ProTest

property based testing

Simon Thompson, Clara Benac Earle

University of Kent, Universidad Politécnica de Madrid

QuviQ
ERICSSON
The University Of Sheffield.
AVANCEZ
1829
Erlang
Erlang Solutions Ltd.
formerly Erlang Training and Consulting Ltd.
University of Kent
Computing
lambdastream
POLITÉCNICA
"Ingeniamos el futuro"
Norway
Bergen
Uppsala
Helsinki
Sankt-Peterburg
(Санкт-Петербург)
Tallinn
Göteborg
Sea
United Kingdom
Belfast
Cork
Cardiff
London
Bremen
Berlin
Bydgoszcz
Nederland
Netherlands
Hannover
Polska
Poland
Беларусь
Belarus
Mogilev
Gomel
België
Belgique
Belgium
Deutschland
Germany
Dresden
Kraków
Kiev
(Київ)
Frankfurt am Main
Česká Rep
Czech Rep
Ostrava
Le Havre
Paris
Reims
München
Slovensko
Slovakia
Україна
Ukraine
Celtic Sea
Rennes
Strasbourg
Wien
Österreich
Austria
Moldova
Chişinău
Nantes
Bern
Magyarország
Hungary
France
Milano
Zagreb
Beograd
(Београд)
România
Romania
Odessa
(Одеса)
Bay of
Lyon
Torino
Bologna
Hrvatska
Croatia
Bordeaux
Montpellier
Nice
Србија
Serbia
Bucureşti
Toulouse
Marseille
Italia
Italy
България
Bulgaria
Porto
Barcelona
Roma
Bari
Prishtinë
Скопје
Istanbul
Tirane
Portugal
Lisboa
Tyrrhenian Sea
Napoli
Ελλάς
Greece
Bursa
Ankara
Eskişehir
Izmir
Konya
Palermo
Catania
Athina
(Αθήνα)
Antalya

ProTest
property based testing

ProTest goals

Integrate property-based testing into the development life cycle:

- Property discovery
- Test and property evolution
- Property monitoring
- Analysing concurrent systems

Property-based testing

Describe the required behaviour of a system using logical properties …

… or abstract state machines.

Test the properties against random data.

Test machine compliance by random execution sequences.

ProTest tools

McErlang

PULSE

Exago
Onviso

QuviQ

QuickCheck

State Chum

Focus for this talk

McErlang

PULSE

Exago

Onviso

QuviQ

QuickCheck

State Chum

ProTest
property based testing

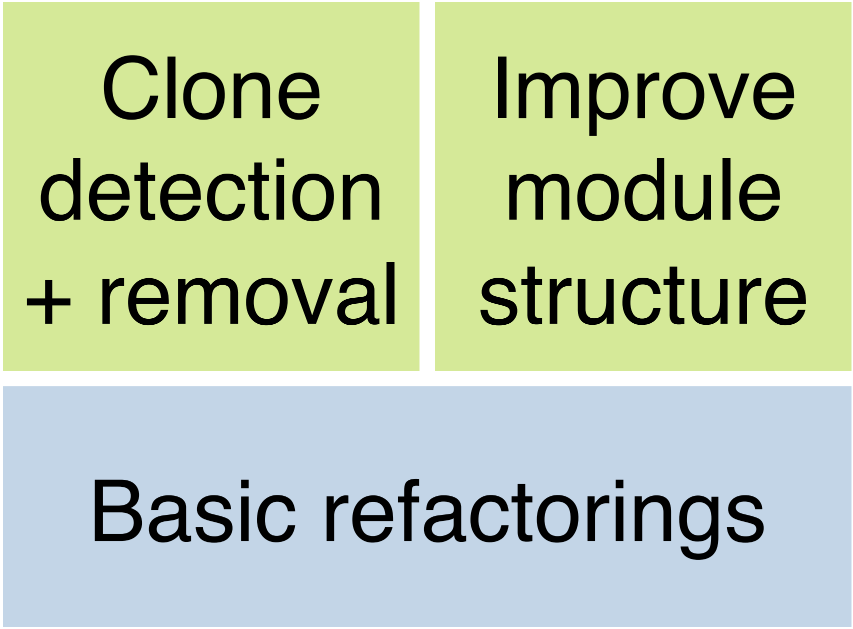
Wrangler

Interactive refactoring tool for Erlang

Integrated into Emacs and Eclipse / ErlIDE

Multiple modules

Structural, process, macro refactorings



Clone detection + removal

Improve module structure

Basic refactorings

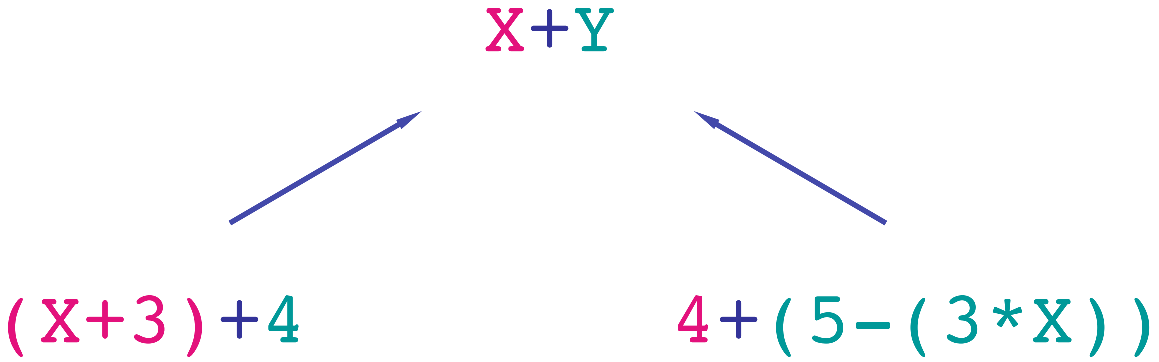
Refactoring and testing

- Clone detection and elimination in test code
- Property extraction through clone detection and FSM inference.
- Refactoring code and tests: frameworks.
- Refactoring tests in a framework.

Refactoring and testing

- Clone detection and elimination in test code
- Property extraction through clone detection and FSM inference.
- Refactoring code and tests: frameworks.
- Refactoring tests in a framework.

What is ‘similar’ code?



X+Y

(X+3)+4

4+(5-(3*X))

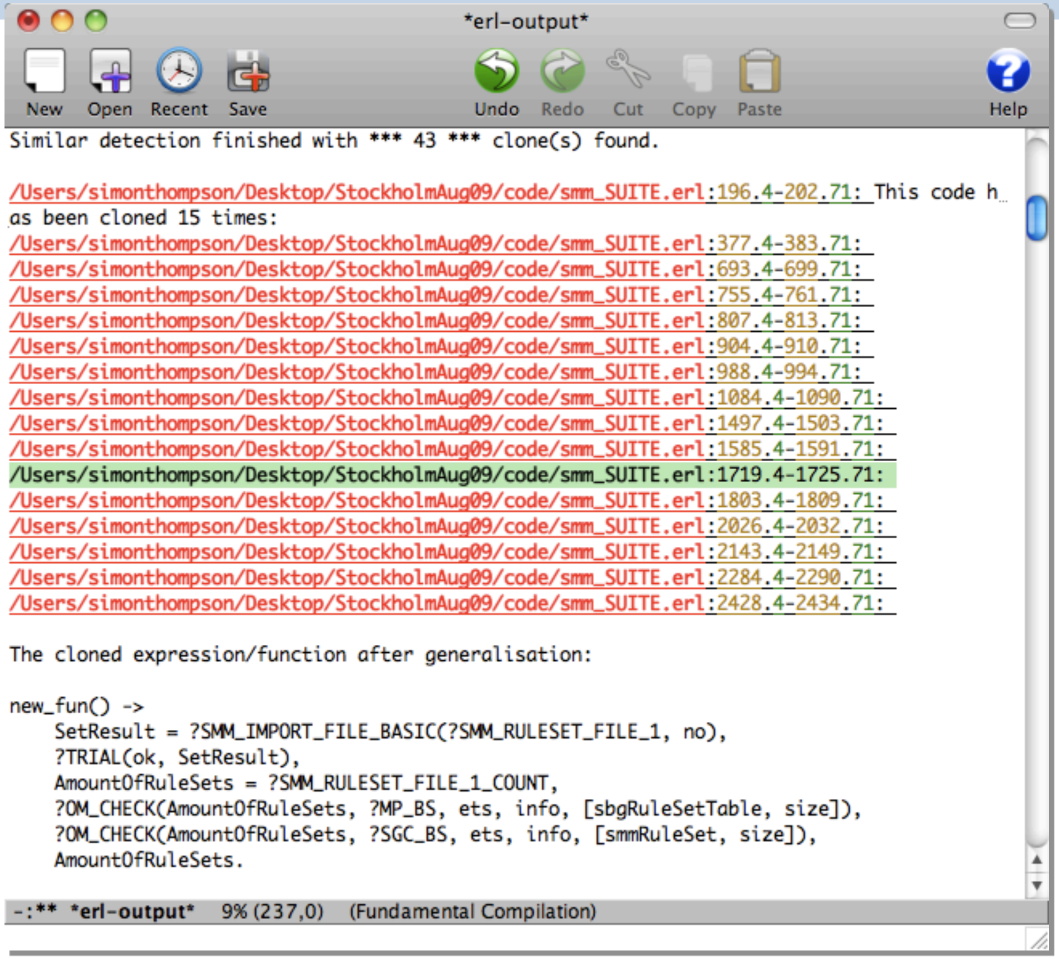
The anti-unification gives the (most specific) common generalisation.

ProTest property based testing

Step 1

The largest clone class has 15 members.

The suggested function has no parameters, so the code is literally repeated.



```
Similar detection finished with *** 43 *** clone(s) found.

/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:196.4-202.71: This code h…
as been cloned 15 times:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:377.4-383.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:693.4-699.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:755.4-761.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:807.4-813.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:904.4-910.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:988.4-994.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:1084.4-1090.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:1497.4-1503.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:1585.4-1591.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:1719.4-1725.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:1803.4-1809.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:2026.4-2032.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:2143.4-2149.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:2284.4-2290.71:
/Users/simonthompson/Desktop/StockholmAug09/code/smm_SUITE.erl:2428.4-2434.71:

The cloned expression/function after generalisation:

new_fun() ->
    SetResult = ?SMM_IMPORT_FILE_BASIC(?SMM_RULESET_FILE_1, no),
    ?TRIAL(ok, SetResult),
    AmountOfRuleSets = ?SMM_RULESET_FILE_1_COUNT,
    ?OM_CHECK(AmountOfRuleSets, ?MP_BS, ets, info, [sbgRuleSetTable, size]),
    ?OM_CHECK(AmountOfRuleSets, ?SGC_BS, ets, info, [smmRuleSet, size]),
    AmountOfRuleSets.
-:** *erl-output*    9% (237,0)    (Fundamental Compilation)
```

ProTest
property based testing

The general pattern

Identify a clone.

Introduce the corresponding generalisation.

Eliminate all the clone instances.

So what’s the complication?

What is the complication?

Which clone to choose?

Include all the code?

How to name functions and variables?

When and how to generalise?

'Widows' and 'orphans'

Clone elimination and testing

Copy and paste … many hands.

Shorter, more comprehensible and better structured code.

Emphatically not “push button” …

Need domain expert involvement.

Refactoring and testing

- Clone detection and elimination in test code
- Property extraction through clone detection and FSM inference.
- Refactoring code and tests: frameworks.
- Refactoring tests in a framework.

Property discovery in Wrangler

Find (test) code that is similar …

… build a common abstraction

… accumulate the instances

… and generalise the instances.

Example:

Test code from Ericsson: different media and codecs.

Generalisation to all medium/codec combinations.

Refactoring and testing

- Clone detection and elimination in test code
- Property extraction through clone detection and FSM inference.
- Refactoring code and tests: frameworks.
- Refactoring tests in a framework.

Testing frameworks

EUnit, Common Test and Quick Check each give a template for writing tests and a platform for performing them.

Want to refactor code and test code in step.

Extend refactorings while observing

- Naming conventions
- Macros
- Callbacks
- Meta-programming
- Coding patterns

Quick Check example

Callbacks, macros and meta-programming.

```
-export( …, command/1, postcondition/3, … ,prop/0]).

command({N}) when N<10 ->
     frequency([{3,{call,nat_gen,next,[]}},
                {1,{call,nat_gen,stop,[]}}]); …

postcondition({N},{call,nat_gen,next,_},R)-> R == N; …

prop() ->
  ?FORALL(Commands,commands(?MODULE),
    begin {_H,_S,Result} = run_commands(?MODULE,Commands),
          Result == ok end).
```

Quick Check example

Callbacks, macros and meta-programming.

```
-export( …, command/1, postcondition/3, … ,prop/0]).

command({N}) when N<10 ->
     frequency([{3,{call,nat_gen,next,[]}},
                {1,{call,nat_gen,stop,[]}}]); …

postcondition({N},{call,nat_gen,next,_},R)-> R == N; …

prop() ->
  ?FORALL(Commands,commands(?MODULE),
    begin {_H,_S,Result} = run_commands(?MODULE,Commands),
          Result == ok end).
```

Refactoring and testing

- Clone detection and elimination in test code
- Property extraction through clone detection and FSM inference.
- Refactoring code and tests: frameworks.
- Refactoring tests in a framework.

Refactoring within QuickCheck

FSM-based testing: transform state variable from simple value to record.

Stylised usage supports robust transformation.

Spinoff to OTP libs.

Property refactorings:

Introduce local definitions (LET)

Merge local definitions and quantifiers (FORALL).

[EUnit too …]

www.cs.kent.ac.uk/projects/wrangler/

→ GettingStarted

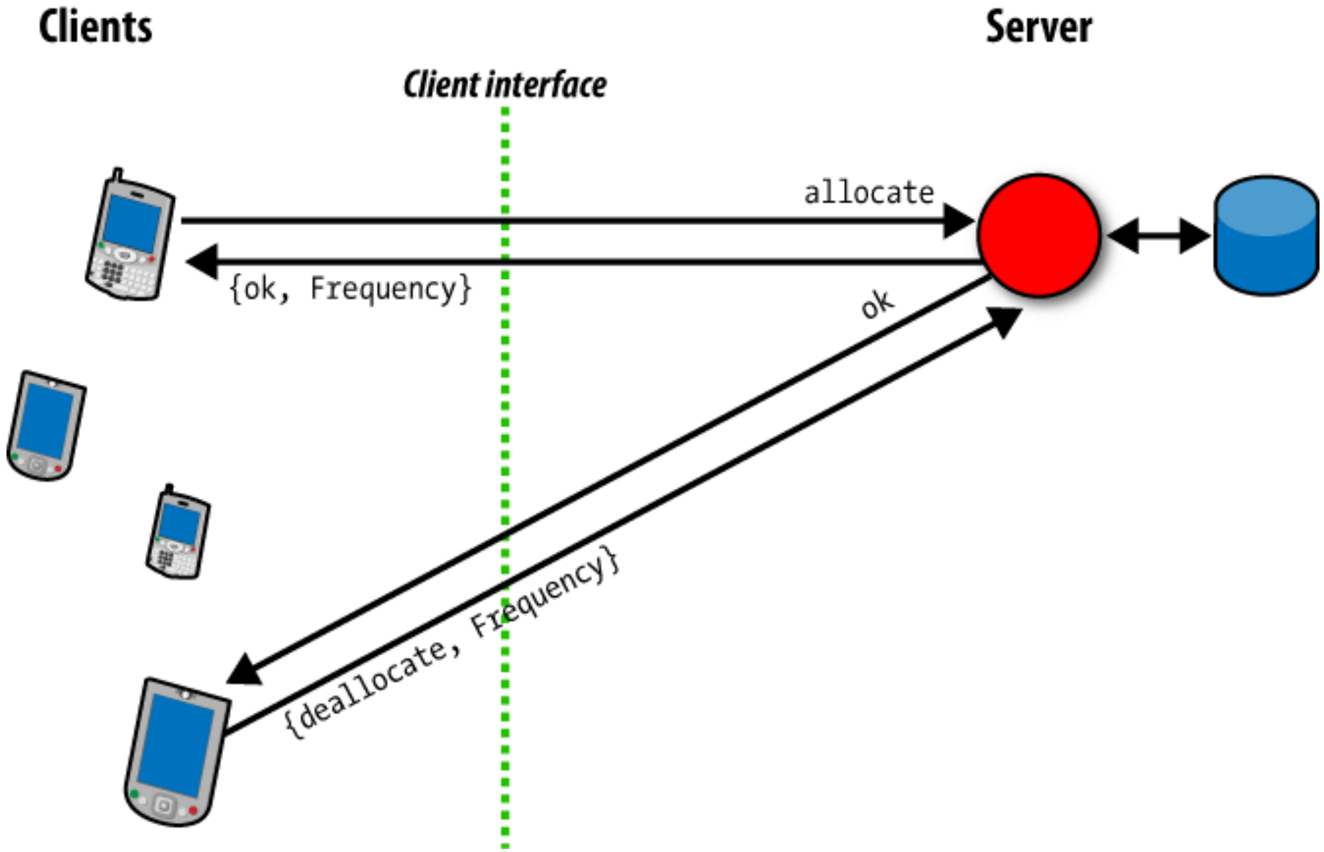
Inferring QuickCheck state machines from Eunit test sets

Thomas Arts, Simon Thompson

Chalmers University, University of Kent

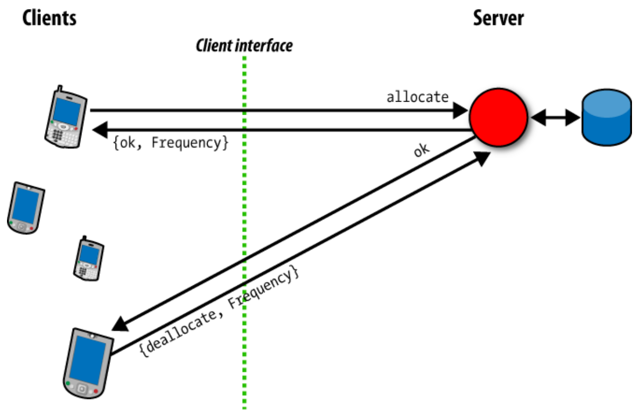
ProTest
property based testing

Server for mobile frequencies



Server for mobile frequencies

State-based system allows allocation and de-allocation of frequencies from an initial list, once system is started.



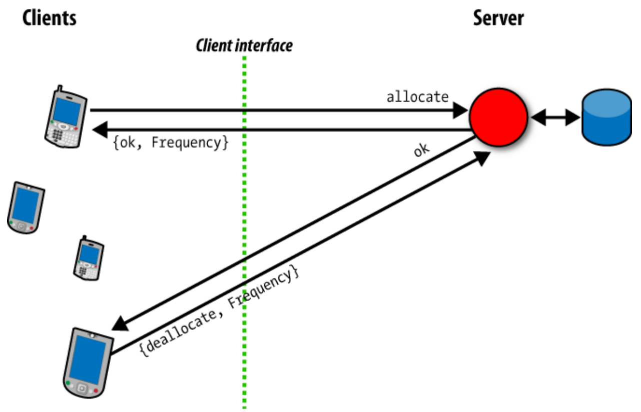
```
-spec start([integer()]) -> pid().
-spec stop() -> ok.
-spec allocate() -> {ok,integer()} |
                    {error, no_frequency}.
-spec deallocate(integer()) -> ok.
```

ProTest
property based testing

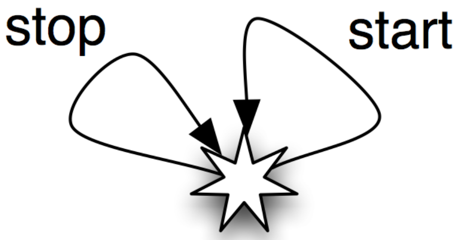
Testing start/stop behaviour

EUnit is a unit testing framework for Erlang.

Test start / stop behaviour.



```
startstop_test() ->
    ?assertMatch( … ,start([])),
    ?assertMatch(ok,stop()),
    ?assertMatch( … ,start([1])),
    ?assertMatch(ok,stop()).
```

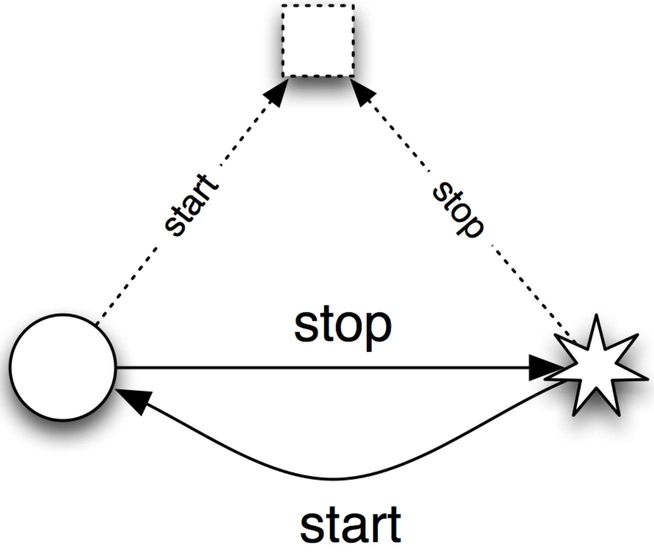


Final test set

```
startstop_test() ->
   ?assertMatch( … ,start([])),
   ?assertMatch(ok,stop()),
   ?assertMatch( … ,start([1])),
   ?assertMatch(ok,stop()).

stop_without_start_test() ->
     ?assertException(_,_,stop()).

start_twice_test_() ->
     {setup,
      fun() -> start([]) end,
      fun(_) -> stop() end,
      fun() -> ?assertException(_,_,start([])) end}.
```



Improved testing through inductive machine inference

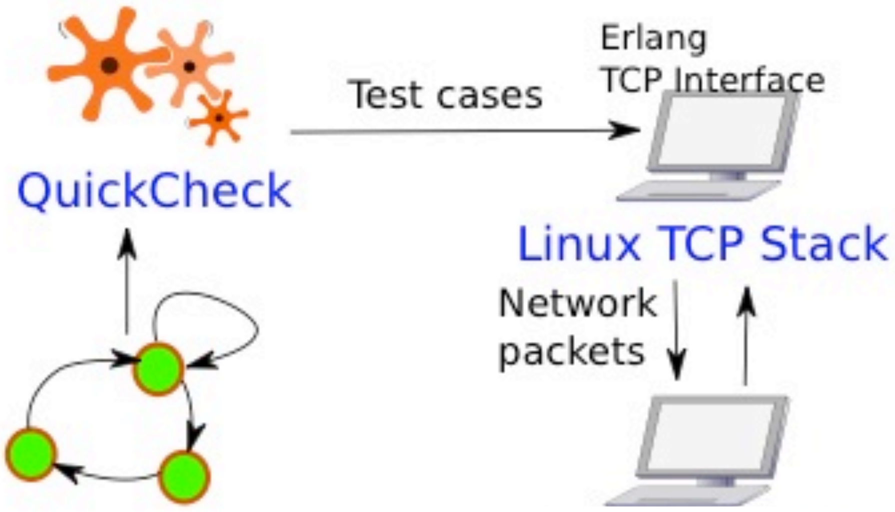
Neil Walkinshaw, John Derrick

University of Sheffield

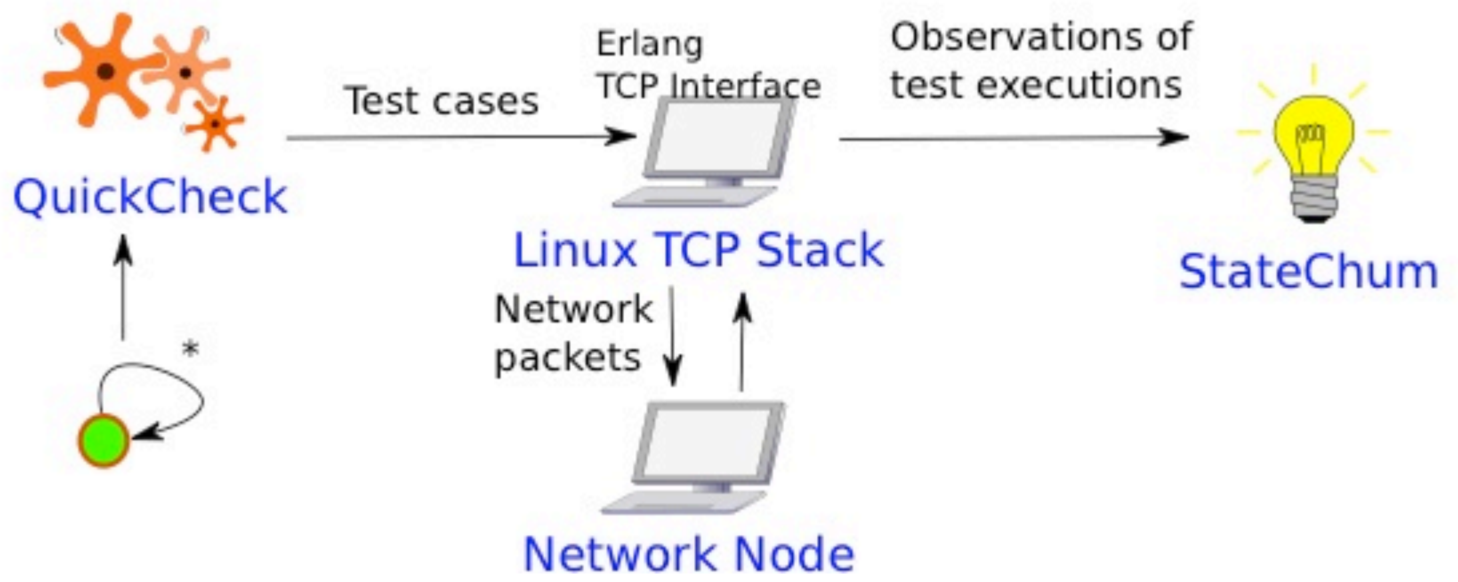
ProTest
property based testing

FSM-based testing

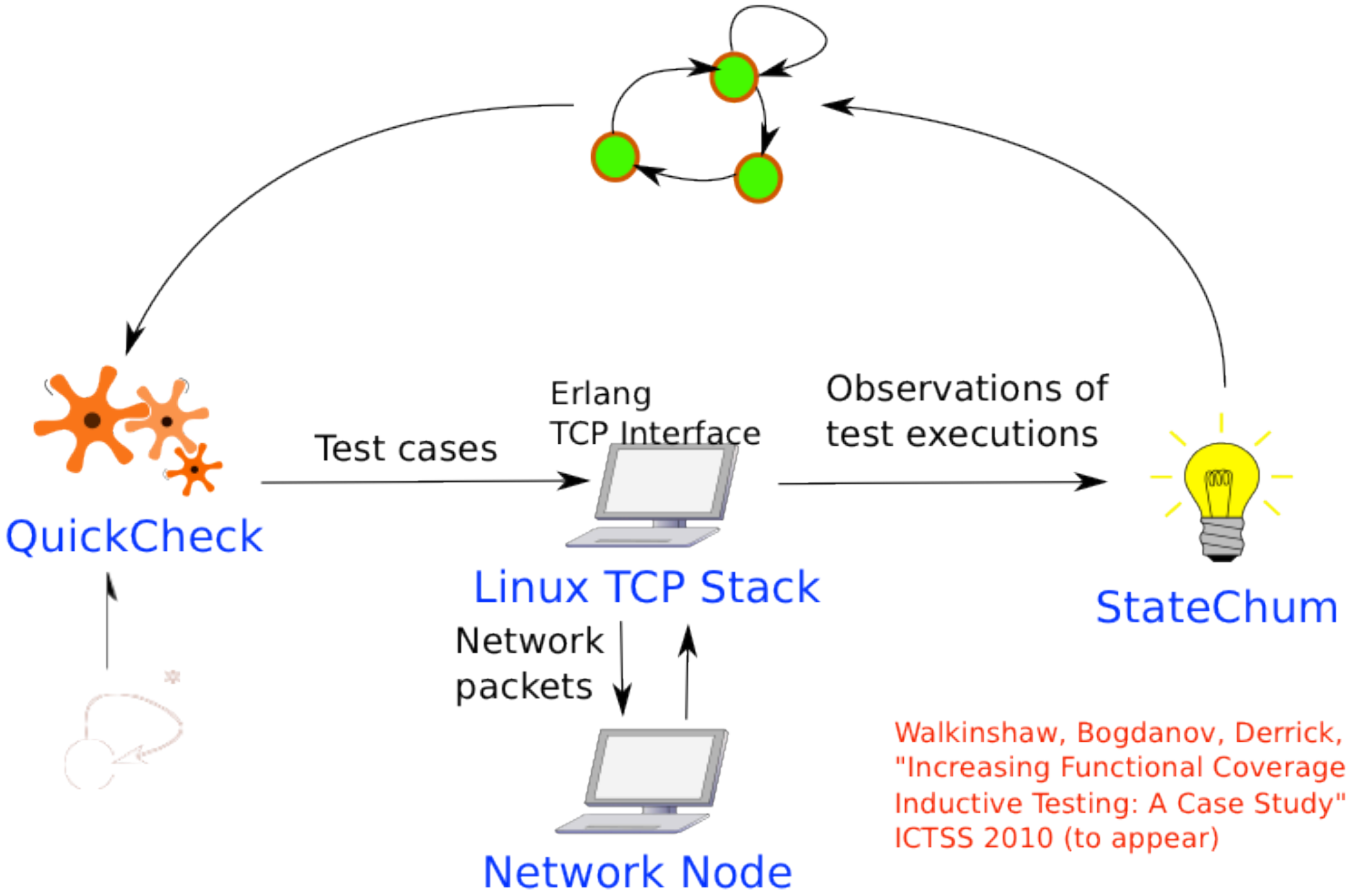
J. Paris and T. Arts,
Automatically Testing TCP/IP Implementations using QuickCheck,
8th ACM SIGPLAN Workshop on Erlang, 2009



Observe test executions



… and improve the FSM



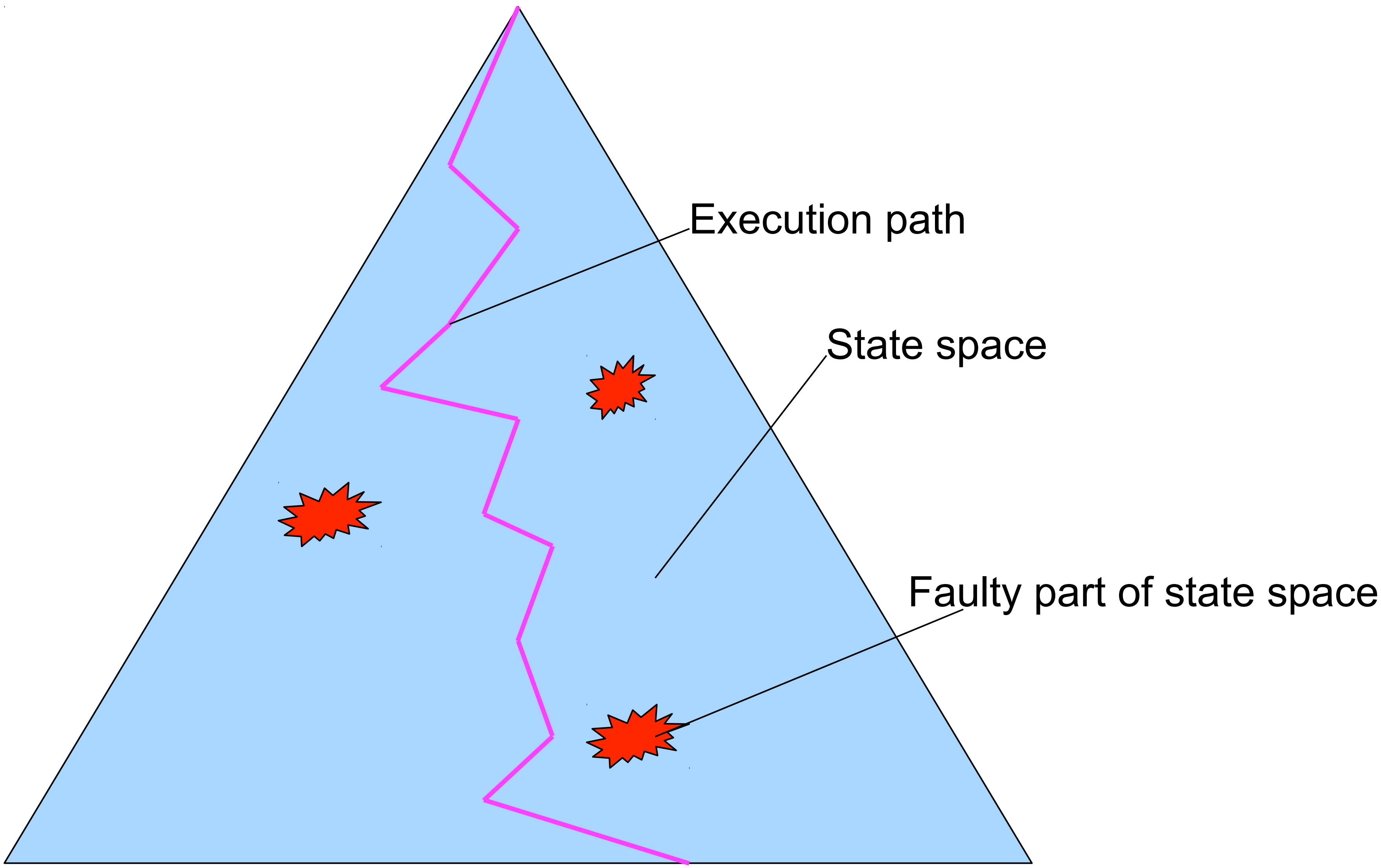
Walkinshaw, Bogdanov, Derrick, Paris - "Increasing Functional Coverage by Inductive Testing: A Case Study" ICTSS 2010 (to appear)

QuickCheck and McErlang integration

Clara Benac Earle, Lars-Åke Fredlund

UPM

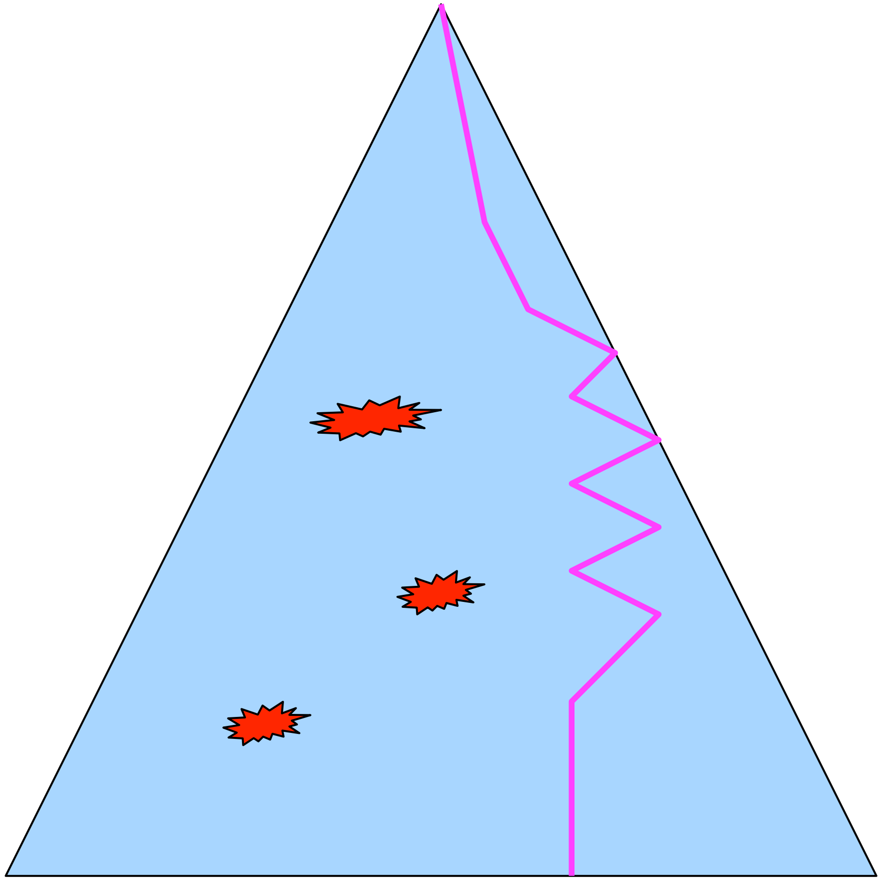
ProTest
property based testing



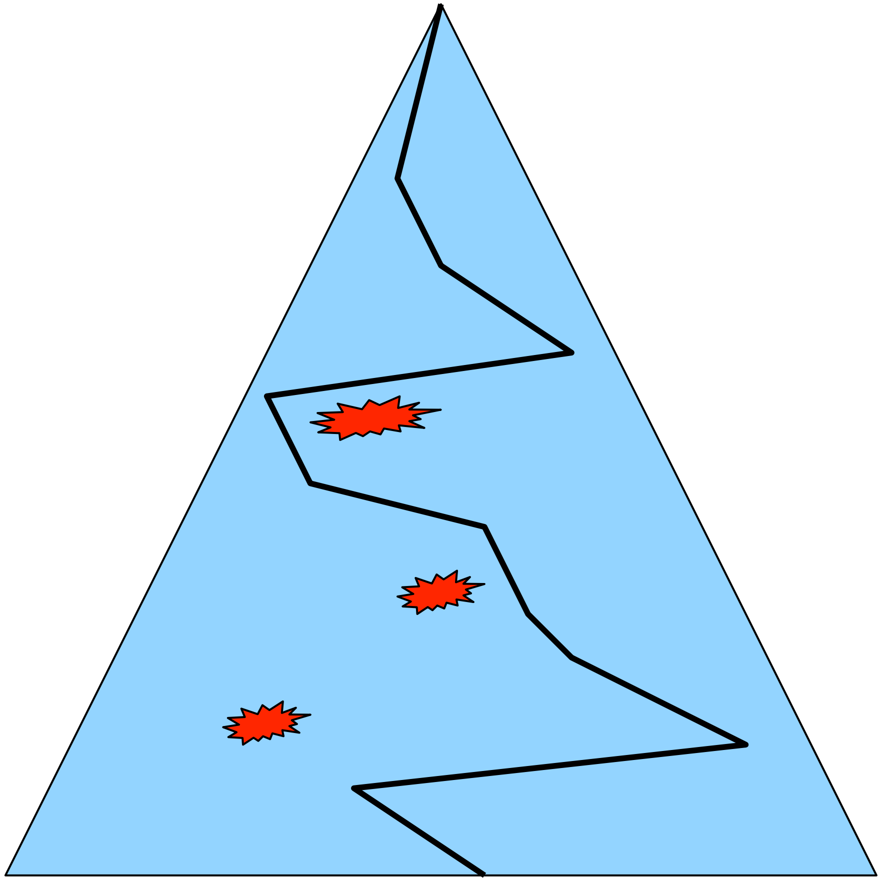
Execution path

State space

Faulty part of state space



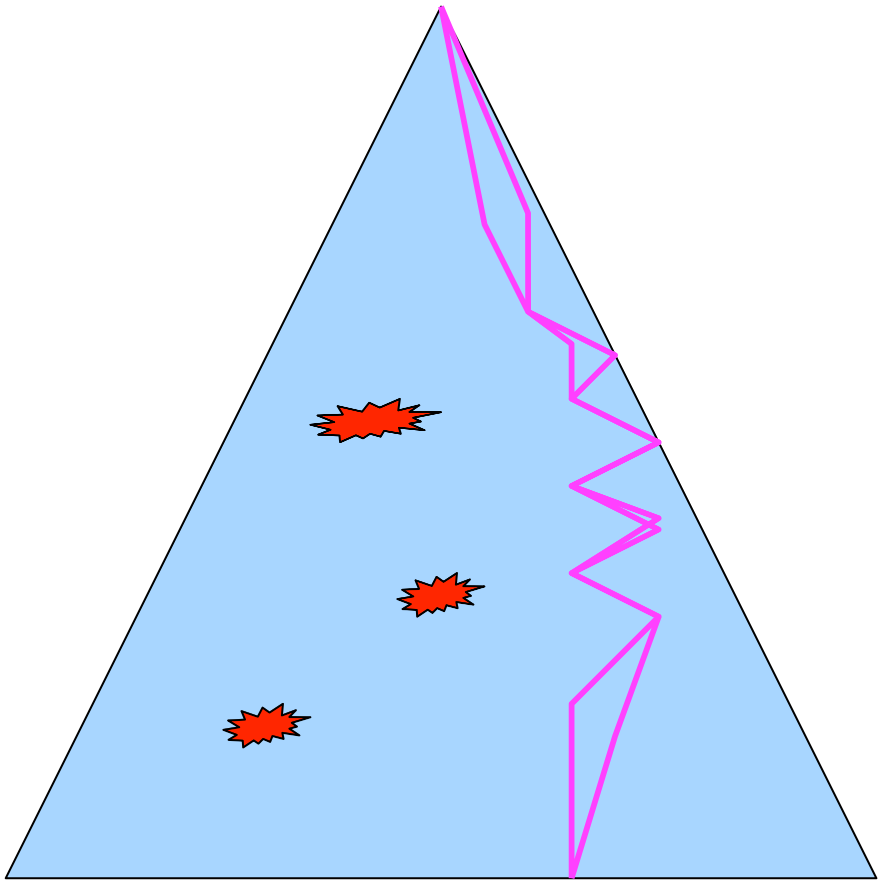
QuickCheck



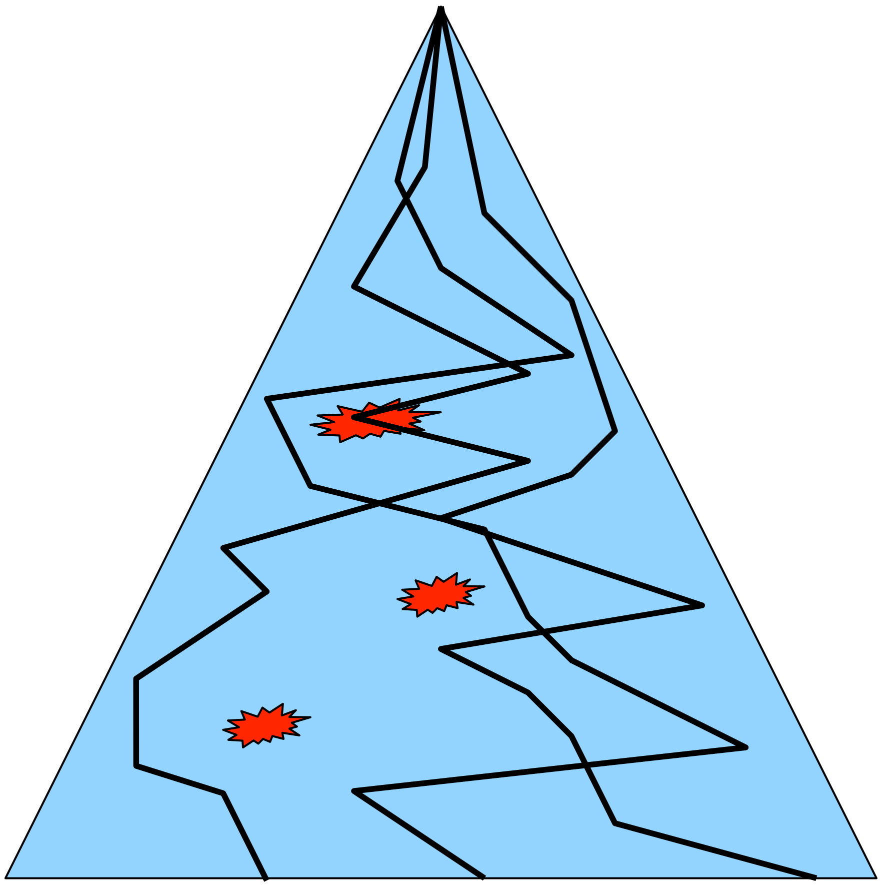
QuickCheck + PULSE

ProTest
property based testing

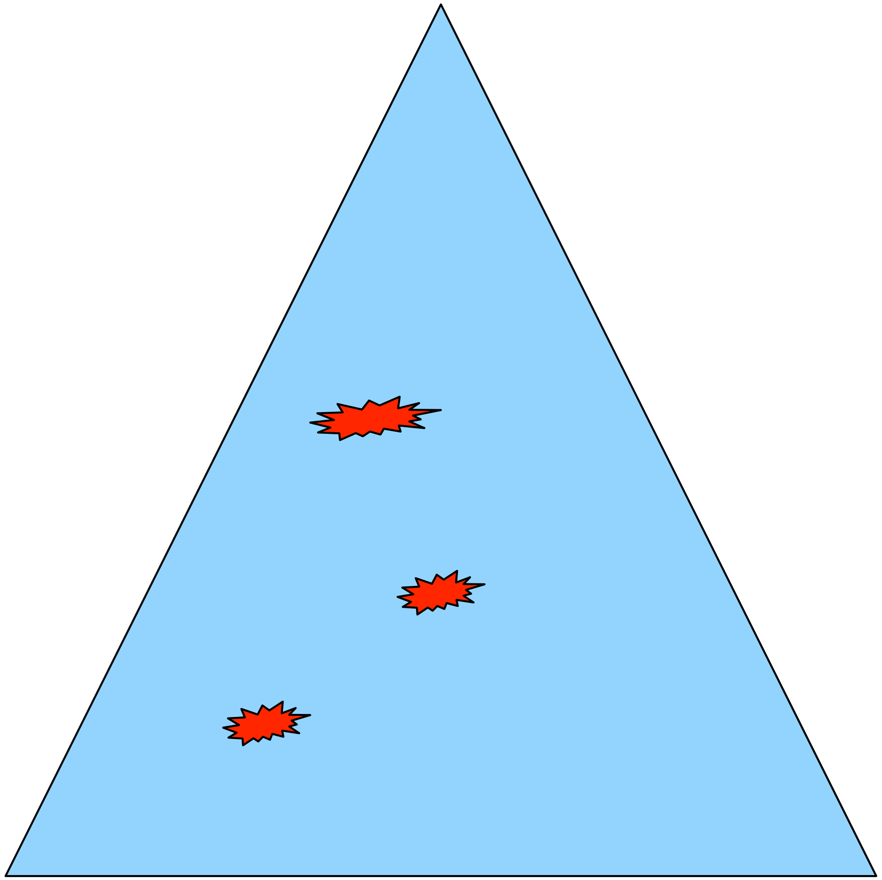
Repeat test N times – `?ALWAYS` macro



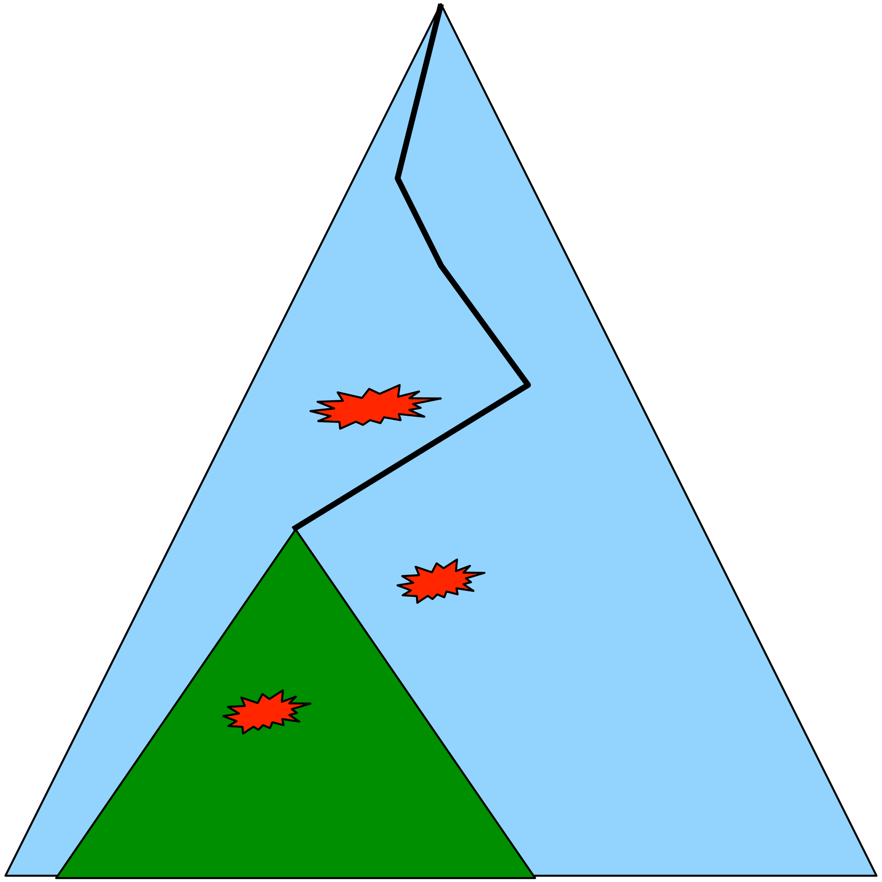
QuickCheck



QuickCheck + PULSE



QuickCheck + McErlang
optimal case

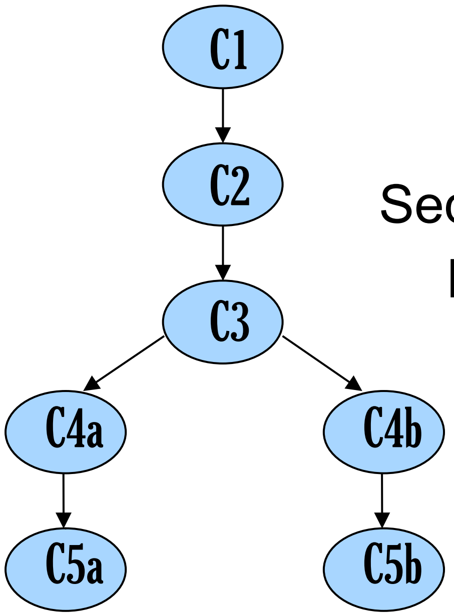


QuickCheck + McErlang
more common case

QuickCheck and McErlang integration

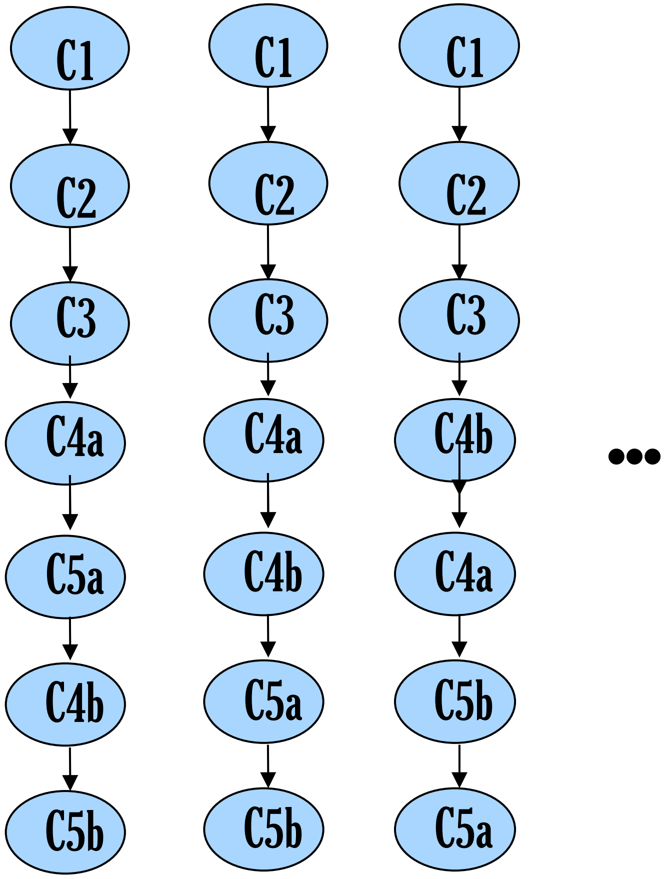
- The goal is to provide easy access to the power of model checking to QuickCheck users
- And to make McErlang more accessible through QuickCheck (generators, commands)
- We focus on the QuickCheck state machine library `eqc_statem`
- The `parallel_commands` is a suitable first functionality to integrate

Parallel commands

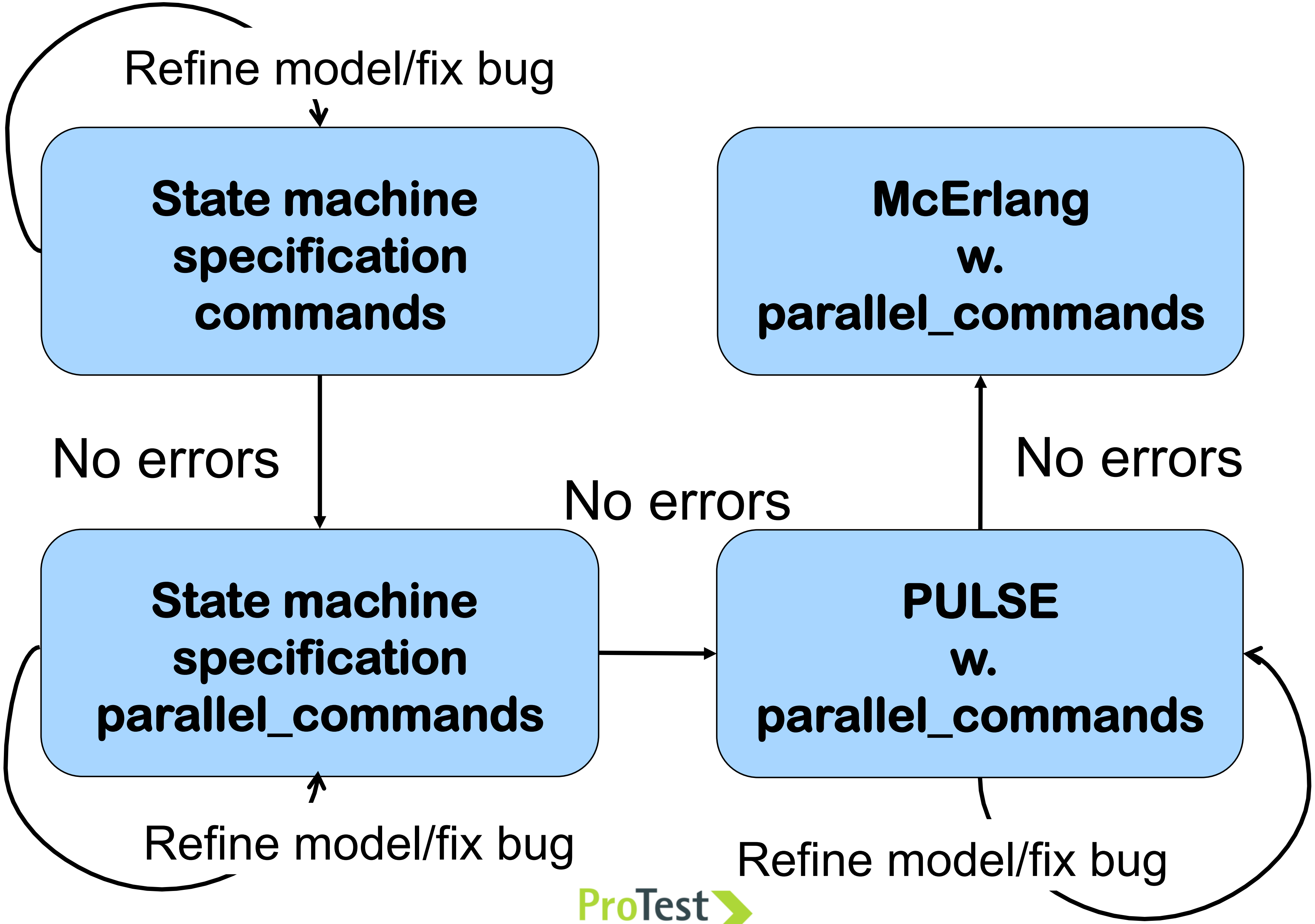


Sequential prefix

Parallel execution



Is there a linear execution “equivalent” to the parallel one?
(such that all command results are the same)



Refine model/fix bug

State machine specification commands

No errors

State machine specification parallel_commands

Refine model/fix bug

No errors

PULSE w. parallel_commands

Refine model/fix bug

No errors

McErlang w. parallel_commands

ProTest
property based testing

Implementation - basic QuickCheck

```
prop_testsomething() →
  ?FORALL(PCmds, parallel_commands(?MODULE),
    begin
      {H,S,Res} =
          run_parallel_commands(PCmds),
      ?WHENFAIL(io:format(...),
                Res == ok)
    end).
```

Implementation - PULSE

```
prop_testsomething() →
  ?FORALL(PCmds, parallel_commands(?MODULE),
    ?PULSE(
      [<instrumented-modules>], %Optional?
      {H,S,Res},
      begin
        run_parallel_commands(PCmds)
      end,
      ?WHENFAIL(io:format(...),
                Res == ok))).
```

Implementation - McErlang

```
prop_testsomething() →
  ?FORALL(PCmds, parallel_commands(?MODULE),
    ?MCERLANG(
      [<instrumented-modules>], %Optional?
      {H,S,Res},
      begin
        run_parallel_commands(PCmds)
      end,
      ?WHENFAIL(io:format(...),
                Res == ok))).
```

Behind the scenes

- Some QuickCheck code compiled with McErlang
- A McErlang application (usable standalone)
- Making McErlang behave better as a testing tool with finite resources:
  - Memory bounded tables
  - Time limit for model checking runs

https://babel.ls.fi.upm.es/trac/McErlang/wiki/QuickCheck/McErlang

Which verification method to use?

- How large is the state space?
- What is the density of faults?
- How critical is the application?
- What resources (memory/time) do we have?
- Is it better to generate many test cases?

… or to run the same test case many times?

… or explore more of its state space?

- We want to do more experiments and compare!

Conclusions

- Next release of QuickCheck will likely ship with McErlang integrated

- Benefits to QuickCheck: finding more bugs
- Benefits to McErlang: more users