TACAS 2001 5 April 2001

# The Erlang Verification Tool

Dilian Gurov
Lars-åke Fredlund
S.I.C.S.

Thomas Noll
RWTH Aachen

## 1. Motivation

- Erlang developed and widely used at Ericsson for programming telecommunication applications.

- Software of a highly *concurrent* and *dynamic* nature: hard to debug and test. Alternative: *code verification*.

- Feasible: core Erlang economic and clean: compact and elegant formalization.

- Challenges: dynamic process creation, unbounded data.

- Needed: rich verification framework, tool support.

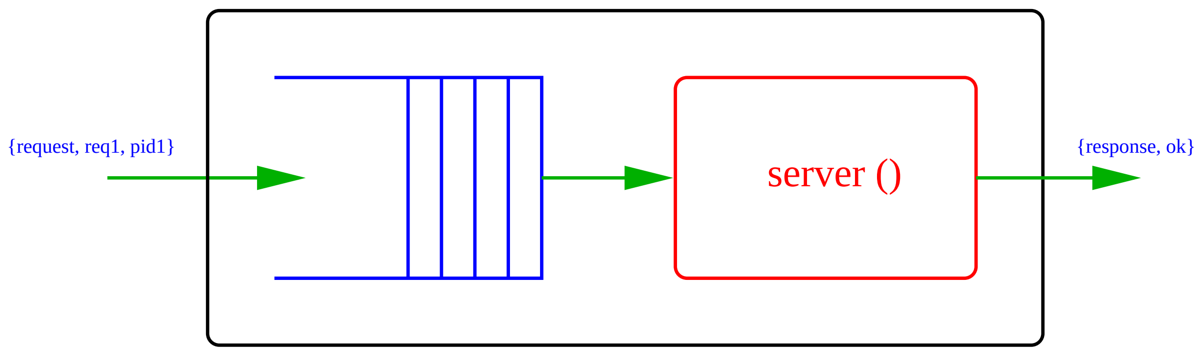
## Example

A server process:

```
server () ->
  receive
    {request, Request, ClientPid} ->
      ClientPid ! {response, handle (Request)}
  end,
  server ().

handle (Request) ->
  ok.
```

Evaluation context: $\mathtt{proc}\langle \mathtt{server()}, Pid, Queue \rangle$



System property:

STAB - *stabilizes on internal and output actions.*

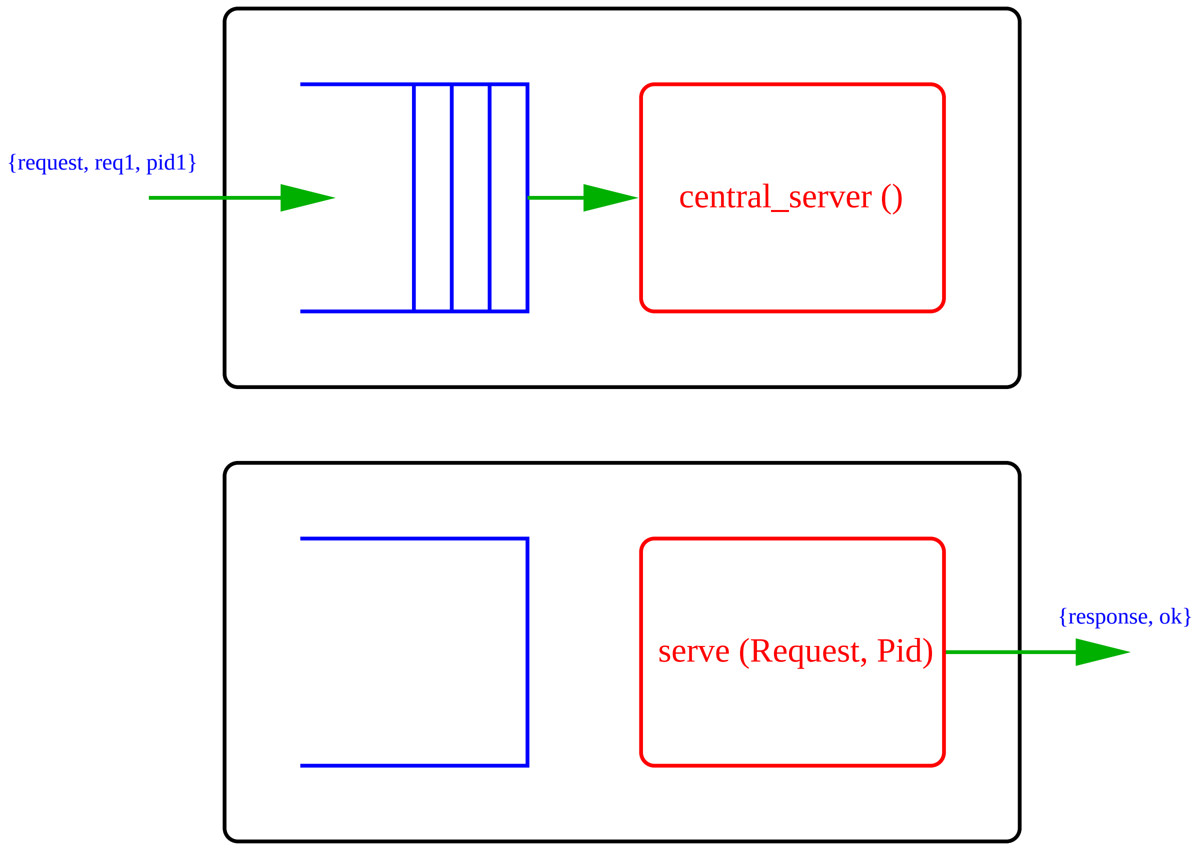
## A concurrent implementation:

```
central_server () ->
  receive
    {request, Request, ClientPid} ->
      spawn (serve, [Request, ClientPid])
  end,
  central_server ().

serve (Request, ClientPid) ->
  ClientPid ! {response, handle (Request)}.

handle (Request) ->
  ok.
```

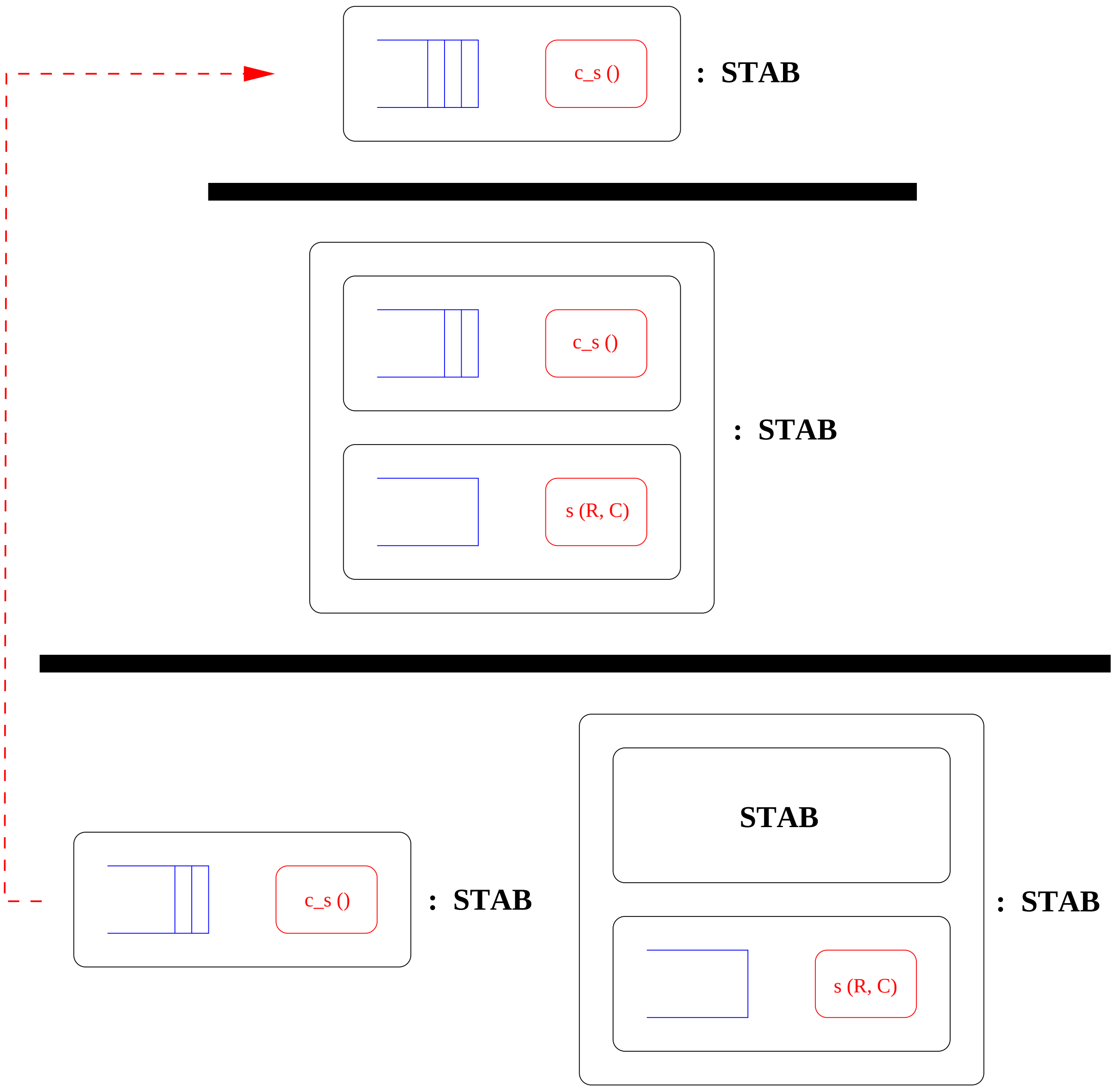
## Concurrent server after spawning:



Should possess STAB.

How can one prove this?

## Proof schema:



## 2. Verification Framework

### Erlang Semantics

Hierarchy of *components*:
- values: numbers, lists, message queues
- expressions: `receive ... end`
- processes: $\texttt{proc}\langle\texttt{server()},\texttt{Pid},\texttt{Queue}\rangle$
- systems: $P_1||\ldots||P_n$

Operational small-step semantics: *labelled transitions* between component states; labels capture *side effects*.

Compositional: *transition rules* relate the behaviours at different layers.

$$\frac{e \xrightarrow{pid!V} e'}{\langle e, pid', Q\rangle \xrightarrow{pid!V} \langle e', pid', Q\rangle}\ pid \neq pid'$$

## Property Specification Language

Captures the *structural* and the *interaction* properties of components, i.e. the possible *sequences of choices of side-effects*.

Consists of:
- State predicates: *is_value*, *local_pid*
- First-order predicate logic: $\neg$, $\vee$, $\exists x.$, $=$
- Modalities: $\langle\alpha\rangle\,\phi$, $[\alpha]\,\phi$
- Fixed-point operators: $\mu Z.\phi$, $\nu Z.\phi$

Example formulas:

$$\text{STAB} \overset{\text{def}}{=} \mu Z.\,[\tau, !]\,Z$$
$$\text{STAB} \Leftarrow [\tau, !]\ \text{STAB}$$

$$\begin{aligned} x : \text{Nat} \Leftarrow& \\ &\quad x = 0 \\ &\vee \exists y{:}\text{Nat}.\, x = S(y) \end{aligned}$$

## Proof System

*Gentzen*-style sequent calculus:

- assertions: $s : \phi$, $s \xrightarrow{\alpha} s'$, $\kappa_1 < \kappa_2$

- sequents: $\phi_1, \phi_2, \phi_3 \vdash \psi_1, \psi_2$

- structural and logical rules,

- dynamical rules:

$$\text{(Box-R)}\ \frac{\Gamma, s \xrightarrow{\alpha} X \vdash X : \phi, \Delta}{\Gamma \vdash s : [\alpha]\,\phi, \Delta}$$

- operational semantics rules:

$$\text{(Prefix-L)}\ \frac{\Gamma[s/X] \vdash \Delta[s/X]}{\Gamma, \alpha.s \xrightarrow{\alpha} X \vdash \Delta}$$

- term-cut rule:

$$\text{(TermCut)}\ \frac{\begin{array}{c}\Gamma \vdash Q : \psi, \Delta \\ \Gamma, X : \psi \vdash P : \phi, \Delta\end{array}}{\Gamma \vdash P[Q/X] : \phi, \Delta}$$

- fixed-point induction:

$$\text{(Approx-R)}\ \frac{\Gamma \vdash s : (\nu Z.\phi)^{\kappa}, \Delta}{\Gamma \vdash s : \nu Z.\phi, \Delta}$$

$$\text{(Unfold-R)}\ \frac{\Gamma, \kappa' < \kappa \vdash s : \phi[(\nu Z.\phi)^{\kappa'}/Z], \Delta}{\Gamma \vdash s : (\nu Z.\phi)^{\kappa}, \Delta}$$

- global *discharge* rule:
  instance checking,
  progress,
  global consistency.

Proof system is:
- sound,
- complete for the fragment with process variables as the only process terms.

## 3. The Erlang Verification Tool

Verification process: goal-directed proof-tree construction.

Verification tool: proof assistant with facilities for semi-automatic proof search.

Proof reuse: sharing sub-proofs; using lemmas.

Lazy proof discovery: meta-variables, incremental discharge.

Automation: *tacticals* and *scripts*, lifting the level of reasoning. Libraries of application-specific tactics, scripts, lemmas.

Graphical user interface: context-sensitive access to *proof resourses* such as proof structure, tactics, scripts, lemmas.

## 4. A Proof Example

Stabilization property:

```
stabilizes: erlang_system -> prop <=
   (forall Pid:erlangPid .
    forall V:erlangValue .
      [Pid!message(V)] stabilizes)
/\ ([estep] stabilizes);
```

Initial proof goal:

```
declare P:erlangPid, Q:erlang_queue in
Q : queue |- proc<central_server (), P, Q> : stabilizes
```

where **queue** is the Erlang queue type:

```
queue: erlang_queue -> prop <=
  \Q:erlang_queue .
       Q = eps
    \/ (exists V:erlangValue .
        exists Q1:erlang_queue .
        exists Q2:erlang_queue .
          Q = Q1@[[V]]@Q2 /\ (is_queue Q1@Q2));
```

## Proof script:

```
loop (
  case_by [
    (sp_and
      (sp_sat_sysproc_r 1)
      (sp_not (sp_sat_is_queue_var_r 1)),    t_queue_flat_r 1),
    (sp_and
      (sp_sat_sysproc_r 1)
      (sp_unfoldable_r 1),                   t_gen_unfold_r 1)
  ]
);
```

## Stops at:

```
{1} Q2@[[{request,Req,ClPid}]]@Q3 : queue(K),
{2} Q1 = Q2@Q3,
{3} not (P = P1)
|-
{1} proc<begin P1, central_server () end, P, Q1> ||
    proc<serve (Req, ClPid), P1, eps>
    : stabilizes
```

Induction on system structure through term-cut, yielding an induction basis:

```
{1} Q2@[[{request,Req,ClPid}]]@Q3 : queue(K),
{2} Q1 = Q2@Q3
|-
{1} proc<begin P1, central_server () end, P, Q1>
    : stabilizes
```

and an induction step:

```
{1} X : stabilizes
|-
{1} X || proc<serve (Req, ClPid), P1, eps> : stabilizes
```

We apply the same script to the induction basis and obtain:

```
{1} Q2@[[{request,Req,ClPid}]]@Q3 : queue(K),
{2} Q1 = Q2@Q3
|-
{1} proc<central_server (), P, Q1> : stabilizes,
```

which is “almost” dischargable. Need queue invariant maintenance and progress. We unfold `queue(K)` via `t_gen_unfold_l`:

```
{1} Q2@[[{request,Req,ClPid}]]@Q3 = Q4@[[V]]@Q5,
{2} Q4@Q5 : queue(K'),
{3} K' < K,
{4} Q1 = Q2@Q3
|-
{1} proc<central_server (), P, Q1> : stabilizes,
```

followed by `t_queue_invar`, and obtain a dischargable goal:

```
{1} Q2@[[{request,Req,ClPid}]]@Q3 = Q4@[[V]]@Q5,
{2} Q1 : queue(K'),
{3} K' < K,
{4} Q1 = Q2@Q3
|-
{1} proc<central_server (), P, Q1> : stabilizes,
```

plus two queue properties.

The induction step is handled similarly.

## Case Studies

Purpose: Evaluate the methodology, get feedback for improving the tool.

Most significant case studies so far:
- *Mnesia*, a distributed database lookup manager.
- *Billing Agent*, a server spawning agents to handle incoming requests.
- *Set-as-process*, a prototypical distributed resource manager.
- *Leader-election* protocol.
- *Quicksort*, a side-effect-free function.

Conclusion: Verification is feasible. General reasoning principles can be extracted, semi-automated and re-applied.