Abstraction and Model Checking of Core Erlang Programs in Maude

– Martin Neuhäußer and Thomas Noll –

Software Modeling and Verification Group (MOVES)
RWTH Aachen University

6th Workshop on Rewriting Logic and its Applications, Vienna, April 2006

What is Core Erlang?

What is Core Erlang?

A strict functional language
with succinct syntax
based upon leightweight processes
and interprocess communication.

Creation of a new process

The evaluation of the built–in function

```
call 'erlang':'spawn'(Module, Function_name, Arguments)
```

creates a new process.

Creation of a new process

The evaluation of the built–in function

```
call 'erlang':'spawn'(Module, Function_name, Arguments)
```

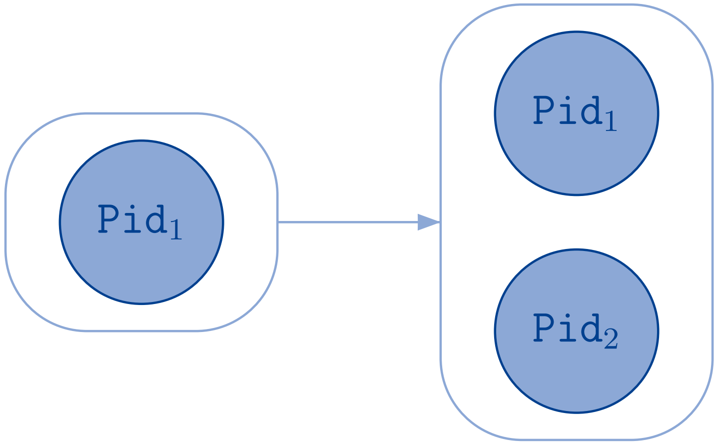
creates a new process.

- spawn returns as soon as the new process is created.
- Evaluates to the **unique identifier** of the created process.
- The new process autonomously starts to evaluate the function call

```
call Module : Function_name (Arguments).
```

- If the evaluation ends, its result is discarded.

  ⇝ Interprocess communication and side effects are a necessity!



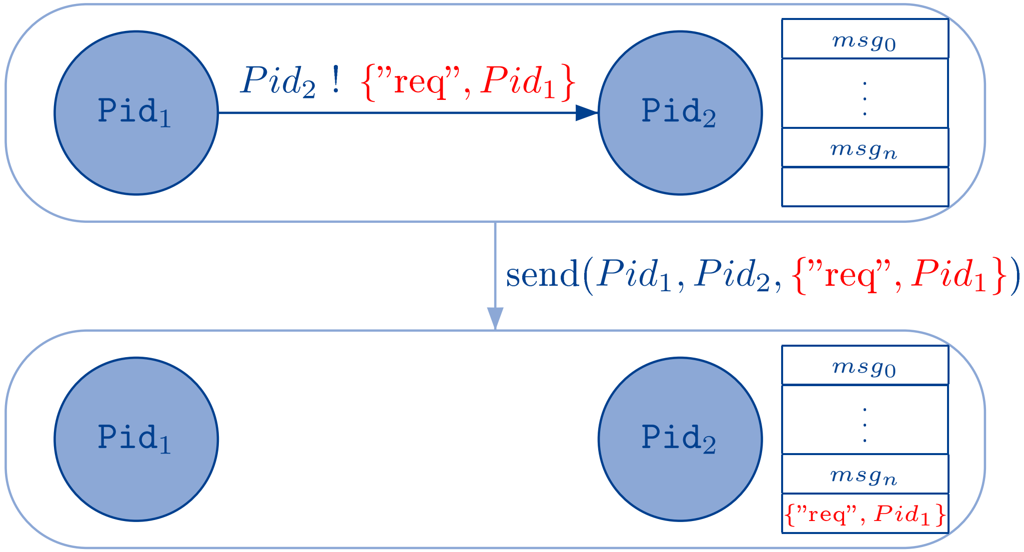
Sending and reception of messages

- **Sending of messages:**

  The evaluation of an expression

```
call 'erlang':'!' (Rcv, Expr)
```

  - first evaluates its arguments `Rcv` and `Expr`
  - and appends the message to the receiver's mailbox.



Sending and reception of messages

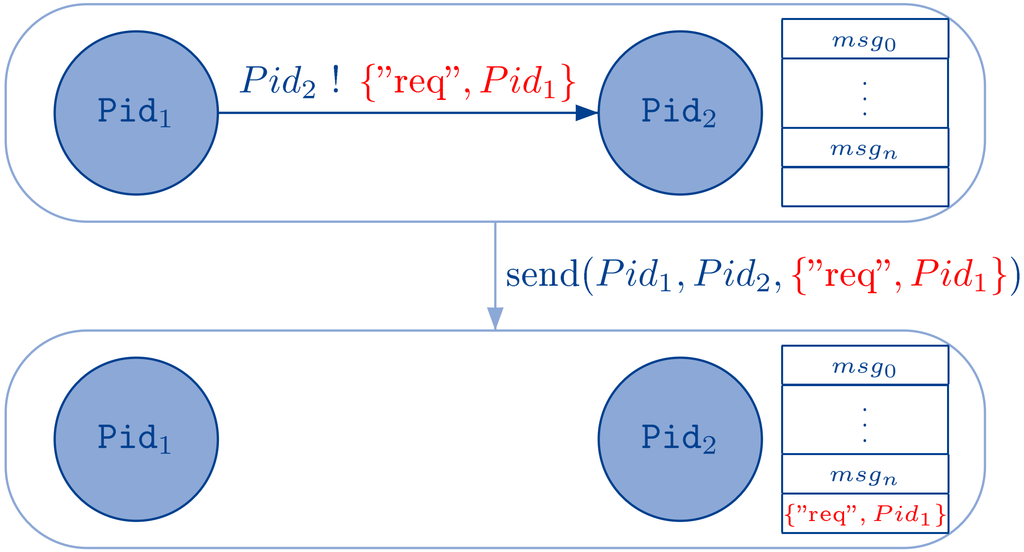
- **Sending of messages:**

  The evaluation of an expression

  ```
  call 'erlang':'!' (Rcv, Expr)
  ```

  – first evaluates its arguments `Rcv` and `Expr`

  – and appends the message to the receiver's mailbox.



- **Reception of messages:**

  receive

  Pat_1 **when** g_1 -> $Expr_1$

  Pat_2 **when** g_2 -> $Expr_2$

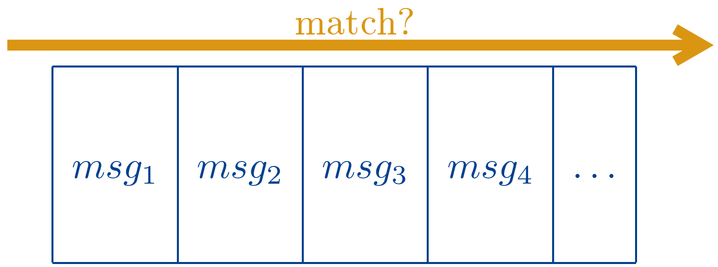
  ⋮ ⋮ ⋮

  Pat_n **when** g_n -> $Expr_n$

  after *Timeout* -> *TimeoutExpr*

  – The oldest matching message is received first.

  – Clauses are tried in order of appearence.



What is it all about?

Goal: Verifying properties of Core Erlang programs by means of transition system models

Approach:

- Formally define the semantics of Core Erlang.
- Operationalize the semantics by transferring it into a Rewriting Logic specification.
- Use abstractions to reduce the state space of the resulting transition systems.
- Automatically derive the transition system model of a given Core Erlang program (MAUDE).

Verification:

If the set of reachable states is finite, apply model checking techniques to verify properties.

A first sublanguage: Sequential Core Erlang

- Regard only the local aspects of expression evaluation.
- Side effects are formalized by non-determinism.

  $\hookrightarrow$ Non-determinism is resolved later by considering the entire system

Transition system T_e only captures the local behaviour of an expression!

A first sublanguage: Sequential Core Erlang

- Regard only the local aspects of expression evaluation.
- Side effects are formalized by non-determinism.

$\hookrightarrow$ Non-determinism is resolved later by considering the entire system

Transition system T_e only captures the local behaviour of an expression!

A first example:

- Sequencing operator `do`:

Example: `do 17 apply 'simex'/0()`

$\hookrightarrow$ The first subexpression is fully evaluated. Semantics: Discard its value and continue!

$$\frac{}{\underline{\texttt{do}}\ val\ e \xrightarrow{\tau}_e e} \quad (\text{Seq}_1)$$

A first sublanguage: Sequential Core Erlang

- Regard only the local aspects of expression evaluation.
- Side effects are formalized by non-determinism.

↪ Non-determinism is resolved later by considering the entire system

Transition system T_e only captures the local behaviour of an expression!

A first example:

- Sequencing operator `do`:

Example: `do 17 apply 'simex'/0()`

↪ The first subexpression is fully evaluated. Semantics: Discard its value and continue!

$$\frac{}{\texttt{do}\ val\ e \xrightarrow{\tau}_e e} \quad (\text{Seq}_1)$$

- **But** what about the evaluation of the first subexpression?

Consider for example: `do call 'erlang':'!'(Rcv,Msg) apply 'proceed'/0()`

↪ Before evaluation of the `do`-operator can proceed, its first argument must be evaluated:

$$\frac{e_1 \xrightarrow{\alpha}_e e_1'}{\texttt{do}\ e_1\ e_2 \xrightarrow{\alpha}_e \texttt{do}\ e_1'\ e_2} \quad (\text{Seq}_2)$$

Pattern matching expressions

- `case` expressions:

$$\frac{\exists i.\ \big(\mathsf{match}(val, cl_i) = e' \wedge \forall j < i.\ \mathsf{match}(val, cl_j) = \bot\big)}{\underline{\texttt{case}}\ val\ \underline{\texttt{of}}\ cl_1\ \cdots\ cl_k\ \underline{\texttt{end}}\ \xrightarrow{\tau}_e\ e'} \quad (\mathrm{Case}_1)$$

Pattern matching expressions

- $\underline{\texttt{case}}$ expressions:

$$\frac{\exists i.\ \big(\mathsf{match}(val, cl_i) = e' \wedge \forall j < i.\ \mathsf{match}(val, cl_j) = \bot\big)}{\underline{\texttt{case}}\ val\ \underline{\texttt{of}}\ cl_1\ \cdots\ cl_k\ \underline{\texttt{end}}\ \xrightarrow{\tau}_e\ e'} \quad (\mathrm{Case}_1)$$

- $\underline{\texttt{receive}}$ expressions:

  qmatch predicate holds iff a matching message is in the mailbox:

$$\mathsf{qmatch}(q, cl_1, \ldots, cl_k) := \exists q_1, q_2 \in \mathsf{Const}^*, c \in \mathsf{Const}, i \in \{1, \ldots, k\}.\quad q = q_1 \cdot c \cdot q_2 \wedge \mathsf{match}(c, cl_i) \neq \bot$$

Reception of the first matching message (c):

$$\frac{\neg\mathsf{qmatch}(q, cl_1, \ldots, cl_k) \qquad \underline{\texttt{case}}\ c\ \underline{\texttt{of}}\ cl_1 \ldots cl_k\ \underline{\texttt{end}} \xrightarrow{\tau}_e e' \qquad c_t \in \mathsf{Num} \cup \{\texttt{'infinity'}\}}{\underline{\texttt{receive}}\ cl_1 \cdots cl_k\ \underline{\texttt{after}}\ c_t\ \texttt{->}\ e_t \xrightarrow{\mathsf{recv}(q,c)}_e e'} \quad (\mathrm{Rcv}_1)$$

Note: The prefix qc of the process' mailbox is guessed nondeterministically!

$\hookrightarrow$ Reflected by the transition label $\mathsf{recv}(q, c)$

Global states and the transition system T_s:

- τ transitions are autonomous evaluation steps.

  $\hookrightarrow$ can be lifted to the system layer semantics directly:

$$\frac{e \xrightarrow{\tau}_e e'}{S \cup \{(e, i, q, L, t)\} \xrightarrow{\tau}_s S \cup \{(e', i, q, L, t)\}} \quad \text{(SeqCore)}$$

- Sending of messages:

  $\hookrightarrow$ By considering process systems, we can formalize message transmission:

$$\frac{e_i \xrightarrow{j!c}_e e_i'}{S \cup \{(e_i, i, q_i, L_i, t_i), (e_j, j, q_j, L_j, t_j)\} \xrightarrow{\mathsf{send}(i,j,c)}_s S \cup \{(e_i', i, q_i, L_i, t_i), (e_j, j, q_j \cdot c, L_j, t_j)\}} \quad (\text{Send}_1)$$

Global states and the transition system T_s:

- τ transitions are autonomous evaluation steps.
  ↪ can be lifted to the system layer semantics directly:

$$\frac{e \xrightarrow{\tau}_e e'}{S \cup \{(e,i,q,L,t)\} \xrightarrow{\tau}_s S \cup \{(e',i,q,L,t)\}} \quad \text{(SeqCore)}$$

- Sending of messages:
  ↪ By considering process systems, we can formalize message transmission:

$$\frac{e_i \xrightarrow{j!c}_e e_i'}{S \cup \{(e_i,i,q_i,L_i,t_i),(e_j,j,q_j,L_j,t_j)\} \xrightarrow{\mathsf{send}(i,j,c)}_s S \cup \{(e_i',i,q_i,L_i,t_i),(e_j,j,q_j \cdot c,L_j,t_j)\}} \quad (\text{Send}_1)$$

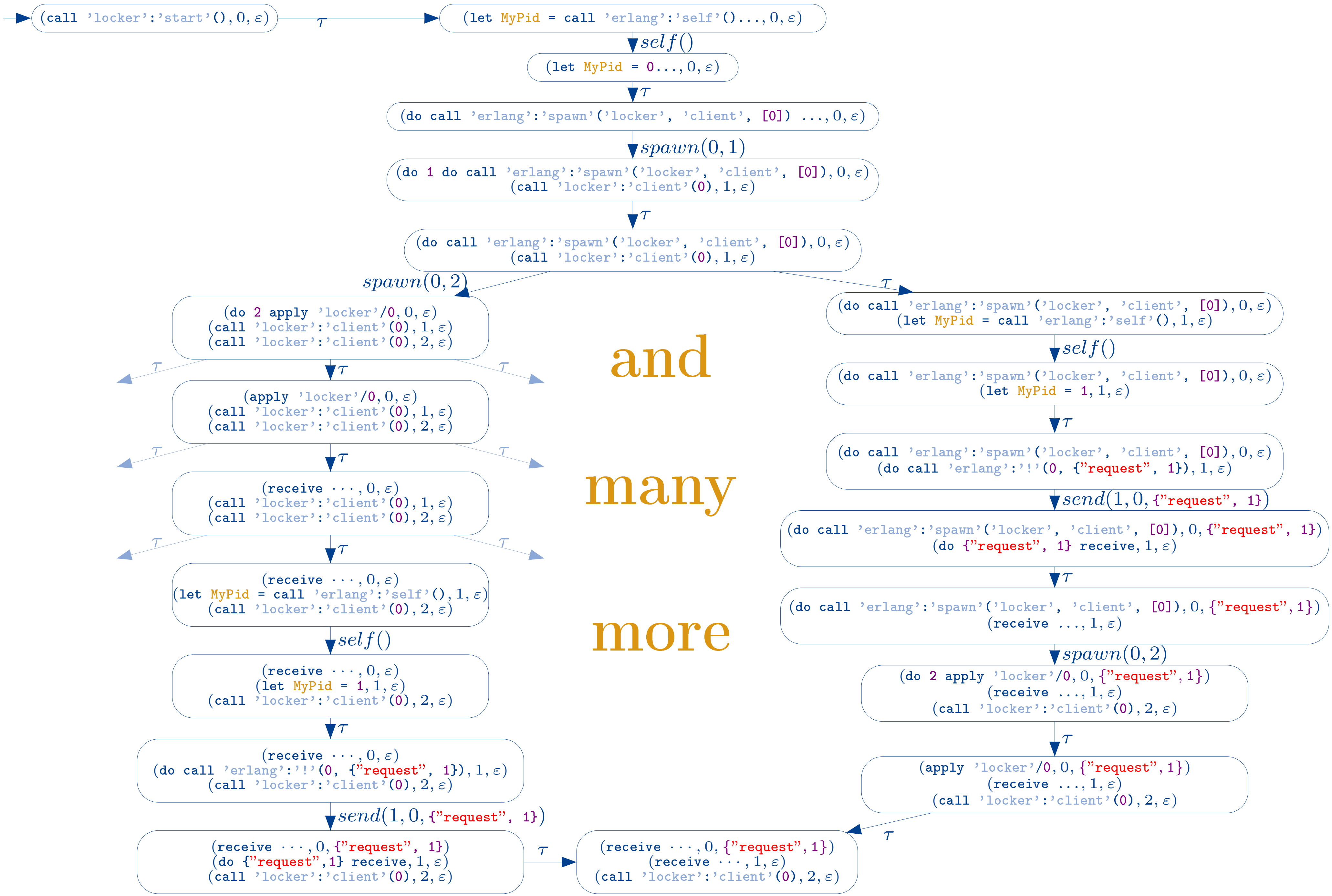
- Message reception:

$$\frac{e \xrightarrow{\mathsf{recv}(q_1,c)}_e e'}{S \cup \{(e,i,q_1 \cdot c \cdot q_2,L,t)\} \xrightarrow{\mathsf{recv}(i,c)}_s S \cup \{(e',i,q_1 \cdot q_2,L,t)\}} \quad \text{(Recv)}$$

Example: A simple mutual exclusion protocol in Core Erlang:

```
'locker'/0 = fun () ->
  receive
    {"request",Client} when 'true' -> do
      call 'erlang':'!'(Client, "ok")
      receive
        {"release",From} when
          call 'erlang':'=:='(From,Client)
          -> apply 'locker'/0()
      after 'infinity' -> 'false'
  after 'infinity' -> 'true'
```

```
'client'/1 = fun (LockerPid) ->
  let MyPid = call 'erlang':'self'() in do
    call 'erlang':'!'(LockerPid, {"request", MyPid})
    receive
      "ok" when 'true' -> do
        %% critical section
        call 'erlang':'!'(LockerPid, {"release", MyPid})
        apply 'client'/1(LockerPid)
    after 'infinity' -> 'false'
```



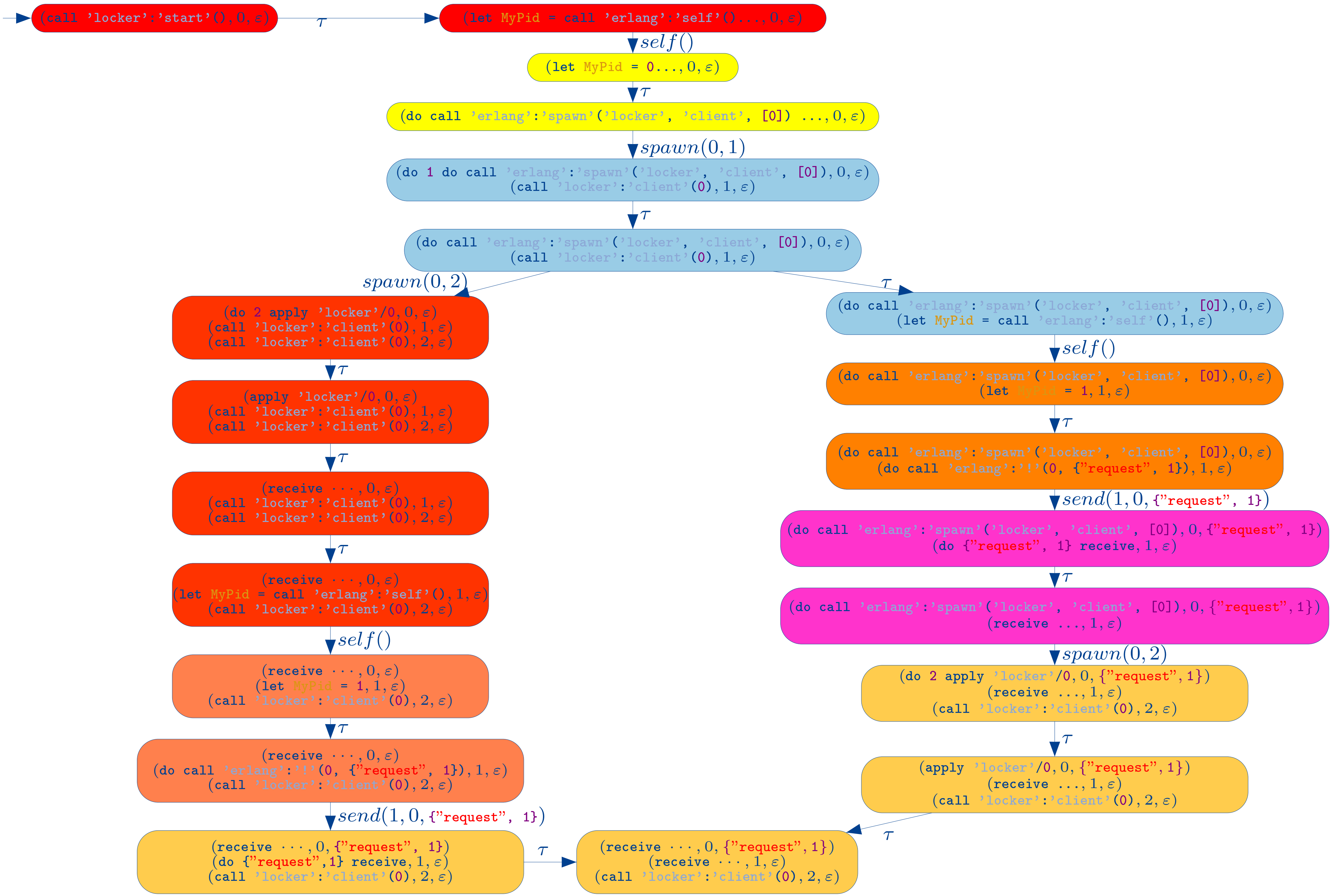
(call 'locker':'start'(), 0, ε)
τ
(let MyPid = call 'erlang':'self'()..., 0, ε)
self()
(let MyPid = 0..., 0, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]) ..., 0, ε)
spawn(0, 1)
(do 1 do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(call 'locker':'client'(0), 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(call 'locker':'client'(0), 1, ε)
spawn(0, 2)
(do 2 apply 'locker'/0, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(apply 'locker'/0, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(let MyPid = call 'erlang':'self'(), 1, ε)
(call 'locker':'client'(0), 2, ε)
self()
(receive ···, 0, ε)
(let MyPid = 1, 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(do call 'erlang':'!'(0, {"request", 1}), 1, ε)
(call 'locker':'client'(0), 2, ε)
send(1, 0, {"request", 1})
(receive ···, 0, {"request", 1})
(do {"request", 1} receive, 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, {"request", 1})
(receive ···, 1, ε)
(call 'locker':'client'(0), 2, ε)
and
many
more
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(let MyPid = call 'erlang':'self'(), 1, ε)
self()
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(let MyPid = 1, 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(do call 'erlang':'!'(0, {"request", 1}), 1, ε)
send(1, 0, {"request", 1})
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, {"request", 1})
(do {"request", 1} receive, 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, {"request", 1})
(receive ..., 1, ε)
spawn(0, 2)
(do 2 apply 'locker'/0, 0, {"request", 1})
(receive ..., 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(apply 'locker'/0, 0, {"request", 1})
(receive ..., 1, ε)
(call 'locker':'client'(0), 2, ε)
τ

Abstracting from τ evaluation steps

$TS_{/\sim} := (\mathcal{S}_{/\sim}, \underline{\text{Act}}, \rightarrow, [s_0]_\sim)$, where

- States are the equivalence classes in $\mathcal{S}_{/\sim}$
- Actions: $\underline{\text{Act}} := \underline{\text{Act}}_s \setminus \{\tau\}$
- Transition relation $\rightarrow \subseteq \mathcal{S}_{/\sim} \times \underline{\text{Act}} \times \mathcal{S}_{/\sim}$
- $[s_0]_\sim \in \mathcal{S}_{/\sim}$ as initial state

1. $\overset{\tau}{\longleftrightarrow}{}_s^*$ denotes the reflexive, symmetric and transitive closure of $\overset{\tau}{\longrightarrow}_s$.
2. Equivalence relation: $\sim := \overset{\tau}{\longleftrightarrow}{}_s^*$


(call 'locker':'start'(), 0, ε)
τ
(let MyPid = call 'erlang':'self'()..., 0, ε)
self()
(let MyPid = 0..., 0, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]) ..., 0, ε)
spawn(0, 1)
(do 1 do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(call 'locker':'client'(0), 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(call 'locker':'client'(0), 1, ε)
spawn(0, 2)
(do 2 apply 'locker'/0, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(apply 'locker'/0, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(call 'locker':'client'(0), 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(let MyPid = call 'erlang':'self'(), 1, ε)
(call 'locker':'client'(0), 2, ε)
self()
(receive ···, 0, ε)
(let MyPid = 1, 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, ε)
(do call 'erlang':'!'(0, {"request", 1}), 1, ε)
(call 'locker':'client'(0), 2, ε)
send(1, 0, {"request", 1})
(receive ···, 0, {"request", 1})
(do {"request",1} receive, 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(receive ···, 0, {"request", 1})
(receive ···, 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(let MyPid = call 'erlang':'self'(), 1, ε)
self()
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(let MyPid = 1, 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, ε)
(do call 'erlang':'!'(0, {"request", 1}), 1, ε)
send(1, 0, {"request", 1})
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, {"request", 1})
(do {"request", 1} receive, 1, ε)
τ
(do call 'erlang':'spawn'('locker', 'client', [0]), 0, {"request", 1})
(receive ..., 1, ε)
spawn(0, 2)
(do 2 apply 'locker'/0, 0, {"request", 1})
(receive ..., 1, ε)
(call 'locker':'client'(0), 2, ε)
τ
(apply 'locker'/0, 0, {"request", 1})
(receive ..., 1, ε)
(call 'locker':'client'(0), 2, ε)
τ

What is MAUDE?

Specification language based on José Meseguer's Rewriting Logic.

Interpreter for parameterized Rewriting Logic theories.

Developed at the University of Illinois at Urbana-Champaign.

MAUDE **preliminaries:**

1. **Membership equational logic theory** $(\Omega, \mathcal{E})$ where
   - $\Omega = ((\mathcal{K}, \Sigma), \varphi)$ denotes a many kinded signature and
   - $\mathcal{E}$ denotes the set of equations.
   - $\mathcal{E} = ER \uplus A$ where A are equational attributes (associativity, commutativity, identity) and ER are (directed) equations

     $\hookrightarrow$ equational rewriting/simplification

   $(\Omega, \mathcal{E})$ allows equational simplification of a term into a $\mathcal{E}$ normal form.

   Precondition: The directed equations in ER are confluent and terminating modulo A

MAUDE preliminaries:

1. **Membership equational logic theory** $(\Omega, \mathcal{E})$ where
   - $\Omega = ((\mathcal{K}, \Sigma), \varphi)$ denotes a many kinded signature and
   - $\mathcal{E}$ denotes the set of equations.
   - $\mathcal{E} = ER \uplus A$ where A are equational attributes (associativity, commutativity, identity) and ER are (directed) equations

     $\hookrightarrow$ equational rewriting/simplification

   $(\Omega, \mathcal{E})$ allows equational simplification of a term into a $\mathcal{E}$ normal form.

   Precondition: The directed equations in ER are confluent and terminating modulo A

2. **Rewriting logic theory** $(\Omega, \mathcal{E}, \phi, R)$ extends the MEL theory:
   - $((\mathcal{K}, \Sigma), \varphi)$ is the signature,
   - $(\Omega, \mathcal{E})$ is the underlying MEL theory and
   - $\phi : \Sigma \to 2^{\mathcal{N}}$ defines frozen argument positions.
   - R denotes the set of rewriting rules

     $\hookrightarrow$ needs not to be confluent!

Idea: Normalize term wrt. $ER \uplus A$ and then apply the rewriting rules from R!
$\hookrightarrow$ Coherence properties between $ER \uplus A$ and R must be fulfilled!

Representation of processes and process systems in Maude

- **Processes:**

```
op <□|□|□|□|□|□|□|□> : Label SysResult Expr Pid Mailbox ProcessLinks TrapExit ModEnv -> Process .
```

Label, SysResult and ModEnv are needed in order to operationalize the semantics

- **Process systems:**

```
op empty-processes : -> Processes [ctor] .
op □||□: Processes Processes -> Processes [ctor assoc comm id: empty-processes] .
```

subsort relation: Process $\sqsubset_{Spec}$ Processes

- **Process environments:**

```
op ((□,□,□,□)) : SysLabel Processes ModEnv PidSequence -> ProcessEnvironment .
```

↪ Process environments constitute the states of our transition system.

Specify the equivalence $\sim$ using the equational theory $(\Omega, \mathcal{E})$:

Example:

$$\frac{}{\underline{\texttt{do}}\ val\ e \xrightarrow{\tau}_e e}\ (\text{Seq}_1) \qquad \frac{e_1 \xrightarrow{\alpha}_e e_1'}{\underline{\texttt{do}}\ e_1\ e_2 \xrightarrow{\alpha}_e \underline{\texttt{do}}\ e_1'\ e_2}\ (\text{Seq}_2)$$

Specify the equivalence $\sim$ using the equational theory $(\Omega, \mathcal{E})$:

Example:

$$\frac{}{\underline{\texttt{do}}\ val\ e \xrightarrow{\tau}_e e}\ (\text{Seq}_1) \qquad \frac{e_1 \xrightarrow{\alpha}_e e_1'}{\underline{\texttt{do}}\ e_1\ e_2 \xrightarrow{\alpha}_e \underline{\texttt{do}}\ e_1'\ e_2}\ (\text{Seq}_2)$$

- Evaluation of the do operator itself:

```
eq [norm-do] :
    <tau|#no-res|do C EX2|PID|MBOX|LINKS|TRAP|ME> =
    <tau|#no-res|EX2|PID|MBOX|LINKS|TRAP|ME> .
```

- Evaluation of the first subexpression:

```
ceq [norm-do] :
    <tau|RES|do EX1 EX2|PID|MBOX|LINKS|TRAP|ME> =
    <#filterExit(ESL)|RES1|do EX1' EX2|PID|MBOX|LINKS|TRAP|ME>
        if not(EX1 :: Const)
        /\ <ESL|RES1|EX1'|PID|MBOX|LINKS|TRAP|ME> :=
           <tau|RES|EX1|PID|MBOX|LINKS|TRAP|ME> .
```

Rewriting rules define the transition relation $R_{\rightarrow}$:

Idea: Specify $\rightarrow \subseteq \mathcal{S}_{/\sim} \times \mathcal{S}_{/\sim}$ by rewriting rules R!

Note: Operationally, process systems are available as normal forms wrt. $(\Sigma, E \cup A)$ only!

Example: Inference rule specifying message reception:

$$\frac{e \xrightarrow{\mathsf{recv}(q_1,c)}_e e'}{S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\} \xrightarrow{\mathsf{recv}(i,c)}_s S \cup \{(e', i, q_1 \cdot q_2, L, t)\}} \quad \text{(Recv)}$$

Rewriting rules define the transition relation $R_{\rightarrow}$:

Idea: Specify $\rightarrow \subseteq \mathcal{S}_{/\sim} \times \mathcal{S}_{/\sim}$ by rewriting rules R!
Note: Operationally, process systems are available as normal forms wrt. $(\Sigma, E \cup A)$ only!

Example: Inference rule specifying message reception:

$$\frac{e \xrightarrow{\mathsf{recv}(q_1,c)}_e e'}{S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\} \xrightarrow{\mathsf{recv}(i,c)}_s S \cup \{(e', i, q_1 \cdot q_2, L, t)\}} \quad \text{(Recv)}$$

The corresponding conditional rewrite rule:

```
crl [sys-receive] :
    (SL, <receive(C)|#no-res|EX|PID|MBOX|LINKS|TRAP|ME> || PRCS, ME', PIDS) =>
    (sys-receive(PID, C),
         <tau|#no-res|EX|PID|MBOX1|LINKS|TRAP|ME> || PRCS, ME', PIDS)
  if MBOX1 := #extractMessage(MBOX|C) .
```

Remark: Receivable messages are observed on expression layer but removed on system layer!

Soundness and completeness

- Semantic point of view:

$$[s]_{\mathcal{E}} = [s]_{A \cup ER} \xrightarrow{R_{/A \cup ER}} [s']_{A \cup ER} = [s']_{\mathcal{E}}$$

- Operational point of view:

$$[s]_A \xrightarrow{ER_{/A}}{}^{*}\!| \xrightarrow{R_{/A}} [s']_A$$

Do they coincide?

Soundness and completeness

- Semantic point of view:

$$[s]_{\mathcal{E}} = [s]_{A \cup ER} \xrightarrow{R_{/A \cup ER}} [s']_{A \cup ER} = [s']_{\mathcal{E}}$$

- Operational point of view:

$$[s]_{A} \xrightarrow{ER_{/A}}{}^{*}\!\!| \xrightarrow{R_{/A}} [s']_{A}$$

Do they coincide?

Yes, they do!

Defining predicates

States of $TS_{/\sim}$ are represented by $(\Omega, \mathcal{E})$ normal forms.
$\hookrightarrow$ Associate predicates to these terms:

$s \models send(i, j, c)$	"process i just sent message c to process j"
$s \models receive(i, c)$	"process i just received c"

Remark: If $s \models send(i, j, c)$ is valid, the respective state was reached by this transition.

Defining predicates

States of $TS_{/\sim}$ are represented by $(\Omega, \mathcal{E})$ normal forms.
$\hookrightarrow$ Associate predicates to these terms:

$s \models send(i, j, c)$	"process i just sent message c to process j"
$s \models receive(i, c)$	"process i just received c"

Remark: If $s \models send(i, j, c)$ is valid, the respective state was reached by this transition.

Model checking the mutual exclusion protocol:

- As long as the first client is in its critical section, the second cannot enter

$$\varphi_1 = scheduler(0,1,2) \rightarrow \Box\, (send(0,1,\text{"}ok\text{"}) \rightarrow (\neg send(0,2,\text{"}ok\text{"})\ \mathcal{U}\ send(1,0,\{\text{"}rel\text{"},1\})))$$

- Eventually, the second client enters the critical section:

$$\varphi_2 = scheduler(0,1) \rightarrow \Diamond\, (send(0,2,\text{"}ok\text{"}))$$

$$\varphi_3 = scheduler(0,1,2) \rightarrow \Diamond\, (send(0,2,\text{"}ok\text{"}))$$

But: In general (unfair scheduling), φ_2 is not fulfilled:
$\leadsto$ Counterexample: The first client enters whereas the second client starves.

Future Work

1. Real Time Maude:

   Extend the Core Erlang semantics with a notion of time.

2. Case studies:

   More examples to see how this approach scales.

Thank you for your attention!

Any questions?

Tool available at `http://www.marneu.com/`