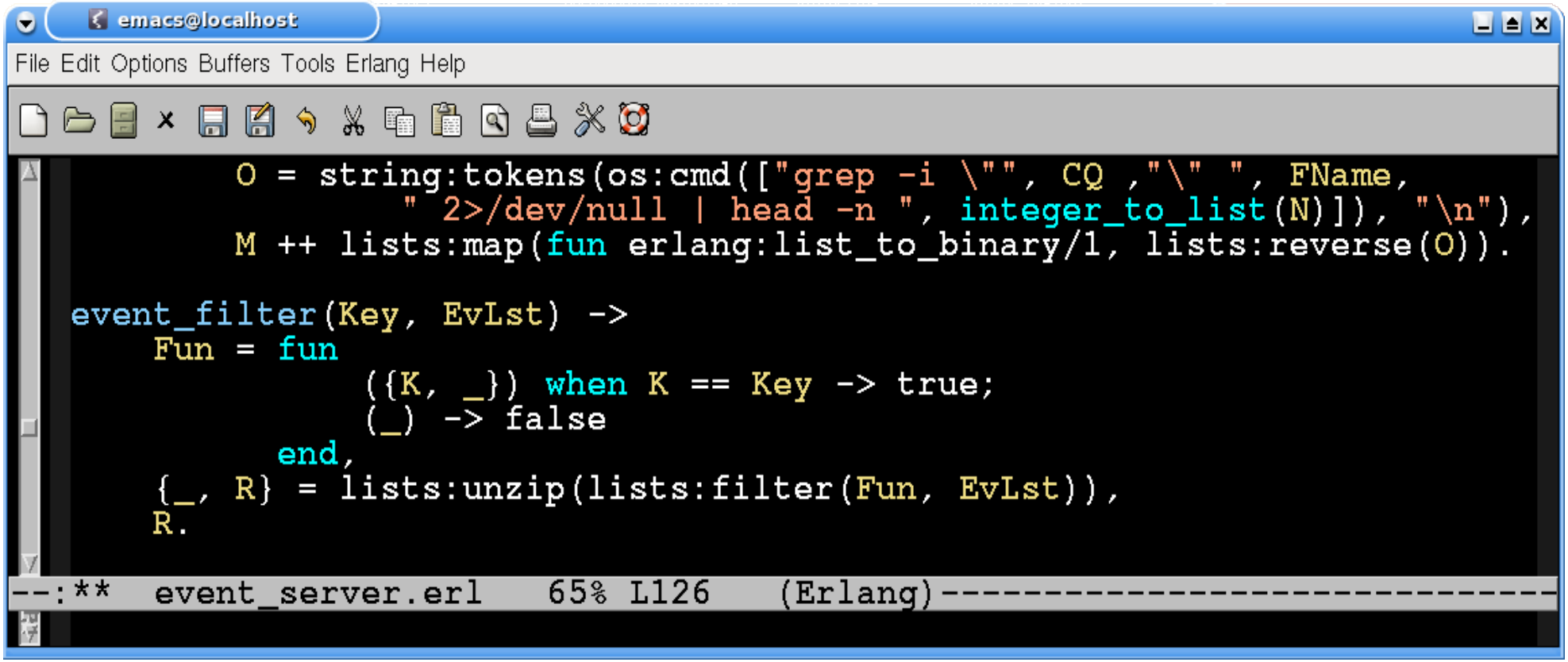
Cool Tools for Modern Erlang Program Development

Kostis Sagonas

Erlang program development




```
> erlc file.erl
```

```
> rebar compile
```

What this talk is about

- Overview some Erlang software development tools that I and my students have built
- Convince you
  - of their value and benefits of using them
  - why they should have a key role in your development environment
- For tools that are mature
  - give some hints/advice on how to properly use them
  - show some new goodies
- For new tools, show/demo their capabilities

Tool #1

Dialyzer

Dialyzer: A defect detection tool

- Uses static analysis to identify discrepancies in Erlang code bases
  - code points where something is wrong
    - often a bug
    - or in any case something that needs fixing
- Fully automatic
- Extremely easy to use
- Fast and scalable
- Sound for defect detection
  - “Dialyzer is never wrong”

Dialyzer

- Part of the Erlang/OTP distribution since 2007
- Detects
  - Definite type errors
  - API violations
  - Unreachable and dead code
  - Opacity violations
  - Concurrency errors

    Data races (`-Wrace_conditions`)
- Experimental extensions with
  - Stronger type inference
  - Detection of message passing errors & deadlocks

How to use Dialyzer

- First build a PLT (needs to be done once)

```
> dialyzer --build_plt --apps erts kernel stdlib
```

- Once this finishes, analyze your application

```
> cd my_app
> erlc +debug_info -o ebin src/*.erl
> dialyzer ebin
```

- If there are unknown functions, you may need to add more stuff to the PLT

```
> dialyzer --add_to_plt --apps mnesia inets
```

Used more and more out there

https://github.com/klacke/yaws/commits/master

ıbe Research

2011-02-26

Fix Dialyzer warning: remove dead code

tuncer (author)
February 24, 2011

vinoski (committer)
February 27, 2011

Fix unused clause Dialyzer warnings in yaws_rss

tuncer (author)
February 23, 2011

vinoski (committer)
February 27, 2011

Fix zlib:deflate/3 Dialyzer warning

tuncer (author)
February 23, 2011

vinoski (committer)
February 27, 2011

2011-02-25

dialyzer


klacke (author)
February 26, 2011

more dialyzer bugs, soap_srv startup code that could never run


klacke (author)
February 26, 2011

Some minor bugs, and some dead code removed, all found by Tuncers run of dialyzer


klacke (author)
February 25, 2011

compile bug


klacke (author)
February 25, 2011

2011-02-24

io:format leftover


klacke (author)
February 24, 2011

2011-02-23

Fixed dialyzer bugs found by Tuncer, Issue #54


klacke (author)
February 23, 2011

Erlang code bases ‘dialyzed’

agner alice aliter beamjs beehive beepbeep bert-erl bitcask cacherl cacherl cecho ced chicagoboss chordial chordjerl couchbeam couchdb cowboy disco distel dynomite edbi efene effigy egearmand-server egeoip egitd ehotp ejabberd ejson eldap elib elib1 elock ememcached enet eopenid eper epgsql epm epmail erlang-amf erlang-collectd erlang-couchdb erlang-facebook erlang-js erlang-jukebox erlang-mysql erlang-mysql-driver erlang-oauth erlang-protobuffs erlang-rfc4627 erlang-rtmp erlang-twitter erlang-uuid erlang-websocket erlangit erlaws erlawys erldis erlgmail erlguten erlide erlmongo erls3 erlsom erlsyslog erlwebsockserver erlydtl erlyweb ernie esdl esmtp eswf etap etorrent ewgi excavator exmpp fermal fragmentron fuzed gen-nb gen-paxos gen-smtp getopt giza gproc herml hovercraft ibrowse innostore iserve jsonerl jungerl ktuo leex lfe libgeoip-erlang log-roller log4erl lzjb-erlang mcd meck merle misultin mochiweb mongo-erlang-driver mustache-erl natter neotoma ngerlguten nitrogen openpoker osmos pgsql phoebus php-app playdar preach proper protobuffs rabbit rebar redis-erl refactorerl reia reverl reversehttp riak rogueunlike s3imagehost scalaris server sfmt-erlang sgte simple-bridge socket-io-erlang sqlite-erlang sshrpc stoplight tcerl thrift-erl tora triq ubf webmachine wings yatce yatsy yaws yxa zotonic

http://dialyzer.softlab.ntua.gr/

Dialyzer's site at softlab.ntua.gr

Home | Current warnings | Heisenbug warnings | Intersection warnings | Warning-free applications | Contact

Welcome to Dialyzer's site at the Software Engineering Laboratory of NTUA

This site contains information for Dialyzer, the DIscrepancy AnaLYZer for ERlang applications.

Dialyzer is a static analysis tool that identifies software discrepancies such as definite type errors, code which has become dead or unreachable due to some programming error, unnecessary tests, etc. in single Erlang modules or entire (sets of) applications. Dialyzer starts its analysis from either debug-compiled BEAM bytecode or from Erlang source code. The file and line number of a discrepancy is reported along with an indication of what the discrepancy is about. Dialyzer bases its analysis on the concept of success typings which allows for sound warnings (no false positives).

You will soon find here papers about Dialyzer and tutorials for its suggested use (coming soon!)

You can find more information on how to use Dialyzer here.

For the time being, you can find results of continuously running Dialyzer on a set of open-source applications whose code is updated periodically.

In particular, under:

- Current warnings: you can find warnings as produced by Dialyzer in the current Erlang/OTP version
- Heisenbug warnings: you can find warnings as produced by an experimental version of Dialyzer that detects some kinds of possible concurrency errors
- Intersection warnings: you can find warnings as produced by an experimental version of Dialyzer that employs a stronger type inference which tracks argument-result dependencies.
- Finally, under Warning-free applications you can see the set of code bases for which dialyzer was run but no warnings were emitted (in any of the above categories)

If you'd like your application to be added (or removed) from these runs, don't hesitate to contact us!

The first versions of Dialyzer were created by Kostis Sagonas and Tobias Lindahl. People actively working and maintaining Dialyzer are Kostis Sagonas, Maria Christakis and Stavros Aronis.

Be nice to your fellow developers!

Expose type information:
make it part of the code

Exposing type information

Can happen in either of the following ways:

- Add explicit type guards in key places in the code
  - Ensures the validity of the information
  - Has a runtime cost – typically small
  - Programs may not be prepared to handle failures
- Add type declarations and function specs
  - Documents functions and module interfaces
  - Incurs no runtime overhead
  - Can be used by dialyzer to detect contract violations
  - Can also be handy for other tools (as we will see later)

Turning `@specs` into `-specs`

Often Edoc `@spec` annotations

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

Can easily be turned into `-spec` declarations

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

Turning `@specs` into `-specs`

In some other cases

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

Type declarations may need to be added

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

Or, better, they may already exist in some modules

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

Turning `@specs` into `-specs`

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

- Not surprising – they are comments after all!

I strongly recommend converting `@specs` gradually and fixing the erroneous ones using Dialyzer

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

Strengthening underspecified `-specs`

Can take place semi-automatically using Dialyzer

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

```
refac_duplicated_code.erl:42:
   Type specification for duplicated_code/3 ::
     ([filename()],[integer()],[integer()]) -> term()
   is a supertype of the success typing:
     ([string()],[integer()],[integer()]) -> {'ok',string()}
```

Document module interfaces

Add `-spec` declarations
for all exported functions

Finding missing `-specs`

New compiler option helps in detecting these:

```
> erlc +warn_missing_spec -I../hrl refac_rename_var.erl
./refac_rename_var.erl:666: Warning:
    missing specification for function pre_cond_check/4
```

Tool #2

Typer

TypEr: A type annotator

- Part of Erlang/OTP since 2008
- Displays the function specs which
  - already exist in a module/set of modules, or
  - are inferred by dialyzer
- Can help in adding missing specs to files

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

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

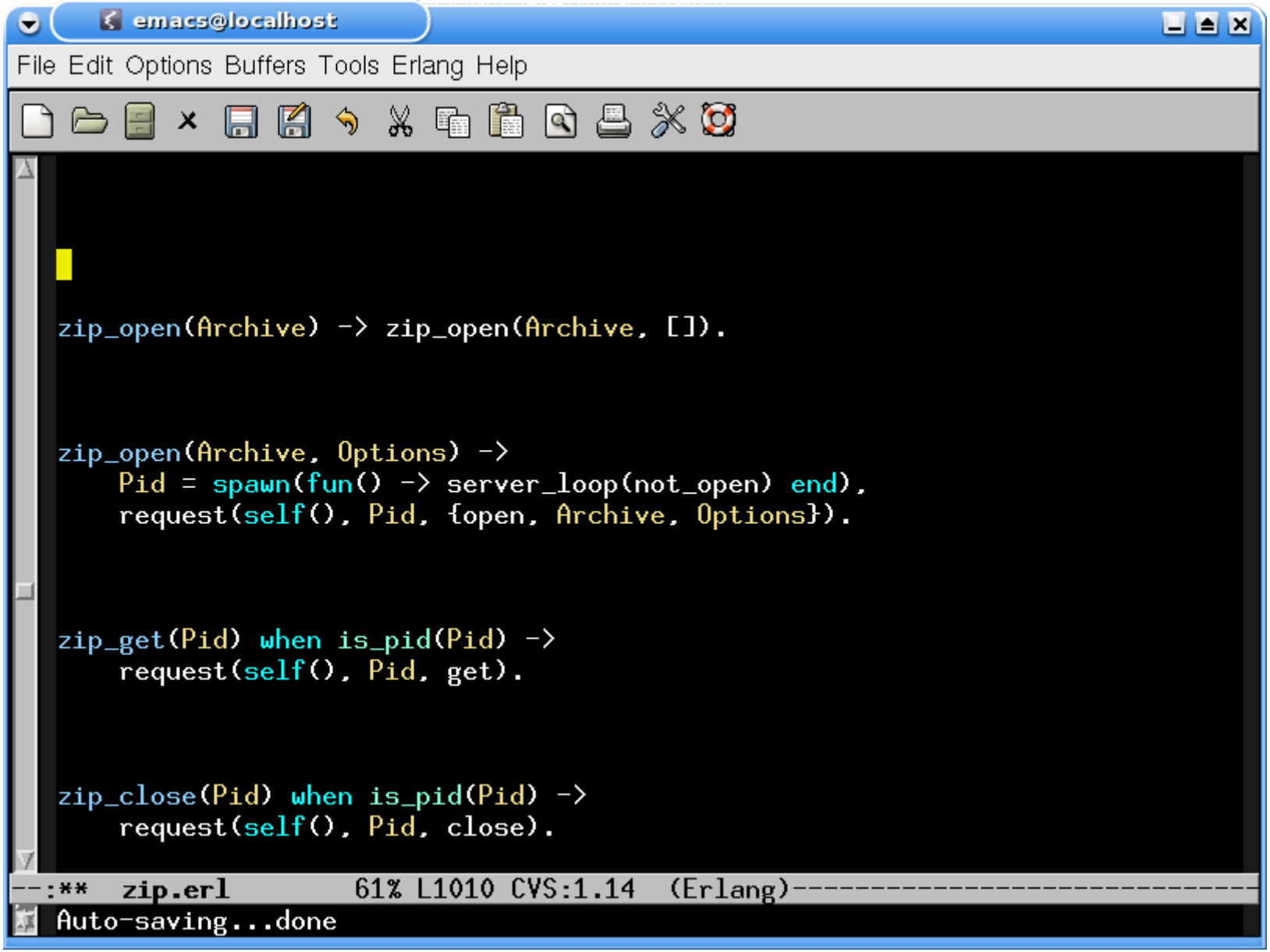
- Can also automatically annotate files

Add types to record fields

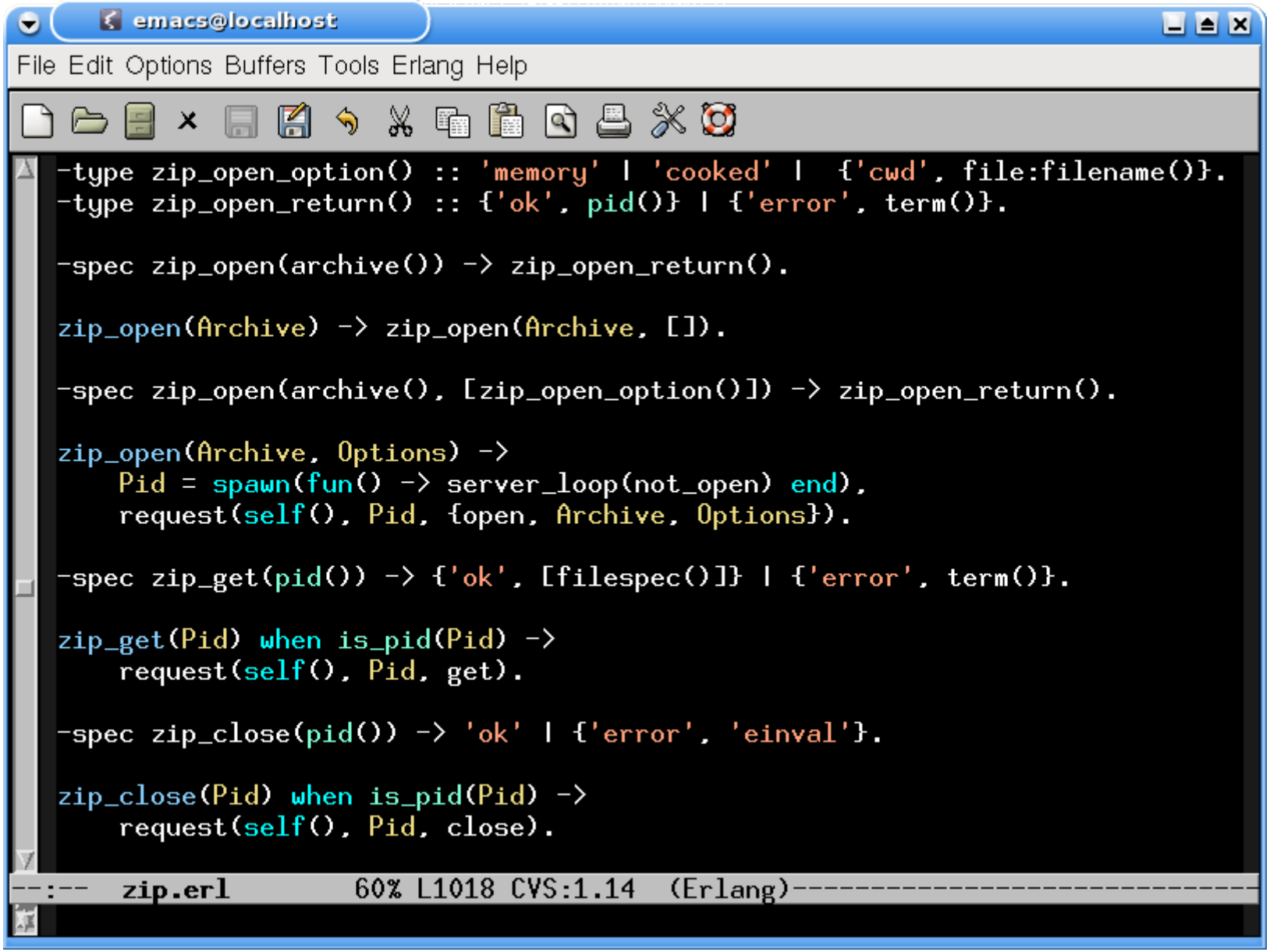
```
-record(hostent,
        {
          h_name,              %% official name of host
          h_aliases = [],      %% alias list
          h_addrtype,          %% host address type
          h_length,            %% length of address
          h_addr_list = []     %% list of addresses from ...
         })
```

```
-record(hostent,
        {
          h_name               :: hostname(), %% official...
          h_aliases = []       :: [hostname()],
          h_addrtype           :: 'inet' | 'inet6',
          h_length             :: non_neg_integer(), %% ...
          h_addr_list = []     :: [ip_address()]     %% ...
        })
```

How Erlang modules used to look like



How modern Erlang modules look



Tool #3

Tidier

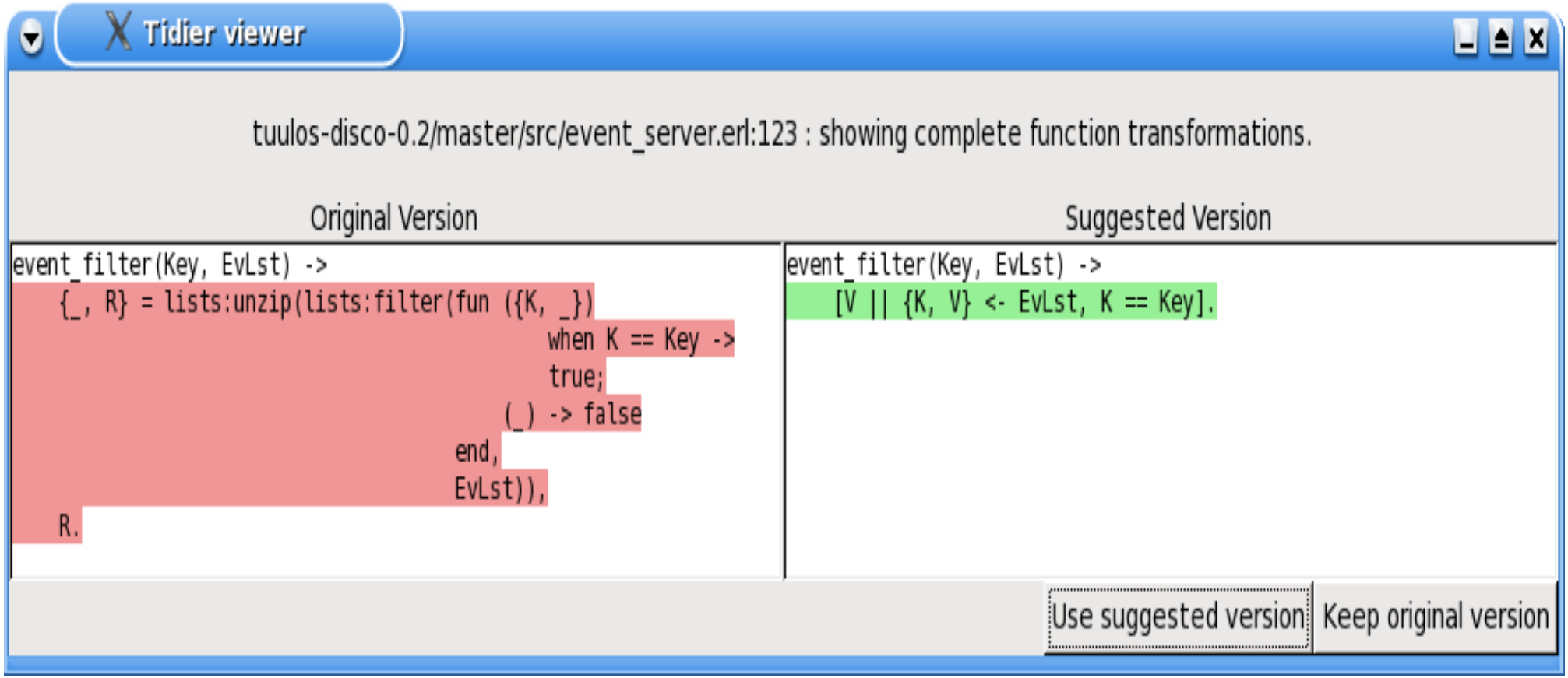
Tidier: An automatic refactoring tool

- Uses static analysis and transformation to:
  - Clean up Erlang code at the source level
  - Modernize outdated language constructs
  - Eliminate certain bad code smells from programs
  - Improve performance of applications

Properties of the transformations

- **Semantics preserving**
  - All transformations are conservative
- **Improving some aspect of the code**
  - Newer instead of an older/obsolete constructs
  - Smaller and/or more elegant code
  - Redundancy elimination
  - Performance improvement
- **Syntactically pleasing and natural**
  - Similar to what an expert Erlang programmer would have written if transforming the code by hand

Tidier in action



http://tidier.softlab.ntua.gr/

Current set of transformations

- Simple transformations and modernizations
- Record transformations
- List comprehension transformations
- Code simplifications and specializations
- Redundancy elimination transformations
- List comprehension simplifications
- Zip, unzip and deforestations
- Transformations improving runtime performance

lib/kernel/src/group.erl:368

```
case get_value(binary, Opts, case get(read_mode) of
                                 binary -> true;
                                 _ -> false
                             end) of
    true -> ...
```

```
case get_value(binary, Opts, get(read_mode) =:= binary) of
    true -> ...
```

lib/hipe/cerl/cerl_to_icode.erl:2370

```
is_pure_op(N, A) ->
    case is_bool_op(N, A) of
        true -> true;
        false ->
            case is_comp_op(N, A) of
                true -> true;
                false -> is_type_test(N, A)
            end
    end.
```

```
is_pure_op(N, A) ->
    is_bool_op(N, A) orelse is_comp_op(N, A)
                     orelse is_type_test(N, A).
```

lib/inviso/src/inviso_tool_sh.erl:1638

```
get_all_tracing_nodes_rtstates(RTStates) ->
  lists:map(fun ({N,_,_}) -> N end,
            lists:filter(fun ({_,{tracing,_},_}) ->
                                 true;
                             (_) -> false
                         end,
                         RTStates)).
```

```
get_all_tracing_nodes_rtstates(RTStates) ->
  [N || {N,{tracing,_},_} <- RTStates].
```

wrangler/src/refac_rename_fun.erl:344

```
lists:map(fun ({_, X}) -> X end,
          lists:filter(fun (X) ->
                           case X of
                             {atom, _X} -> true;
                             _ -> false
                           end
                       end,
                       R))
```

```
[X || {atom, X} <- R]
```

yaws/src/yaws_ls.erl:255

```
mkrandbytes(N) ->
  list_to_binary(lists:map(fun(N) ->
                               random:uniform(256) - 1
                           end, lists:seq(1,N))).
```

```
mkrandbytes(N) ->
  << <<(random:uniform(256)-1)>> || _ <- lists:seq(1,N)>>.
```

disco-0.2/master/src/event_server.erl:123

```
event_filter(Key, EvList) ->
    Fun = fun ({K, _}) when K == Key ->
                  true;
              (_) ->
                  false
          end,
    {_, R} = lists:unzip(lists:filter(Fun, EvList)),
    R.
```

```
event_filter(Key, EvList) ->
    [V || {K, V} <- EvList, K == Key].
```

Quote from a **tidier** user

I just ran a little demo for tidier here for ..., ..., ..., and .... Many laughs and comments like "*whose code is that? Mine?!!*" and a couple of "*I didn't know you could write that like that*".

I'd like to force everyone to set it up and run tidier on the code they are responsible for, as a learning experience for many of the more junior developers (and for some senior ones as well, apparently...).

Tool #4

PropEr

PropEr: A property-based testing tool

- Inspired by QuickCheck
- Available open source under GPL
- Has support for
  - Writing properties and test case generators

    `?FORALL/3, ?IMPLIES, ?SUCHTHAT/3, ?SHRINK/2, ?LAZY/1, ?WHENFAIL/2, ?LET/3, ?SIZED/2, aggregate/2, choose2, oneof/1, ...`
  - Concurrent/parallel “statem” testing
- Fully integrated with the language of types and specs
  - Generators often come for free!

Testing simple properties (1)

```
-module(simple_props).

-export([delete/2]).
%% Properties are automatically exported.
-include_lib("proper/include/proper.hrl").

delete(X, L) ->
    delete(X, L, []).

delete(_, [], Acc) ->
    lists:reverse(Acc);
delete(X, [X|Rest], Acc) ->
    lists:reverse(Acc) ++ Rest;
delete(X, [Y|Rest], Acc) ->
    delete(X, Rest, [Y|Acc]).
```

```
prop_delete() ->
  ?FORALL({X,L}, {integer(),list(integer())},
          not lists:member(X, delete(X,L))).
```

Testing simple properties (2)

```
%% Testing the base64 module:
%%   encode should be symmetric to decode:

prop_enc_dec() ->
  ?FORALL(Msg, union([binary(), list(range(1,255))]),
       begin
         EncDecMsg = base64:decode(base64:encode(Msg)),
         case is_binary(Msg) of
           true  -> EncDecMsg =:= Msg;
           false -> EncDecMsg =:= list_to_binary(Msg)
         end
       end).
```

Automatically testing specs

```
-module(specs).

-export([divide/2, filter/2, max/1]).

-spec divide(integer(), integer()) -> integer().
divide(A, B) ->
    A div B.

-spec filter(fun((T) -> term()), [T]) -> [T].
filter(Fun, List) ->
    lists:filter(Fun, List).

-spec max([T]) -> T.
max(List) ->
    lists:max(List).
```

Automatically using types as generators

- We want to test that **`array:new/0`** can handle any combination of options
- Why write a custom generator (which may rot)?
- We can use the type in that file as a generator!

```
-type array_opt() :: 'fixed' | non_neg_integer()
                   | {'default', term()}
                   | {'fixed', boolean()}
                   | {'size', non_neg_integer()}.
-type array_opts() :: array_opt() | [array_opt()].
```

```
-module(types).
-include_lib("proper/include/proper.hrl").

prop_new_array_opts() ->
    ?FORALL(Opts, array:array_opts(),
            array:is_array(array:new(Opts))).
```

Tool #5

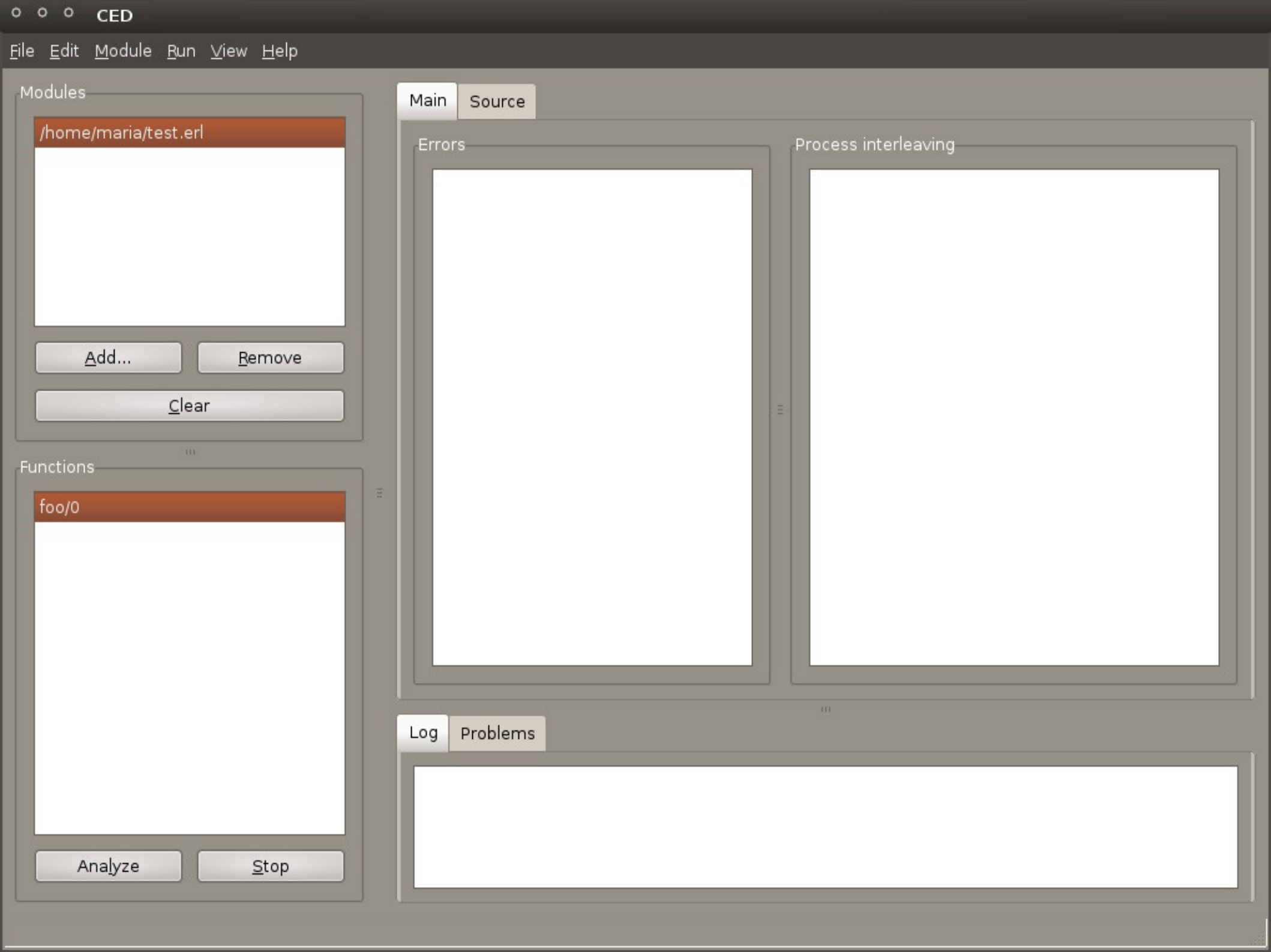
CED

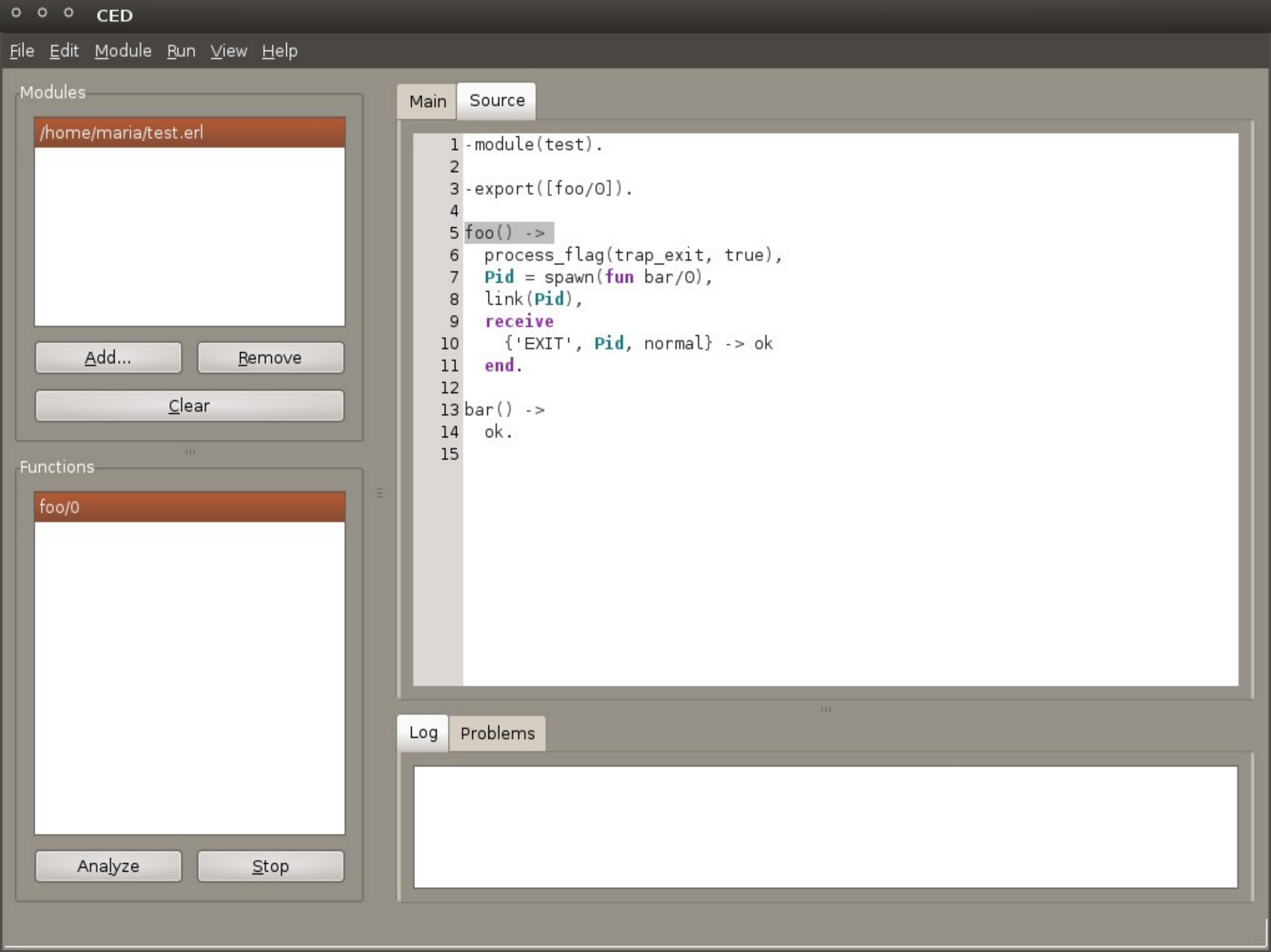
CED: Concurrency Error Detector

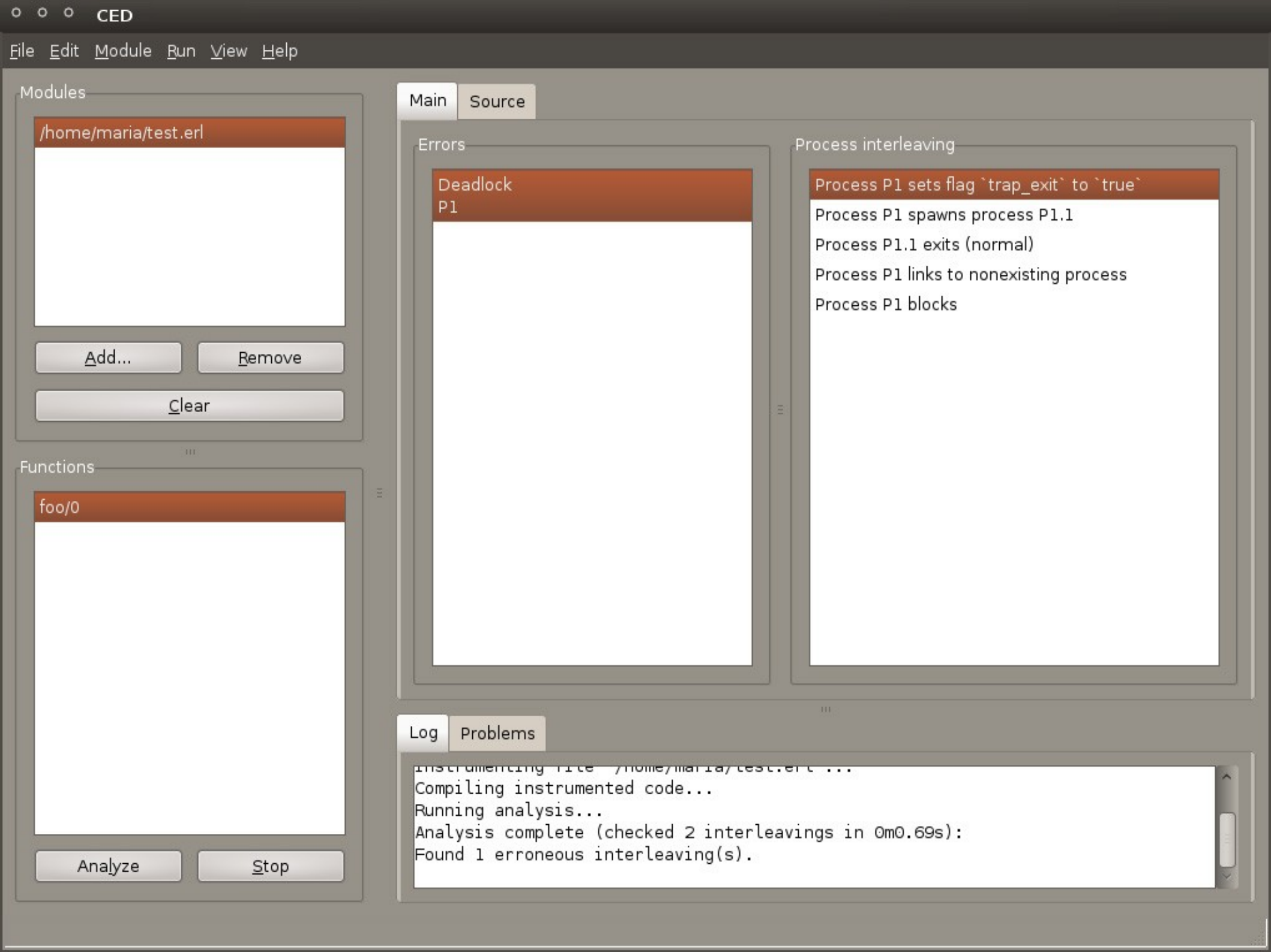
- Detects some concurrency errors (or verifies their absence) by controlling process interleavings
  - Systematically explores program state space
- Uses existing tests to detect
  - Exceptions (due to race conditions)
  - Assertion violations
  - Deadlocks
- Can consistently reproduce an erroneous execution

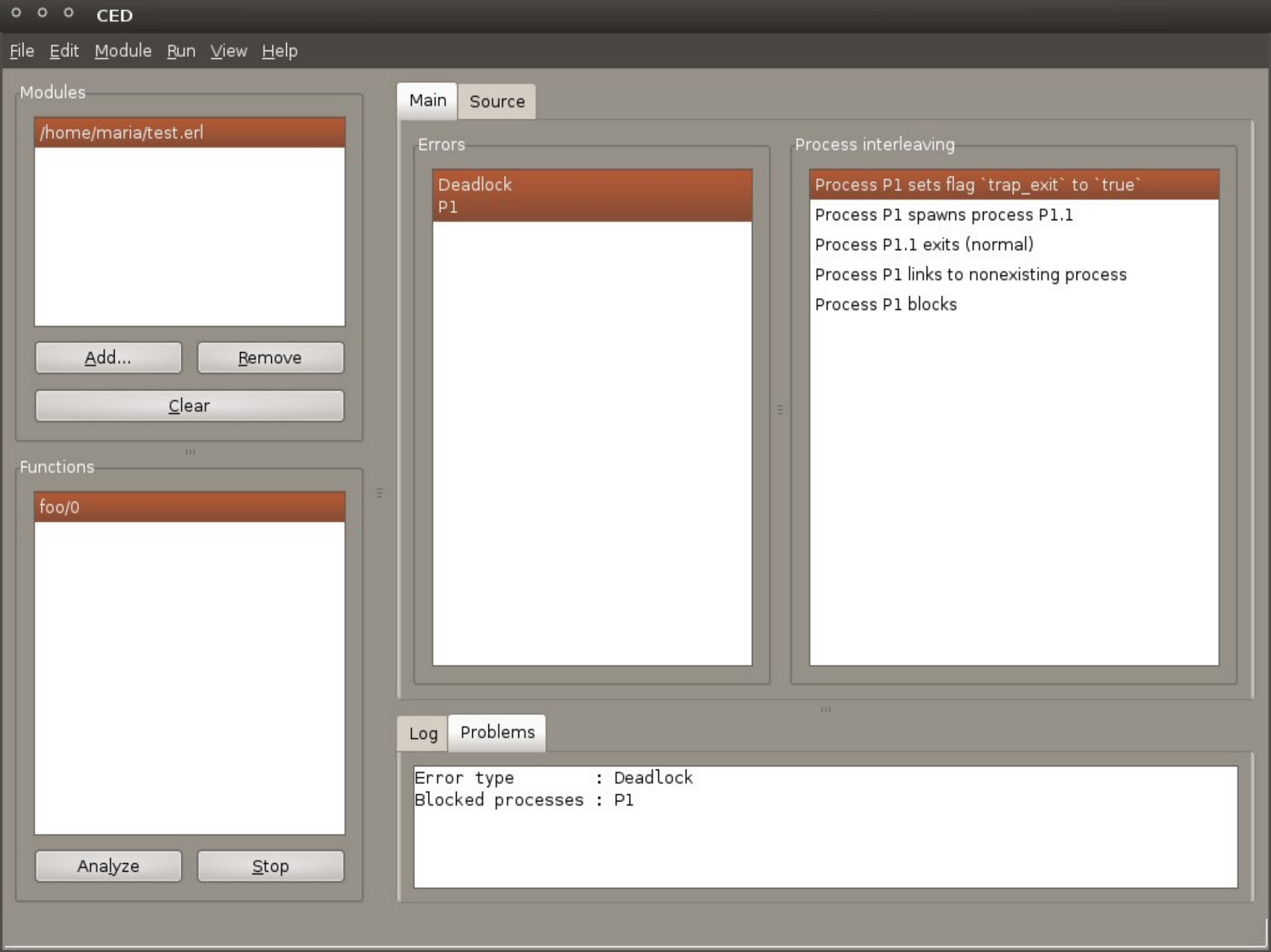
CED: Example

```
-module(test).

-export([foo/0]).

foo() ->
  process_flag(trap_exit, true),
  Pid = spawn(fun bar/0),
  link(Pid),
  receive
    {'EXIT', Pid, normal} -> ok
  end.

bar() ->
  ok.
```


CED
File Edit Module Run View Help
Modules
/home/maria/test.erl
Add...
Remove
Clear
Functions
foo/0
Analyze
Stop
Main
Source
Errors
Process interleaving
Log
Problems


CED
File Edit Module Run View Help
Modules
/home/maria/test.erl
Add...
Remove
Clear
Functions
foo/0
Analyze
Stop
Main
Source
1 -module(test).
2
3 -export([foo/0]).
4
5 foo() ->
6 process_flag(trap_exit, true),
7 Pid = spawn(fun bar/0),
8 link(Pid),
9 receive
10 {'EXIT', Pid, normal} -> ok
11 end.
12
13 bar() ->
14 ok.
15
Log
Problems


CED
File Edit Module Run View Help
Modules
/home/maria/test.erl
Add...
Remove
Clear
Functions
foo/0
Analyze
Stop
Main
Source
Errors
Deadlock
P1
Process interleaving
Process P1 sets flag `trap_exit` to `true`
Process P1 spawns process P1.1
Process P1.1 exits (normal)
Process P1 links to nonexisting process
Process P1 blocks
Log
Problems
Compiling instrumented code...
Running analysis...
Analysis complete (checked 2 interleavings in 0m0.69s):
Found 1 erroneous interleaving(s).


CED
File Edit Module Run View Help
Modules
/home/maria/test.erl
Add...
Remove
Clear
Functions
foo/0
Analyze
Stop
Main
Source
Errors
Deadlock
P1
Process interleaving
Process P1 sets flag `trap_exit` to `true`
Process P1 spawns process P1.1
Process P1.1 exits (normal)
Process P1 links to nonexisting process
Process P1 blocks
Log
Problems
Error type : Deadlock
Blocked processes : P1

CED: Future extensions

- Use partial order reduction to speed up execution
  - by avoiding redundant interleavings
- Allow selective instrumentation
- Enhance compatibility with **eunit** & **common_test**

Concluding remarks

Described the uses of some tools for modern Erlang program development:

- **Dialyzer**: Automatically identifies bugs or issues in ths code that need to be fixed
- **Typer**: Displays/adds type information to programs
- **Tidier**: Cleans up Erlang source code
- **Proper**: Performs semi-automatic property based testing
- **CED**: Systematically runs a test suite under all/some process inter-leavings

Where can I find these tools?

Dialyzer & Typer

- They are part of Erlang/OTP

Tidier

- Use of the tool is free via tidier's web site

  `http://tidier.softlab.ntua.gr/`

- The tool is also available by purchasing a license

Proper & CED

- They are open source

  `https://github.com/manopapad/proper/`

  `https://github.com/mariachris/CED/`

A team effort

Dialyzer

- Tobias Lindahl (UU/Klarna)
- Maria Christakis & Stavros Aronis (NTUA)

Typer

- Tobias Lindahl & Bingwen He (UU)

Tidier

- Thanassis Avgerinos (NTUA/CMU)

Proper

- Manolis Papadakis & Eirini Arvaniti (NTUA)

CED

- Alkis Gotovos & Maria Christakis (NTUA)