions;

process migration when dealing with variable workloads and long-running applications, processes may need to be migrated from one cluster node to another: it allows both to balance the computing load of the cluster and ensure that processes exchanging high volumes of data are executed on the same machine (without wasting network bandwidth);

fault tolerance long-running distributed applications may have to deal with hardware failures or software bugs that, starting from one or more components, may influence the whole system. If a complete failure is not allowed, then errors should be identified, reported and handled in run-time.

A HPTC framework should help solving all these issues. Furthermore, it must deal with existing numerical code: it is quite uncommon that low-level routines are developed from scratch, since an enormous amount of well-tested mathematical functions is available. In particular, linear algebra packages such as BLAS [9, 10] and LAPACK [2] are the foundation of almost all numerical applications; they also have several highly optimized implementations, either provided by hardware vendors (such as Intel™ [16] or AMD™ [1]) or available on the Web as open source software (such as *ATLAS (Automatically Tuned Linear Algebra Subprograms)* [30]). In addition, research centers and universities usually have relevant collections of home brewed numerical code. Any good HPTC solution must allow to reuse existing software as easily as possible.

3. A Foreign Function Interface for Erlang/OTP

As a direct consequence of the previous paragraph, one of the primary requirements of our Erlang-based HPTC solution is that existing numerical code could be reused without excessive complications. By solving this issue, it also becomes easier to (re)implement performance-critical portions of an application in C, and call the optimized routines from Erlang.

Unfortunately, the traditional Erlang/OTP solution for interfacing external code (i.e. developing a linked-in driver) shows several shortcomings. It gives power and flexibility (like the capability to handle asynchronous execution) — but the resulting API is complex even for developers who just need to perform some synchronous native function calls. This is the case with numerical libraries: they are usually composed by tens or hundreds of functions, and binding all of them with an Erlang linked-in driver requires a relevant amount of boilerplate code for data (de)serialization and type conversions. This code inflation, of course, increases the possibility of introducing bugs. This issue has been addressed with tools like EDTK (Erlang Driver Toolkit) [13] and DryvERL [18] that autogenerate most of the glue code — but the procedure is still not straightforward. Furthermore, the glue code itself may introduce latencies in native function calls.

For all these reasons, we developed an Erlang FFI (Foreign Function Interface) that simplifies the creation of Erlang bindings to native libraries. It does not offer the full potential of linked-in drivers: it just performs synchronous calls with Erlang-to-C and C-to-Erlang type translations — but it is done in a simple, automatic and efficient way.

In order to use our FFI, it is necessary to load an existing shared library, and obtain a port that will work as a handle for further C function calls:

```
ok = erl_ddll:load_library("/lib", libc),
Port1 = open_port("libc").
```

It is now possible to perform direct C calls through that port:

```
Pointer1 = ffi:raw_call(Port1,
                        {malloc, 1024},
                        {pointer, size_t}),
ok = ffi:raw_call(Port1,
                  {free, Pointer1},
                  {void, pointer}).
```

The first tuple provided to `raw_call/3` contains the C function name and arguments, while the second one is the function signature — i.e. a tuple with the function return type followed by the arguments types.

This FFI API is very easy to use, but introduces noticeable overhead: each call must perform a C function symbol lookup and build dynamic C call structures. In order to reduce such delays, our FFI allows to preload functions and compile call structures in advance:

```
ok = erl_ddll:load_library("/lib", libc,
                   [{preload,
                     [{puts, {sint, nonnull}},
                      {putchar, {sint, sint}},
                      {malloc, {nonnull, size_t}},
                      {free, {void, nonnull}}]}]),
Port2 = open_port("libc").
```

The preloading mechanism allows to call C functions without specifying function signatures again:

```
Pointer2 = ffi:raw_call(Port, {3, 1024}),
ffi:raw_call(Port, {4, Pointer2}).
```

The "3" and "4" occurrences above represent the positions of `malloc()` and `free()` in the preload list given as argument to `erl_ddll:load_library/3`. Even if this API is less developer-friendly, it can be used in the inner parts of an Erlang library binding, possibly with the help of tools like SWIG [4].

Since we are working on numerical applications, we have performed a benchmarks on our FFI by executing a sequence of 5 matrix multiplications. We called the BLAS function `sgemm()` (see section 4) from ATLAS 3.6.0 in different ways:

- natively: C code with `cblas_sgemm()` invocations;

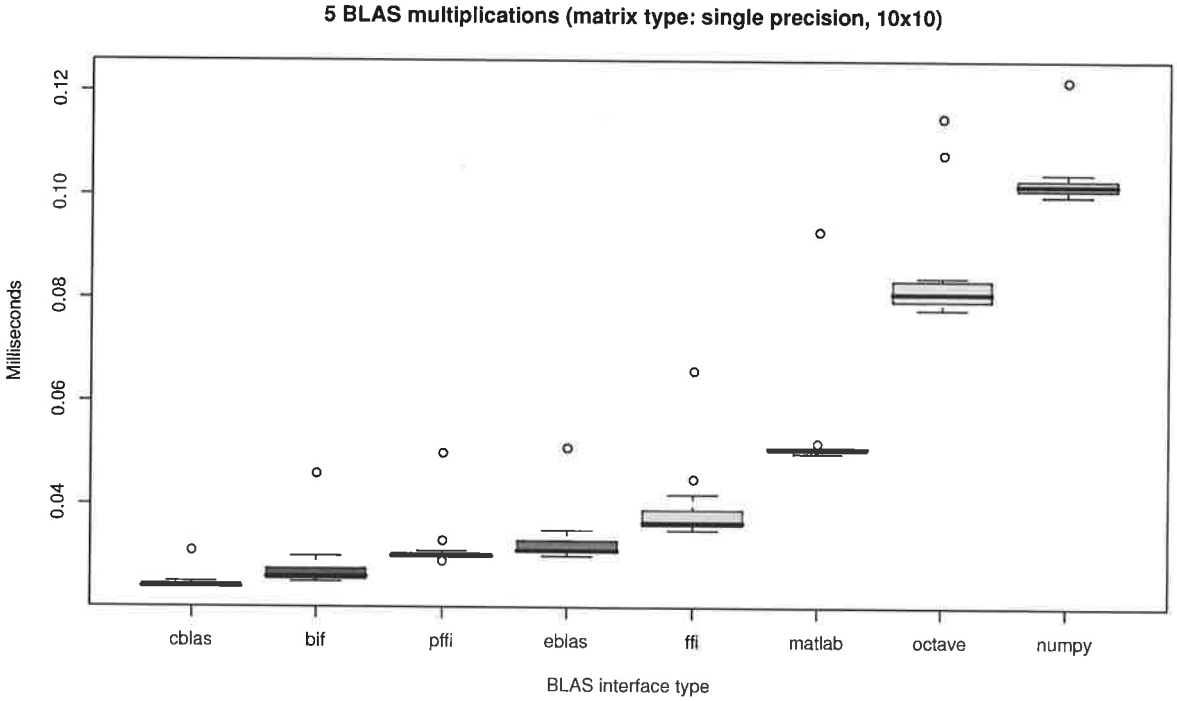


Figure 1. Box plot with benchmarking results of a sequence of 5 matrix multiplications (BLAS function `sgemm()`) performed in different ways: *cblas* (direct call in C through the CBLAS interface), *bif* (dedicated Erlang BIF, developed for testing pourposes), *pffi* (Erlang FFI with symbol preloading), *eblas* (higher-level BLAS interface based on pffi), *ffi* (Erlang FFI, without symbol preloading). The last three columns show the timings of the same multiplications on Matlab™ R14, GNU Octave 3.0.0 and NumPy 1.0.4 with Python 2.5. All the measurements have been done by linking against the same BLAS implementation (ATLAS 3.6.0, optimized for SSE2) on the same platform (Ubuntu™ 7.10, Intel™ Pentium™ 4, 2800 Mhz). The box plot summarizes 20 repetitions of the benchmark.

- in Erlang, with different strategies:
  - using a dedicated BIF, developed for testing purposes;
  - using our FFI, without function preloading;
  - using our FFI, *with* function preloading;
- Matlab™ R14 (by setting its environment variables to make it load the required ATLAS version);
- GNU Octave [11] (linked against ATLAS);
- NumPy [23] (linked against ATLAS).

The timings are summarized in fig. 1. We have chosen box plots [29] for visualization, because they allow us to emphasize the median value (bold line in each column) as well as the presence of jitter and outliers (shown as isolated points above/below each pair of "box and whiskers") among the repetitions of the benchmark.

The plots show that the FFI/preloading combination offers a very good tradeoff: it introduces a small overhead compared to native calls or dedicated BIFs, and its latency is still less than Matlab™. Furthermore, BLAS routines have been interfaced in Erlang with just a few lines of code (mostly necessary for function preloading): developing a linked-in driver for achieving the same result would have required a greater effort (without guaranteeing comparable performances).

Our Erlang FFI has been implemented as a series of patches for OTP R11B-5 [25], and published as an Erlang Enhancement Proposal [24].

4. A BLAS binding for Erlang/OTP

BLAS [9, 10] is the *de facto* standard for numerical computing. It is a set of Fortran routines, also available for the C language (under the name of CBLAS). Its functions are organized in levels: level 1 (vector-vector operations), level 2 (matrix-vector) and level 3 (matrix-matrix). All routines work on memory buffers containing matrix or vector data, and support different numerical types distinguished by the prefix of the function name: single precision (`s`), double precision (`d`), and complex numbers with single (`c`) or double (`z`) precision. This prefix will be indicated with `<X>` in the examples below.

The peculiar form of the BLAS API causes some issues when designing language bindings and integrating existing code. The following paragraphs summarize the problems, and, after listing some additional requirements, present our solution.

Memory layout BLAS operations may access memory buffers in different ways: when dealing with matrices, for example, function parameters allow to use column-major (Fortran-style) or row-major (C-style) modes, with or without transposition; furthermore, each row/column may be separated from the others by any number of unused values: a 10x10 row-major untransposed matrix, for example, could be accessed within a 10x20 one simply by passing 10 as row length, and 20 as the so-called *leading dimension* (i.e. the distance between the first elements of two consecutive rows). These access patterns must be made available through the library binding: they allow to integrate existing numerical routines even if they follow different memory layouts for numerical data.

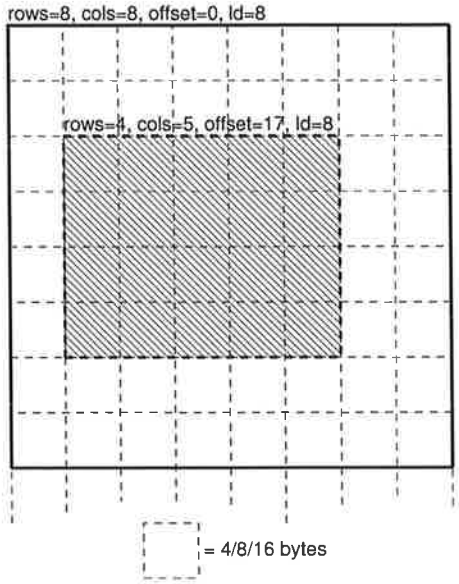


Figure 2. Example of matrix slicing: a 4x5 matrix is indexed within a binary containing the rows and columns of an 8x8 matrix. Each cell contains a numeric value, sized between 4 and 16 bytes depending on the matrix type.

Destructive API Most BLAS functions overwrite one of the arguments with the result of their operations. Furthermore, basic operators may not be directly available — but they may be obtained by calling more complex functions. For example, the contents of two memory buffers `A` and `B` could be multiplied and stored in a buffer `C` by calling the `<X>gemm()` function — which, given the floating point arguments α and β, performs the following operation:

$$C \leftarrow \alpha AB + \beta C$$

Thus, by passing $\alpha = 1.0$ and $\beta = 0.0$, we obtain $C \leftarrow AB$ as we needed.

As another example, there are no BLAS functions that implement matrix addition, and thus it is necessary to implement loops of vector additions (one per row or column). But vector addition, in turn, is only implemented with a function, called `<X>axpy()`, that performs the following operations:

$$Y \leftarrow \alpha X + Y$$

Since one of the operands is overwritten after the addition, we need to make a copy if we want its value to be preserved.

API requirements An easy-to-use BLAS-based API for handling vectors and matrices in Erlang must satisfy several requirements:

1. matrices and vectors should be represented with standard Erlang terms, that could be sent between local or remote processes using standard message passing. In-memory copies should be avoided whenever possible;
2. new matrices and vectors are often created by *slicing* (for example, a vector may be extracted from a column of a matrix). This operation should be as efficient as possible, minimizing data copying and memory usage (and thus maximizing performance);
3. there should be a functional, side-effect-free and Erlang-style API for matrix/vector operations, that could be used without caveats;

```
blas:init(),

%% Create a 3x3 identity matrix
I = blas:eye(s,   % Precision: 's'ingle or 'd'ouble
             3),  % Rows and columns
V = blas:vector(s, 3, [1.0, 2.0, 3.0]),

%% Functional API example: blas:mul/2
V2 = blas:mul(blas:mul(2.0, I), V),
VL = blas:to_list(blas:transpose(V2)),
%% VL is:
%% [[2.00000,4.00000,6.00000]]

%% Procedural API example: blas:mul/3
VTarget = blas:vector(s, 3), % Random data
blas:mul(blas:mul(2.0, I), V, VTarget),
VL = blas:to_list(blas:transpose(VTarget)).
%% VTarget has been overwritten, thus matching VL
```

Figure 3. Usage example of the Erlang BLAS binding.

4. there should also be a procedural interface, easier to use than the underlying BLAS library, allowing to overwrite existing memory buffers — thus reducing memory allocations and garbage collections;
5. lastly, there should be a one-to-one mapping between BLAS routines and Erlang functions, in order to achieve the maximum performance when necessary.

Implementation Facing all the requirements above, matrices have been implemented with Erlang records:

```
-record(matrix, {
    type,   % Atom: s, d, c, z
    rows,   % Number of rows
    cols,   % Number of columns
    ld,     % Leading dimension
    trans,  % Transposition indicator
    offset, % Offset from beginning of binary data
    data    % Refcounted binary with matrix data
   }).
```

The same for vectors:

```
-record(vector, {
    type,   % Atom: s, d, c, z
    length, % Number of elements
    inc,    % Distance between elements
    trans,  % Transposition indicator
    offset, % Offset from beginning of binary data
    data    % Refcounted binary with vector data
   }).
```

These representations allow vector and matrices to be treated as regular Erlang terms. The BLAS library binding ensures that binaries assigned to the `data` field are *always* reference-counted, even for small matrices or vectors: this is necessary in order to make the procedural BLAS API work as expected (when small binaries are copied instead of referenced, destructive updates may be "lost"). For this purpose, the binding always creates data binaries bigger than the heap binary size limit (64 bytes on OTP R11B and R12B [12]).

Matrix and vector slicing is obtained by adjusting the `rows`, `cols`, `ld`, `length`, `inc` and `offset` fields of the matrix/vector records: they allow to use the same data binary to represent different matrices or vectors (see figure 2).

```
%% Create a 3x3 identity matrix
I = eye(3);
%% The following expression is equivalent to:
%%   V = blas:transpose(
%%         blas:vector(s, 3, [1.0, 2.0, 3.0]))
V = [1.0, 2.0, 3.0]';

%% The following expression is equivalent to:
%%   V2 = blas:mul(blas:mul(2.0, I), V)
V2 = 2 * I * V;
%% Result:
%%   V2 = [2.00000, 4.00000, 6.00000]'

%% Function definition
function y = fn(x, t, data)
  y = -x * 3;
end;

%% Function integration (4th-order Runge-Kutta)
Y = rk4(fn,              % Function to integrate
        3.0,             % Initial value
        [0.0, 0.1, 0.2], % Integration points
        []);             % Data (unused)
%% Integration result (on final point):
%%   Y = 1.64652
```

Figure 4. Usage example of the Matlang language.

The resulting BLAS API for can be used as shown in fig. 3. The benchmark results in fig. 1, in the `eblas` column, show that the overhead introduced by the Erlang binding is moderate, and its run-time checks do not eliminate the performance advantage given by our FFI over Matlab™ or GNU Octave.

5. A Matlab™-style language

At this point, Erlang and our BLAS binding are the basic elements for assembling HPTC applications. But there are relevant drawbacks:

1. the numerical code is verbose, expecially compared to equivalent routines developed with mathematically-oriented languages like Matlab™ or GNU Octave;
2. even if the BLAS binding provides a procedural API with side effects, it may be difficult to use efficiently: all the optimizations (and the risks of destructive updates on shared data) are left in the hands of the developer;
3. engineers and physicists (who, as we said in the introduction, are the main target of our work) are not usually accustomed to Erlang and functional programming: they expect to use some procedural language, possibly with Matlab™-like data types and syntax.

For all these reasons, we decided to develop a procedural, Matlab™-like programming language that compiles into Core Erlang [5, 6] by translating matrix and vector operations into BLAS function calls (performed through our Erlang BLAS binding).

We called such language Matlang. A code sample is shown on figure 4.

The main difference between Matlang and Matlab™ is our treatment of functions as first-class objects: in the code sample we see how the `fn` variable (created when defining the homonym function) is passed to the integration routine `rk4()`. Since Matlab™ does not support higher-order functions, the same behaviour could be simulated by passing the `'fn'` string to `rk4()` — which, in turn, would need to call `eval()` on its argument.

There are also other minor differences, mostly due to unsupported syntactic quirks (for an in-depth overview of the problems in implementing a complete Matlab™ language parser, see [17]).

The main mismatch between Matlang (or any imperative language in general) and Core Erlang is the single-assignment semantics. Since multiple assignments cannot be translated directly, we followed a two-step strategy:

1. the Matlang parse tree is converted to SSA (Static Single Assignment) form [8];
2. the SSA form is compiled in Core Erlang: Matlang `if` statements are converted into Core Erlang `case` switches, while `for` and `while` loops are turned into `letrec`s.

In the second phase, ϕ-functions inserted during the first step are used to decide which variables must be returned by each Core Erlang statement.

In its current incarnation, Matlang is a dynamically typed language, and the compiler generates relevant amounts of run-time checks that ensure correct typing of expressions. We are, however, enhancing to the compiler in order to obtain a statically typed language: more details are available in the conclusions (section 9).

6. An Erlang framework for HPTC

In section 2 we have seen four main issues that an HPTC framework must solve: cluster assembly, process migration, minimization of in-memory copies, fault tolerance.

In order to solve them, we defined a simple model that abstracts a generic HPTC application:

> A distributed numerical application is a set of looping numerical *processes* connected by predefined communication channels (called *buses*)

We implemented this model in a framework, called *FLOW*, that provides an API for building, running, monitoring and controlling distributed applications. The developer only needs to define the bus topology and the functions being looped by each process, while FLOW takes care of distributing the computation load on a cluster of computers, dispatch communications and monitor the system behaviour.

More in detail, a FLOW process is defined by:

- an unique identifier;
- one or more *input ports*;
- one or more *output ports*;
- a *core function*, with arguments and return values mapped respectively to input and output ports.

Ports are characterized by an unique identifier and a *signature* that specifies which data types it handles. For example, an output port may be called `Output1` and produce three floating-point values, called x, y and z.

Buses can connect one output port to one or more input ports, provided that they have compatible type signatures (the `Output1` port may, thus, be connected to an input port that expects three floating-point values as well).

As an example of FLOW application, we illustrate the simulation of a mechanical system: a single-degree-of-freedom compound pendulum (fig. 5), composed by a rod RS with length l that oscillates around point S constrained to O (the reference system origin).

The pendulum motion can be described with the following ordinary differential equation (ODE):

$$\ddot{\theta} = \frac{-3g \sin \theta}{2l}$$

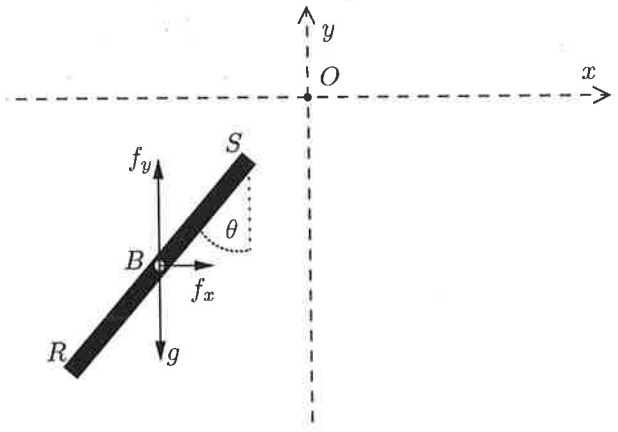


Figure 5. Schema of compound pendulum simulation: the rod RS falls subject to gravity g, but is constrained to O by artificial forces f_x and f_y applied to its barycenter B.

By integrating such equation, we can obtain the pendulum angle θ and its angular velocity $\dot{\theta}$ at any time instant.

But we could also follow a different approach:

- the rod is simulated as unconstrained and free-falling under gravity g. Its position is indicated with barycenter B (with coordinates (B_x, B_y)) and angle θ, while its horizontal, vertical and angular velocities are indicated respectively with $\dot{B}_x$, $\dot{B}_y$ and $\dot{\theta}$;
- the pendulum constraint is simulated by two artificial forces (f_x and f_y) applied to B. They are periodically recomputed so that, every instant, the rod is moved to a position that makes point S coincide with point O. In other words, f_x and f_y simulate the pendulum constraint reactions.

The artificial forces are computed using well-known control theory techniques: in a feedback-based loop, their intensity changes depending on the current position, velocity and angle of the pendulum. More details on this simulation scheme are available in [7]: it allows to model complex constrained mechanical systems, achieving parallelization and numerical stability (even if its benefits may not be apparent from our simple compound pendulum example).

This kind of pendulum simulation can be modeled with two processes:

- a process called `Constraint` that, given the rod state vector $(B_x, \dot{B}_x, B_y, \dot{B}_y, \theta, \dot{\theta})$ as input, computes the constraint forces f_x and f_y;
- a process called `Rod` that, given the current constraint forces and the rod state vector, computes an updated state vector.

These processes should, thus, exchange the current state vector and constraint forces. They should remain idle until new inputs are available. This model can be represented with a diagram, as seen in fig. 6 (on the left).

This kind of diagram can be translated directly into a list of `FlowChildSpecs`, i.e. a data structure that describes a FLOW application in terms of processes and buses (fig. 6 on the right).

The `FLOWChildSpecs` in the example define two processes and two buses. As anticipated above, the processes are characterized by an unique identifier (`Rod` and `Constraint`), input/output ports and a core function that is executed in an infinite loop (`core_Constraint/3` and `core_Rod/3`). When new data is available on the input ports, it is passed as argument to the core function; after its execution, the return values are written on the output ports.

The core functions must respect the following signature:

```
core_fn(State, Params, Inputs) ->
    ...
    {State1, Outputs}.
```

where `State` is a term representing the current internal state, `Params` contains constant parameters, and `Inputs` is a list containing all the values coming from the process input ports. The return tuple contains an updated internal state (`State1`) and a list of values that will be sent through the process output ports. When the core function is called for the first time, the `State` parameter takes the value of the `state0` field from the process `FLOWChildSpec`. The `Params` parameter, instead, comes from the `params` field.

Returning to our pendulum, we have two buses, `B_state` and `B_fxfy`, which carry respectively the rod state and constraint forces. The `input_process` and `output_processes` fields specify which I/O ports get connected by each bus.

Since the pendulum simulation has a circular structure, both `Rod` and `Constraint` have to wait until the other process writes some values on its output port. In order to start the computational loop, the `Rod` process has a field, `output0`, with an initial value (i.e. the initial rod state) that FLOW writes directly on the `State` port.

Lastly, the `node` field, found in both processes and buses, is a hint that FLOW uses for load balancing: objects with the same `node` value will be spawned on the same Erlang VM, while different values will cause processes and buses to be distributed among different VMs.

6.1 Run-time services

At run-time, the FLOW framework works like the Erlang/OTP supervisor behaviour: it spawns, monitors and eventually restarts FLOW processes when they die because of errors. The same happens for buses.

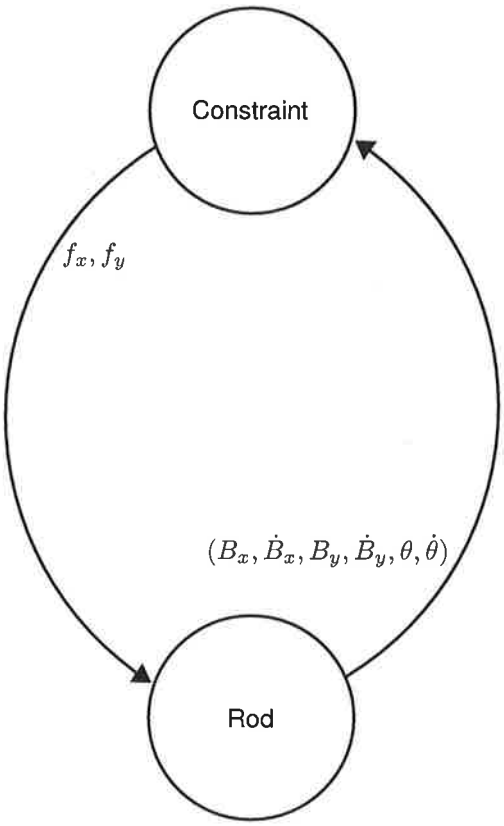
FLOW also provides several functions for the run-time management of an HPTC application, allowing to:

- obtain the `FLOWChildSpec` of a running application, in order to reconstruct its topology;
- stop/resume processes or data dispatching over a bus;
- add, remove or replace processes;
- add, remove or replace bus connections;
- replace the core function of a process, or change its internal state and parameters;
- migrate processes from one cluster node to another, without halting the execution.

These functions allow, for example, to connect to a running FLOW application, attach a monitoring process to a bus and observe the values being dispatched. The monitoring process can be detached when needed. Returning to the pendulum example, we developed an ESDL-based [14] real-time visualization tool that can be attached to the `B_state` bus (fig. 7) with the following sequence of Erlang statements:

```
Pid = spawn(pendulum_monitor, start, []),
ok = flow_bus:add_output_process(
        {pendulum, 'B_state'}, Pid).
```

where `pendulum` is the identifier of the running FLOW application, and the `{pendulum, 'B_state'}` tuple identifies the bus.




```
[%% FLOW process specs
 [
  {type, process},
  {id, 'Constraint'},
  {core, fun core_Constraint/3},
  {node, n1},
  {params, {1.0, 36.0, ...}}, % Constraint coefficients
  {state0, 0},
  {input_ports, [{'State', {vector}}]},
  {output_ports, [{'Forces', {float, float}}]}
 ],
 [
  {type, process},
  {id, 'Rod'},
  {core, fun core_Rod/3},
  {node, n2},
  {params, {9.80665, 1.0, ...}}, % Gravity, rod length...
  {state0, {blas:vector(...)}},
  {output0, [{'State', blas:vector(...)}]},
  {input_ports, [{'Forces', {float, float}}]},
  {output_ports, [{'State', {vector}}]}
 ],
 %% FLOW bus specs
 [
  {type, bus},
  {id, 'B_state'},
  {node, n2},
  {input_process, {'Rod', 'State'}},
  {output_processes, [{'Constraint', 'State'}]}
 ],
 [
  {type, bus},
  {id, 'B_fxfy'},
  {node, n1},
  {input_process, {'Constraint', 'Forces'}},
  {output_processes, [{'Rod', 'Forces'}]}
 ]
].
```

Figure 6. Example of FLOW diagram with corresponding list of `FLOWChildSpecs`. The communication channels are modeled with the `B_state` and `B_fxfy` buses.

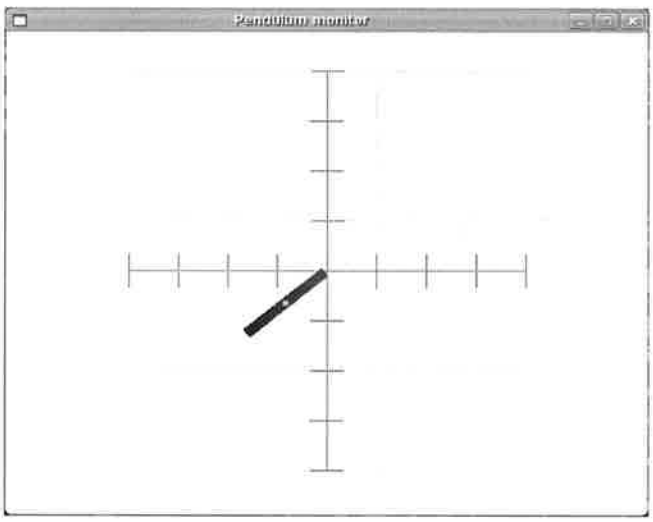


Figure 7. Real-time visualization tool for the compound pendulum simulator. The barycenter is emphasized in the middle of the rod. Since the screenshot was taken when the simulation was still converging, we can see that the constraint is not completely satisfied: the rod extremity is not centered in the reference system origin.

When the monitor is not useful anymore, it can be detached by executing:

```
ok = flow_bus:remove_output_process(
         {pendulum, 'B_state'}, Pid).
```

Running processes can also be migrated with a simple function call:

```
{ok, NewPid} = flow_supervisor:migrate_process(
                    {pendulum, 'Rod'}, 'vm@host.com').
```

It causes the `Rod` process to be moved from its current cluster node to `vm@host.com`, keeping its internal state. Its input and output buses are automatically redirected, without halting the execution of the whole `pendulum` application.

7. An IDE for HPTC applications

Even if the FLOW framework takes care of the low-level details, modeling an large HPTC application could be a very long task: in particular, specifying a list of `FLOWChildSpecs` by hand is tedious and error-prone. And we, as we wrote from the beginning, cannot expect that our target users become proficient enough with Erlang to handle it.

For these reasons, we developed an Integrated Development Environment (IDE), called *ClusterL*, that allows to autogenerate and run a FLOW application with a visual, point-and-click approach.

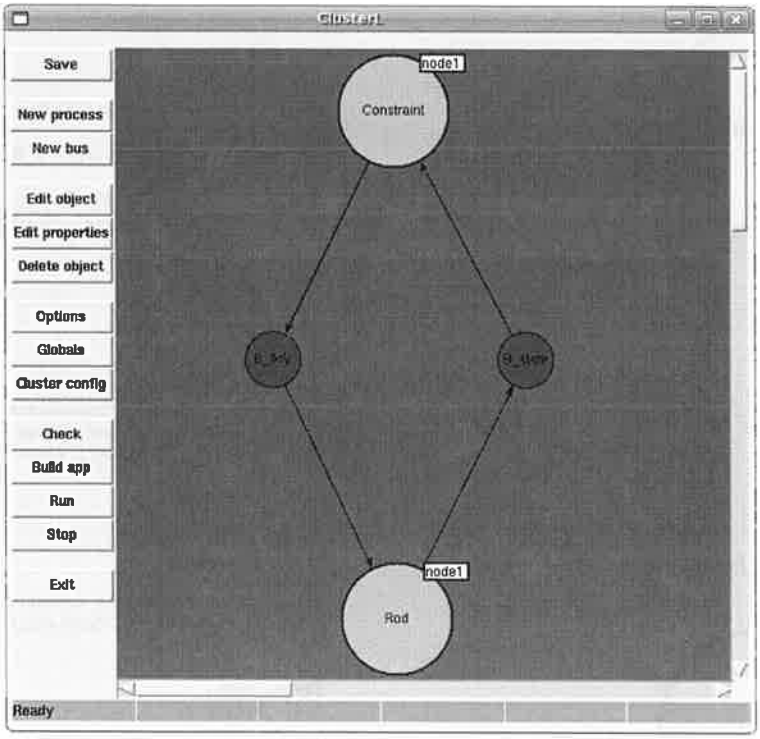


Figure 8. ClusterL IDE workspace, with compound pendulum simulation diagram.

In its current form, it provides a workspace for placing FLOW processes and buses, and a toolbar for editing them.

A screenshot of the interface can be seen in fig. 8: the similarities between the contents of the workspace and the pendulum diagram in fig. 6 are apparent.

ClusterL allows to program FLOW processes either using Erlang or Matlang. Code editing is performed with an editing dialog shown in fig. 10. The GUI allows to define all the process fields seen in the `FLOWChildSpec` (section 6): I/O ports names and types, initial process state, initial outputs, clustering hints, etc. Processes can be connected by adding buses — and, as one may expect, the IDE checks whether the selected input and output ports have compatible type signatures.

ClusterL also introduces some metaphors for representing particular types of processes:

- *source* processes, i.e. processes without input ports that only generate output data. They can abstract external components like read-only files or interfaces to hardware sensors;
- *sink* processes, i.e. processes that have input ports but no output ports. They can be used to represent monitoring or logging routines, or interfaces to hardware devices that provide no input;
- *nested* processes, i.e. processes composed by several subprocesses. This abstraction can represent reusable sub-systems, and allows to assemble large distributed applications without cluttering the GUI with tens or hundreds of objects.

When editing is done, the buttons on the toolbar allow to check the consistency of the process graph, and autogenerate a FLOW-based application — that could be run either from the toolbar, or independently as a stand-alone product. The toolbar also allows to configure a cluster of Erlang VMs, and decide how the application will be distributed on the available nodes.

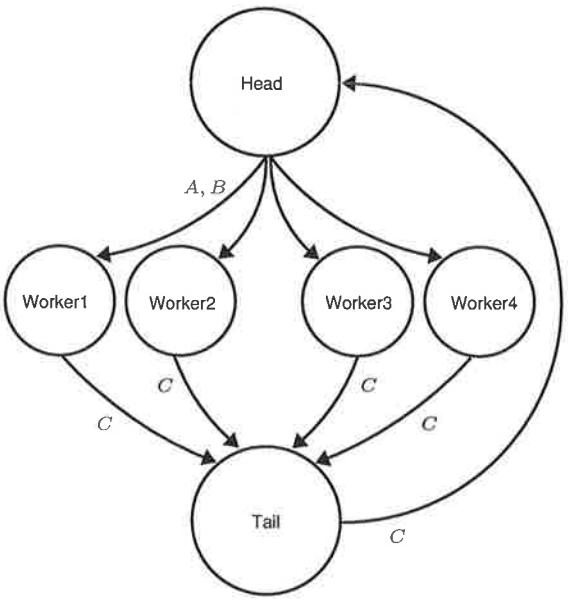


Figure 9. Schema of parallel benchmarking application: the `Head` process sends two matrices, A and B, to four worker processes, which perform a sequence of BLAS operations (20 for each worker) and dispatch results (matrix C), to `Tail`. When `Tail` collects all four copies of C, it sends one of them to `Head`, which in turn wakes up and sends A and B again.

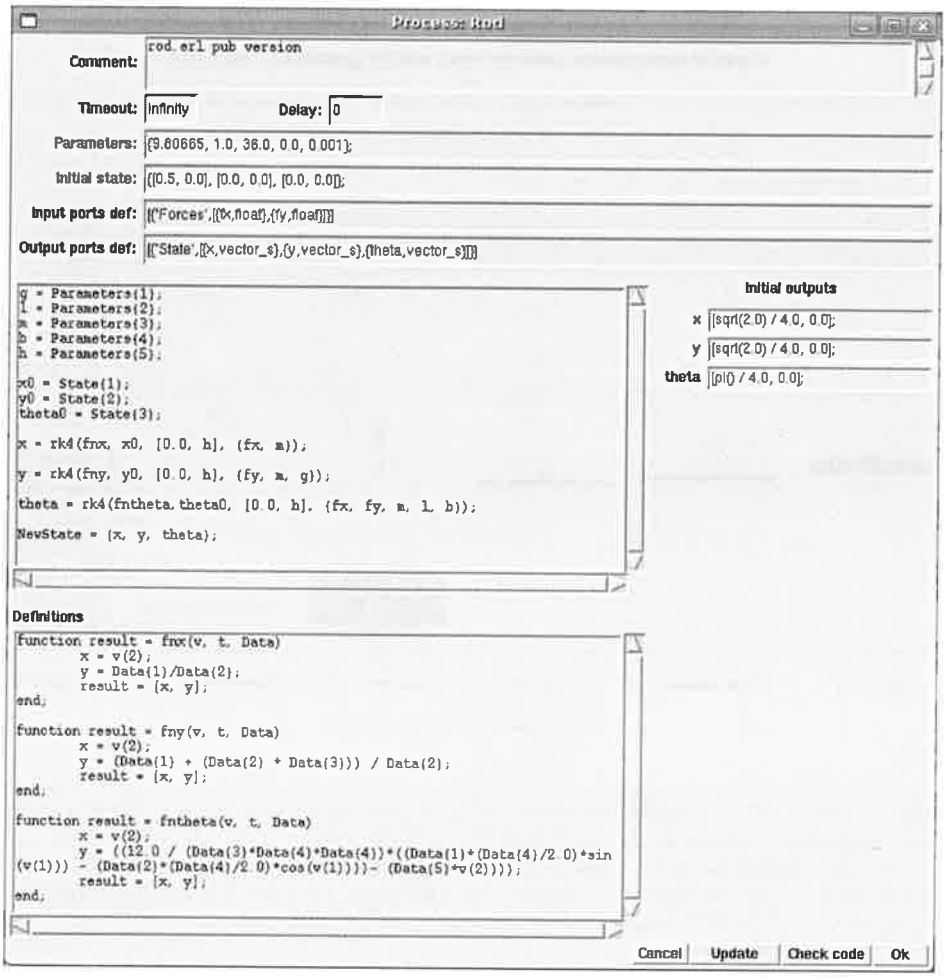


Figure 10. ClusterL IDE process editing dialog. The application is still in alpha stage, and some Erlang code snippets appear where a GUI has not yet been defined (for example, in the text entries for defining I/O ports).

Our experience so far shows that the ClusterL visual approach helps Erlang-unaware physicists and engineers to develop distributed systems.

8. Parallel benchmark

In order to measure the performance of the FLOW framework and our Erlang BLAS binding in respect to "classical" HPTC solutions, we developed a simple parallel benchmark (shown in fig. 9): four worker processes wait for two matrices A and B from a coordinator process ("head"), perform a sequence of 20 BLAS operations each, and send their result C to another process ("tail") — which, in turn, sends C to "head". The timing runs in the "head" process, and measures the number of milliseconds elapsed between A and B are sent, and C is received.

We implemented this parallel benchmark in two ways:

- in Erlang, using ClusterL to generate an application based on FLOW and our BLAS binding;
- in C, with an *ad hoc* program based on BLAS and MPI, using blocking functions for sending and receiving messages (`MPI_Send()` and `MPI_Recv()`).

We deployed the benchmark on two hardware configurations: a single dual-core workstation, and a small cluster with two dual-core PCs connected over a dedicated 1-Gigabit Ethernet LAN. In the latter case, each node ran three processes (two workers together with either "head" or "tail").

Given this setup, we tried two different solutions for the Erlang-based measurements:

1. we launched one SMP-enabled Erlang VM on each cluster node;
2. later, we tried with two non-SMP Erlang VMs on each cluster node. The VMs were bound to different CPU cores using the Linux™ `taskset` utility [19], and each VM ran a single worker process.

For the MPI tests, we used MPICH2 [3] with its default configuration, except for the `--ncpus` option of `mpd` (which we used to indicate the number of CPU cores in each node).

The benchmark results are summarized in fig. 11 and 12. The plots show that, at least for this benchmark, our Erlang-based HPTC solution can reach the same performance of an *ad hoc* application written using C, BLAS and MPI (and thus lacking the run-time services provided by the FLOW framework). Launching several Erlang virtual machines per node (with `taskset`) instead of a single SMP-enabled VM can reduce jitter and increase performance predictability — but multiple VMs also increase the amount

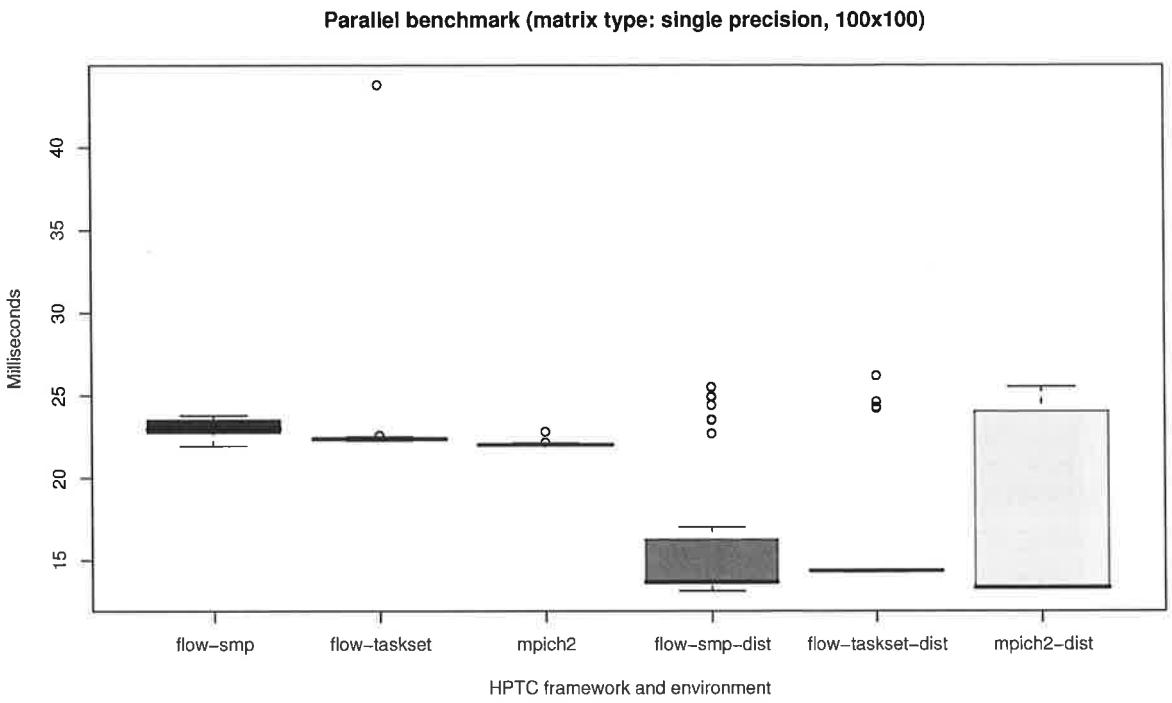


Figure 11. Box plot with the results of the parallel benchmark in fig. 9, repeated 40 times with 100x100 matrix size. The first three columns show the timings obtained on a single workstation (dual-core AMD™ Athlon™ 64 4200+ with 2GB RAM, Ubuntu 8.04 and MPICH2 1.0.6p1); the latter three columns (with the `-dist` suffix in their label) show the results after distributing the benchmark on two workstations (with the same hw/sw configuration above) connected over a dedicated 1-Gigabit Ethernet LAN. The FLOW benchmarks have been performed both by running a single SMP Erlang VM on each cluster node (columns with `-smp`), and by launching a non-SMP Erlang VM for each CPU core (with the `taskset` utility).

of data serialized and sent through sockets, possibly degrading performance (as we can see in fig. 12).

There are, however, some "cheats" that guarantee the efficiency of the FLOW benchmarks: the Erlang VM always tries to handle and send binaries by reference instead of copying them (while MPICH2 uses highly optimized in-memory copying, as required by the MPI standard); furthermore, FLOW buses can automatically detect if two or more target processes are running on a remote node: if it happens, then data is sent only once to a remote dispatcher process, which in turn handles multiple delivery to its local targets. MPI does not implement these capabilities — but our "cheats" are just some advantages that derive from the choice of using a very high-level language (Erlang) and framework (FLOW) that can automatically perform such optimizations.

9. Conclusions and future developments

This paper illustrated how we extended Erlang/OTP, and then used it, as a foundation for building High-performance Technical Computing applications:

1. our Foreign Function Interface allows to interface native code in a simple and efficient way;
2. our BLAS binding represents an use case of the FFI, and allows Erlang to achieve native-speed number crunching capabilities;
3. the Matlang language allows to write Erlang-based numerical code with a concise and mathematically-oriented syntax, familiar to physicists and engineers accustomed to tools like Matlab™;
4. the FLOW framework allows to define and control distributed numerical applications in Erlang, freeing the developer from low-level tasks;
5. the ClusterL IDE can be used for the rapid development of FLOW-based applications, even by users that know nothing about Erlang.

With these solutions, we were able to satisfy the four HPTC framework requirements outlined in section 2. The parallel benchmarks we've performed also show that the overall performance is very good, and comparable to classical solutions based on lower-level libraries.

The most important aspect from our point of view of framework developers, however, is that the resulting HPTC toolkit has been built, and can be extended, using an high-level, concurrency-oriented and fault-tolerant language and platform — i.e. Erlang/OTP. It is an enormous advantage in terms of productivity, that our final users can notice in terms of rapid development and quick response to customization requests. We were prepared to pay this advantage by losing some performance over "traditional" HPTC solutions — but so far the price appears to be moderate.

Even if the results we've obtained are positive and encouraging, there are still several enhancements we are working on. More in detail:

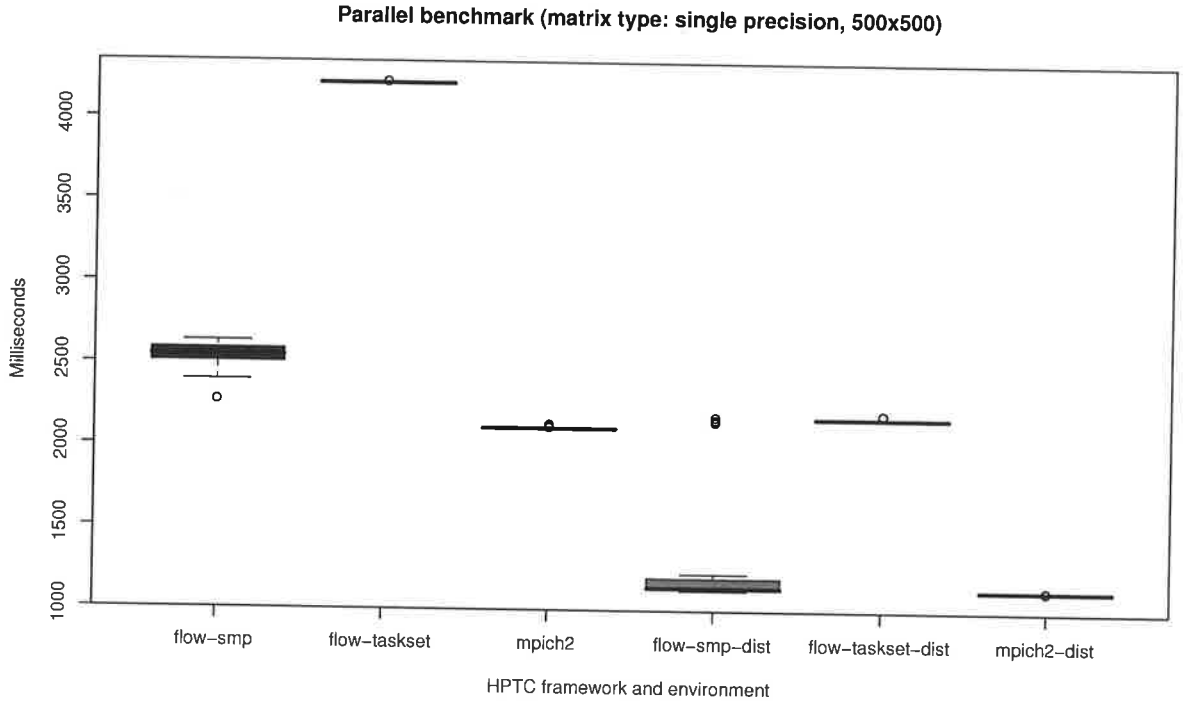


Figure 12. Box plot with the results of the parallel benchmark in fig. 9, repeated 40 times with 500x500 matrix size. When the amount of data exchanged among processes increases, the SMP-enabled Erlang VM guarantees better performances, reaching the numbers of MPICH2. Separate non-SMP Erlang VMs, on the other hand, are slowed down by the need to serialize and transmit huge memory buffers, instead of simply sharing them among processes.

1. FLOW should be extended with the concept of nested processes (that, as reported in section 7, are currently implemented by the ClusterL IDE). We should, in other words, add support for entities similar to Simulink™ "systems";
2. the FLOW run-time control functions should be made available through an user-friendly graphical tool, allowing to connect to a running FLOW-based application and manage it. Users should be able to infer the connection graph of buses and processes, instantiate and connect monitoring tools, migrate a process around cluster nodes, etc. — all without touching the Erlang shell;
3. the ClusterL GUI should be improved, possibly by using wxErlang [15] or some other modern widget toolkit;
4. since port signatures in FLOW processes are statically typed, Matlang could be treated as a statically typed language, too. This would allow to remove most of the run-time checks and increase execution speed. Static type checks could also reduce the amount of bugs, expecially in large projects;
5. static typing could also allow to optimize matrices and vectors handling in Matlang, that (in its current form) never uses the procedural API provided by the BLAS binding. The compiler should perform more static analysis, and decide whether certain optimizations (like overwriting an unused matrix with the partial results of a computation) are legal within some code block.

References

[1] Advanced Micro Devices™, Inc. AMD Core Math Library (ACML). `http://developer.amd.com/cpu/Libraries/acml`.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, Washington, DC, USA, 1990. IEEE Computer Society.

[3] Argonne National Laboratory. MPICH2: an high-performance, portable implementation of the MPI standard. `http://www.mcs.anl.gov/research/projects/mpich2`.

[4] David M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop*, Berkeley, CA, USA, 1996. USENIX Association.

[5] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, September 2001.

[6] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Petterson, and Robert Virding. Core Erlang 1.0.3 language specification, 2004. `http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`.

[7] M. D. Compere and R. G. Longoria. Combined DAE and sliding mode control methods for simulation of constrained mechanical systems. *Journal of dynamic systems, measurement and control*, 122:691–697, December 2000.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[9] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, Spring 2002.

[10] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard II. *International Journal of High Performance Applications and Supercomputing*, 16(2):115–199, Summer 2002.

[11] John W. Eaton. GNU Octave numerical computation language. `http://www.gnu.org/software/octave/`.

[12] Ericsson AB™. Erlang efficiency guide: Constructing and matching binaries, 2008.

[13] Scott Lystig Fritchie. The evolution of Erlang drivers and the Erlang driver toolkit. In *ERLANG '02: Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 34–44, New York, NY, USA, 2002. ACM.

[14] Dan Gudmundsson. ESDL, a SDL and OpenGL™ driver for Erlang/OTP. `http://esdl.sourceforge.net`.

[15] Dan Gudmundsson. wxErlang, an Erlang binding to wxWidgets. `http://www.erlang.org/~dgud/wxerlang/`.

[16] Intel™ Corporation. Intel™ Math kernel library. `http://www.intel.com/cd/software/products/asmo-na/eng/266858.htm`.

[17] Pramod G. Joisha, Abhay Kanhere, Prithviraj Banerjee, U. Nagaraj Shenoy, and Alok Choudhary. The design and implementation of a parser and scanner for the MATLAB language in the MATCH compiler. Technical Report CPDCTR9909017, Center for Parallel and Distributed Computing, Electrical and Computer Engineering Department, Technological Institute, 2145 Sheridan Road, Northwestern University, IL 602083118, September 1999.

[18] Romain Lenglet and Shigeru Chiba. Dryverl: a flexible Erlang/C binding compiler. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 21–31, New York, NY, USA, 2006. ACM.

[19] Robert M. Love. The `taskset` on-line manpage from the Linux™ User manual. `http://www.linuxcommand.org/man_pages/taskset1.html`.

[20] Message Passing Interface Forum (MPIF). MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[21] Message Passing Interface Forum (MPIF). MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, 1996.

[22] National Instruments™. The LabVIEW™ Development environment. `http://www.ni.com/labview/`.

[23] NumPy development team. NumPy, numerical package for python. `http://numpy.scipy.org/`.

[24] Alceste Scalas. Erlang enhancement proposal 7: Foreign function interface, September 2007. `http://erlang.org/eeps/eep-0007.html`.

[25] Alceste Scalas. Home page of the foreign function interface (FFI) for Erlang/OTP, 2008. `http://muvara.org/crs4/erlang/ffi/`.

[26] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[27] The Mathworks™. Matlab™: the language of technical computing. `http://www.mathworks.com/products/matlab/`.

[28] The Mathworks™. Simulink™ : Simulation and model-based design. `http://www.mathworks.com/products/simulink/`.

[29] John W. Tukey. *Exploratory data analysis*. Behavioral Science: Quantitative Methods. Addison-Wesley, Reading, Massachusetts, 1977.

[30] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, 1997.

Refactoring with Wrangler, updated

Data and process refactorings, and integration with Eclipse

Huiqing Li and Simon Thompson
Computing Laboratory, University of Kent
{H.Li,S.J.Thompson}@kent.ac.uk

György Orosz and Melinda Toth
Eötvös Loránd University, Budapest,
Computing Laboratory, University of Kent
{G.Orosz,M.Toth}@kent.ac.uk

Abstract

Wrangler is a refactoring tool for Erlang, implemented in Erlang. This paper reports the latest developments in Wrangler, which include improved user experience, the introduction of a number of data- and process-related refactorings, and also the implementation of an Eclipse plug-in which, together with Erlide, provides refactoring support for Erlang in Eclipse.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) l anguages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

General Terms Languages, Design

Keywords Erlang, Wrangler, Eclipse, Erlide, refactoring, tuple, record, process, slicing

1. Introduction

Refactoring [10] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs. While it is possible to refactor a program by hand, tool support is considered invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools [24] can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the application of the refactoring itself, thus making refactoring less painful and less error-prone.

Whilst the bulk of refactoring tools that have been developed have supported object-oriented programming, there is an increasing interest in refactoring tools for functional and concurrent languages. For Haskell there is HaRe [20, 14, 15], which is embedded in both the Emacs [3] and Vim [4] editors. A prototype of a refactoring tool for Clean is also available [28].

We have recently developed the Wrangler tool for refactoring Erlang [6, 5] programs [19, 16, 17, 18], and in [21] we and the team from Eötvös Loránd University, Budapest jointly reported work on our system and their RefactorErl tool [23, 25]. In this paper we describe the latest developments in Wrangler, which include the introduction of a number of new refactorings, and also the implementation of an Eclipse plug-in which, together with Erlide, provides refactoring support for Erlang in Eclipse.

The rest of the paper is organized as follows. Section 2 gives a short overview of the Wrangler tool for refactoring Erlang programs. Section 3 reports several improvements to the Wrangler user experience. The next two sections describe the data-related refactorings: Section 4 the tupling of function arguments, and Section 5 the introduction of records. We move to discussing process-related refactorings in Section 6. The integration of Wrangler with Eclipse and Erlide is the subject of Section 7. Finally, we draw some conclusions and point to further work in Section 8.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

2. Wrangler

Wrangler is a refactoring tool which supports interactive refactoring for Erlang programs. It is integrated with Emacs [3] and now also with Eclipse [1]. Snapshots of Wrangler embedded in Emacs and Eclipse are shown in Figure 1 and Figure 11. It uses Distel [12] to manage the communication between the refactoring tool and Emacs, and on the other hand the Eclipse integration uses RPC (Remote Procedure Call) to manage the communication.

Wrangler supports more than a dozen refactorings: *Rename variable/module/function*, *Generalise function definition*, *Move function definition to another module*, *Function extraction*, *Fold expression against function*, *Tuple function parameters*, *From tuple to record*, *Rename a process*, *Register a process*, *Add a tag to messages*, and *From function to process*. There are two functionalities for duplicated code detection: *expression search* within a single module and *duplicated code detection* across multiple modules.

2.1 Tool Structure

Every refactoring has two main parts: side-condition checking and transformation. In most cases the side-condition checking is more complex than the transformation itself, because it requires a lot of syntactic and semantic information to be collected and analysed, in order, for example, to ensure that the binding structure of the program is unaffected, or the way in which messages are passed between processes is unchanged. Figure 2 gives an overview the refactoring workflow in Wrangler.

Wrangler uses the standard Erlang parser, slightly modified to include more layout information, to parse an Erlang program into "parse trees", and the SyntaxTools [8] library to build the Abstract Syntax Tree (AST) representation of the program from the parse trees. The AST generated is then annotated with various kinds of syntactic and semantic information including locations, comments, syntax category information, binding structure information, hence becoming the term Annotated Abstract Syntax Tree (AAST). Both

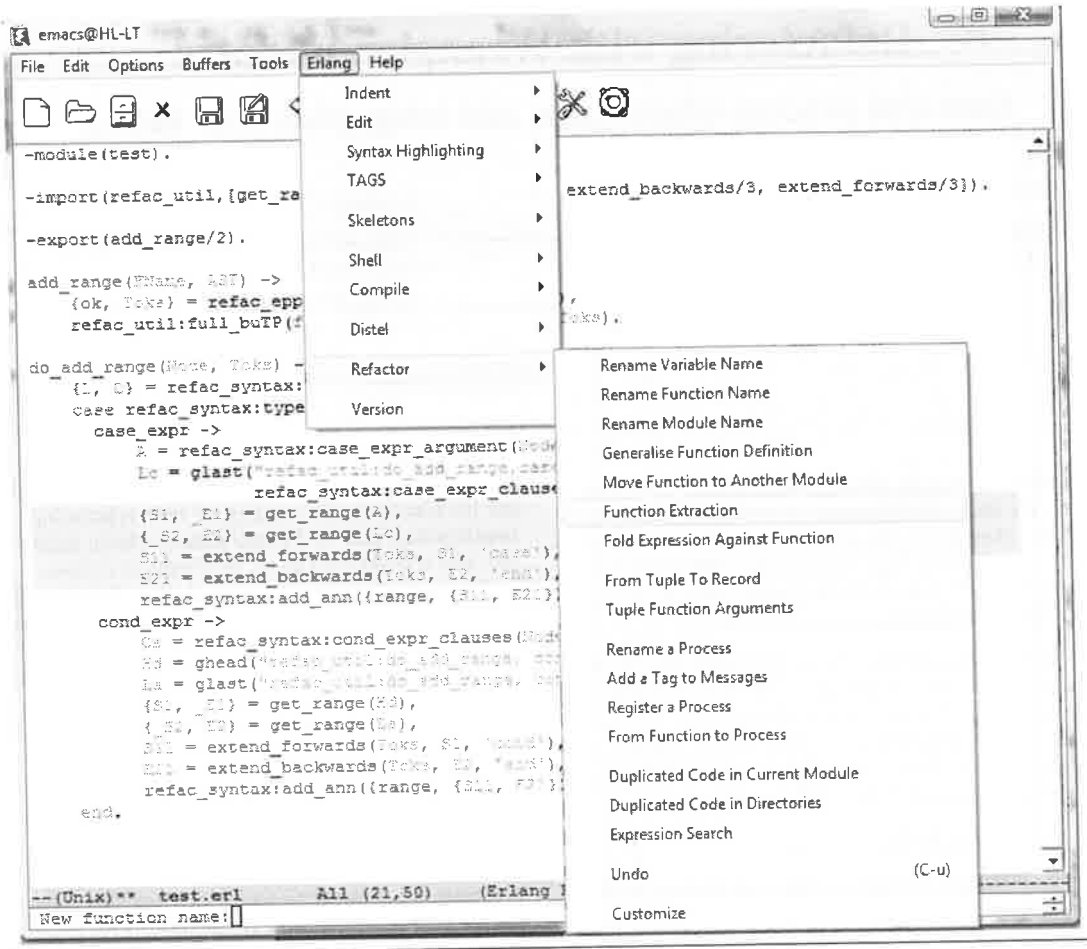


Figure 1. A snapshot of Wrangler in Emacs

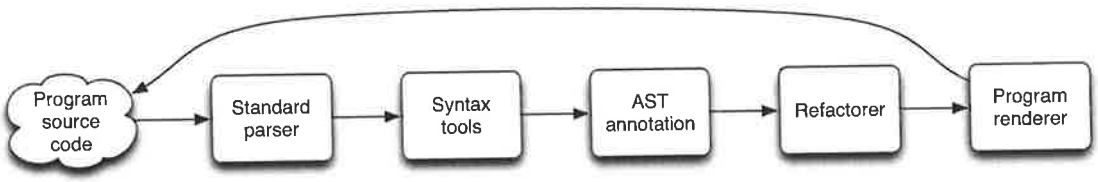


Figure 2. The Wrangler workflow

side-condition checking and program transformation operate over the AAST: during condition checking the conditions typically collate information gathered by walking the tree, and the transformations themselves are also typically accomplished by a tree-walking algorithm.

To perform a refactoring, the refactoring engine first gathers the necessary data and checks that all the side-conditions are satisfied, and then performs the necessary transformation if the previous check succeeds. Most of the refactorings need some user interaction when the refactoring is initiated (typically, a prompt for a new name), and/or during the refactoring process to allow the user to guide the refactoring process. All the refactorings supported by Wrangler are module-aware, supporting the refactoring of multiple-module projects.

Wrangler preserves the original layout of the program as much as possible, and functions/attributes that are not affected by a refactoring have the layout/comments unchanged after a refactoring. More about layout preservation is given in next section.

In order for users to be able to undertake refactoring in a speculative way as a part of their software development process, it is important to be able to undo any transformation. This can be done in Emacs, but if any edits have been performed after the last refactoring these will be lost; in the Eclipse embedding, the undo streams for edits and refactorings are fully integrated.

3. Improved User Experience

To better meet its users' expectations, the infrastructure of Wrangler has been modified in several ways, including improved program appearance preservation, support for refactoring code with syntax errors, and enhanced efficiency in the refactoring process. More details follow in the reminder of this section.

3.1 Program Appearance Preservation

By program appearance preservation, we mean that the refactored program should preserve the original program's layout and comment information as much as possible. Programmers would be reluctant to use a refactoring tool which reformats their code and makes it unrecognisable to them. Comment information is valuable for program understanding and long-term maintenance, therefore should never be discarded by the refactorer. Wrangler was designed to preserve comments, but not layout, from the very beginning.

Originally, we decided to use a pretty-printer to format the transformed program hoping that the layout produced would be acceptable by Erlang programmers; however, we soon discovered that this was not ideal. Our own refactoring experience suggested that sometimes the new layout produced could be so different from the original one that we would rather not to do the refactoring, hence we needed a better way to render the transformed program.

With the current implementation of Wrangler, program appearance preservation is achieved by making use of both token stream and AST. Erlang's standard token scanner discards both whitespace and comments from the source, however we have extended it to keep both. Location information, which is kept in both token stream and AST, is used to map the AST representation of an syntax phrase – such as a function, an attribute and so forth – to its token stream representation. After a refactoring, only those functions/attributes that are affected by the refactoring process are formatted by a pretty-printer, and all the other function/attributes are rendered by extracting the source from the token stream, therefore have their layout completely unchanged. The pretty-printer used by Wrangler respects the original layout, such as line width, of each function/attribute to be printed, and in most cases produces a layout very similar to the original layout of the function/attribute.

3.2 Refactoring Code with Syntax Errors

Wrangler has also been extended to accept Erlang programs that contain syntax errors or macro definitions that cannot be parsed by SyntaxTools. When the program under consideration has syntax errors or unparsable macros, functions/attributes to which these errors/macros belong are not refactored by the refactoring process, however warnings asking for manual inspection of those parts of the program will be given by Wrangler.

This feature was made possible in Wrangler by two facts. Firstly, the Erlang parser is self-recoverable, i.e., a function/attribute that does not parse does not stop the parser from parsing the code following it; secondly, location information kept in the AST and token stream allows us to extract the source code for those syntactically erroneous functions/attributes from the token stream, and put them back into the program during the program rendering process.

3.3 Efficiency Enhancement

While the program analysis and transformation needed by each refactoring may be different, all refactorings need to parse the program under consideration and annotate the AST produced, as shown in Figure 2. When refactoring a large project, a considerable amount of time could be spent on program parsing and annotation, and this could slow down the refactoring process. Therefore, it would be preferable if we could avoid the parsing and annotation process when it is possible, or parse and annotate the program when the refactoring engine is idle.

With Erlang as the implementation language of Wrangler, reusing of AAST is naturally achievable using Erlang processes, and indeed that is the approach we have adopted. With the latest implementation of Wrangler, a `gen_server` process, called *AST_server*, is dedicated to AST management. If an AAST is needed, the refactorer engine will ask *AST_server* for it. With *AST_server*, an Erlang module is parsed only when its AAST does not exist or is out-of-date. The refactoring engine also informs the *AST_server* when a module has been refactored, which will then update its AAST repository in the background. In a similar way, there is also a process in charge of maintaining the function/module callgraph in the background.

4. Tuple Function Parameters

The refactoring *Tuple Function Parameters* groups a number of consecutive arguments of a function into a tuple. This refactoring also modifies the arguments to the call sites of the function, and affects multiple modules if the function is exported, therefore has a global effect.

To apply this refactoring in Wrangler, the user first points the cursor to a function parameter or an application argument in the editor, then selects *Tuple Function Arguments* from the *Refactor* menu, after that the refactorer will prompt for the number of elements that are to form the new tuple.

Tuple Function Arguments has the following side-conditions:

- The indicated position in the editor must be a formal argument of a function definition or an application argument.
- The desired length of the tuple (*m*, say) must be valid. If the chosen parameter is the *n*-th element of the function arguments, and then *m+n-1* should not be larger than the arity of the function.
- The new function produced with a reduced arity should not conflict with existing functions.
- The function must not be an OTP callback function.
- As a design decision, we ask the user to initiate the refactoring from the module where the function is defined.

The example in Figure 3 illustrates an application of this refactoring which groups the first two parameters of function `f/3` into a tuple. Function `f/3` is exported by its defining module and used by another module, and in this case both the definition of `f/3` and its application in the other module `tup2` are changed. The `export` attribution is also affected by the refactoring.

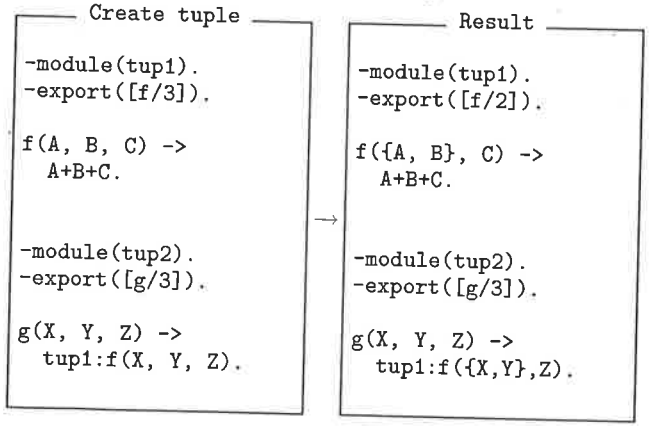


Figure 3. Tupling the first two arguments of the function `f`.

In the case that the function under consideration is used in an implicit *fun* application or a meta-function application, Wrangler will issue a warning message asking the user to check and modify manually if necessary.

5. Introduce Records

Erlang's principal data structuring mechanism is the *tuple*, which corresponds to the C structure, or indeed to tuples in other functional programming languages. The Erlang *record* allows tuple fields to be named, allowing programmers more flexibility in implementation by hiding some of the data representation. One example of this would be to allow a programmer to add a field to an existing record.

Thus, the process of turning a tuple into a record is a natural refactoring, which we call *From Tuple to Record*. This has been explored by the RefactorErl team [22], but to date remains a prototype in that system. We have chosen to take a bottom-up approach to implementing it in the work reported here.

Specifically we have chosen to implement the refactoring which transforms a tuple function parameter into a record expression. This refactoring modifies both the definition of the function and its application sites across the program. If the given record name does not exist, a new record definition is created by the refactorer.

In the remainder of this section we report the design and implementation of *From Tuple to Record*, and then explore ways in which this should be extended.

5.1 From Tuple to Record

To apply this refactoring in Wrangler, first mark a tuple in the editor, which should be a function parameter or an application argument, then select *From Tuple to Record* from the *Refactor* menu, and after that the refactorer will prompt for the record name and the record field names. As a design decision, the user should initiate the refactoring from the module where the function is defined.

A number of side-conditions are necessitated by this refactoring, and they are:

- The starting and ending positions of the selected text should delimit a tuple, which is a function parameter or an application argument.
- The given record name and field names should be atoms, and the record name should not have been used as a record name.
- The number of the field names given must be equal to the selected tuple size and must be distinct.

The example in Figure 4 shows the application of *From Tuple to Record* to the first argument of function `f/2`. A new record, named `rec`, with two fields has been created, and both the definition of `f/3` and its application in `g/1` have been changed. Since `f/2` is not exported by its defining module, this refactoring has a local effect; whereas the example in Figure 5 illustrates an application of this refactoring which affects multiple modules. In the latter example, both the definition of `f/3` and its application in the other module, `record2`, are affected. A record definition is created in both module `record1` and module `record2`. The resulting program could be further refactored by lifting the record definition into a `.hrl` file.

5.2 Types and the refactoring

The example in 6 illustrates refactoring a function which has more the one function clause, and can be applied to both tuples and lists. In this case the refactoring needs to analyze the function calls to the transformed function to decide whether an argument is a tuple (which will become a record) or not. In general this is not decidable, and so it will be necessary to add some run-time type checking (using `case` for example) to decide whether the argument

Create record expression

```
-module(record).
-export([g/1]).

f({A, B}, C) ->
    A+B+C.

g(X) ->
    f({X, 2*X},3*X).
```

Result

```
-module(record).
-export([g/1]).
-record(rec,{first,second}).

f(#rec{first=A, second=B},C) ->
    A+B+C.

g(X) ->
  f(#rec{first=X,second=2*X},3*X).
```

Figure 4. An example of *From Tuple to Record* affecting a single module

Type example

```
f({A, B}, C) ->
  A+B+C;
f([],C)-> 9.

h(X) ->
  Y = {X, X},
  f(Y, 5),
  S = []
  f(S, 3*X),
  Z = mod:app(X),
  f(Z, X).
```

Figure 6. Function with multiple clauses

is a record or not. This will clutter up the code, but serves as a warning to the possible user of a refactoring like this.

5.3 Replace tuple with record

In order to inform the next steps of our work, we have undertaken a case study of the Erlang Standard Library in order to discover the most used patterns of record usage. The three that we have discovered are

Replace tuple with record in a function body. Instead of accessing a tuple literally, we can name the record in the function argument and access it directly.

Using record update. If a tuple expression is a variant of another tuple expression, the former can be defined from the latter using record update syntax.

Record access. Access to components of a record can be given by an access expression, rather than by a pattern match of the whole record.

Used in combination, these transformations allow a user to hide the representation of a data type, thus giving a more abstract, and thus more flexible, interface to the data. It remains a research challenge to provide the appropriate interface to this collection of refactorings, so that a 'batch' application of them to a whole set of

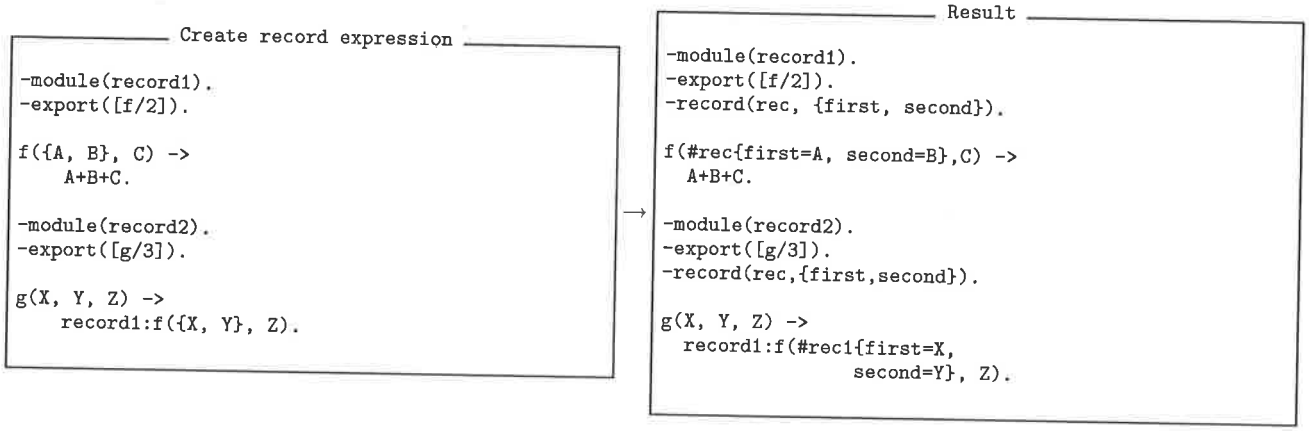


Figure 5. An example of application of *From Tuple to Record* affecting multiple modules

functions which operate over a given (conceptual) data type can be devised.

6. Process-related Refactorings

Built-in support for lightweight processes is one of the strengths that distinguish Erlang from other programming languages. Erlang programs are made of lots of processes. These processes can communicate with each other by sending messages. In Erlang, programming with processes is easy, needs only three new primitives: **spawn**, **send (!)** and **receive**; however, undisciplined use of processes could make the program hard to understand and maintain. For example, some typical process-related bad code smells include

- Code for implementing a single process spans across multiple modules, or code for more than one kind of process exist in the same module.
- Use process and message passing when a function call can be used, or use sequential function calls to model parallel activity.
- Name of a registered process does not reflect its role or functionality.
- Send/receive untagged messages.
- Non tail-recursive functions, especially non tail-recursive servers.
- Register a process that only lives a short time, or not register a process that lives a long time
- Not use generic OTP libraries, such as the generic server, when doing so is more appropriate.

Most of the above bad code smells can be detected, and refactored out step by step manually. However, after having examined a few basic refactorings, such as *register a process*, *add a tag to messages*, we realised that the dynamic nature of the language and the implicitness of process and communication structure of an Erlang program present a challenge for tool support of automated process-related refactorings, or at least some of them.

For example, the refactoring *register a process* registers a process with a name provided by the user, and replaces the receiving process identifier in a send expression with the process name if the process identifier refers to, and only refers to, the selected process. An example application of this refactoring is shown in Figure 7. For this refactoring to be behaviour preserving, the following side-conditions are necessary:

- The process name provided by the user should be an atom, and should not have been used as a process name in the program under consideration.
- The selected process should not have been registered.
- Should multiple instances of the process exist during run time, they should not co-exist at the same time.

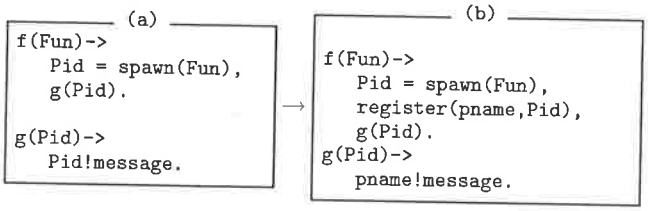


Figure 7. Register a process

If all the side-conditions are met, we are then able to proceed with the transformation. However, when replacing a process identifier in a `send` expression with the process name, we must make sure that the process identifier *only* refers to the process selected. For instance, in the example shown in Figure 8, the `Pid` in expression `Pid!message` should not be replaced by `pname` because this Pid is associated with multiple process instances.

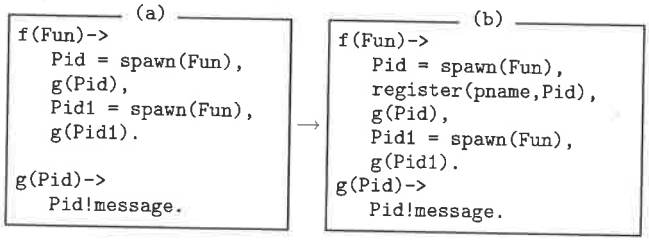


Figure 8. Register a process

Even though this refactoring is very basic, neither its side-condition analysis or its transformation is straightforward to carry out due to the dynamic feature of Erlang and the design of Erlang's process system. Next, we summarise the major challenges that we have encountered when process-oriented refactoring is concerned.

- Processes in an Erlang program are syntactically implicit. Unlike some other concurrency-oriented programming languages,

such as Pict [27] in which processes and channels are syntactically marked out, Erlang does not have a syntax category designed especially to identify processes. In an Erlang program, a process is created by the application of `spawn/1` or its variants. `spawn/1` itself is just an Erlang built-in function. For example, the expression

```
Pid = spawn(Fun)
```

creates a new concurrent process that evaluates `Fun`, and returns a `Pid` whose value identifies the process.

- Implicit connection between a process identifier and the process identified. The `spawn` expression above also reveals the fact that what identifies a process is not the name of the process identifier, but the actual value. Since a variable can take part in computation, or pass its value to other variables, it is possible that two or more process identifiers have the same value, therefore refer to the same process. Deciding whether two or more process identifiers refer to the same process statically needs data-flow analysis. Furthermore, as the Erlang type system only provides run-time rather than static type checking, even whether a variable stands for a process identifier or not is not always clear from the static view of the program.

  While it is possible to name a process using the function `register/2` provided by Erlang, it is not always desirable to do so especially if a process only lives a short time, and sometimes it is not possible to do so as pointed out by the side-conditions of *Register a process*.

- The process communication structure is implicit. Processes in an Erlang program communicate with each other by message passing. `Pid!Message` sends `Message` to the process identified by `Pid`, and returns the message itself; `receive...end` receives a message that has been sent to a process. Because of the indirect connection between a process identifier and the `send/receive` expressions of the identified process, trying to establish a connection between a `send` expression in one process and the corresponding `receive` expression in another process is difficult, not even to mention the mapping between particular messages sent/received. This is particularly obvious when the refactoring *Add a tag to messages* is concerned. This refactoring tries to add a tag to all the messages received (or sent) by a particular process, and obviously it needs to find out where these messages are sent from.

- Unlike functions or modules, a process in Erlang does not have a clear syntactically specified body or scope. Statically a process consists of the collection of functions that are reachable from the entry function/expression of this process. But, it is possible for multiple processes to share code, even `send/receive` expressions. Sharing of `send/receive` expressions makes it difficult to refactor messages sent/received, since it potentially affect all those processes sharing the code, as well as those processes that communicate with them.

- Process context dependent evaluations. Erlang is a language with side-effects. Some of the built-in functions provided by Erlang depend on the context of the current calling process. A particular example is the function `self/1`, which returns the process identifier of the calling process. Hence, care has to be taken if a refactoring changes the execution context of an expression. Examples of this kind of refactorings include *From function to process*, *From process to function*, *Spawn a new process to execute an expression* etc.

As mentioned before, Wrangler uses *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs. The annotation information includes binding information of variables and functions, syntax category, location, comment information, tokens, etc. Together with some fundamental functionalities for function call graph construction, module graph construction, side-effect analysis, etc, the existing infrastructure provides enough information to proceed with most refactorings regarding to the pure functional part of the language, but not with most process-related refactorings because of the challenges presented above.

To support process-related refactorings, we have extended our work in two aspects. Firstly, we have extended the existing AAST representation of Erlang programs with process information; secondly, we have exploited the use of slicing techniques to help the refactoring process. As a design strategy, Wrangler always try to extract as much necessary information as possible by static analysis, and minimise the amount of information needed from the user.

The remaining of this section is organised as follows. We first describe the annotation of AAST with process information, then discuss program slicing and its uses within the refactoring context. Finally, a summary of the process-related refactorings supported by the current implementation of Wrangler is given.

6.1 Annotate AST with Process Information

In an Erlang program, the only way to create a process is via the application of `spawn`, which creates a new concurrent process and returns a process identifier. But because process identifiers can be passed to other functions as parameters or returned values, or even passed to other processes by messages, sometimes it is not clear which process an identifier refers to. With this analysis, we aim to establish a static connection between a process identifier occurrence and the process identified. Due to the syntactic implicitness of Erlang processes, we use the `spawn` expression to represent the process created. In Wrangler, a particular `spawn` expression is identified by the combination of the spawn expression itself, the enclosing function of the spawn expression and the relative location of the spawn expression within the function. Location is needed to resolve the cases when two or more lexically the same spawn expressions occur in the same function.

As an example, given the sample code (a) in Figure 8, this analysis will annotate each occurrence of `Pid` in function `f/0` with

```
{pid, [{spawn(Fun), {mod, f, 1}, 1}}],
```

in which `pid` means the variable represents a process identifier, `spawn(Fun)` is the spawn expression that creates this identifier, `{mod,f,1}` refers to the enclosing function of the spawn expression, and the last integer `1` means that the spawn expression is the first spawn expression in this function. Here we assume that the name of the module to which the sample code belongs is `mod`. However, the occurrences of `Pid` in function `g/1` will be annotated with the following information because of the multiple application sites of this function:

```
{pid, [{spawn(Fun), {mod, f, 1}, 1},
       {spawn(Fun), {mod, f, 1}, 2}]}.
```

With this kind of annotation, we are able to check whether two process identifiers refer to the same process or not by looking at the spawn expressions associated with them. The basic annotation algorithm used by Wrangler works as follows:

1. Construct the call graph for functions, and sort it topologically based on the dependencies between functions.
2. Within each function definition, annotate every occurrence of `spawn` application expression with process identifier information as illustrated above.

3. Analyze the call graph in a bottom-up order to propagate process information within each function definition through function application (when a function returns a process identifier), pattern matching and the binding structure of variables whenever it is possible. In the case that a function returns a process identifier, the return type of this function is also recorded.
4. Analyze the call graph in a top-down order to propagate process information from the call-sites to local function definitions.
5. Repeat from step 3 until a fix-point has been reached.

Apart from `spawn` expressions, process identifiers returned by other built-in functions, such as `self/1`, could also be annotated in a similar way.

So far, this algorithm does not handle complex pattern matching and message passing, therefore only partial process information is annotated into the AAST. However, methods have been taken to indicate whether the information annotated to a process identifier is complete or not.

User input is still needed when an undicidable situation occurs, but we try to reduce this kind of situations by the use of slicing techniques when it is possible.

6.2 Program Slicing

Apart from annotating AAST with process information, we have also exploited the use of program slicing techniques to reduce the number of uncertainties encountered by the refactoring engine by marking out the scope of the program that needs to be analysed or transformed.

The concept of program slicing was first introduced by Weiser. In [30], Weiser defines a program slice S as *a reduced executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P*. The slicing process generally starts for a *slicing criterion*, which represents the point in the code whose impact is to be observed with respect to the entire program. A backward slice contains all parts of a program that may have an effect on the criterion in question; by contrast, forward slices contain all parts of a program that may be affected by the selected criterion. Program slicing has its applications in many areas, such as debugging, code understanding, reverse engineering, program testing, etc. Program slicing itself could also be refactorings. For example, a function returning a tuple could be sliced into two functions, each of which returns an element of the tuple.

Within the context of refactoring Erlang programs, we have mainly exploited the use of static program slicing to reduce the scope of the program to be analysed, with the hope to reduce those undicidable cases for which Wrangler needs to ask for user's input or issue warning messages in order to proceed with the refactoring process. Both forward and backward inter-function slicing of Erlang programs have been implemented. In this paper we are not going into the details of the implementation, instead we focus on benefits of slicing during the refactoring process.

6.2.1 Forward slicing

Given an expression or a subset of the arguments of an Erlang function, Wrangler's forward slicer returns all parts of the program that may be affected by the value of the selected expression or arguments by employing data dependency analysis. The slicing algorithm operates cross function borders if the returned value of the function depends on the slicing criterion or any expression that depends on the slicing criterion is passed as a parameter to a function defined within the application in question. For instance, the example code (b) in Figure 9 shows the slicing result for the first `spawn(Fun)` expression in function `f/1`.

The major benefit of forward slicing is that it gives a clear scope of the program which might be dependent on the selected criterion,

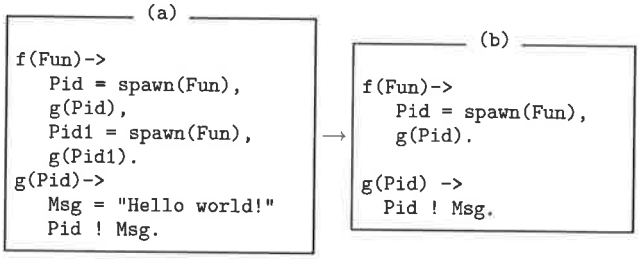


Figure 9. Forward slicing

therefore a confined scope for program analysis if only the parts of the program that depend on the slicing criterion is necessary to be analysed. For example, to check whether a spawned process has been registered by other processes, we only need to check those registration expressions that belong to the slice produced by taking the spawn expression as slicing criterion. Reducing the analysis scope also reduces the number of undicidable situations encountered.

6.2.2 Backward slicing

In contrast to forward slicing, backward slicing uses a backward traversal of the data dependency flow from the point of interest given in the slicing criterion, and returns the parts of the program that could potentially affect the value of the selected expression. Depending on the applications of the computed slices, some will require that the returned slice is executable, while others only need the relevant expressions to be returned without checking whether those expressions form a syntactically well-formed program or not. With Wrangler, backward slicing has been used mainly with two scenarios. More details follow.

- Slice in order to evaluate. In some situations, it would help the refactoring process if Wrangler could know the possible values of a specific variable or expression. One approach is to use the functionalities provided by the module `erl_eval`, which defines an Erlang meta interpreter for expressions. For example, the function `erl_eval:exprs/2`, or its variants, can be used to evaluate a sequence of expressions in an abstract syntax representation. *First slice then evaluate* could ensure that only those expressions which could affect the value of the selected expression will be evaluated. More than that, in the case that the expression sequence to be evaluated depends on some formal parameters of the enclosing function, inter-function slicing provides more chances for the evaluation to be successful.

  For instance, with refactorings such as *rename a registered process*, *register a process*, Wrangler needs to know the process names that have already been used by the program, however this is not always straightforward when a process name can be dynamically composed as shown in the example code (a) in Figure 10. Taking the variable `ProcessName` from the expression `register(ProcessName, Pid)` as the slicing criterion, Wrangler's backward slicer will return the expression shown in part (b) in Figure 10. If there are multiple applications of the enclosing function of the slicing criterion, or functions that call this function either directly or indirectly, the slicer will return a list of expressions, each of which corresponds to a non-recursive call chain that leads to the function containing the slicing criterion. Note that it is not always the case that the produced slices can be evaluated, because of the lack of bindings for some functions for example, but again one strategy of Wrangler is to extract as much as information needed as possible.

(a)

```
start() ->
  Prefix = "ch1",
  State = [1,2]
  start(Prefix, State).

start(Prefix, State) ->
   ProcessName=list_to_atom(Prefix++"_proc"),
   Pid=spawn(ch1, init,[ProcessName, State]),
   register(ProcessName, Pid).
```

(b)

```
fun(Prefix) ->
   ProcessName = list_to_atom(Prefix++"_proc"),
   ProcessName
end (begin Prefix = "ch1", Prefix end).
```

Figure 10. Backward slicing

- Like forward slicing, backward slicing can also be used to refine the scope of analysis. For example, taking a process identifier as the slicing criterion, backward could help to locate where the process is spawned, and even the initial function of the process identified.

The current slicing algorithms implemented in Wrangler do not handle process communication, and this aspect will be further investigated in the future.

6.3 Process-related refactorings supported by Wrangler

A number of process-related refactorings have been implemented using the enhanced infrastructure of Wrangler, and they are:

- *Register a process*, which register a process identifier with a user-provided name, and replaces the use of the process identifier in a `send` expression with the use of the process name whenever this is safe. Registering a process with a name allows any process in the system to communicate with the process without knowing its Pid.
- *From function to process*, which turns a function definition into a process, and all the calls to this function into communication with the new process. This refactoring provides potential for memorisation of the computed results and adding new functionalities.
- *Rename a registered process*, which renames a process' registered name to a user-provided new name. The main challenge of this refactoring is to detect whether an atom with the same name in the program presents a process name or not.
- *Add a tag to the messages sent/received by a process*, which adds a tag to all the messages received (or sent) by a process. This refactoring affects not only the process where the refactoring is initiated, but also the other processes which commutate with it. The refactoring does not distinguish individual messages received (or sent) by a process, therefore all the messages belonging to the processes involved will be added the same tag. The tags added can then be renamed manually by the user to distinguish different kinds of messages. While not ideal, this refactoring still help to mark out a clear scope that needs inspection.

7. Eclipse integration

There are some imitations to the way in which Wrangler is integrated into the Emacs editor, and so we have investigated integrating Wrangler in Integrated Development Environment (IDE). In doing this we aimed to make as few changes to Wrangler as necessary, and to use it as a 'black box' to provide services to the IDE. On the other hand, this integration work provides a perspective on the design of Wrangler (and indeed Eclipse and its refactoring model) and we discuss this at the end of the section. Before that we describe the background to the work, and then give an overview of the integration work; full details of this work are given in the project report, [26].

7.1 Emacs

Emacs [3] is an highly configurable text editor with syntax highlight tool, debugger interface among many other features, but – as its name says *Editor MACroS* – it is just an editor with additional functionalities. What is more the fundamentals of the current version were originally written in 1984, when the developers of the tool, in a very understandable way, did not address refactoring support.

So the support provided by Emacs for various code transformation scenarios is not as good as it might be. To be more specific

- A typical refactoring will affect a complete project, rather than a single file. When integrating a refactoring tool with Emacs it therefore becomes necessary to define a notion of project, by, for instance, specifying a set of search paths.
- A number of refactorings – such as those which move a definition from one module to another, or those which rename a module – affect the way in which a project is built using 'make' or other systems. Changes made within the editor-embedded refactorer will not by default be reflected in the build infrastructure of the system.
- Emacs has a notion of 'undo', related to the editing operations; a refactoring tool will also provide a separate 'undo' operation; it is not at all clear how the two separate 'undo' operations can be put together.

Taken together these arguments against editor-embedded refactoring systems prompted us to investigate ways in which Wrangler could be integrated with an IDE.

7.2 Eclipse

The best developed open source IDE is Eclipse [1, 13], which *is an open source community whose projects are focused on building an extensible development platform, ... for building, deploying and managing software across the entire software lifecycle. Many people know us ... as a Java IDE but Eclipse is much more than [that]* [1]. In particular Eclipse has a plug-in architecture [9] which supports the integration of new functionality for Java and other languages. Plug-in distribution and update is provided by the Eclipse organisation.

For us, the most important thing is the refactoring support of Eclipse. It provides a very well documented refactoring API, the Eclipse Language Toolkit (LTK) [11], with fully support for integration into various aspects of the infrastructure of Eclipse, including

- the refactoring menu,
- refactoring previews, and,
- 'undo' and 'redo' support.

The LTK is described in more detail in Section 7.4 below, when we describe how Wrangler refactorings are integrated into Eclipse. Integration of this sort has already been developed for the Ruby language [29].

Eclipse is designed to be a universal tool platform and provides several extension points and APIs to extend it. The basis of Eclipse

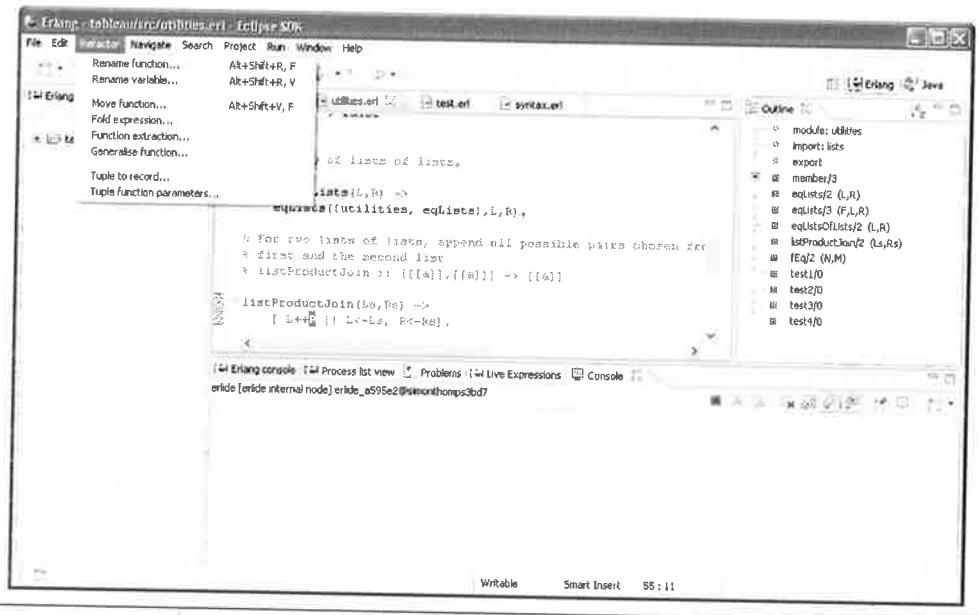


Figure 11. Wrangler in Erlide

is the kernel (or *runtime*), which loads plug-ins as needed. On top of this are four components

Workspace. The Workspace component handles the resources, including files, directories, projects, connections. Every modification of a resource is handled by the Workspace component; it also stores the history of each resource, letting the user undo or redo changes.

Workbench. The Workbench is the graphical interface next to the kernel. It is implemented in Eclipse's own Standard Widget Toolkit (SWT), giving OS native look-and-feel. It manages all the views, editors, and user actions as well. Of course it is also extensible using its extension points.

Team. This provides support for working with CVS / SVN repositories among other version management systems.

Help. This supports the definition and contribution of many kind of documentation.

Plug-ins can declare extension points, which can be used by others to extend its functionality in a controlled way. The Wrangler plug-in uses the following extension points:

org.eclipse.ui.editorActions: This allows plug-ins to add menus and toolbars to the workbench, when the selected editor type becomes active. In our case this was used to add the Refactor menu.

org.eclipse.ui.bindings: A binding is used to define relations between sets of conditions, commands and keybindings, and is used to create shortcuts for refactorings.

org.eclipse.ui.commands: This is used to create commands and command categories. A command is an abstract representation of a semantic behaviour; in our case it makes the connection between actions and bindings.

7.3 Erlide

The Erlide [2] plug-in provides an Eclipse-based development environment for Erlang, with features including a built-in console, automated build tool, syntax highlighting, code completion and debugging support, outline and running processes view and live expression evaluation; see Figure 11. The Erlide backend is a Java interface for an Erlang node [7]. It provides thread safe RPCs (Remote Procedure Call) to each node, and each project is linked to a backend. This backend starts and stops when the project is opened or closed.

7.4 Integrating Refactorings using the LTK

The LTK provides a toolkit for integrating refactorings into Eclipse. This has a number of advantages, such as integrating them with the preview mechanism and the undo/redo mechanism, but it does provide a somewhat different workflow for refactorings than that assumed by Wrangler. An initial problem was that Wrangler is designed to modify source files, and we needed first to modify it so that it returns a new copy of the file. More fundamentally, the LTK workflow follows this pattern

1. The user initiates the refactoring.
2. An initial check is made of some of the preconditions.
3. User interactions (e.g. getting a new variable name).
4. According to the user input, another check is called; if no error occurs, the changes are calculated.
5. A preview dialog appears (optionally), then the calculated changes are applied if required.

while the Wrangler workflow is thus:

1. The user initiates the refactoring.
2. User interactions.
3. Applying the refactoring (within the Wrangler system)
   (a) checking conditions
   (b) calculating modifications
   (c) applying them to the AAST
   (d) writing them back to a new source file

Clearly, the Wrangler workflow will not allow the initial check (LTK 2) and so this stage becomes trivial, with user interactions

(LTK 3) preceding the call to Wrangler (LTK 4). This call will generate a new source file, from which a set of differences, calculated using an open source 'diff' tool, can be generated, as required by LTK 4. This 'diff' set forms the input for the final stage (LTK 5).

This correspondence gives a high-level overview of the way that a number of refactorings, such as *renaming functions and variables*, and *tupling of arguments*, can be integrated into Erlide and Eclipse. We next turn to some of the difficulties presented by the integration exercise.

7.5 Integration challenges

The model presented in the last section allows information to be gathered prior to any further processing, and this supports certain kinds of refactoring as discussed above. However, others require a more fine grained interaction. This includes *function generalization* and *folding expressions against function definitions*, which we discuss now.

In function generalization, a user selects a sub-expression of the function body, provides a new parameter name, and once this is done the user will be prompted by Wrangler for further confirmation in the case that the expression contains free variables or potentially causes a side-effect. This extra interaction is accommodated in the plug-in by means of Eclipse pop-up windows.

Folding instances of a function body into a call to that function will in general result in multiple instances of that body, and so multiple requests to the user for confirmation. In order to integrate this, it was necessary to change the Wrangler workflow for this refactoring, to return all the candidates in a single step, then to be iterated through within Erlide.

In both those cases, it was necessary to modify the refactoring to fit the LTK model of a refactoring. Some refactorings appear to go beyond the LTK model entirely. Any refactoring which modifies the files used by a system – such as renaming a module, or creating a new module by moving a definition to a non-existent module – cannot be accommodated in the LTK model.[1]

Other 'refactorings' – like clone detection – are not quite refactorings, and it would be artificial to include them in the LTK interface; we are currently investigating including them in a general 'search' interface.

7.6 Reflections on Wrangler

The LTK workflow presented in Section 7.4 suggests that the architecture of Wrangler might be modified to fit more tightly into Eclipse. In particular, it would be possible to refactor the preconditions of refactorings into two parts.

- The first part could be checked independently of the user input: in the example of 'rename function' this might include checking that the current position of the cursor is on a function identifier.
- The second part will use the user input – in our example the new name for the function – and check that, for instance, this name is not already used in the module, or imported from another module.

As we have noted earlier, the output of Wrangler after a refactoring is a new file, from which we calculate a 'diff' set; it would be possible to modify Wrangler to produce a 'diff' set directly. We aim to investigate these modifications in the months to come, and to continue our overall project to integrate Wrangler as tightly as possible into Eclipse and Erlide.

[1] The Eclipse Java refactoring systems allows these file changes, but note that it was implemented prior to the definition of the LTK.

8. Conclusions and future work

It is clear that as we look at more advanced refactorings – such as those involving wholesale transformation of data representations, or others which address inter-process communication – then more complicated analyses are required. Indeed, we would contend that for these more advanced transformations it is impossible to make them automatic, and that the role of the refactoring tool becomes one of a refactoring assistant, which can provide support for various aspects of the refactoring process, rather than a completely automated process. Perhaps this should be no surprise, as this is the case in machine proof, where theorem-provers and proof assistants co-exist, and there is more than a little in common between meaning-preserving refactoring and proof. We therefore expect that our work will take us towards more complex, user-driven, interactions.

We also see in the work that we report there is a substantial investment in infrastructure in any tool building of this sort. While it may not be evident from the high-level report of the Eclipse integration that we provided, the project report [26] shows this was not a trivial, or even a straightforward exercise, and considerable work remains to be done. Nevertheless we expect to contribute our refactoring tools to the general Erlide project, which shows great promise.

On the same theme we and the team from Eötvös Loránd University hope to evolve a common infrastructure between our two systems, so that user can take advantage of the two in a seamless way. This common infrastructure will also allow us to test the two systems against each other.

The Kent team would like to acknowledge the support of the UK EPSRC in funding work on Wrangler, as well as support provided by Vlad Dumitrescu for his work on Erlide, the members of the Erlide development mailing list, and the members of the Eclipse JDT development mailing list for support provided in the port of Wrangler to Eclipse.

References

[1] Eclipse - an open development platform. `http://www.eclipse.org/`.

[2] Erlide - the Erlang IDE. `http://erlide.sourceforge.net/`.

[3] The Emacs Editor. `http://www.gnu.org/software/emacs/`.

[4] The Vim Editor. `http://www.vim.org/`.

[5] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

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

[7] D. Byrne. Integrating Java and Erlang. `http://www.theserverside.com/tt/articles/article.tss?l=IntegratingJavaandErlang`.

[8] R. Carlsson. Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3`, 2004.

[9] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley, 2006.

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

[11] L. Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. 2006.

[12] L. Gorrie. Distel: Distributed Emacs Lisp (for Erlang). In *The Proceedings of Eighth International Erlang/OTP User onference*, Stockholm, Sweden, November 2002.

[13] S. Holzner. *Eclipse Cookbook*. O'Reilly, 2004.

[14] H. Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Kent, UK, September 2006.

[15] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, August 2003.

[16] H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. Di Penta and L. Moonen, editors, *SCAM2006*, 2006.

[17] H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007*, Freiburg, Germany, September 2007.

[18] H. Li and S. Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In P. Achten, P. Koopman, and M. T. Morazn, editors, *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming(TFP)*, The Netherlands, May 2008.

[19] H. Li and S. Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, San Francisco, California, USA, January 2008.

[20] H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.

[21] Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring Erlang Programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.

[22] László Lövei, Zoltán Horváth, Tamás Kozsik, and Roland Király. Introducing Records by Refactoring. In *Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 18–28, New York, NY, USA, 2007. ACM.

[23] László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring Erlang Programs. *Periodica Polytechnica – Electrical Engineering (to appear)*, 2007.

[24] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[25] Tamás Nagy and Anikó Víg. Erlang Refactor Tool. Master's thesis, ELTE, Budapest, Hungary, 2007.

[26] G. Orosz. The Eclipse Integration of the Wrangler Erlang Refactor Tool. Technical report, Computing Laboratory, University of Kent, UK, 2008.

[27] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[28] P. Diviánszky R. Szabó-Nacsa and Z. Horváth. Prototype Environment for Refactoring Clean Programs. In *The Fourth Conference of PhD Students in Computer Science*, Szeged, Hungary, 2004.

[29] L. Felber T. Corbat and M. Stocker. Refactoring Support for the Eclipse Ruby Development tools. Master's thesis, University of Applied Sciences, Rapperswil, Switzerland, 2006.

[30] M. Weiser. Program Slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Gradual Typing of Erlang Programs: A Wrangler Experience

Konstantinos Sagonas Daniel Luna
School of Electrical and Computer Engineering, National Technical University of Athens, Greece
Department of Information Technology, Uppsala University, Sweden
kostis@cs.ntua.gr daniel.luna@it.uu.se

Abstract

Currently most Erlang programs contain no or very little type information. This sometimes makes them unreliable, hard to use, and difficult to understand and maintain. In this paper we describe our experiences from using static analysis tools to gradually add type information to a medium sized Erlang application that we did not write ourselves: the code base of Wrangler. We carefully document the approach we followed, the exact steps we took, and discuss possible difficulties that one is expected to deal with and the effort which is required in the process. We also show the type of software defects that are typically brought forward, the opportunities for code refactoring and improvement, and the expected benefits from embarking in such a project. We have chosen Wrangler for our experiment because the process is better explained on a code base which is small enough so that the interested reader can retrace its steps, yet large enough to make the experiment quite challenging and the experiences worth writing about. However, we have also done something similar on large parts of Erlang/OTP. The result can partly be seen in the source code of Erlang/OTP R12B-3.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contract; F.3.3 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Documentation, Languages, Reliability

Keywords Erlang, software defect detection, contracts, Dialyzer

1. Introduction

Almost all Erlang applications have so far been written without type information being explicitly present in their code. Of course, this is hardly surprising. After all, Erlang is a dynamically typed language where type information is only implicit during program development. Program testing typically uncovers many typos and type errors and these are corrected in the process. In many cases, type information in the form of (Edoc) comments is added in programs in order to document the intended interfaces of key functions and modules which are part of the API.

In our experience, this mode of developing Erlang programs is far from ideal. Even after extensive testing, many typos and type errors remain in the code. Often these errors appear in the not so commonly executed paths such as those handling serious error situations. Also, type information in the form of comments is often unreliable as it is not checked regularly by the compiler. Such documentation sooner or later is bound to suffer from code rot.

For a number of years now we have been trying to ameliorate this situation by developing and releasing tools that support and promote a different mode of program development in Erlang. Namely, one where most typos, type errors, interface abuses and other software defects are identified automatically using whole program static analysis rather than testing, and where type information is automatically added in the program code, becomes a part of the code, is perhaps manually refined by the programmer and is subsequently automatically checked for validity after program modifications. What's interesting in our approach is that all these are achieved *without* imposing any (restrictive) static type system in the language. Instead, programs can be typed *as gradually as desired* and the programmer has total control of the amount of type information that she wishes to expose and publicly document.

During the last year, we have been practicing this approach on a considerably large part of the Erlang/OTP system. Indeed, nowadays the entire code of the Dialyzer and Typer tools, a large part of the code of the High Performance native code compiler for Erlang (HiPE), and many modules of the standard libraries of Erlang/OTP R12B-3 come with explicit type information. The process has uncovered many software defects, identified some dubious interfaces and a significant number of discrepancies between the published documentation and the actual behavior of key functions of the standard libraries. In the code of Erlang/OTP, the whole process has often been slow and painful, partly because one has to worry about maintaining backwards compatibility and partly because it involves a considerable amount of communication with the Erlang/OTP developers. Nevertheless, overall it has been very rewording and clearly worth its while. The resulting code is cleaner, easier to understand and maintain, more robust, and much better documented.

This paper aims to document in detail the steps of the program development mode we advocate and have been practicing all this time; both on code produced by our group and on code of Erlang/OTP. By doing so, others who are possibly interested in gradually typing existing Erlang applications can explicitly see what's involved in the process. In particular, they can see both the benefits and costs of using our tools as well as many pitfalls that the more "traditional" mode of Erlang code development involves.

We decided to start with a handicap: we do this experiment on code that we did not write ourselves and for that reason possibly not fully grasp. Also, for the experiment to be interesting, we wanted that the code should be of significant size and publicly available so that others can retrace our steps. After looking around at a handful of open source Erlang projects, we opted for the code of Wrangler, a refactoring tool for Erlang [3].

Erlang'08, September 27, 2008, Victoria, BC, Canada.

```
refac_atom_info.erl:715: Guard test length(M::atom()) can never succeed
refac_batch_rename_mod.erl:161: The call erlang:exit('error',[1..255,...]) will fail
             since it differs in argument position 1 from the success typing arguments: (pid() | port(),any())
refac_util.erl:921: Call to missing or unexported function refac_syntax:class_body/1
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())
```

Figure 1. The main defects of Wrangler 0.1 as identified by Dialyzer

The code of Wrangler has various interesting characteristics with respect to what we want to do. First, it has been developed by researchers who are experts in typed functional programming. For this reason, we expected that Wrangler's code base would be written in a type disciplined manner and would not contain (m)any type errors. Second, we expected that its code base would contain an interesting set of uses of higher order functions — possibly more than in most Erlang code bases out there — and this would be challenging for our tools and approach. Third, the authors of Wrangler have been heavily involved in a project related to testing Erlang programs and have used Wrangler in conjunction with sophisticated testing technology such as QuviQ's QuickCheck tool for Erlang [2]. Finally, the authors of Wrangler are aware of the tools of our group: as they acknowledge in Wrangler's homepage they 'make use some of the ideas from Dialyzer'. In short, we expected that this would be a relatively easy task. Let's see what we found.

2. Using Dialyzer on Wrangler

We started our experiment with the first action we recommend to any Erlang project: use Dialyzer [4]. Dialyzer is a static program analyzer that is really easy to use and is particularly good in identifying software defects which may be hidden in Erlang code, especially in program paths which are not exercised by testing. Indeed, as we will see below, it is quite common that these defects remain unnoticed for a long period of time.

2.1 The first experiment: Dialyzer on Wrangler 0.1

To learn something about Wrangler's evolution, we started by obtaining the first version of Wrangler, which was publicly released on the 25th of January 2007. We executed the following commands:

```
> wget http://www.cs.kent.ac.uk/projects/forse/wrangler/
        distel3.3-wrangler/distel-wrangler-0.1.tar.gz
> tar zxvf distel-wrangler-0.1.tar.gz
> cd distel-wrangler-0.1/wrangler
> wc *.erl
   2229    7247   73088 refac_atom_info.erl
            ... 24 more lines suppressed ...
  34784  137281 1198955 total
```

As we can see from the output of the last command, the main body of the code of Wrangler 0.1 contains a total of 25 modules comprising of about 35,000 lines of code. Out of these modules, many are modified versions of Erlang/OTP modules (of the `syntax_tools` application, the `compiler`, and two supporting modules of `dialyzer`).

We postponed making the Wrangler system because we wanted to shake its code first. Instead, we run Dialyzer v1.8.1 as follows:

```
> dialyzer --src -c *.erl
```

This analyzed all Wrangler modules and generated 67 warnings in less than 2 minutes. About 50 of these warnings concerned the `refac_epp` module and were warnings of the form `'Function F/A will never be called'`. Such warnings are typically side-effects of some failing or contract-violating function calls earlier in the same module which in turn makes calls to these functions unreachable. Indeed, these warnings were produced because Dialyzer also identified two calls to the `file:open/2` function which violate both its published documentation at `www.erlang.org` and the explicit type information which exists for this function in the source code of the `file` module of Erlang/OTP R12B-3. We manually modified the two offending calls to this function by changing them from the old-fashioned one:

```
file:open(Name, read)
```

which is still allowed for backwards compatibility to the more kosher and documentation-conforming one:

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

In the process, we performed a similar change to two calls to function `file:path_open/3`. Doing these changes took about two minutes of our time and reduced the number of Dialyzer warnings to 15. About half of these warnings concern modules `refac_compile`, `refac_sys_core_fold` and `refac_v3_core` which are clones of the corresponding modules of Erlang/OTP with only minor modifications. These warnings are genuine errors that have been fixed in Erlang/OTP R12B. We concentrate on four of the remaining warnings that are specific to the code of Wrangler. These warnings are shown in Figure 1.

The first of them concerns a guard that will never succeed. This typically signifies a genuine bug or is a sign of severe programmer confusion. Indeed, very few Erlang programmers fancy writing guards that always fail. In this case the Dialyzer warning identifies a programming error. The corresponding code is shown in Figure 2. As can be seen, `M` is an atom and the call to `length/1` will always fail in this case. However, since this call occurs in a guard context its failure is silenced and can easily remain undetected by testing.

```
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)))
      ...
```

Figure 2. Portion of the code of `refac_atom_info.erl`

The second warning identifies a call to the `exit` function with the wrong arity. The corresponding code checks for an error condition and if the condition is met it wants to exit the Wrangler process most probably with a tagged two tuple where the first element is the atom `error`. Instead, it constructs the call:

```
exit(error,"Can not infer new module names, ...")
```

This is a particularly nasty bug that is very hard to detect by testing. The problem is that this code will abort execution alright, but will do so with a significantly different message than the programmer

```
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;
```

Figure 3. Portion of the code of `refac_util.erl`

intended. (The `erlang:exit/2` function throws an exceptions and exits a process in Erlang but expects a different type of term in the first argument and will throw a different exception if called with an atom in the first argument.)

The third warning is simple but quite common in Erlang. The code contains a call to a non-existing function (of an existing module). One does not need Dialyzer to detect this error; the `xref` tool would also have detected it.

The last warning is the most interesting one. The corresponding code is shown in Figure 3. To somebody not very familiar with the idiocyncrancies of the Erlang parser this code looks correct. The problem is that and binds stronger than == in Erlang and so the case expression in the code is parsed as:

```
case (filelib:is_regular(File) and
      filename:extension(File)) == Ext of
```

that is, the code in Figure 3 effectively tries to test a boolean value with the value of Ext, instead of being parsed the way that the programmer intended:

```
case filelib:is_regular(File) and
    (filename:extension(File) == Ext) of
```

This bug can be fixed either by adding explicit parentheses as above or by using the `andalso` operator instead of and.

Overall, we spent about half and hour understanding and fixing the software defects of Wrangler 0.1 that were identified by Dialyzer. We started from this version of Wrangler because we wanted to see which of Wrangler's defects are long-lived and managed to survive from the first to the current release.

2.2 The second experiment: Dialyzer on Wrangler 0.3

At the time of writing this section (early June 2008), version 0.3 was the most recent snapshot of Wrangler. It was released on the 7th of January 2008, almost a year after version 0.1. The structure of Wrangler's source code has changed a bit and some of the modules of Wrangler 0.1 that were from Erlang/OTP are no longer present. However, many modules of the `syntax_tools` application are still present and some new modules have been added. Including those modules, Wrangler's code consists of 25 modules and about 27,000 lines of code. We run Dialyzer as follows:

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

After about 50 seconds, Dialyzer produced warnings many of which were in file `refac_epp` and were due to using an atom rather than a list for the options argument of calls to functions of the `file` module. After manually fixing this issue, about 20 warnings remained.

Some of these warnings were due to confusing one library function with another one and abusing its interface. The `lists` module provides a `concat/1` function. Its published documentation at `www.erlang.org` reads:

```
%% concat(L) concatenate the list representation of
%%  the elements in L - the elements in L can be atoms,
%%  numbers or strings.  Returns a list of characters.

-type concat_thing() ::
        atom() | integer() | float() | string().
-spec concat([concat_thing()]) -> string().

concat(List) ->
    flatmap(fun thing_to_list/1, List).

thing_to_list(X) when is_integer(X) ->
    ...
```

Figure 4. `lists:concat/1` function annotated with a contract

```
concat(Things) -> string()

  Types:
    Things = [Thing]
    Thing = atom() | integer() | float() | string()

  Concatenates the text representation of the elements
  of Things.  The elements of Things can be atoms,
  integers, floats or strings.
```

However, the current implementation of the `concat/1` function is more liberal than its documentation claims it is. For example, its implementation in Erlang/OTP R12B-3 allows calls where each `Thing` is a tuple:

```
Eshell V5.6.3  (abort with ^G)
1> lists:concat([[{a,1},{b,2}],[{c,3}]]).
[{a,1},{b,2},{c,3}]
```

Note that the result in this case is not a string. The code of Wrangler is relying on an undocumented behaviour of a library function.

Misunderstanding or abusing the interface of some library function is a very common software defect in dynamically typed languages such as Erlang. We consider this problem quite severe because an application might give the impression of working alright but this remains so *only* until the library has the same observable undocumented behavior. Of course, this is something that is not guaranteed by the library developers. We have noticed this phenomenon happening again and again — even in our own code! — in Erlang applications. For this reason, we have designed and proposed a *contract language* for Erlang [1] and have already annotated key libraries of Erlang/OTP with their documented interface. Indeed, in Erlang/OTP R12B, the corresponding code in the `lists` module reads as shown in Figure 4. Due to the presence of these contracts, Dialyzer can easily detect such interface abuses and warn the user about them.

In this particular case, the problem is easily fixed. The code of Wrangler can simply use the `lists:append/1` function which has the behaviour that its authors are after. There are 13 calls in total to `lists:concat/1` that should become calls to `lists:append/1`.

After this fix, Dialyzer reports 10 warnings in total. The main ones, those related to Wrangler files not from Erlang/OTP, are shown in Figure 5.

The first and last of them are familiar. They are identical to those in Wrangler 0.1 and have remained unaffected by code evolution and undetected by testing and uses of Wrangler. As mentioned, it is not very surprising that the first of them has remained undetected since the defect appears in error-detection code which is notoriously hard to exercise.

```
refac_batch_rename_mod.erl:161: The call erlang:exit('error',[1..255,...]) will fail
                since it differs in argument position 1 from the success typing arguments: (pid() | port(),any())
refac_duplicated_code.erl:441: The pattern {'error', _Reason} can never match the type 'false' | {'value',tuple()}
refac_fold_expression.erl:97: The pattern {'error', 'reason'} can never match the type {'error','none'} | {'ok',_}
refac_move_fun.erl:137: The pattern {'eror', Reason} can never match the type {'error',_}
refac_util.erl:921: Call to missing or unexported function refac_syntax:class_body/1
```

Figure 5. The main defects of Wrangler 0.3 as identified by Dialyzer

```
trim_clones(FileNames, Cs, MinLength, MinClones) ->
  ...
    case lists:keysearch(File1, 1, AnnASTs) of
      {value, {File1, AnnAST}} ->
        ...
      {error, _Reason} -> {false, {Range, Len, F}}
    end
  ...
```

Figure 6. Portion of the code of `refac_duplicated_code.erl`

The second warning is due to confusion about the possible return values of the `lists:keysearch/3` function. The offending code is shown in Figure 6. We have seen similar defects in various other Erlang code bases. The remaining warnings are simple typos in error checking code. Similar defects have a tendency to remain unnoticed for a long time.

We manually corrected these problems but for the last one (the call to the missing function) which we did not know how to fix. The whole process, including referring to Erlang/OTP's documentation and code to verify issues related to `lists:concat/1` vs. `lists:append/1`, took us a bit more than two hours. With an almost warning-free code base, we could start adding contracts to the code of Wrangler in order to robustify its API and in the hope of identifying more defects and interface abuses. Let's see where this got us.

3. Adding Contracts to Wrangler

The second action we recommend to any Erlang application is to expose as much type information about functions and modules as possible and make this information part of the code. Typically, type information is only implicit in most Erlang programs. Making it more explicit can happen in the following two ways:

Add explicit type guards in key places in the code. Such an action has the advantage that it exposes type information to static analysis tools such as Dialyzer and at the same time ensures that calls to these functions will fail if they violate these type tests during program execution. One disadvantage is that there is a runtime cost associated with this action, but this cost is typically quite small. A more serious disadvantage is that programs may not be prepared to gracefully handle such failures.

Add type declarations and contracts. Type declarations can give convenient names to key data structures which can then be used to document function and module interfaces. Such type information can then be used by Dialyzer to detect interface violations without occurring any runtime overhead. Quite often such information already exists in comments: either in Edoc format or even in plain text.

Of course, these two methods of exposing type information are not mutually exclusive and projects can employ the combination that is best suited for each situation in hand.

In the case of Wrangler 0.3, its source code already contains a fair amount of `@spec` annotations (336 in total). However, the bulk of these annotations is in files that are minor modifications of Erlang/OTP modules. Because for the more up-to-date version of some of these modules (the ones in Erlang/OTP R12B-3) we had already performed a similar action to the one we will describe in this section, we decided to focus on the `@spec` annotations in modules that have been written entirely by Wrangler's authors. There are 15 such modules but three of them (`refac_module_graph`, `wrangler_distel` and `wrangler_options`) contain no annotations. In the remaining 12 modules there are 54 `@spec` annotations in total. Their breakdown according to module is shown in Table 1.

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

Table 1. Number of `@specs` in modules of Wrangler 0.3; modules with no `@specs` and modules from Erlang/OTP have been excluded

3.1 Turning `@spec` annotations into `-spec` declarations

At least syntacticly, converting an existing `@spec` annotation into a `-spec` declaration is a rather straightforward procedure. For example, in `refac_batch_rename_mod.erl` the `@spec` annotation:

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

can immediately be turned into:

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

The single quotes around the atoms are not really needed, but we recommend their use so that it is clear to the reader what e.g. is supposed to be the atom `'ok'`, which denotes a singleton type in the language of types, rather the `ok()` type where the programmer has mistakenly forgotten the parentheses.

Quite often, one also needs to make up names for types which are not built-in types. For example, `refac_duplicated_code.erl` contains the following `@spec` annotation:

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

which, after making some educated guess, can be turned into:

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

If one continues this way, she is quickly faced with a problem. Because `@spec` annotations are not routinely checked by the compiler or any static analysis tool, many of them have suffered from severe code rot and have become inaccurate, outdated, or even completely wrong. For example, to be correct, let alone precise, the above `-spec` declaration should actually read:

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

Note that the problem is not in the type declaration that we introduced but in that the original `@spec` annotation that the file contained is not correct.

Out of curiosity, we performed the following experiment. We converted all 54 `@spec` annotations of Wrangler 0.3 to `-spec` declarations and added very loose type declarations for type names which were not documented in the code: we basically mapped most of these types to `any()`. This makes the contracts containing these types as forgiving as possible. We then run Dialyzer on the Wrangler files. Dialyzer reported a total of 164 warnings! Recall that this was on a set of files which were warning-free without any `-spec` declarations. This is not the first time we experienced this behaviour: Edoc annotations need to be treated with caution.

In our experience, the 'convert all `@specs` at once' approach is very crude. The user is simply overwhelmed by the number of warnings that Dialyzer reports. We recommend the following approach instead.

Start from some easy files. Easy files are either those that do not contain many `@spec` annotations or those that depend on only few other modules. This way, one has the chance to run Dialyzer on a *single* module at a time and correct the defects that Dialyzer identifies on a module-local basis. Then continue this way until all modules have been processed. Note that this is not guaranteed to result with a set of files which, when considered together, can be analyzed by Dialyzer without any warnings. If the warnings that are produced are too many, then analyze the modules by considering the strongly connected components that they form, fix warnings in the process, and expand on this set until all modules can be analyzed warning-free.

Fixing warnings of only one or of a small set of modules is usually quite easy. For example, for the `refac_rename_var` module, one gets the following warning from Dialyzer.

```
refac_rename_var.erl:66:
  The call  cond_check(..., ..., NewName1::atom())
  breaks the contract (..., ..., NewName::string())
```

where one can immediately see that there is something wrong in the last argument of this function; either in the call on line 66 or in the contract of the function (i.e., the `-spec` declaration that we added). Finding out which of these two is to blame is a bit more tricky, especially if one is unfamiliar with the code. Quite often though the module has some code part that gives a strong indication about where to assign blame.

We followed the approach we describe above and converted all `@specs` to `-specs` ending up with a set of modules for which Dialyzer gave no warnings when run on a single module at a time. In the process we had to fix a total of ten erroneous specs out of the 54 original ones. The 'local' column of Table 2 shows how these are partitioned per module. We then run Dialyzer on the complete set of modules, which resulted in a total of 42 warnings. (In fact, only 17 if one excludes warnings that are quite clearly a side-effect of some other warning.) In any case, 42 is a much more manageable number than 164. Most warnings were due to eight additional specs in the code of Wrangler 0.3 being erroneous, which we also corrected. Their modules are indicated in the 'global' column of Table 2. The whole process took about six hours. Of course, it would have taken us less time had we been familiar with Wrangler's code.

module	`@specs`	wrong `@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

3.2 Fixing defects exposed by `-spec` declarations

When `-spec` declarations become part of the code, interesting software defects are exposed by Dialyzer. For example, the Wrangler file `refac_util.erl` contains the following `@spec` annotation:

```
@spec pos_to_var_name(Node::syntaxTree(), Pos::Pos) ->
        {'ok', {atom(), {Pos, Pos}}} | ...
```

To ease exposition, let us drop the variable names for referring to types, introduce a type declaration for what the authors of Wrangler denote as `Pos`, and fix this annotation so that its return type is actually correct. The intended specification for function `refac_util:pos_to_var_name/2` should read:

```
-type pos() :: {integer(), integer()}.
-spec pos_to_var_name(Node::syntaxTree(), Pos::pos()) ->
        {'ok', {atom(), {pos(), pos()}, cat()}} | ...
```

where `cat()` is some type. In `refac_rename_var.erl` this function is used as shown in Figure 7. In this code, Dialyzer warns that the equality test between `DefinePos`, which is a two tuple, and a singleton list will always fail. Once again, this is a very difficult bug to spot or discover by testing because it is in code which handles exceptional cases. (Under typical executions, the code goes to the `true` branch anyway.)

```
rename_var(Fname, Line, Col, NewName, SearchPaths) ->
  ...
  case refac_util:pos_to_var_name(AST, {Line,Col}) of
    {ok, {VarName, {_, DefinePos}, C}} ->
      if DefinePos == [{0,0}] ->
          {error, "Renaming of ... is not supported!"};
        true ->
          ... % code that renames the variable here
          case cond_check(AST1, DefinePos, NewName) of
            ...
```

Figure 7. Portion of the code of `refac_rename_var.erl`

Once this problem gets exposed, Dialyzer also warns about other problems further down in the code. Figure 8 shows a small portion of the code of the `cond_check/3` function. The call to `lists:any/2` demands that `Pos`, which comes from `DefinePos`

```
cond_check(Tree, Pos, NewName) ->
  ...
  BdVars = lists:map(fun(_, B, _) -> B end, ...),
  Clash = lists:any(fun(bound, Bds) ->
                      ...
                      F_Member = fun (P) -> ... end,
                      lists:any(F_Member, Pos) and ...
                    end, BdVars),
  ...
```

Figure 8. Portion of the code of `refac_rename_var.erl`

in Figure 7, is a list. This code will surely fail if ever executed. We could not decipher what exactly this `lists:any/2` call and two similar occurrences further down in the code of `cond_check/3` try to do, so we did the best action we could think of: we simply wrapped the `Pos` variables in a list. This silenced all but one Dialyzer warnings on the complete set of files of Wrangler 0.3.

3.3 Strengthening and factoring `-type` declarations

Since we were unfamiliar with Wrangler's code, when adding contracts we initially mapped most types mentioned in `@spec` annotations (like for example the types `syntaxTree()` and `cat()` in the example of the previous section) to the type `any()`. This is the most general type of the type system, representing the set of all Erlang terms. Mapping these types to `any()` has the property that Dialyzer will not report any contract violations due to a mistake in the definitions of these types. On the other hand, it is clear that in most cases these type names denote only a subset of all Erlang terms and mapping them to `any()` is a gross overapproximation. We can and should do better than that.

However, unless one is pretty certain about the values of types, we recommend that initially one is not overly zealous in constraining them. The reason is that over-constrained type declarations can result in a lot of warnings from Dialyzer. As a result, it might be quite hard to find the culprits and correct these warnings in conjunction with erroneous `-spec` declarations. We instead recommend that one first tries to come to a state where the existing `-spec` declarations do not result in any warnings from Dialyzer and only then start constraining the types. Indeed, this is the approach we followed when typing Wrangler.

Sometimes, Edoc `@type` annotations already exist in the files and these can be changed to the corresponding `-type` declarations. Some other times, type declarations are pretty obvious, as e.g. for the case of the `filename()` type that we mapped to `string()`. Finally, often information about types exists in comments or types are pretty clear from the structure of terms and the names of variables. This is for example what we did for `pos()`. In various parts of the code, it was mentioned that this type denotes a pair of integers. Thus, we initially added the declaration:

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

and corrected the warnings reported by Dialyzer. None of them was related to this declaration. Then, looking deeper in the code, we realized that `pos()` denotes the line and column numbers of a position in the program source; the position `{0,0}` was used to denote the default position or the absense of position information. We subsequently refined its declaration to exclude negative integers:

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

For safety, a Wrangler programmer might want to further constrain this type to appropriate integer ranges for lines and columns that a source file might contain. For example, the above declaration can be refined to:

```
-type pos() :: {0..100000, 0..200}.
```

In short: like applications, types can be *gradually* refined and strengthened up to the point that the programmer wishes to expose information about sets of values and impose constraints on their uses. This way, programs can protect themselves from accidentally violating these constraints.

Once types are declared, often one notices that the same type definition appears in more than one file. For example, the above type declaration for `pos()` was added and refined in a total of four Wrangler files. It is of course bad software engineering practice to have the same information in different places in the code. One can either place this type definition in a common header file which can then be included by all files that need it, or place it in only one file, say `m.erl`, and then in all other files can use the notation `m:t()` to refer to this `t()` type definition that module `m` contains. For Wrangler, since a `wrangler.hrl` file already existed, we opted for factoring all type declarations that were used in more than one module to this header file.

3.4 Strengthening underspecified `-spec` declarations

The next step is to gradually strengthen some `-spec` declarations, because quite often many of them are underspecified. For example, in the code of Wrangler about a third of all `@spec` annotations specify a return type of `term()` for the corresponding functions. Obviously, this return type is not very precise; most of these functions return terms with a statically known structure.

Luckily, when specs become part of the code, there is an easy automatic way to discover the underspecified ones among them:

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

Running this command revealed a total of 19 underspecified `-spec` declarations (out of the 54 ones). This was after we strengthened the `-type` declarations; the number would have been 24 if we had not done so.

Correcting the underspecified declarations is quite easy. For example, for one of them Dialyzer reports:

```
refac_duplicated_code.erl:53:
  Type specification for duplicated_code/3 ::
    ([filename()],[byte()],[byte()]) -> any()
  is a supertype of the success typing:
    ([string()],[byte()],[byte()]) -> {'ok',[1..255,...]}
```

and of course it is a simple matter to change the return type in the `-spec` declaration of this function from `any()` to either the return type which is reported by Dialyzer (denoting a two tuple where the second element is a non-empty string) or to the slightly underspecified but much more readable type `{'ok',string()}`.

It is important to note that the success typing information reported by the `-Wunderspecs` option of Dialyzer is a conservative approximation of the behaviour of the function which is safe to use and can be copied and pasted in the file as is. Its use will never result in any additional Dialyzer warnings. Dialyzer does not really need its presence because it is the one that it infers. But there is a good reason to explicitly add this information in the file: it provides useful documentation and from that point on its consistency with the code can be statically checked by Dialyzer.

Sometimes this search for underspecified contracts uncovers repeated patterns which are so common that they deserve their own type declaration. For example, the Dialyzer call above revealed that many Wrangler files define an auxiliary function `application_info/1` that returns a two tuple of the form `{{_,_}, non_neg_integer()}`. Turns out that the two underscores are always atoms and the non-negative integer represents the arity of a function. We thus added the following type declaration:

```
-type appl_info() :: {{atom(),atom()}, arity()}.
```

```
> 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
 Unknown functions: [{refac_syntax,get_ann,1}, ...,
                     {refac_util,envs_bounds_frees,1}, ..., {refac_util,write_refactored_files,1}]
%% File: "refac_rename_var.erl"
%% ----------------------------
-spec pre_cond_check(tuple(),_,_,atom()) -> bool().
-spec rename(Tree::syntaxTree(),DefinePos::pos(),NewName::atom()) -> {syntaxTree(),bool()}.
-spec rename_var(FileName::filename(),...,SearchPaths::[string()]) -> {'ok',string()} | {'error',string()}.
```

```
> typer --show-exported -I ../hrl refac_rename_var.erl -T refac_util.erl
 Unknown functions: [{refac_syntax,get_ann,1}, ...,
                     {refac_util,parse_annotate_file,4},{refac_util,post_refac_check,3}]
%% File: "refac_rename_var.erl"
%% ----------------------------
-spec pre_cond_check(tuple(),non_neg_integer(),non_neg_integer(),atom()) -> bool().
-spec rename(Tree::syntaxTree(),DefinePos::pos(),NewName::atom()) -> {syntaxTree(),bool()}.
-spec rename_var(FileName::filename(),...,SearchPaths::[string()]) -> {'ok',string()} | {'error',string()}.
```

Figure 9. Finding missing contracts for exported functions of module `refac_rename_var` using Typer

in the header file of Wrangler although we could refine the two `atom()` types even further.

With the help of Dialyzer, many underspecified contracts can be strengthened more or less automatically. However, one should be aware that Dialyzer does not report all underspecified contracts. Instead, Dialyzer only reports those `-spec` declarations that are found strictly more general than the corresponding *success typings* that it infers for these functions [5]. If there exists even one argument position in the `-spec` declaration which is more specific than the corresponding success typing, Dialyzer will not report these declarations as underspecified. For this reason, one might want to manually inspect all `-spec` declarations to spot arguments and return values whose types are underspecified. In fact, this is what we did for Wrangler 0.3. After we corrected underspecified contracts which Dialyzer reported, we used `grep` to detect `-spec` declarations with an occurrence of the `term()` or `any()` type and manually corrected these. There were an additionally nine such `-spec` declarations. The whole process described in this subsection took about two hours to complete.

module	@specs	
	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

3.5 Adding `-spec` declarations for exported functions

To ease development and maintainability of Erlang applications, we recommend that modules contain `-spec` declarations for all their exported functions. This way, at least their public interface is documented and Dialyzer can detect possible violations. To help detect modules whose public interface is not documented, we introduced a new compiler option in Erlang/OTP R12B-3, called `warn_missing_spec`, which warns about missing `-spec` declarations for all exported functions of a module. We used this option on the files of Wrangler 0.3 which are not from Erlang/OTP. The number of existing and missing `specs` for exported functions for these modules is shown in Table 3. As can be seen, only half of the exported functions have a publicly documented interface.

With the help of this new compiler option and of the Typer tool the missing function specifications can also be generated semi-automatically. For example, Figure 9 shows the three commands we used to find the missing contract of module `refac_rename_var`. The first command uses the new compiler option to see the exported functions without specifications; there is only one of them in this module. Subsequently, Typer is used to generate specifications for all exported functions in this module. For all functions with existing specifications (e.g. functions `rename/3` and `rename_var/5` in this case) Typer is printing them as these appear in the file. But Typer also generates conservative approximations of specifications for the remaining functions. As can be seen, the first attempt to generate such a specification for function `pre_cond_check/4` was only partly successful. The generated specification contains no type information for the second and third argument of the function because Typer also complained that it does not know anything about functions of modules `refac_syntax` and `refac_util` that the `refac_rename_var` module is using. By instructing Typer to *trust* the existing function specifications of file `refac_util.erl` (but recall that this module has specifications for only half of its functions), Typer is able to infer an accurate specification for function `pre_cond_check/4`.

Actually, in this particular case, we happened to be somewhat lucky. Module `refac_util` contained type specifications which are sufficient for Typer to infer a relatively accurate type information for `pre_cond_check/4`. However, often this is not the case. In those situations, we recommend that the user starts from *leaf* modules (i.e., modules which do not call functions from other modules), use Typer to annotate their exported functions with contracts, and continue bottom up in the module dependency graph until all modules are annotated with contracts.

One can even be brave and use the `--annotate` option of Typer, which will automatically insert the generated `-specs` in the source code of the file(s) on which Typer is run.

Of course, one must always keep in mind that the specifications that Typer generates are conservative approximations (in fact, they are *success typings*) and will never contain any constraints that are not present or enforced by the source code of the module. In other words, these automatically generated specifications are correct but possibly imprecise. In most cases, the user needs to refine them manually, both in order to strengthen them and in order to use appropriate type names for their arguments. For example, the occurrence of `tuple()` in the specification of `pre_cond_check/4` denotes a `syntaxTree()`.

4. Contacting the Authors of Wrangler

At this point, instead of proceeding on our own, we decided to get in touch with the authors of Wrangler. We sent them our paper with the information it contains up to this point.

In the beginning of July 2008, the code of Wrangler had been extended and somewhat changed compared with the version of January 2008 that we were looking at, but most of our steps could easily be retraced even in the development version of Wrangler. The Wrangler authors confirmed our findings. They also added `-spec` declarations for most exported functions of Wrangler modules. Unfortunately, they added these specifications in one go and were subsequently confronted with many Dialyzer warnings that they could not figure out their cause. So, they asked for our help. Of course, the culprit was that some of the `-specs` that they added were in conflict with the functions' uses. In other words, the Wrangler authors did not only confirm our findings but also corroborated our opinion that converting all `@spec` annotations into `-spec` declarations in one go is something not recommendable in code bases of significant size.

With our help, the erroneous function specifications which were resulting in warnings from Dialyzer were corrected. There were eight of them in a total of about 150 `-spec` declarations. In the process, some of the specifications written by the authors of Wrangler were tightened and a few more were added by us. The end result was a Wrangler code base which was totally free from Dialyzer warnings, more robust, and with better documentation about its main functions. The Wrangler authors were happier but we were still not fully satisfied...

5. Testing Contracts of Wrangler

What troubled us was the following. Because Dialyzer's analysis is conservative and based on approximations, Dialyzer never reports a code discrepancy if it is not absolutely certain that there is something wrong with the code. In particular, all `-spec` declarations are trusted and are assumed correct unless Dialyzer discovers a clear conflict between their definitions and uses. For functions with no calls, for functions whose calls are with arguments whose types are not precise enough, or in cases where the return value is not involved in any explicit pattern matching, contract violations will not be detected or reported.

For this reason, we have created yet another tool that, given a test suite, dynamically checks the validity of `-spec` declarations in a set of files. This tool is not yet publicly available and its interface is subject to changes so we will only describe its main idea here.

Currently, the tool starts with a set of `.beam` files and a test suite which can be called from some top-level function (e.g. `mytest:run/N`) possibly with some arguments. For all files which have been compiled with `debug_info` on (and thus whose `-specs` are retained in the byte code), it will employ runtime monitoring to check the validity of their contracts and record all violations it detects while the test suite is running. The recording of all contact violations happens using the Erlang error logger and can be saved in a file, if so desired. The contract checker is straightforward to use for code bases with an already existing test suite. The only drawback, albeit a serious one, is that the test suite will run significantly slower. However, because all calls to contract-annotated functions originating from non debug-compiled modules will not be checked, the user can fully control which parts of the code base will be contract checked and the amount of runtime overhead to the test suite.

The authors of Wrangler provided us with a small test suite that we used to test the validity of `-spec` declarations in files that were somehow "touched" by this test suite. These files contained a total of 106 `-specs` out of which 55 were checked at least once; the remaining 51 concerned functions that were not called by the test suite. The contract checker detected a total of six contract violations: two in calls to functions and four cases where functions returned a value of different type than promised.

Two of the contract violations involved functions `get_toks/1` and `concat_toks/1` of the heavily called `refac_util` module. They were both due to an erroneous declaration of the `token()` type by the Wrangler authors. This type was declared as:

```
-type token() :: {'var', pos(), atom()}
               | {'integer', pos(), integer()}
               | {'float', pos(), float()}
               | {'char', pos(), char()}
               | {'string', pos(), string()}
               | {'atom', pos(), atom()}
               | {atom(), pos()}
```

but failed to account for the fact that the lexical analyzer also returns white spaces and comments as tokens. We extended this declaration by including the following two cases:

```
               | {'whitespace', pos(), whitespace()}
               | {'comment', pos(), string()}.
```

and added an appropriate definition for the `whitespace()` type.

The `refac_util` module contained another contract violation. The function `get_bound_vars/1` was declared as:

```
%% @doc Return the bound variables of an AST node.

-spec get_bound_vars(Node::syntaxTree()) -> [atom()].
get_bound_vars(Node) ->
  get_bound_vars_1(refac_syntax:get_ann(Node)).
```

failing to account for the fact that a variable annotation can occasionally be a two tuple containing an atom and a position (e.g. `{'Self',{77,11}}`).

The forth violation concerns function `fold_expression/3` of the `refac_fold_expression` module. Its contract reads:

```
-spec fold_expression(filename(),integer(),integer()) ->
        {'ok', [filename()]} | {'error', string()}
```

but it is clear from the code, shown in Figure 10, that this function returns something different than a list of filenames (strings) when the last argument to the `fold_expression/4` function is `emacs`.

A similar, though not the same violation, concerned the return type of `refac_move_fun:move_fun/6`. Finally, the last violation was detected in the contract of function `refac_gen:generalise/5` whose last argument was erroneously specified as being a `dir()` when in fact it should be `[dir()]` (i.e. a list of directories).

After the corresponding changes, the contract checker reported no violations when running Wrangler's test suite. Of course, this does not mean that Wrangler's contracts were not erroneous anymore. Instead, it just means that contracts which were exercised by the test suite accurately reflect their common uses.

```
fold_expression(FileName, Line, Col) ->
    fold_expression(FileName, Line, Col, emacs).

fold_expression(FileName, Line, Col, Editor) ->
  case refac_util:parse_annotate_file(FileName, true, []) of
    {ok, {AnnAST, _Info}} ->
       ...
       Candidates = search_candidate_exprs(AnnAST, FunName, FunClauseDef),
       case Candidates of
         [] -> {error, "No expressions that are suitable for folding against ..."};
          _ -> Regions = case Editor of
                           emacs ->
                             lists:map(fun({{{StartLine, StartCol}, {EndLine, EndCol}},NewExp}) ->
                                           {StartLine, StartCol, EndLine, EndCol, NewExp, {FunClauseDef, ClauseIndex}}
                                       end, Candidates);
                           eclipse -> Candidates
                         end,
               {ok, Regions}  %% or {ok, FunClauseDef, Regions}? CHECK THIS.
       end;
    {error, Reason} -> {error, Reason}
  ...
```

Figure 10. Portion of the code of `refac_fold_expression.erl`

6. Concluding Remarks

In this paper we described in detail the steps needed to gradually type the code base of an existing Erlang application. We carefully documented the methodology we advocate, the effort that is requires, and the pitfalls that it may involve. In most code bases the process is far from straightforward, but with the help of the static and dynamic analysis tools we have developed it can at least be performed semi-automatically.

In our experience, what we have described for the code base of Wrangler in no way refects on its quality as an application. In fact, it is quite typical for most Erlang applications out there on which we have applied Dialyzer. Type information is not a panacea, but having it as part of the code helps in catching some easy to detect programming errors, documents intended uses of functions and results in code which is easier to understand and whose correctness is easier to maintain.

Acknowledgements

The research of the second author has been supported in part by a grant from the Swedish Research Council (Vetenskapsrådet). We thank Huiqing Li and Simon Thompson for confirming our findings, giving us access to their repository and sending us a test suite for Wrangler.

References

[1] M. Jiménez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in Erlang and its interaction with success typings. In *Proceedings of the 2007 ACM SIGPLAN Erlang Workshop*, pages 11–17, New York, NY, USA, Sept. 2007. ACM Press.

[2] H. Li and S. Thompson. Testing Erlang refactorings with QuickCheck. In *Pre-proceedings of Implementation of Functional Languages*, Sept. 2007.

[3] H. Li and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 199–203. ACM Press, Jan. 2008.

[4] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[5] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.

Refactoring Module Structure *

László Lövei Csaba Hoch Hanna Köllő Tamás Nagy Anikó Nagyné Víg Dániel Horpácsi
Róbert Kitlei Roland Király

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{lovei,hoch,khi,n_tamas,viganiko,daniel_h,kitlei,kiralyroland}@inf.elte.hu

Abstract

This paper focuses on restructuring software written in Erlang. In large software projects, it is a common problem that internal structural complexity can grow to an extent where maintenance becomes impossible. This situation can be avoided by careful design, building loosely coupled components with strictly defined interfaces. However, when these design decisions are not made in the right time, it becomes necessary to split an already working software into such components, without breaking its functionality. There is strong industrial demand for such transformations in refactoring legacy code.

A refactoring tool is very useful in the execution of such a restructuring. This paper shows that the semantical analysis required for refactoring is also useful for making suggestions on clustering. Existing analysis results are used to cover the whole process of module restructuring, starting with planning the new structure, and finishing by making the necessary source code transformations.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Design, Languages

Keywords Erlang, refactoring, clustering modules

1. Introduction

In a large software, there are many interconnected program entities such as modules, functions, records, macros that form a complex graph. Sometimes this graph grows to the magnitude that no programmer is fully aware of the structure of the program code. Such code is hardly maintainable, therefore the entities must be grouped into smaller sets, called clusters, so that each cluster is small enough to be maintained effectively.

Refactoring [4] is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. A refactoring tool is the perfect means to analyse the connections between the entities and then automatically suggests one or more clustering solutions for the user of the tool. After this, the code can be restructured according to the suggested clusters. The restructuring of the code can be done manually or preferably by using the refactoring tool for this task.

Earlier papers [8, 9, 11] have reported on the design and implementation of two refactoring tools developed in cooperation by Eötvös Loránd University, Budapest, and University of Kent, Canterbury for the functional programming language Erlang and the OTP middleware.

Our refactoring tool, named RefactorErl [6], creates a formal semantical graph model from Erlang source code and stores the graph in a relational database. The graph contains semantical analysis results, it can be modified on the syntax tree level, and the original source code is reproducible from there. The tool has a user interface provided as an Emacs minor mode to help performing refactoring steps.

This paper presents the design of an analysis and restructuring method of large Erlang programs using the existing framework of RefactorErl.

1.1 Overview of Erlang

Erlang/OTP [3] is a functional programming language and environment developed by Ericsson, designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency, whereas OTP is a set of design principles and libraries that support building fault-tolerant systems [2].

The structural elements of Erlang are relatively simple. There are no classes and inheritance, no complex type definitions, and no global variables. Program code consists of functions which are organized into modules. Modules have a list of exported functions, these can be called from other modules; non-exported functions can only be used locally. Variables are always local to a function, they have no type declaration, and they get a value once which can't be changed.

The only complicating factor is the presence of a preprocessor, which handles file inclusions, macro substitutions, and conditional compilation. Macros are the only means to create global named constants (which are essential in protocol descriptions), and they are usually placed in header files to be included in more modules. During restructuring, when functions are moved between modules, these file inclusions have to be taken care of to make necessary macro definitions available.

There is one more language construct that is worth mentioning. Although Erlang is a weakly typed language, there is one type declaration construct: *records* are widely used to describe process states, act as database table rows, and in many other places to simplify usage of large data constructs. Their impact on restructuring is the same as of macros, they must be made visible using file inclusion at every place of usage.

* Supported by ELTE IKKK and Ericsson Hungary.

Erlang'08, September 27, 2008, Victoria, BC, Canada.

1.2 Module restructuring workflow

Now suppose we have a large software written in Erlang, with too many internal dependencies. How can we split it into a few smaller, more maintainable parts based on module functionality?

First, we need to explore every single dependency in the software. This is where RefactorErl excels, see Sec. 2. for a summary of our previous work, an overview of the refactoring tool which hosts the analysis and restructuring tools, and introduction to the call graph analysis work that was done to support clustering.

Using these dependencies, we can group modules which show similar functionality. Existing clustering algorithms are used in this phase, Sec. 3. analyzes the possibilities and arising problems.

When we have several different clusterings, we need to choose the best one. A fitness function is used for this purpose, which gives a higher score for better clusterings – details are in Sec. 4.

There is one last design step. Those parts of the software which can't take part in clustering—typically library modules and header files, which are used in too many modules—need to be distributed into the clusters. This is done by splitting these files, moving every part into the cluster which uses it, as described in Sec. 5.

This last part requires code transformation, which can be supported by a refactoring tool. The refactoring steps necessary for this work are collected in Sec. 6.

2. RefactorErl for analysis and refactoring

From the refactoring point of view, the most important characteristic of a programming language is the extent of semantical information available by static analysis. As Erlang is a functional language, most language constructs can be analysed easily. Side effects are restricted to message passing and built-in functions, variables are assigned a value only once in their life, and the code is organised into modules with explicit interface definitions and static export and import lists.

However, Erlang has certain features that make it hard, or even impossible to understand the behaviour of a program by static analysis. Many rules of the language (for example the typing rules) fall into the category of dynamic semantics. In some cases the meaning of certain constructs can only be found out during runtime. A prominent example is the dynamic dispatch of functions, and running dynamically constructed code is also possible. These features, for example make it impossible to give a complete analysis of the call structure of an Erlang program statically.

In order to define a refactoring, one has to define the side-conditions (the conditions that are necessary for the safe execution of the refactoring), the transformation (the intended change on the code) and possibly the compensations (further transformations that should be performed to keep the refactored code consistent and preserve its behaviour). The operation of a refactoring is based on different sorts of static analysis that collect syntactic and semantical information. The obtained information can often be used by many refactorings. For example, all variable-related refactorings (rename a variable, eliminate a variable, merge subexpression duplicates etc.) require *scope* and *visibility* information on the variables.

We developed a refactoring framework [7] which is able to analyse and store any kind of Erlang source code, even code which contains non-parsable macros in a way that makes it possible to restore the original files, retaining whitespace and comments. We have improved the preprocessor in order to enable it to handle all macro structures encountered. Now industrial code can be processed while still retaining the full source code. Support of file inclusion allows of specifying multiple include file search directories, and application directories for supporting `include_lib` directives. Conditional compilation support is supported, too. The actual conditions are stored in a kind of semantic node called environmental node. These can also hold other kinds of configuration information such as include directories. Macro and record definitions are stored but not expanded, unlike include files.

There are several semantical analyses implemented. For example, when the tool loads a record definition into the graph (a form with type `record`), the analyzer module creates a semantical record object as child of the containing file (with link `record`) and links the definition to the semantical object with label `recdef`.

The records can be referred by expressions. When an expression refers to a record and there is no existing semantical object for this record, it is created automatically. The reference is represented by a link between the expression and the record with label `recref`.

2.1 Call graph analysis

Call graph analysis aims to give the exact function dependency relations in a given amount of code. This means that if the scope of the analysis is wider the result will be more accurate, but the smaller scope's call graph can never be invalidated by the new data. Just new edges can appear.

Data flow graph on the other hand aims to give back the flow of data through the functions. It shows how parameters, global variables are used and passed around the system.

In Erlang to have an accurate call graph, data flow analysis has to be done. This is the result of the highly dynamic nature of the language, a result of how functions are treated. It is possible for example to call functions from data we receive from different parts of the system using the `erlang:apply` built-in function. The source code of such data does not clearly show that it will be used in a function call. In different parts of the system, it can be used for a different purposes. This is because functions are identified by `atoms`. `Atoms` can be created dynamically with for example the `erlang:list_to_atom` function.

There are a subset of function calls which do not need data flow analysis. These are the static function calls. Where every element of the function call are known at compile time.

The dynamic calls – where the called function is not known at compile time – can be further categorized based on how much information is present at compilation time. Naturally, the less information given, the harder the analysis is. There are some edge cases where the analysis is impossible. In these cases the analysis' aim is to limit the possible functions which the call could refer to.

2.1.1 Static call analysis

The static analysis aims to create the function call graph of the static calls. This analysis is straightforward if we use the results of the semantical analysis which is incorporated in our refactoring system [8]. Collecting the function calls, and then sorting them based on which function they are in and which function they call is essentially the work that has to be done to create the call graph for the static calls.

By creating a different interface to retrieve the existing data, the clustering algorithm's data collection part can be easier. Of course this interface will be used to retrieve the dynamic function call data as well. By providing a common interface for the two different data we further ease the clustering algorithm's initial data collection complexity.

2.1.2 Dynamic call analysis

While building the static call graph is a relatively lightweight job, the dynamic call graph building takes significantly more time and resource. By default this part of the analysis is not done. The data retrieval does not change whether this part of the analysis is done or not. Of course the dynamic analysis possibly adds more data to the graph resulting in a more accurate call graph.

The analysis method is based on the Observer design pattern. It means that we have entities (variables, atoms, tuples etc.) which are loosely connected to each other. During the analysis, new connections and entities are created as well. Connections represent the dependencies between certain entities.

When an entity finds out more information about itself (for example a variable finds out its possible values), it sends this information to the other entities which are connected to it. Because connections can be created to an entity after its analysis is finished, the entity is not deleted after its analysis is finished.

Entities are modeled with Erlang processes, and data propagation with message sending. This approach creates the opportunity of parallel computation, because there is no strong order how the entities should processed. There is one further advantage which is re-computation after changes happen. If there are changes in the underlying code, the graph can be adjusted to it with creating and deleting entities and edges. In other words there is no need to re-compute the whole graph.

To start the analysis we need initial entities which will be further analysed. These are the dynamic calls unknown values. For example the values of the `erlang:apply/3` function's parameters. Further analysis is done by investigating entity types and surroundings. This could result in new edges and new entities which have to be further analysed and so on. When the value of the initial entities is found out with the analysis, it is made available through the same interface as the static call graph data.

3. Clustering modules and functions

The clustering algorithm sorts entities into groups. The entities can be modules or functions in the current implementation. The groups are called *clusters*.

We have chosen the hierarchical clustering algorithm [1] as the one which gives the most practical result: the user does not have to determine the number of clusters in advance, but can choose among the results with different cluster counts.

The main concept of the clustering method we use is the *attribute matrix*. The rows of this matrix are the entities, and the columns are the attributes of the entities (e.g. functions and records). One element of the matrix describes the relation between an entity and an attribute.

The clustering algorithm works on the attribute matrix. The algorithm can be parametrized with functions, which describe the distance of two entities and the properties of new clusters. This property can be either the distance of the new cluster from the existing clusters, or the attributes of the new cluster. The clustering algorithm works in the following way: In the beginning, each entity forms a separate cluster. Then, in each step, the two closest clusters are selected and unified. This process continues until there is only one cluster. The intermediate states contain a possible clustering of the entities. The output of the algorithm is the list of these possible clusterings. If the number of entities is n, this output contains n clusterings, and the ith clustering contains $n - i$ clusters.

3.1 Using the clustering algorithm

To obtain the result of the clustering algorithm, the following steps have to be done:

1. The attribute matrix has to be created.
2. The attribute matrix can be filtered.
3. The attribute matrix has to be transformed in certain cases.
4. The actual clustering has to be done.

Each of these steps is described in the following sections.

3.2 Creating the attribute matrix

Before running the clustering algorithm, the attribute matrix is created. The matrix describes the relation between the entities and attributes. The entities are either modules or functions. Attributes can be functions, records and macros. There may be also attributes such as size, which indicates the number of other entities represented by the entity. It becomes important when performing the clustering, because the clusters are represented as entities, as well, and these entities contain many modules.

3.2.1 Filtering the attribute matrix

The attribute matrix can be filtered before clustering the entities. During the filtering, the entities and attributes that are not wanted to take part in the clustering algorithm are removed from the matrix. Typically, library modules are removed from the entities and module internal functions are removed from the attributes. Filtering is done by a function that is parametrized with filtering functions, which describe which entities and attributes should be removed.

The two mentioned filtering algorithms are implemented, but the user can define filtering functions and `fun` expressions, as well.

3.2.2 Transformation of the attribute matrix

The clustering algorithm works with attribute matrices whose elements are numbers that describe the weight of the connection between the entity and the attribute. If an attribute matrix does not satisfy this condition, it has to be transformed to that form. A general transformer function can be used for this operation, which is parametrized by the transformation function that transforms one element of the matrix. Various transformation functions were tested, e.g. a function, with which the weight of the connection is that how many times the attribute is used by the entity.

3.3 Running the clustering algorithm

After creating the attribute matrix that contains weights, the clustering algorithm can be started.

Two clustering algorithms are implemented in the tool. Both algorithms work the way described in the beginning of section 3, but their exact ways of calculation are different. The first algorithm uses entity matrix in each step, the second one uses attribute matrix in each step. (They both use attribute matrix as a basis.)

The entity matrix user algorithm Using this algorithm the entity matrix needs to be created from the attribute matrix. The rows and columns of the entity matrix are entities. One element describes the distance between the two entities.

The entity matrix can be created from the attribute matrix using a function of the tool. The way of calculating distances needs be specified when calling this function. Functions are provided that can be used for this calculation, but the user can define own functions, as well.

The function that implements the rest of the algorithm is also parametrized: it is parametrized by the function that is used to calculate the distance of new clusters from the existing clusters. Functions that can be given as parameters are provided, but the user can also use own functions.

The attribute matrix user algorithm This algorithm does not use entity matrix, only attribute matrix. Two functions have to be specified as arguments: a function that calculates the distances between two entities, and a function that calculates the attributes of new clusters. The same distance calculator functions can be used as in the entity matrix user algorithm. For calculation of the attributes of new clusters, there are functions provided and the user can define own functions, too.

Distance calculator functions The distance calculator functions are used in both clustering algorithms. They calculate the distance of two entities based on the attribute matrix. Their arguments are the names and the attributes of the two entities.

We have implemented more than 10 distance calculator functions. There are some based on the literature, e.g. the 'Jaccard' distance calculator function, which is a generic distance calculator function based on the number of common attributes. There are others, based on function call structure and record usage. A kind of antigravity has also been implemented, because otherwise big clusters tend to grow more easily than small ones, and a few huge clusters will come to existence next to a lot of small ones.

Cluster distance calculator functions The cluster distance calculator functions are used in the entity matrix user algorithm. They are used to calculate the distance of new clusters from the existing clusters.

Their arguments are:

- Size of the first original entity.
- Size of the second original entity.
- Size of the other entity.
- Distance of the two original entities.
- Distance of the first original and the other entity.
- Distance of the second original and the other entity.

We have implemented 7 distance calculator functions. The one called `ward/6` seems to be working in the most sensible way. The advantage of this function is that it can prevent the clustering algorithm from creating one dominant buffer while there are a lot of clusters with very few elements (typically one). Instead, unification of large clusters is slowed, and unification of small clusters is quickened.

Cluster attribute calculator functions The cluster attribute calculator functions are used in the attribute matrix user algorithm. They are used to calculate the attributes of new clusters.

There are simple functions, e.g. in which an attribute of the new cluster is the sum of the same attributes of the original clusters. There are also more sophisticated ones, in which the calculation of the attribute may depend on the type of the attribute. The original attributes can be summed, merged, the average can be calculated, etc.

3.4 Experiences

Parametrization of clustering It seems to us that the attribute matrix user algorithm is better than the entity matrix user algorithm. It works better with our little, specific examples, where the correct clustering is easy to see.

Without taking into account the size of the clusters explicitly, the results show that usually one dominant cluster is created and many small ones around it. With using an appropriate weighting function, the results are satisfactory.

In the case of the entity matrix user algorithm, the distance calculator function `ward/6` succeeds to prevent creating one dominant and many small buffers.

Speed of clustering Clustering the modules of our refactoring tool (22 modules with 342 functions) takes 6 seconds. Clustering an industrial software with approximately 100 modules and 5000 functions takes two minutes. However, clustering the functions of our tool takes 37 minutes. This indicates that the run-time complexity of clustering is quadratic in the number of entities.

4. Fitness function

We have implemented various clustering algorithms, some of them with various parametrization possibilities, and the need to automatically compare these algorithms has arisen. The informal description of a good clustering was that elements logically close to each other should be in the same cluster and independent or loosely connected elements should be in different clusters. We assumed that logical closeness can be approximated by elements of program code such as function calls between modules, joint record usage and possibly macro usage.

We have constructed a fitness function that, from a result of a clustering (a list of clusters), computes the fitness value – a real number so that a better clustering gets a higher fitness value.

4.1 MQ metric

In the clustering literature, there is a widespread method for measuring the fitness of a given clustering, namely the MQ metric [5]. This metric is independent of the entities of the clustering task, it is based on only graph-theoretical notations.

4.1.1 First approach of implementing the MQ metric

Cluster factor Assume that there are n clusters, numbered from 1 to n. For each cluster i we compute the cluster factor, CF_i, as follows:

$$CF_i = \frac{2\mu_i}{2\mu_i - \sum_{j=1}^{n}(\varepsilon_{ij} + \varepsilon_{ji})}$$

where ε_{ij} is the number of connections between cluster i and cluster j. (Each pair of modules is either connected or not.) μ_i is the number of connections inside cluster i. (Each pair of modules is either connected or not.)

After this, the MQ is defined as the sum of cluster factors for each cluster.

$$MQ = \sum_{i=1}^{n} CF_i$$

Note: if we compute the MQ metric of a very disadvantageous clustering, it is possible that there will be far more external connections than internal ones, and this will lead to negative clustering factors. This is unwanted because it distorts the fitness-space, i.e. out of two disadvantageous clusterings the better can have less fitness value than the worse. To avoid this, we set the cluster factor to 0 whenever it would be negative.

Range of fitness values The MQ metric (in this approach) is a nonnegative number for every possible clustering.

Special cases:

- There is only one cluster, and every module belongs to it. The fitness value for this clustering is 1.
- Each module is in a separate cluster. The fitness value for this clustering is 0, because there are no internal connections at all.

In general, the fitness value is a nonnegative number, with no upper border.

Disadvantages There is a large group of clusterings which have their fitness value 0. Usually all of these are very disadvantageous clusterings so we do not have to deal with this, since we are interested in the few most fit clusterings. But there are cases in which we might want to compare even these very bad clustering results and therefore we need a fitness function which gives useful information on these clusterings, too.

4.1.2 Second approach of implementing the MQ metric

There is another version of the MQ metric in literature which gives the same fitness-space as the previously defined one, i.e. if one

clustering is better than the other in the first version, then it will be better in the second too, and vice versa.

Intraconnectivity of a cluster For each cluster we calculate the intraconnectivity value, which, for a given cluster i is the following:

$$A_i = \frac{\mu_i}{N_i^2}$$

where μ_i is the same as before, and N_i is the number of modules in cluster i.

Interconnectivity of two clusters For each pair of clusters noted with i and j, the interconnectivity value is:

$$E_{ij} = \begin{cases} 0 & i = j \\ \frac{\varepsilon_{ij}}{2N_iN_j} & \text{otherwise} \end{cases}$$

where ε_{ij} is the same as above, and N_i, N_j is the number of modules in cluster i, and cluster j respectively.

Finally, the MQ metric is defined as:

$$MQ = \begin{cases} A_1 & n = 1 \\ \frac{1}{k}\sum_{i=1}^{n} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i=1}^{n} E_{ij} & \text{otherwise} \end{cases}$$

This version of the MQ metric gives a real number between -1 and 1 for every possible clustering. The A_i and E_{ij} parameters are always between 0 and 1, representing the number of existing connections out of all the possible ones.

4.2 Towards an evolutionary algorithm

The fitness function can serve as the key component of some evolutionary algorithms to do find an optimal clustering. We plan to implement several evolutionary algorithms based on this fitness function, such as *steepest-ascent hill climbing*, *simulated annealing* and *genetic algorithm*.

5. Splitting modules and header files

Library modules and header files usually cannot be sensibly put into one cluster, because their contents are used in more clusters. However, if we look at their contents one-by-one, we often find that they are used only in one cluster, so it makes sense to split up these files using this information.

The splitting algorithm sorts functions, records, and macros of modules or header files into smaller files depending on their relations. Functions, records and macros will be called *objects* in this section.

The algorithm implemented in the Refactorerl tool is based on which clusters use which objects, and which objects use which objects. Each of the given modules and header files will be split independently from each other, so let us consider only one of the modules. Splitting header files works in the same way as splitting modules.

After the clustering, there are n clusters. Each cluster contains one or more modules, but none of the clusters contain the module to be split. The aim of splitting is that those objects that are used by one cluster only should be placed into a new module that belongs to it. More precisely, if an object is not used by any other cluster, even indirectly, it can be placed in this new module. With this method the number of dependencies outside the clusters can be reduced.

The number of new modules may vary between 1 and $n + 1$. If every object is used by more than one clusters, the splitting will leave everything in its place. However, if every cluster has objects in the library module which are only used by it, and there are objects which are used by more than one clusters, then a new module will be created for each cluster and the original module will remain with the remaining objects.

The algorithm calculates how to split the graph, but it does not make the actual changes. It only returns the suggestion for splitting.

5.1 An example of using the library

Let us suppose that these are the test files:

```
% a1.erl
-module(a1).
-export([f/0]).
-include("h.hrl").

f() ->
    lib:f2(), #r_a1{}, #rh1{}.

% a2.erl
-module(a2).
-export([f/0]).

f() -> lib:f1(), lib:f2(), lib:f5().

% a3.erl
-module(a3).
-export([f/0]).
-include("h.hrl").

f() -> lib:f4(), lib:f5().

% h.hrl
-record(rh1,{something}).
-record(rh2,{something}).
-record(rh3,{something}).
-record(r_a1,{something}).

% lib.erl
-module(lib).
-export([f1/0,f2/0,f3/0,f4/0,f5/0,f6/0,f7/0]).
-record(r1,{something}).
-record(r2,{something}).
-record(r3,{something}).
-define(MAC1,a).
-define(MAC2,b).
-include("h.hrl").

f1() ->
    f2(), f3(), f6(),
    #r1{}, #rh1{}, #r2{}, #rh2{},
    ?MAC1, ?MODULE.

f2() -> f3(), #r1{}, #rh1{}.

f3() -> f7().

f4() ->
    f6(), f7(), #r2{}, #rh2{}, #r3{}, ?MAC2.

f5() -> #rh3{}.

f6() -> ?MODULE.

f7() ->
    length([1,2]),
    spawn(fun() -> ok end),
    f8().

f8() -> ok.
```

The splitting function returns the following result:

```
[{"/erlang/lib.erl",
  {[{fun_attr,lib,f5,0},
    {fun_attr,lib,f6,0},
    {fun_attr,lib,f7,0},
    {fun_attr,lib,f8,0},
    {rec_attr,"/erlang/lib.erl",r2}],
   [{0,
     [a1,a2],
     [{rec_attr,"/erlang/lib.erl",r1},
      {macro_attr,"/erlang/lib.erl","MAC1"},
      {fun_attr,lib,f1,0},
      {fun_attr,lib,f2,0},
      {fun_attr,lib,f3,0}]},
    {1,
     [a3],
     [{fun_attr,lib,f4,0},
      {rec_attr,"/erlang/lib.erl",r3},
      {macro_attr,"/erlang/lib.erl","MAC2"}]}]}},
 {"/erlang/h.hrl",
  {[{rec_attr,"/erlang/h.hrl",rh3},
    {rec_attr,"/erlang/h.hrl",rh2}],
   [{0,
     [a1,a2],
     [{rec_attr,"/erlang/h.hrl",r_a1},
      {rec_attr,"/erlang/h.hrl",rh1}]}]}}]
```

Each element of the main list specifies how to split a concrete file. The first element specifies how to split `"/erlang/lib.erl"`, the second does the same with `"/erlang/h.hrl"`.

Let us focus on how `"/erlang/lib.erl"` should be split. After the file's name in the tuple there is a list of elements that should not be moved: `f5/0`, `f6/0`, `f7/0`, `f8/0` and the record, the name of which is `r2` and which is defined in the file `"/erlang/lib.erl"`. Then the elements that should be moved into the new file which will belong to cluster 0 follow, after the list `[a1,a2]`, which states that cluster 0 consists of these two modules.

6. Transformations for module restructuring

Clustering modules is really a logical operation which does not involve changing the code itself, but consequently, splitting library modules and header files into parts that are used in the clusters is a typical refactoring operation. This section introduces the basic refactorings that are required to apply the results of clustering and splitting.

These refactoring steps will be available in RefactorErl, *move function* is already completed, and the other two is partially implemented.

6.1 Move function

Move function refactoring enables to move an arbitrary set of functions between modules. Selected functions are copied to the given target module (each clause of each function), deleted in the original module, and export lists are updated accordingly. Calls to these functions are updated by changing module qualifiers to use the new module name.

The refactoring takes care of the proper handling and moving of macros, records and function calls, and ensures their visibility in the previous and new place of the functions. It tries to avoid code duplication as much as possible.

With the help of transformation several functions, which are collected in a list by the tool, can be moved in one step. By examining the content of the list, it can reject functions or make recommendations for creating a proper list. When any function is rejected in the list, the whole transformational step is rejected by the system.

Parameters

- The module from which the functions are to be moved.
- The name and arity of the functions to be moved.
- The name of the module where the functions should be moved.

Side conditions

- The names of the selected functions should not conflict with other functions in the target module, neither with those imported from another module (overloading). Furthermore, the name should be a legal function name in all modules.
- Macro name conflicts must not occur in the target module, that is, macro names used in the functions must refer to the same macro definition in the source and in the target module. This applies to macros used in these macros too.
- Record name conflicts must not occur in the target module, that is, record names used in the functions must refer to the same record definition in the source and in the target module.
- If header file inclusions have to be made during the transformations, these inclusions must not introduce name clashes between macros and records.

Transformation steps and compensations

1. The function bodies to be moved are deleted from their original places with all their clauses.
2. The moved functions are placed at the end of the new module.
3. Functions that appear in the export lists of the original module are removed from there, and a new export list is created from them in the target module right after the last export list in that module.
4. The functions, which are called in a moved function but remain in the original module, are put in an export list in the original module.
5. Moved functions that called from other functions in the original module are exported in the new module and the calls in the original module are changed to include a module qualifier that refers the target module.
6. Moved functions that are referred by qualified names in the moved functions are changed to use the new module name.
7. Moved functions that appear in an import list of the target module are removed from that import list.
8. Moved function that appear in an import list of any module are removed from that import list, and a new import list is created in that module which refers to the moved function using the target module name.
9. Qualified names referring to a moved function in any module are changed to use the name of the target module.
10. Records and macros used in the moved function have to be made visible in the target module, either including the header file in which they are defined (but only when no record or macro name clash is introduced by the inclusion), or copying their definition.

6.2 Move record

This transformation moves a record definition between two files. Source and target files can be either modules or header files, the conditions are slightly different in every case. The goal of the

transformation is to make the record definition available in every place where it is used after the move.

Parameters

- The module or a header file in which the records are defined.
- The name of the records to be moved.
- The name of the module or header file where the records should be moved.

Side conditions

- Record names do not clash with existing record definitions in the target file.
- Modules files cannot be included in other files, so moving to a module file is permitted only if no other modules use the records.
- If header file inclusions have to be made during the transformation, these inclusions must not introduce name clashes between macros and records.

Transformation steps and compensations

1. The record definitions are removed from the source file.
2. The record definitions are placed at the end of the target header file, or before the first function of the target module file.
3. If a record is moved into a header file, then every module that uses the record is changed to include the target header file. This is not an issue when the target is a module file.

6.3 Move macro

This refactoring moves macro definitions between modules and header files. It is similar to *move record*, but it is a little more complex, because macros can refer to each other. Macros can also refer to records and functions, but this does not affect the transformation, because only the definitions are moved, and the context of macro applications remain unchanged, the same records and functions remain accessible everywhere.

Parameters

- The module or a header file where the macros are defined.
- The names of the macros to be moved.
- The name of the module or header file where the macros should be moved.

Side conditions

- Macro names must not clash with existing macro names in the target file.
- Macros can be moved into a module file only if no other module refers to them.
- If header file inclusions have to be made during the transformation, these inclusions must not introduce name clashes between macros and records.
- If a compensation refactoring is necessary (see the next paragraph), it must fulfil these requirements too.

Transformation steps and compensations

- The macro definitions to be moved are removed from the source module or header file.
- The macro definitions are put into the target file. They are placed before any function definition in module files, and if the target file refers to any moved macro in a non-macro definition, the moved macro definitions are placed before these usage points. In header files, if there are no references to the macros, the definitions are placed at the end of the file.
- If the target is a header file, every module that refers to any of the moved macros is changes to include the target header file.
- If the macros to be moved refer to other macros, an availability check is done for every module that uses the moved macros. Every referred macro is made available using the following rules:
  - Macros which are already visible in the module don't need further steps.
  - Unavailable macros defined in a header file are made available by including their defining header file in the module.
  - Unavailable macros defined in module files are made available by moving them into the same place as the originally moved macros, using the same *move macro* transformation. The failure of this compensation means the failure of the whole transformation.

7. Conclusions

The components introduced in the paper are rather lightly connected, some of them work with clusters of modules, and the refactorings are completely stand-alone. Their power lies in using them together: different clustering solutions can be produced, the best one can be selected, the list of neccessary transformations can be generated, and this list can be automatically executed. Extended with the possibility of manual intervention and editing, this covers the whole process of restructuring large systems into smaller ones, and proved to be helpful in industrial applications.

References

[1] Anquetil N., Fourrier, C., Lethbridge T. C.: Experiments with Hierarchical Clustering Algorithms as Software Remodularization Methods Working Conference on Reverse Engineering (1999).

[2] Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, The Royal Institute of Technology, Stockholm, Sweden (2003)

[3] Armstrong, J.: Programming Erlang, Software for a Concurrent World Pragmatic Bookshelf (2007)

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

[5] Harman, M., Swift, S., Mahdavi, K.: An empirical study of the robustness of two module clustering fitness functions. GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation (2005), 1029–1036

[6] Horváth, Z. et al.: Refactoring Erlang Programs. `http://plc.inf.elte.hu/erlang/`

[7] Kitlei, R., Lövei, L., Nagy, T., Nagyn, V. A., Horvth, Z., Csrnyei, Z.: Generic syntactic analyser: ParsErl. International Erlang/OTP User Conference (2007).

[8] Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang programs. To appear in: Periodica Polytechnica – Electrical Engineering (2007) 19 pages.

[9] Li, H., Thompson, S., Lövei, L., Horváth, Z., Kozsik, T., Víg, A., Nagy, T.: Refactoring Erlang programs. In: The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden (2006) http://www.erlang.se/euc/06/, 10 pages.

[10] Lövei, L., Horváth, Z., Király R., Kitlei R.: Static rules for variable scoping in Erlang. To appear in: The 7th International Conference on Applied Informatics, Eger, Hungary, 2007.

[11] Nagy, T., Víg, A.: Erlang refactor tool. Master thesis, Eötvös Loránd University, Budapest, Hungary, 2007.

Author Index

Arts, Thomas 1
Boberg, Jonas 9
Castro, Laura M. 1
Casu, Giovanni 49
Cesarini, Francesco 29
Hoch, Csaba 83
Horpácsi, Dániel 83
Hughes, John 1
Király, Roland 83
Kitlei, Róbert 83
Köllő, Hanna 83
Li, Huiqing 61
Lövei, László 83
Luna, Daniel 73
Nagy, Tamás 21, 83
Nagyné Víg, Anikó 21, 83
Orosz, György 61
Pappalardo, Viviana 29
Pili, Piero 49
Reinefeld, Alexander 41
Sagonas, Konstantinos 73
Santoro, Corrado 29
Scalas, Alceste 49
Schintke, Florian 41
Schütt, Thorsten 41
Thompson, Simon 61
Tóth, Melinda 61