Evolving projects to concurrency with Wrangler

Simon Thompson
University of Kent, UK

Wrangler is a tool

Tool for refactoring Erlang programs.

Inside Emacs and ErlIDE …

… and stand-alone.

Wrangler is a toolkit

Usable from the Erlang shell / command line.

An open API for new refactorings …

… and a DSL for scripting complex changes.

API migration tool (e.g. `dict` to `map`, `regexp` to `re`).

Wrangler is a toolset

Clone detection and module “bad smells”.

Web services testing support.

Plays with testing tools.

Laboratory: symbolic evaluation, slicing, concurrency, parallelisation.

erlang

rerlang

rangler

Wrangler

wrangler | ˈraŋglə |

noun

1 N. Amer. a person in charge of horses or other livestock on a ranch.

• a person who trains and takes care of animals on a film set. *they had three cow wranglers to help with the scene.*

2 a person engaging in a lengthy and complicated dispute. *he was known as the wrangler for the aplomb with which he skewered the professors.*

3 (at Cambridge University) a person placed in the first class of the mathematical tripos.

Wrangler is a tool

Refactoring

"Change how the program works, but not what it does

Part of the programmer's standard toolkit.

Renaming, function extraction, generalisation

Refactoring tools

Avoid the *"tedious and error-prone"*.

Keep it simple.

Cover the bureaucracy.

Whole language …

… whole projects,

… plus tests, …

Demo

Rename,
function extraction,
generalisation,
rename again.

Using Wrangler outside emacs/ErlIDE

Wrangler from the Erlang shell … use module `api_wrangler.erl`

For example:

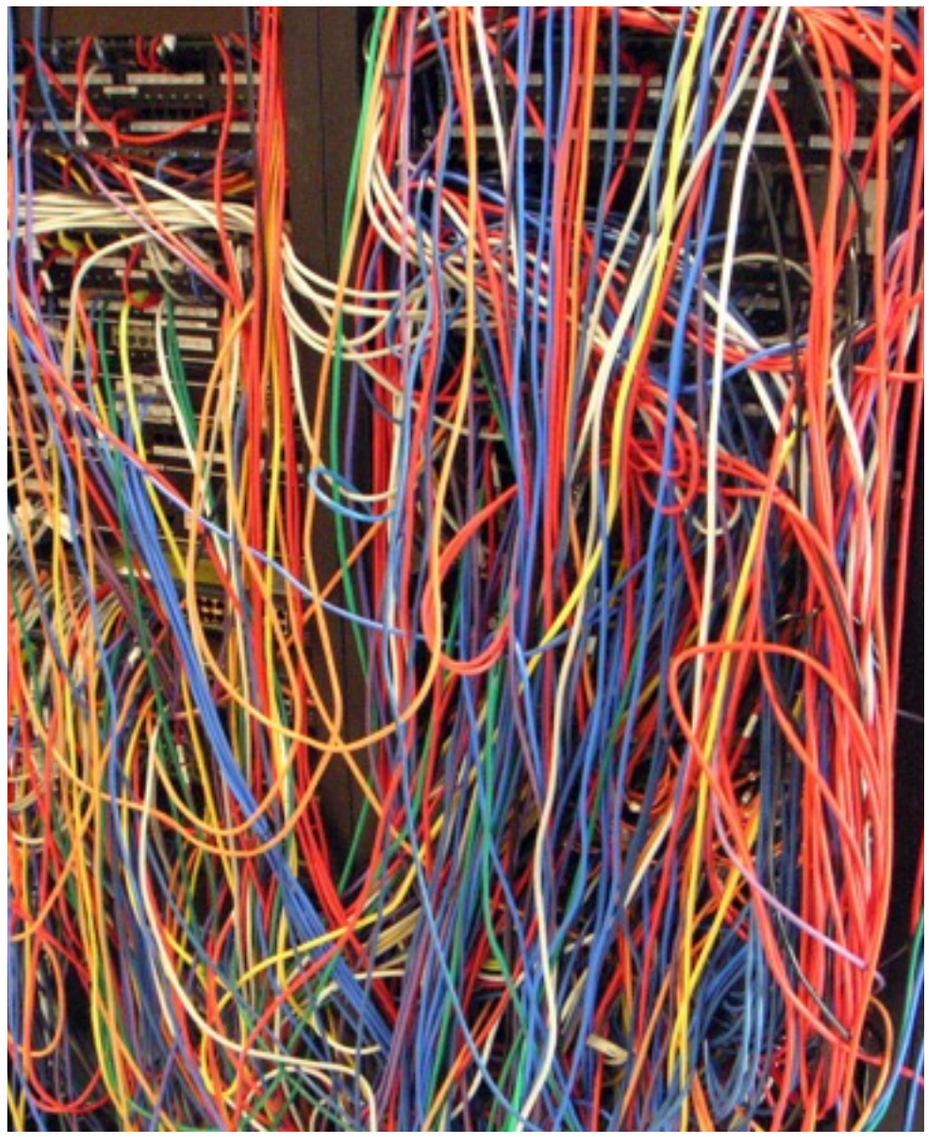
```
1> api_wrangler:start().

2> api_wrangler:rename_mod("main.erl",
                           test,
                           ["c:/cygwin/home/hl/test"]).

3> api_wrangler:stop().
```

More info: `http://refactoringtools.github.io/wrangler`

Wrangler is a toolkit

Wrangler is a toolkit

Wrangler is written in Erlang, all the way down, so you can extend it yourself …



Wrangler is a toolkit

… but we have given you tools to make that extension much easier to deal with.

Key idea

You know Erlang …

… you don't want to have to learn a whole new language or compiler internals to get the job done.

API and DSL

An API to define a completely new refactorings from scratch

… using Erlang concrete syntax,

… also “code inspection”.

DSL for scripting refactorings

… “on steroids”

… embedded in Erlang.

What we've been asked for …

`camelCase` to `camel_case`

Batch module renaming

Removing "bug preconditions" for Quviq.

API migration … for example, `regexp` to `re`.

Introduce `s_group`s.

Wrangler API

Generalisation

Describe expressions in Erlang ...

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N),
       loop_a()
    end.

body(Msg,N) ->
      io:format("ping!~n"),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N,"ping!~n"),
       loop_a()
    end.

body(Msg,N,Str) ->
      io:format(Str),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

Generalisation

... how expressions are transformed ...

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
    end.

body(Msg,N) ->
      io:format("ping!~n"),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N,"ping!~n"),
        loop_a()
    end.

body(Msg,N,Str) ->
      io:format(Str),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

Generalisation

... and its context and scope.

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N),
       loop_a()
    end.

body(Msg,N) ->
       io:format("ping!~n"),
       timer:sleep(500),
       b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N,"ping!~n"),
       loop_a()
    end.

body(Msg,N,Str) ->
       io:format(Str),
       timer:sleep(500),
       b ! {msg, Msg, N - 1}.
```

Generalisation

Pre-conditions for refactorings

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
    end.

body(Msg,N) ->
      io:format("ping!~n"),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

Can't generalise over an expression that contains free variables ...

... or use the same name as an existing variable for the new variable.

Wrangler API

Context is used to define preconditions

Traversals describe how transformations are applied

Rules describe transformations

Templates describe expressions

Introducing concurrency

Key idea

Identify parts of computations that can be performed independently.

Base decisions on intra-function control-flow slicing.

Percept2

Uncovering potential concurrency

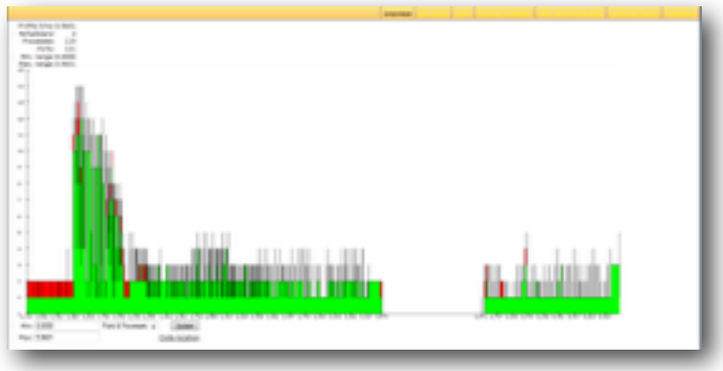
Number of active processes.

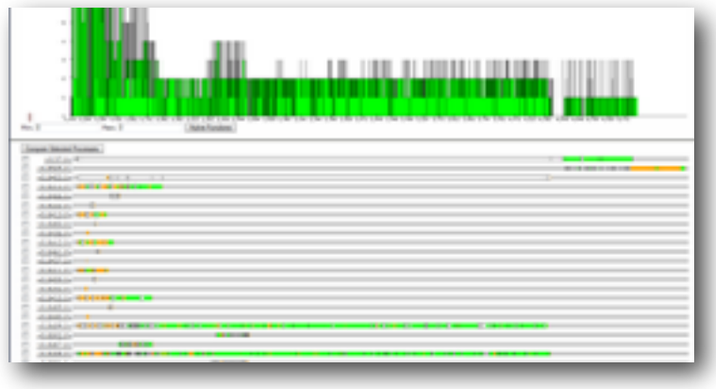
Runnable vs running.

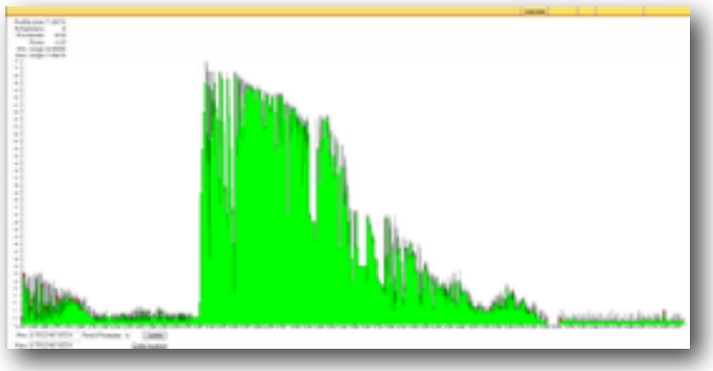
Process id to function definition.

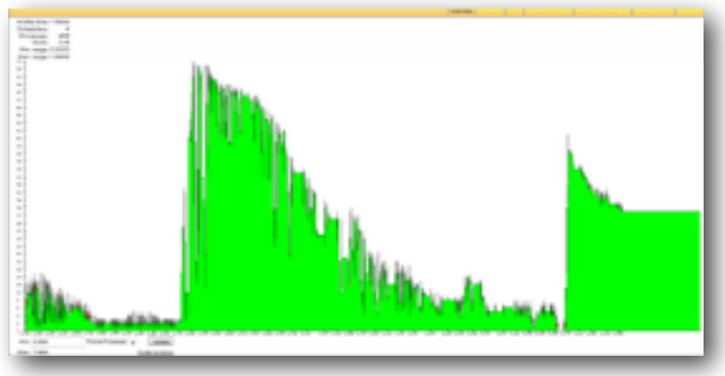
Github: *RefactoringTools/Percept2*.

Also available: htop, etop, …









Introducing concurrency in practice

Some examples are easy …

```
lists:map(…, …)
```

```
[f(X) || X <- Xs]
```

… both can be replaced with a *parallel map* occurrence

```
percept2:pmap(…, …)
```

```
percept2:pmap(f,Xs)
```

But in general need to analyse program structure more carefully.

Pragmatics

Fit with Erlang design philosophy:

- work with OTP behaviours: synchronous to asynchronous calls to a generic server …
- … an instance of a general function transformation;
- Deal with tail recursive functions.

Provide automation in Wrangler.

Analysis

```
readImage(FileName, FileName2) ->

  {ok, #erl_image{format=F1, pixmaps=[PM1]}}
      = erl_img:load(FileName),
  Cols1=PM1#erl_pixmap.pixels,

{ok, #erl_image{format=F2, pixmaps=[PM2]}}
   = erl_img:load(FileName2),
Cols2=PM2#erl_pixmap.pixels,

R1 = [B1||{_A1, B1}<-Cols1],
R2 = [B2||{_A2, B2}<-Cols2],

{R1, F1, R2, F2}.
```

Analysis and transformation

```
readImage(FileName, FileName2) ->

  {ok, #erl_image{format=F1, pixmaps=[PM1]}}
     = erl_img:load(FileName),
  Cols1=PM1#erl_pixmap.pixels,

  {ok, #erl_image{format=F2, pixmaps=[PM2]}}
     = erl_img:load(FileName2),
  Cols2=PM2#erl_pixmap.pixels,

  R1 = [B1||{_A1, B1}<-Cols1],
  R2 = [B2||{_A2, B2}<-Cols2],

  {R1, F1, R2, F2}.
```

```
readImage(FileName, FileName2) ->
  Self = self(),
  Pid = spawn_link(
    fun () ->
      {ok, #erl_image{format=F1,
                      pixmaps=[PM1]}}
         = erl_img:load(FileName),
      Cols1 =PM1#erl_pixmap.pixels,
      R1 =[B1||{_A1, B1}<-Cols],
      Self ! {self(), {R1, F1}}
    end),

  {ok, #erl_image{format=F2, pixmaps=[PM2]}}
     = erl_img:load(FileName2),
  Cols2=PM2#erl_pixmap.pixels,

  R2 = [B2||{_A2, B2}<-Cols2],

  receive {Pid, {R1, F1}} -> {R1, F1} end,

  {R1, F1, R2, F2}.
```

Working with OTP

Fit with Erlang design philosophy: calculate a reply separately from the new state …

```
handle_call(which_children, _From, State) ->
    Resp = lists:map(fun(…) -> {Name, Pid, ChildType, Mods} end,
                     State#state.children),
     {reply, Resp, State};


handle_call(which_children, From, State) ->
    spawn_link(
      fun () ->
              Resp = lists:map(fun(…) -> {Name, Pid, ChildType, Mods} end,
                               State#state.children),
              gen_server:reply(From, Resp)
      end),
    {no_reply, State};
```

The Wrangler API – top level

The top level of the transformation

```
transform(_Args=#args{current_file_name=File,
                      cursor_pos=Pos}) ->
    ?STOP_TD_TP([rule1(Pos)], [File]).
```

`?STOP_TD_TP` … apply the transformation top down;

`rule1(Pos)` … apply rule1 when possible.

The Wrangler API - rule

```
rule1(Pos) ->
   ?RULE(?T("handle_call(Args@@) when Guard@@->
                        Body@@,{reply, Res@, State@};"),
               gen_new_handle_call(_This@, Res@, State@,
                    {Args@@, Guard@@, Body@@, State@}),
               begin
                  {S, E} = api_refac:start_end_loc(_This@),
                  S=<Pos andalso E>=Pos
               end).
```

```
match this / replace with this / if this holds
```

Meta-variables match objects (`Res@`) and sequences (`Body@@`).

The Wrangler API - analyse / generate

```
gen_new_handle_call(C, Res, State,
                          {Args, Guard, Body, State}) ->
    {Slice1, _}=wrangler_slice_new:backward_slice(C, Res),
    {Slice2, _}=wrangler_slice_new:backward_slice(C, State),
    ExprLocs = Slice1 -- Slice2,
    Exprs =[B||B<-Body,
                lists:member(
                  api_refac:start_end_loc(B), ExprLocs)],
    NewBody = Body -- Exprs,
    api_refac:subst(
     ?T("handle_call(Args@@) when Guard@@ ->
            Body@@,
            spawn_link(
                 fun()->
                      Resp= begin Exprs@@ end,
                    gen_server:reply(From, Resp)
                 end),
            {no_reply, State@};"),
       [{'Args@@', Args}, {'Guard@@', Guard},
         {'Body@@', NewBody}, {'State@', State},
         {'Exprs@@', Exprs}]).
```

The Wrangler API - analyse / generate

```
gen_new_handle_call(C, Res, State,
                            {Args, Guard, Body, State}) ->
    {Slice1, _}=wrangler_slice_new:backward_slice(C, Res),
    {Slice2, _}=wrangler_slice_new:backward_slice(C, State),
    ExprLocs = Slice1 -- Slice2,
    Exprs =[B||B<-Body,
               lists:member(
                 api_refac:start_end_loc(B), ExprLocs)],
    NewBody = Body -- Exprs,
    api_refac:subst(
 ?T("handle_call(Args@@) when Guard@@ ->
          Body@@,
          spawn_link(
               fun()->
                    Resp= begin Exprs@@ end,
                  gen_server:reply(From, Resp)
                end),
          {no_reply, State@};"),
       [{'Args@@', Args}, {'Guard@@', Guard},
         {'Body@@', NewBody}, {'State@', State},
         {'Exprs@@', Exprs}]).
```

Demo

Slicing in Wrangler

Handling tail recursion

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};

do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
```

```
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

Handling tail recursion

Work thorough a series of analysis and transformation steps.

- is it tail recursive?
- which is the accumulator?
- partition the body
  - float out: computation that can be separated,
  - new values of parameters, and …
  - … new value of the accumulator;
- and finally repackage.

Is it tail recursive?

Without loss of generality we assume that it has the form …

```
fun_name(Arg_11, . . ., Arg_1n) -> Body1;
. . .
fun_name(Arg_m1, ...., Arg_mn) ->
   BodyExpr1,
   BodyExpr2,
   . . .
   fun_name(NewArg_m1, ..., NewArg_mn).
```

… but obviously the same mechanisms will apply in other forms.

Which is the accumulator?

Value depends on itself (and some others)?

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

No other parameter depends on its value.

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

Its value is not used in the termination condition of the recursion.

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

What can we float out?

From the slice for the accumulator …

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

… not depending on the value of the accumulator,

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

… not overlapping with slices of other parameters.

```
do_grouping([], _, _, _, Acc) -> {ok, Acc};
do_grouping(Nodes, _Size, 1, Counter, Acc) ->
  {ok,  [make_group(Nodes, Counter)|Acc]};
do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
  Group = lists:sublist(Nodes, Size),
  Remain = lists:subtract(Nodes, Group),
  NewGroup = make_group(Group, Counter),
  NewAcc = [NewGroup|Acc],
  do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

Transformed process … top level

```
do_grouping(Nodes, Size, NumGroup,Counter, Acc) ->
  Parent = self(),

  Workers = [spawn(fun() ->  do_grouping_worker_loop(Parent) end)
             || _ <- lists:seq(1, erlang:system_info(schedulers))],

  Pid = spawn_link(
          fun() ->
              do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, 0, 0)
          end),

Pid ! {Nodes, Size, NumGroup, Counter},

receive
  {Pid, Acc} ->
    [P ! stop || P <- Workers],
    Acc
end.
```

Transformed process … under the hood

Each of the workers repeatedly calculates `make_group` on demand.

```
do_grouping_worker_loop(Parent) ->
   receive
      {Group, Size, Counter, Index} ->
          NewGroup = make_group(Group, Counter),
          Parent ! {{worker, self()}, Index, NewGroup},
          do_grouping_worker_loop(Parent);
      stop ->
          ok
   end.
```

Ensure that results collected in the correct order, to preserve semantics.

```
do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, RecvIndex, CurIndex) ->
  receive
     {[], Size, NumGroup, Counter} when  RecvIndex == CurIndex ->
         Parent ! {self(), {ok, Acc}};
     {[], Size, NumGroup, Counter} when  RecvIndex < CurIndex ->
         self() ! {[], Size, NumGroup, Counter},
         do_grouping_dispatch_and_collect_loop(
            Parent, Acc, Workers, RecvIndex, CurIndex);
     {Nodes,_Size, 1, Counter} when RecvIndex == CurIndex ->
          Parent ! {self(), {ok, [make_group(Nodes, Counter)|Acc]}};
     {Nodes, Size, 1, Counter} when RecvIndex < CurIndex ->
          self() ! {Nodes, Size, 1, Counter},
          do_grouping_dispatch_and_collect_loop(
            Parent, Acc, Workers, RecvIndex, CurIndex);
     {Nodes, Size, NumGroup, Counter} ->
          Group = lists:sublist(Nodes, Size),
          Remain = lists:subtract(Nodes, Group),
          Pid = oneof(Workers),
          Pid ! {self(), Group, Size, Counter},
          self() ! {Remain, Size, NumGroup-1, Counter+1},
          do_grouping_dispatch_and_collect_loop(
             Parent, Acc, Workers, RecvIndex, CurIndex+1);
      {{worker, _Pid}, RecvIndex, NewGroup} ->
          NewAcc = [NewGroup|Acc],
          do_grouping_dispatch_and_collect_loop(
             Parent, NewAcc, Workers, RecvIndex+1, CurIndex)
end.
```

Why are these *explicit* transformations?

These are complex transformations … why not just in a compiler?

- So that the step is an explicit part of the development …
- … and in particular logged in a repository.
- Compiler-based transformation notoriously fragile.
- ‘Hand’ intervention is often necessary for optimal results …
- … as recognised by others performing loop parallelisation.

API migration: maps in R17

Can we use maps in our project?

Look for uses of dict … are they map-like?

All the details in my talk at the Erlang User Conference 2014

Wrangler is a toolset

Wrangler as a toolset

- Clone detection
  - Parametrisable
  - Incremental
  - Automated support using the DSL
- Module “bad smell” detection
  - Size, cycles, exports
- Other inspection and refactoring functions
- WSToolkit: PBT for web services

Clone detection and removal

emacs@HL-LT

File Edit Options Buffers Tools Help

```
loop_a() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_a();
        {msg,Msg,N} ->
            io:format("ping!~n"),
            timer:sleep(500),
            b!{msg,Msg,N+1},
            loop_a()
    end.

loop_b() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_b();
        {msg,Msg,N} ->
            io:format("pong!~n"),
            timer:sleep(500),
            a!{msg,Msg,N+1},
            loop_b()
    end.
```

Rename function

Rename variables

Reorder variables

Add to export list

Fold* against the def.

```
--\---  pingpong.erl    Bot L46    Git:master   (Erlang EXT)----------------------------

c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2 ! {msg,Msg,N + 1}.

-1\**-  *erl-output*    40% L11     (Fundamental)-----------------------------------
```

Demo

Clone detection in Wrangler

Not just a script ...

Tracking changing names and positions.

Generating refactoring commands.

Dealing with failure.

User control of execution.

… we're dealing with the pragmatics of composition, rather than just the theory.

Automation

Don't have to describe each command explicitly: allow conditions and generators.

Allow lazy generation … return a refactoring command together with a continuation.

Track names, so that `?current(foo)` gives the 'current' name of an entity `foo` at any point in the refactoring.

Clone removal: top level

Transaction as a whole … non-“atomic” components OK.

Not just an API: `?atomic` etc. modify interpretation of what they enclose …

```
?atomic([?interactive( RENAME FUNCTION )
         ?refac_( RENAME ALL VARIABLES OF THE FORM NewVar* )
         ?repeat_interactive( SWAP ARGUMENTS )
         ?if_then( EXPORT IF NOT ALREADY )
         ?non_atomic( FOLD INSTANCES OF THE CLONE )
        ]).
```

Erlang and the embedded DSL

```
?refac_(rename_var,
        [M,
         begin
           {_, F1, A1} = ?current(M,F,A),
           {F1, A1}
         end,
        fun(X) ->
            re:run(atom_to_list(X), "NewVar*")/=nomatch
        end,
        {user_input, fun({_, _, V}) ->
                       lists:flatten(io_lib:format
                                     "Rename variable ~p to: ", [V]))
                     end},
        SearchPaths])
```

WSToolkit

Property-based testing of stateful systems

Build an abstract model: a single state EFSM

- the state data: typically an Erlang record.
- pre- and post-conditions on calls of API functions
- state data transitions for the API functions

Test the system through random call sequences through the model.

```
prop_state_machine() ->
  ?SETUP(fun setup/0,
    ?FORALL(Cmds, commands(?MODULE),
       begin
         {_H, _S, Res} = run_commands(?MODULE, Cmds),
         Res==ok
       end)).
```

Automation of web services testing

From WSDL description, automated creation of Erlang code for

- data type definitions,
- data generators,
- web services connector module, and
- skeleton `eqc_statem` behaviour.

https://github.com/RefactoringTools/WSToolkit

Evolution and refactoring

Automatic inference of web service interface changes.

A set of domain-specific refactorings in Wrangler.

Automatic creation of Wrangler refactoring scripts.

https://github.com/RefactoringTools/wrangler

Automation … why?

Automation of the boilerplate and the routine …

… gives time to concentrate on the semantic.

Because it’s useful in practice …

www.interoud.com

363 operations: 160 POST and 203 GET; data returned in XML.

Hand-coded QuickCheck state machine tests 98 operations of this web service (27% of the total).

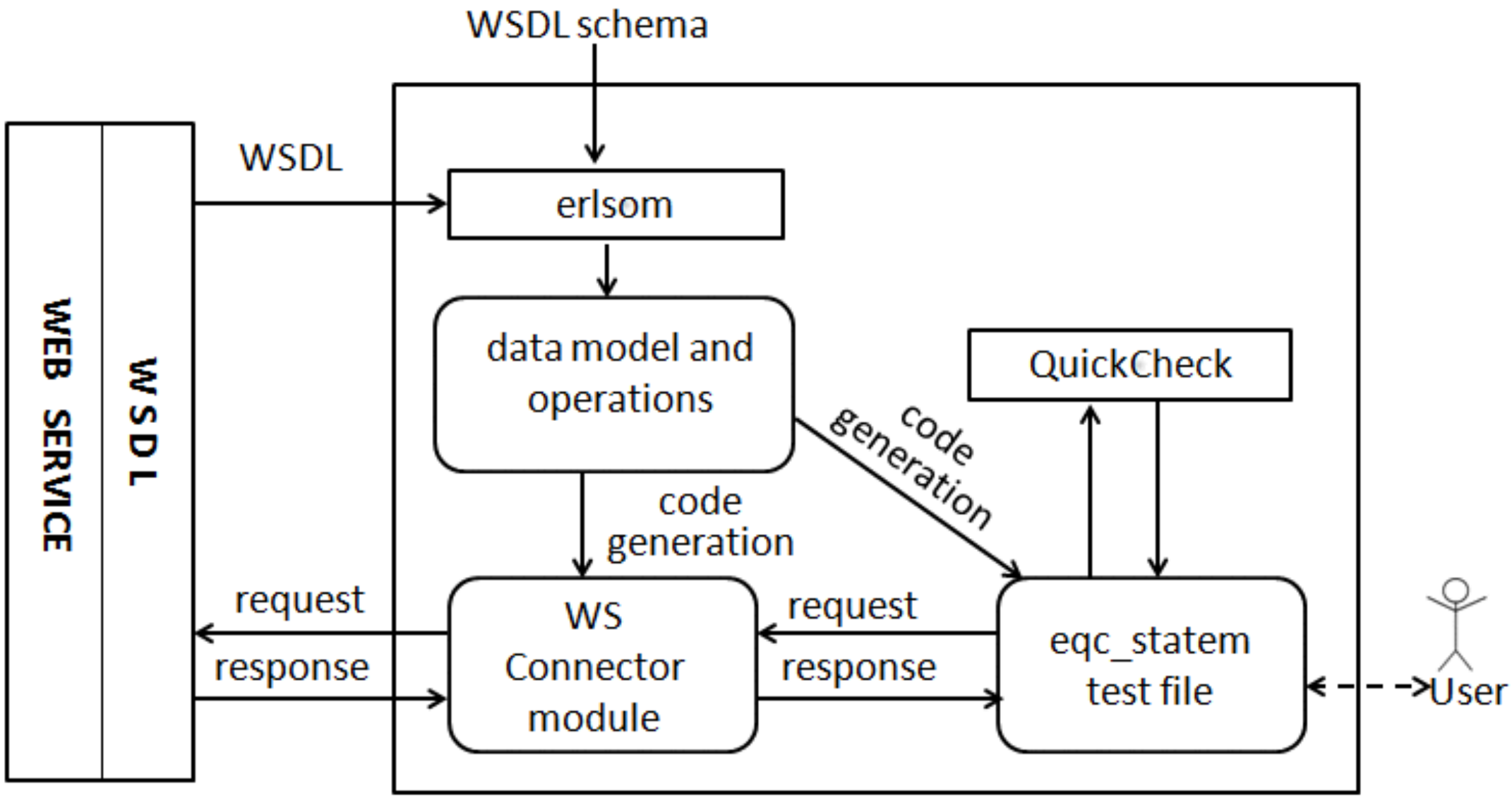
Hand-crafted WS connection module of 1,000 lines of code.

Rapid change: e.g. in September and October 2013, 10 operations added and 15 modified, many by adding new parameters.

Automation leads to

- fewer errors (e.g. in connector module), and
- more robust evolution.

Automation architecture



Connector module

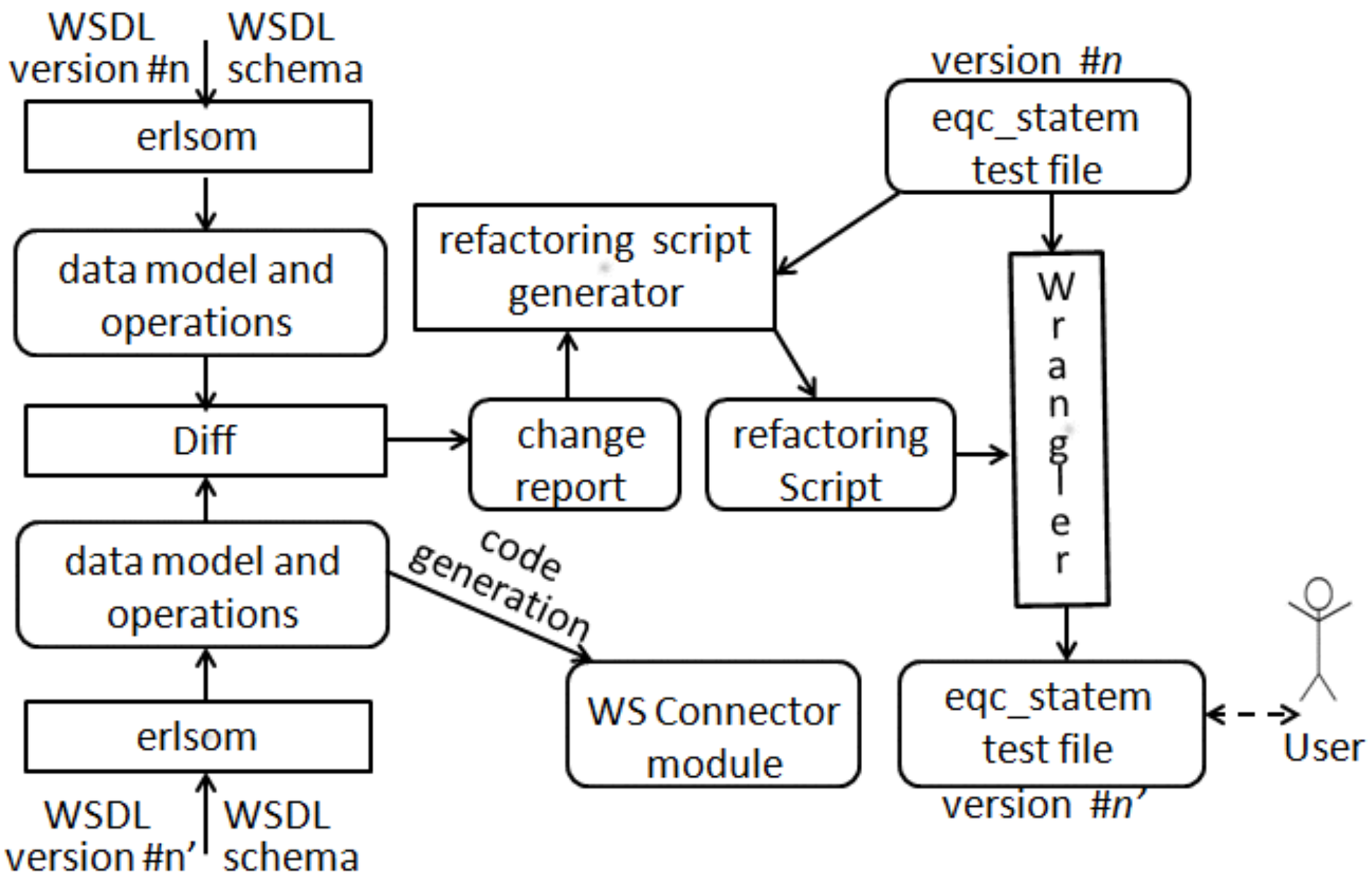
From WSDL operation descriptions …

```
<operation name="GetWeather" pattern="http://www.w3.org/ns/wsdl/in-out">
   <documentation>Get weather report for all major cities around the world.
       </documentation>
   <input element="tns:GetWeather"/>
   <output element="tns:GetWeatherResponse"/>
</operation>
```

… generate Erlang connector functions, to call the service.

```
get_weather(CityName, CountryName)->
  GetParams = generate_get_params(`GetWeather', [CityName, CountryName]),
  Url = add_get_params(?BASE_URL++"/GetWeather",GetParams),
  http_request(`GET', Url,
                fun(Data) ->
                    process_response(`GetWeatherResponse', Data)
                end).
```

Evolution architecture



Inferring changes in WSDL

Old and new WSDL represented as Erlang data structures.

Infer Levenshtein distance …

… plus some domain-specific processing e.g. order, rename, merge.

Typical example: two new operations, plus …

… changes to input and output components and their types.

Domain-specific refactorings

Some refactorings in Wrangler already e.g renaming, but others not.

The PBT use case requires specific extras, e.g. add parameter:

- addition of a field to a tuple, not as another parameter;
- symbolic calls in description of the state machine;
- connector module uses.

These domain-specific refactorings defined using the Wrangler API.

Inferring refactoring scripts

From the changes we infer between WSDL versions we can derive a script for the Wrangler DSL to automate the refactorings.

In the case inferred in the paper (with some …).

```
-module(refac_evolve_api).

composite_refac(_Args=#args{current_file_name=File})
  ?interactive(
    [?refac_(refac_add_op,[File,"find_all_rooms",[],[File]]),
     ?refac_(refac_add_op_arg,[File,"find_devices",1,"SortBy",[File]]),
     ?refac_(refac_add_op_arg, [File,"find_devices",1,"Order",[File]]),
     ?refac_(refac_add_op_arg,[File,"find_devices",1,"Query",[File]]),
     ?refac_(refac_add_op, [File,"delete_device",["DeviceId"],[File]])]).
```

Many thanks to Huiqing Li for her Wrangler work from its inception through to last summer.

Getting involved

https://github.com/RefactoringTools

RELEASE

PROWESS