

Two Testing Tools for the Erlang Ecosystem

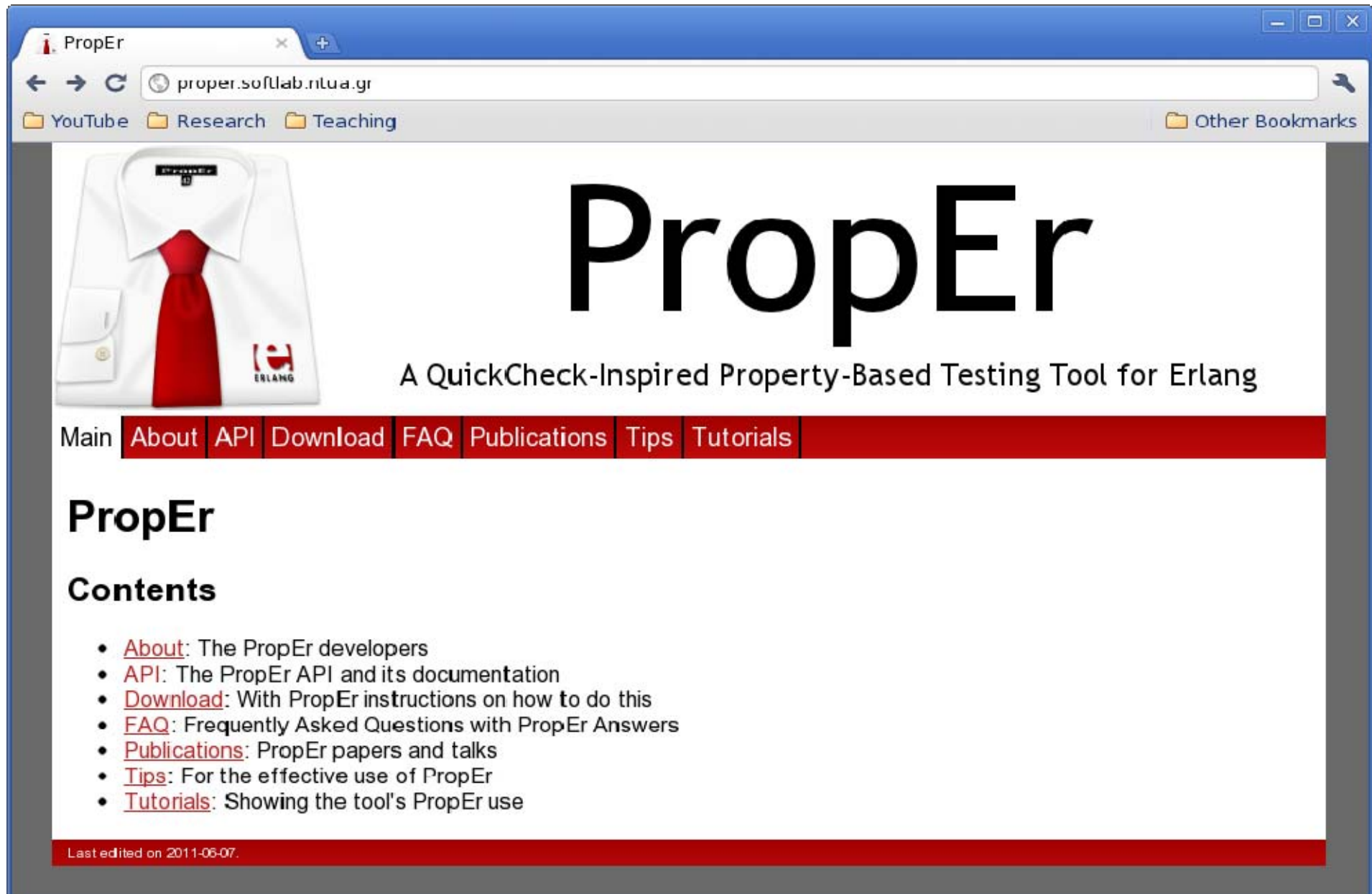
Kostis Sagonas

Some material is joint work with

Andreas Löscher

Stavros Aronis and Scott Lystig Fritchie

PropEr – proper.softlab.ntua.gr



The screenshot shows a web browser window with the address bar containing 'proper.softlab.ntua.gr'. The browser's bookmark bar includes 'YouTube', 'Research', 'Teaching', and 'Other Bookmarks'. The website content features a white dress shirt and a red tie on the left, with the 'PropEr' logo on the shirt. The main heading is 'PropEr' in a large, bold, black font. Below the heading is the subtitle 'A QuickCheck-Inspired Property-Based Testing Tool for Erlang'. A red navigation bar contains the following links: 'Main', 'About', 'API', 'Download', 'FAQ', 'Publications', 'Tips', and 'Tutorials'. The 'PropEr' heading is repeated in a smaller font. Below it is the 'Contents' section, which lists the following items:

- [About](#): The PropEr developers
- [API](#): The PropEr API and its documentation
- [Download](#): With PropEr instructions on how to do this
- [FAQ](#): Frequently Asked Questions with PropEr Answers
- [Publications](#): PropEr papers and talks
- [Tips](#): For the effective use of PropEr
- [Tutorials](#): Showing the tool's PropEr use

At the bottom of the page, a red footer bar contains the text 'Last edited on 2011-06-07.'

PropEr: A property-based testing tool

- Inspired by QuickCheck
- Open source
- 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, ...
 - Stateful (aka “statem” and “fsm”) testing
- Fully integrated with types and specs
 - Generators often come for free!
- Extensions for **targeted** property-based testing

Demo program

```
%% A sorting program, inspired by QuickSort
-module(demo).
-export([sort/1]).

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
    sort([X || X <- Xs, X < P])
    ++ [P] ++ sort([X || X <- Xs, P < X]).
```

```
Eshell v9.2.1 (abort with ^G)
1> demo:sort([]).
[]
2> demo:sort([17,42]).
[17,42]
3> demo:sort([42,17]).
[17,42]
4> demo:sort([3,1,2]).
[1,2,3]
```

A property for the demo program

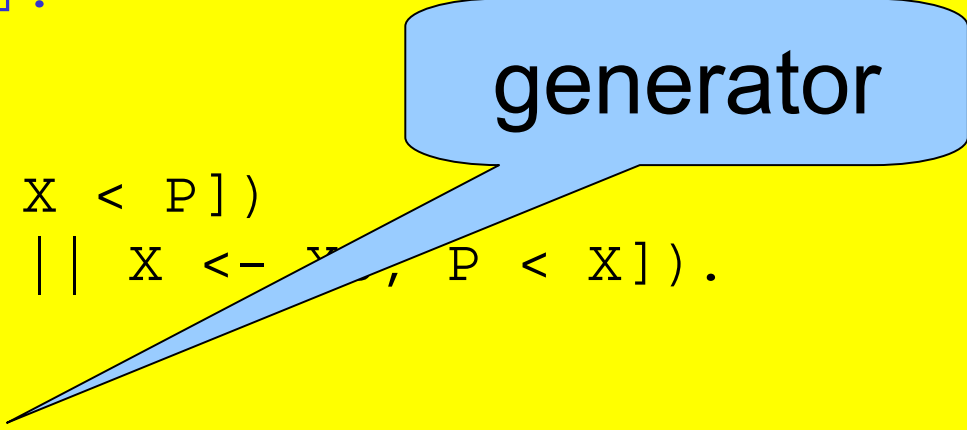
```
-module(demo).
-export([sort/1]).

-include_lib("proper/include/proper.hrl").

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
  sort([X || X <- Xs, X < P])
  ++ [P] ++ sort([X || X <- Xs, P < X]).

prop_ordered() ->
  ?FORALL(L, list(integer()), ordered(sort(L))).

ordered([]) -> true;
ordered([_]) -> true;
ordered([A,B|T]) -> A =< B andalso ordered([B|T]).
```



generator

Testing the ordered property

```
$ erl -pa /path/to/proper/ebin
Erlang/OTP 20 [erts-9.2.1] [...] ...

Eshell V9.2.1 (abort with ^G)
1> c(demo).
{ok,demo}
2> proper:quickcheck(demo:prop_ordered()).
..... 100 dots .....
OK: Passed 100 tests
true
3> proper:quickcheck(demo:prop_ordered(), 4711).
..... 4711 dots .....
OK: Passed 4711 tests
true
```

Runs any number of “random” tests we feel like.

If all tests satisfy the property, the test passes.

Another property for the program

```
-module(demo).
-export([sort/1]).

-include_lib("proper/include/proper.hrl").

-spec sort([T]) -> [T].
sort([]) -> [];
sort([P|Xs]) ->
  sort([X || X <- Xs, X < P])
  ++ [P] ++ sort([X || X <- Xs, P < X]).

prop_ordered() ->
  ?FORALL(L, list(integer()), ordered(sort(L))).

prop_same_length() ->
  ?FORALL(L, list(integer()),
    length(L) == length(sort(L))).

ordered([]) -> ...
```

Testing the same_length property

```
4> c(demo).
{ok,demo}
5> proper:quickcheck(demo:prop_same_length()).
.....!
Failed: After 14 test(s).
[1,3,-3,10,-3]

Shrinking (6 time(s))
[0,0]
false
6> proper:quickcheck(demo:prop_same_length()).
.....!
Failed: After 13 test(s).
[2,-8,-3,1,1]

Shrinking (1 time(s))
[1,1]
false
```

```
sort([]) -> [];
sort([P|Xs]) ->
  sort([X || X <- Xs, X < P])
  ++ [P] ++
  sort([X || X <- Xs, P < X]).
```


Integration with simple types

```
%% Using a user-defined simple type as a generator
-type bf() :: binary() | 'apple' | 'banana' | 'orange'.

prop_same_length() ->
  ?FORALL(L, list(bf()),
    length(L) == length(sort(L))).
```

```
7> c(demo).
{ok,demo}
8> proper:quickcheck(demo:prop_same_length()).
.....!
Failed: After 17 test(s).
[banana,apple,<<134>>,banana,<<42,25,177>>]

Shrinking (2 time(s))
[banana,banana]
false
```

Integration with complex types

```
%% Using a user-defined recursive type as a generator
-type bf() :: binary() | 'apple' | 'banana' | 'orange'.
-type tree(T) :: 'leaf' | {'node', T, tree(T), tree(T)}.

prop_same_length() ->
  ?FORALL(L, list(tree(bf()))),
    length(L) == length(sort(L)).
```

```
9> c(demo).
{ok,demo}
10> proper:quickcheck(demo:prop_same_length()).
.....!
Failed: After 15 test(s).
[{node,banana,{node,<<42>>,leaf,leaf},leaf},...]

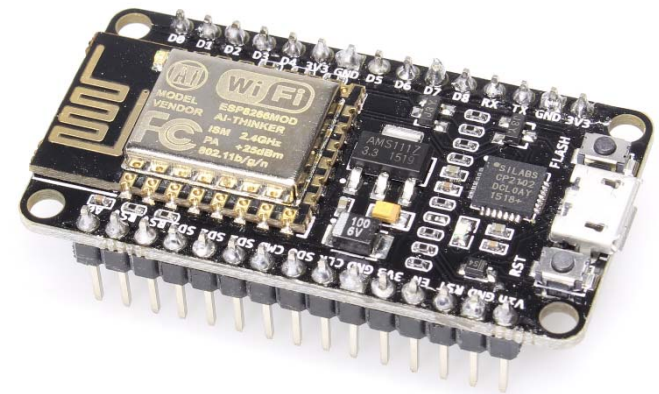
Shrinking (2 time(s))
[{node,banana,{node,banana,leaf,leaf},leaf},
 {node,banana,{node,banana,leaf,leaf},leaf}]
false
```

PBT testing of sensor networks

- Sensor network:
 - Random distribution of UDB server and client nodes
 - Client node periodically sends messages to server node

- Property to test:
 - Has X-MAC for any network a duty-cycle $> 25\%$?

(duty-cycle ::= % time the radio is on)



User-defined generators

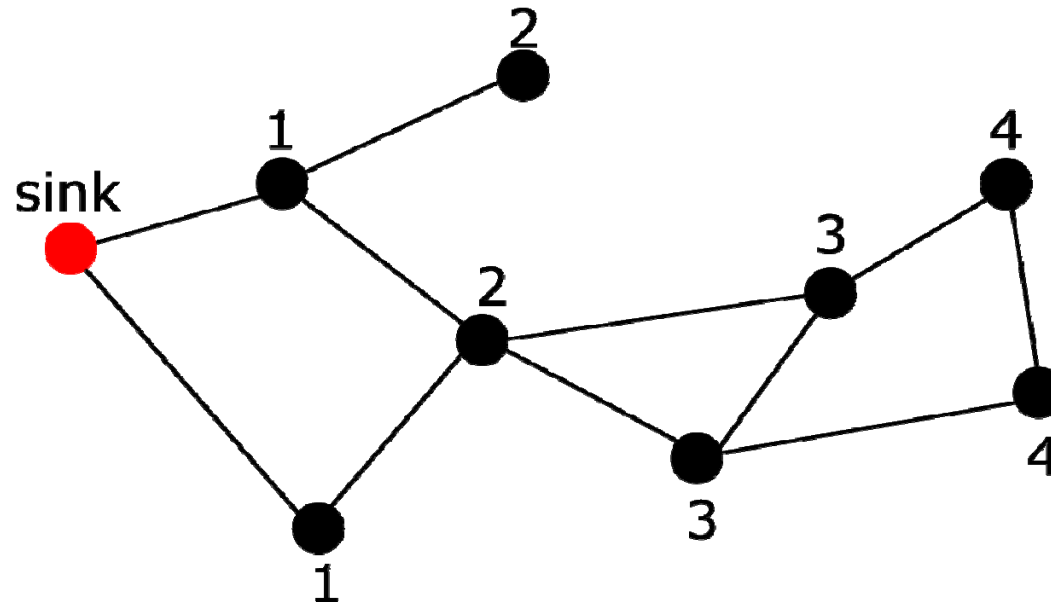
A generator for random graphs of N nodes:

```
graph(N) ->
  Vs = lists:seq(1, N),
  ?LET(Es, list(edge(Vs)), {Vs, lists:usort(Es)}).

edge(Vs) ->
  ?SUCHTHAT({v1,v2}, {oneof(Vs), oneof(Vs)}, v1 < v2).
```

Great: We can generate random sensor networks!

Node distances



On this graph, the maximum distance to sink is 4.

Is there a network with N nodes where the max distance to a sink node is greater than $N/2$?

Testing the max_distance property

```
prop_max_distance(N) ->
  ?FORALL(G, graph(N),
    begin
      D = lists:max(distance_to_sink(G)),
      D < (N div 2)
    end).
```

```
2> proper:quickcheck(demo:prop_max_distance(42)).
..... 100 dots .....
OK: Passed 100 tests
true
3> proper:quickcheck(demo:prop_max_distance(42), 100000).
..... 100000 dots .....
OK: Passed 100000 tests
true
```

Possible solutions

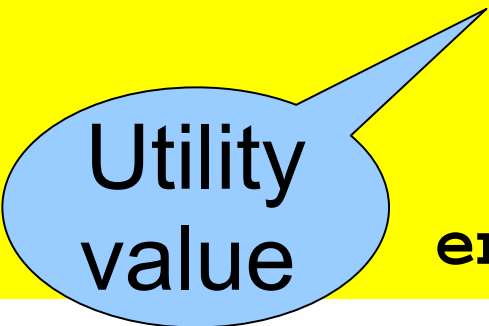
- Write more involved (custom) generators
- Guide the input generation
using a search strategy, and
introducing a feedback-loop in the testing

Targeted Property-Based Testing

- Combines search techniques with PBT.
- Automatically guides input generation towards inputs with high probability of failing.
- Gather information during test execution in the form of **utility values (UVs)**.
- UVs capture how close input came to falsifying a property.

Targeted max_distance property

```
prop_max_distance(N) ->  
  ?FORALL_SA(G, ?TARGET({gen => graph(N)}),  
    begin  
      D = lists:max(distance_to_sink(G)),  
      ?MAXIMIZE(D),  
      D < (N div 2)  
    end).
```



Utility
value

Now the `prop_max_distance(42)` property fails consistently with only a few thousand tests!

Testing the X-MAC protocol

Random PBT

Average amount of tests: **1188**

Average time per tests: 23.5s

Mean Time to Failure: 7h46m

Targeted PBT

Average amount of tests: **200**

Average time per tests: 40.6s

Mean Time to Failure : 2h12m

Testing security properties

$$\frac{i(pc) = \text{Noop}}{\boxed{pc} \mid \boxed{s} \mid \boxed{m} \Rightarrow \boxed{pc+1} \mid \boxed{s} \mid \boxed{m}} \quad (\text{NOOP})$$
$$\frac{i(pc) = \text{Push } v}{\boxed{pc} \mid \boxed{s} \mid \boxed{m} \Rightarrow \boxed{pc+1} \mid \boxed{v : s} \mid \boxed{m}} \quad (\text{PUSH})$$
$$\frac{i(pc) = \text{Pop}}{\boxed{pc} \mid \boxed{v : s} \mid \boxed{m} \Rightarrow \boxed{pc+1} \mid \boxed{s} \mid \boxed{m}} \quad (\text{POP})$$

Definitions for an abstract machine.

Test: Do these definitions fulfill a certain security criteria?

(Noninterference)

Cătălin Hrițcu et al. "Testing noninterference, quickly." *Journal of Functional Programming*, 26 (2016).

Testing security properties

Random PBT

Naive: generate random programs

ByExec: generate program step-by-step one instruction a time;
new instruction should not crash program

	Random PBT	
	Naive	ByExec
ADD	2234,08ms	312,97ms
LOAD	324028,34ms	987,91ms
STORE A	<i>timeout</i>	4668,04ms

Testing security properties

Targeted PBT

List: programs are a list of instructions; using the built-in list generator for Simulated Annealing

ByExec: neighboring program: a program with one more instruction

	Random PBT		Targeted PBT	
	Naive	ByExec	List	ByExec
ADD	2234,08	312,97	319,86	68,49
LOAD	324028,34	987,91	287,23	135,52
STORE A	–	4668,04	1388,09	263,94

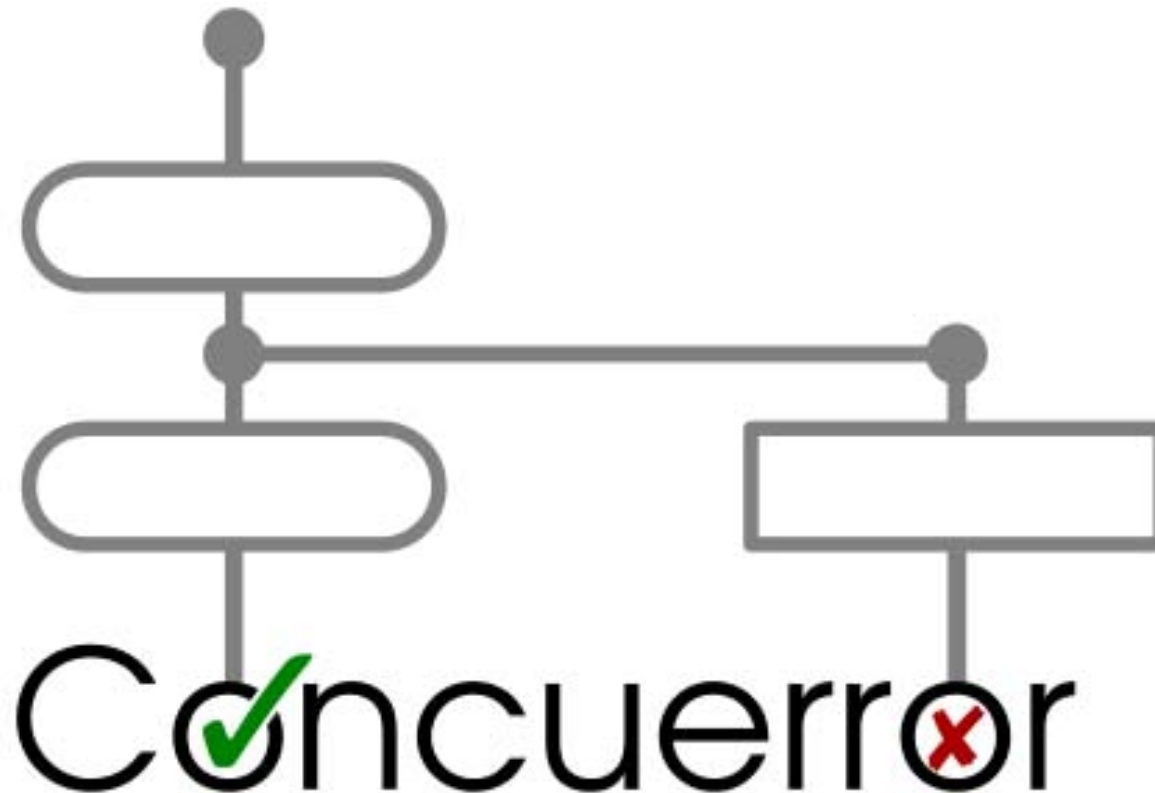
Testing security properties

hand written; ca. 30 lines of additional code

	PBT		Targeted PBT	
	Naive	ByExec	List	ByExec
ADD	2234,08	312,97	319,86	68,49
LOAD	324028,34	987,91	287,23	135,52
STORE A	—	4668,04	1388,09	263,94

1 line of code!

Concuerror – concuerror.com



Stateless Model Checking (SMC)

aka **Systematic Concurrency Testing**

A technique to **detect** concurrency errors or **verify** their absence by exploring all possible ways that concurrent execution can influence a program's outcome.

fully automatic

low memory requirements

applicable to programs with finite executions

How SMC works

Assume that you only have one 'scheduler'.

Run an arbitrary execution of the program...

Then:

Backtrack to a point where some other thread could have been chosen to run...

From there, continue with another execution...

Repeat until all choices have been explored.

Systematic exploration example

Initially: $x = y = 0$

Thread 1

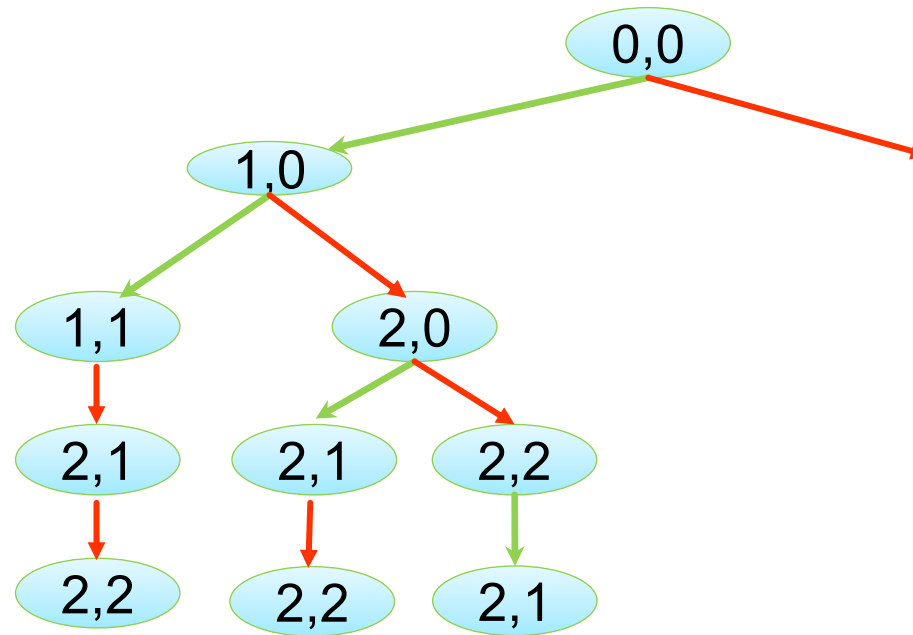
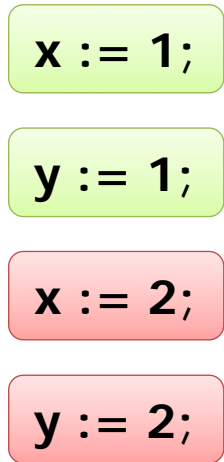
$x := 1;$
 $y := 1;$

Thread 2

$x := 2;$
 $y := 2;$

Correctness Property (at the end)

`assert(x == y);`



Exploration can stop early when a property is violated.

Systematic exploration example

Initially: $x = y = 0$

Thread 1

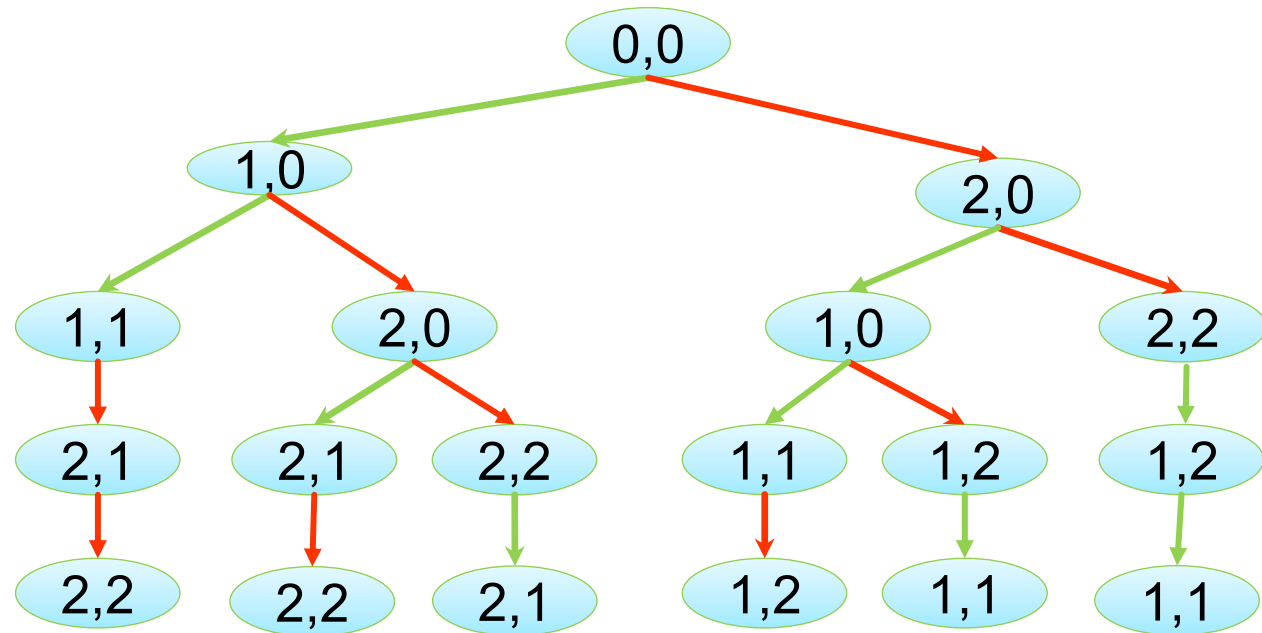
$x := 1;$
 $y := 1;$

Thread 2

$x := 2;$
 $y := 2;$

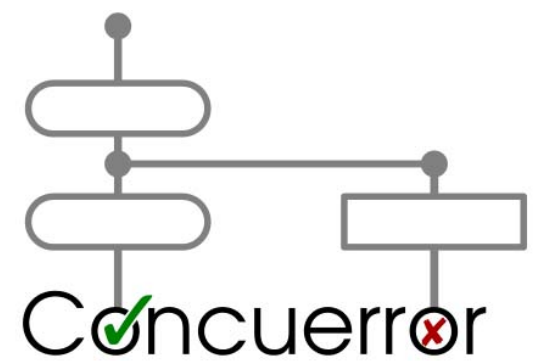
Correctness Property (at the end)

$\text{assert}((x + y) < 7);$



Exploration needs to visit the complete set of traces for properties that hold.

Concuerror



A **stateless model checker** for **Erlang** that systematically explores **all** possible behaviours of a program annotated with some assertions, to

either detect concurrency errors

(in which case it reports the erroneous trace)

or verify their absence

(i.e., that the properties in the assertions hold)

Systematic ≠ Stupid

Literally explore “all traces”?? Too many!

Not all pairs of events are conflicting.

Each explored trace should be **different**.

Partial Order Reduction (POR)

Combinatorial explosion in the number of interleavings.

Initially: $x = y = \dots = z = 0$

Thread 1:
 $x := 1$

Thread 2:
 $y := 1$

Thread N:
 $z := 1$

- Interleavings under naïve exploration: **N!**
- Interleavings needed to cover all behaviors: **1**

Partial Order Reduction (POR)

- ✓ Explore just a subset of all interleavings
- ✓ Still cover all behaviors

Optimal DPOR [POPL'14, JACM'17]

The exploration algorithm

- ... monitors **conflicts** between events
- ... explores additional interleavings **as needed**
- ... completely avoids **equivalent** interleavings

Dynamic: at runtime, using concrete data

Optimal:

explores only one interleaving per equivalence class

does not even initiate redundant ones

Optimal DPOR exploration

Initially: $x = y = 0$

Thread 1

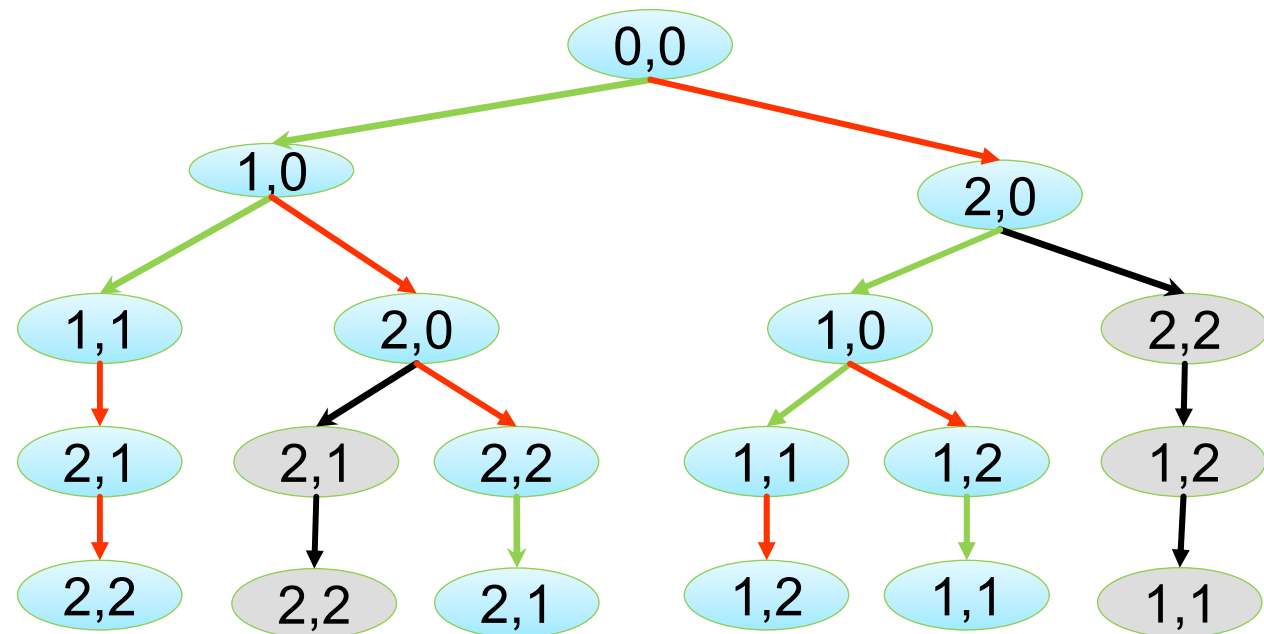
$x := 1;$
 $y := 1;$

Thread 2

$x := 2;$
 $y := 2;$

Correctness Property (at the end)

`assert((x + y) < 7);`



Optimal DPOR will not explore the grey nodes.

Bounding

Explore only a few traces based on some bounding criterion.

E.g., number of times threads can be preempted, delayed, etc.

Very effective for testing!

Not suitable for verification.

Preemption bounded exploration

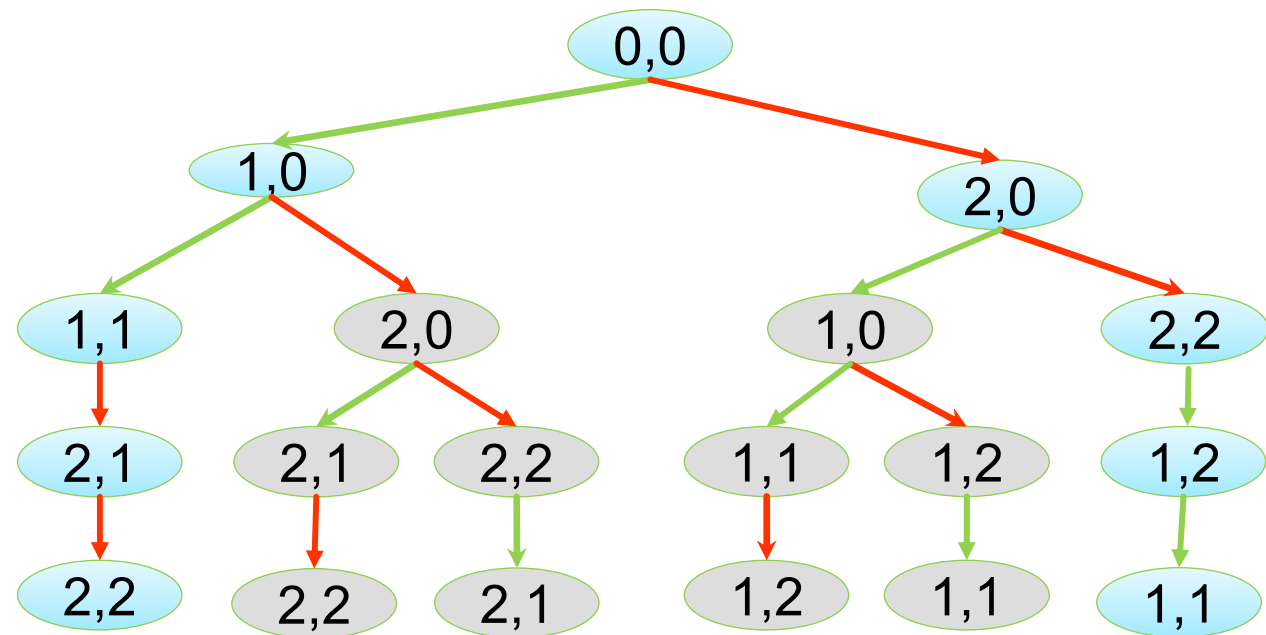
Initially: $x = y = 0$

Thread 1

$x := 1;$
 $y := 1;$

Thread 2

$x := 2;$
 $y := 2;$



With a **preemption bound of 0**,
the grey nodes will not be explored.

Preemption bounded exploration

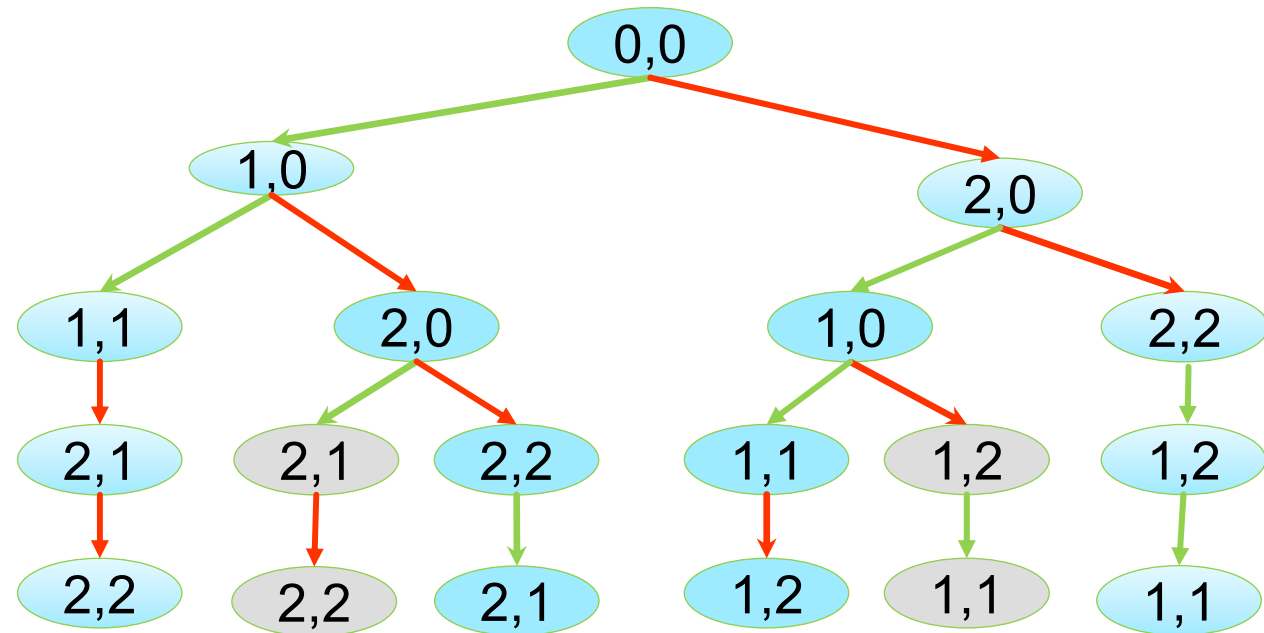
Initially: $x = y = 0$

Thread 1

$x := 1;$
 $y := 1;$

Thread 2

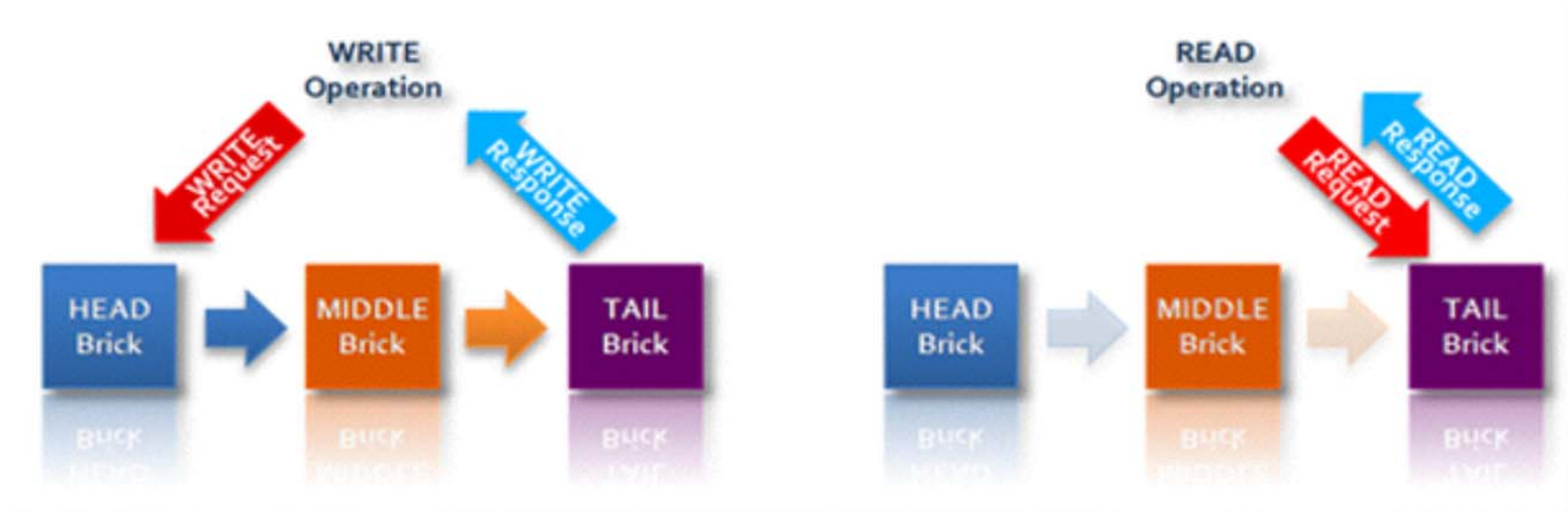
$x := 2;$
 $y := 2;$



With a **preemption bound of 1**,
the grey nodes will not be explored.

Chain replication [OSDI'04]

A variant of master/slave replication.
Strict chain order:



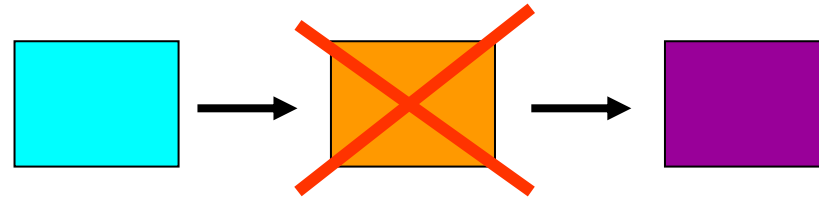
Sequential read @ tail.

Linearizable read @ all.

Dirty read @ head or middle.

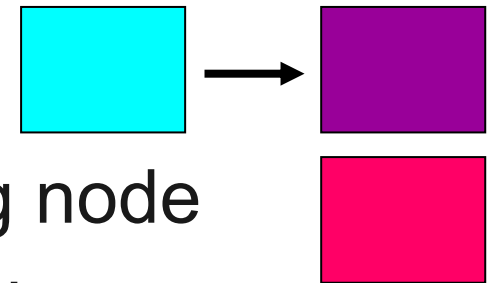
Chain repair


Suppose chain of three servers:



Naive offline repair method:

1. Stop all surviving servers in the chain
2. Copy tail's update history to the repairing node
3. Restart all nodes with the new configuration



A better repair method for CR systems places the repairing node directly on the chain and reads go to  (the old tail).

CORFU [SIGOPS'12, NSDI'17]

Uses Chain Replication with three changes:

1. Responsibility for replication is moved to the client.
2. CORFU's servers implement write-once semantics.
3. Identifies each chain configuration with an epoch #.
 - All clients and servers are aware of the epoch #.
 - The server rejects clients with a different epoch #.
 - A server temporarily stops service if it receives a newer epoch # from a client.

Engineers at VMWare (1)

Investigated methods for chain repair in CORFU

Method #1: ~~Add to the tail~~

Engineers at VMWare (2)

Investigated methods for chain repair in CORFU

Method #2: ~~Add to the head~~



Scott L. Fritchie

@slfritchie

Following

I was all ready to have a celebratory "New algorithm works!" tweet. Then the DPOR model execution w/Concuerror found an invalid case. Ouch.

RETWEET

1

LIKES

5



9:16 AM - 23 Jun 2016

Modeling CORFU in Erlang

Initial model:

- Some (**one** or **two**) servers undergo a chain repair to add **one** more server to their chain.
- Concurrently, **two** other clients try to write **two** different values to the same key.
- While a third client tries to read the key **twice**.

Modeling CORFU in Erlang (cont)

- Servers and clients are modeled as Erlang processes.
- All requests are modeled as messages.

Processes used by the model:

- Central coordinator
- CORFU log servers (2 or 3)
- Layout server process
- CORFU reading client
- CORFU writing clients (2)
- Layout change and data repair process

Correctness properties

Immutability:

Once a value has been written in a key, no other value can be written to it.

Linearizability:

If a read sees a value for a key, subsequent reads for that key must also see the same value.

Three repair methods

1. Add repair node at the tail of the chain.
2. Add repair node at the head of the chain.
3. Add repair node in the middle.
 - Configuration with two healthy servers.
 - Configuration with one healthy server which is “logically split” into two.

Results in (old) Concuerror

Method	Bounded Exploration			Unbounded Exploration		
	Bug?	Traces	Time	Bug?	Traces	Time
1 (Tail)	Yes	638	57s	Yes	3 542 431	144h
2 (Head)	Yes	65	7s	Yes	389	26s
3 (Middle)	No	1257	68s	No	>30 000 000	>750h

Model refinements

Conditional read

Avoid issuing read operations that are sure to not result in violations.

Convert layout server process to an ETS table (instead of a process).

Effect of model refinements

Method #3 (add repair node in the middle)

Concuerror verifies the method

- in 48 hours
- after exploring 3 931 412 traces.

Method #1 (add repair node in the tail)

Even *without* bounding, the error is found in just
19 seconds (212 traces).