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 lightweight processes
and interprocess communication.
Creation of a new process

The evaluation of the built-in function

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

creates a new process.
Creation of a new process

The evaluation of the built-in function

\[
\text{call 'erlang': 'spawn'(Module, Function\_name, Arguments)}
\]

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

\[
\text{call Module : Function\_name (Arguments)}.
\]

- If the evaluation ends, its result is discarded.

> Interprocess communication and side effects are a necessity!

![Diagram of process creation]
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
  
  \[
  \text{call 'erlang':!' (Rcv, Expr)}
  \]
  
  - first evaluates its arguments \( \text{Rcv} \) and \( \text{Expr} \)
  - and appends the message to the receiver’s mailbox.

- **Reception of messages:**
  
  \[
  \text{receive}
  \]
  
  \[
  \begin{align*}
  \text{Pat}_1 & \text{ when } g_1 & \rightarrow & \text{Expr}_1 \\
  \text{Pat}_2 & \text{ when } g_2 & \rightarrow & \text{Expr}_2 \\
  & \vdots & \vdots & \vdots \\
  \text{Pat}_n & \text{ when } g_n & \rightarrow & \text{Expr}_n \\
  \text{after Timeout} & \rightarrow & \text{TimeoutExpr}
  \end{align*}
  \]
  
  - The oldest matching message is received first.
  - Clauses are tried in order of appearance.
Our Intention: Verifying Concurrent Erlang Programs

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.
  \(\rightarrow\) 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.
  \[ \rightarrow \text{Non-determinism is resolved later by considering the entire system} \]

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

A first example:

- Sequencing operator \( \texttt{do} \):
  
  **Example:** \( \texttt{do 17 apply 'simex'/0()} \)
  
  \[ \rightarrow \text{The first subexpression is fully evaluated. Semantics: Discard its value and continue!} \]

\[
\text{do \hspace{1cm} val \hspace{1cm} e \hspace{1cm} } \xrightarrow{\tau} \hspace{1cm} e \hspace{1cm} \text{ (Seq₁)}
\]
A first sublanguage: Sequential Core Erlang

- Regard only the local aspects of expression evaluation.
- Side effects are formalized by non-determinism.
  \[ \rightarrow \text{Non-determinism is resolved later by considering the entire system} \]

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

A first example:

- Sequencing operator \textbf{do}:  

  \[ \text{Example: } \textbf{do} \ 17 \ \textbf{apply} \ ' \text{simex}'/0() \]
  \[ \rightarrow \text{The first subexpression is fully evaluated. Semantics: Discard its value and continue!} \]

\[
\begin{array}{c}
\text{do } val \ e \\
\xrightarrow{\tau} \ e
\end{array} \quad \text{(Seq1)}
\]

- \textbf{But} what about the evaluation of the first subexpression?  

  Consider for example:  
  \[ \text{do call} ' \text{erlang}':':' (Rcv,Msg) \ \textbf{apply} ' \text{proceed}'/0() \]
  \[ \rightarrow \text{Before evaluation of the do-operator can proceed, its first argument must be evaluated:} \]

\[
\begin{array}{c}
e_1 \\
\xrightarrow{\alpha} e_1'
\end{array} \quad \text{(Seq1)}
\]

\[
\begin{array}{c}
\text{do } e_1 e_2 \\
\xrightarrow{\alpha} \textbf{do} \ e_1' \ e_2
\end{array} \quad \text{(Seq2)}
\]
Pattern matching expressions

- **case** expressions:

\[ \exists i. \left( \text{match}(\text{val}, \text{cl}_i) = e' \land \forall j < i. \text{match}(\text{val}, \text{cl}_j) = \bot \right) \]

\[
\text{case } \text{val} \text{ of } \text{cl}_1 \cdots \text{cl}_k \text{ end } \overset{\pi}{\longrightarrow} e'
\]

(Case\(_1\))
Pattern matching expressions

- **case** expressions:

\[ \exists i. \left( \text{match}(val, cl_i) = e' \land \forall j < i. \text{match}(val, cl_j) = \bot \right) \]

\[
\text{case } val \text{ of } cl_1 \cdots cl_k \text{ end } \rightarrow_e e'
\]  

(Case_1)

- **receive** expressions:

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

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

Reception of the first matching message (c):

\[ \neg \text{qmatch}(q, cl_1, \ldots, cl_k) \quad \text{case } c \text{ of } cl_1 \ldots cl_k \text{ end } \rightarrow_e e' \quad c_t \in \text{Num} \cup \{\text{'infinity'}\} \]

(Rcv_1)

\[
\text{receive } cl_1 \cdots cl_k \text{ after } c_t \rightarrow e_t \rightarrow_{\text{ recv}(q, c)} e'
\]

Note: The prefix \( qc \) of the process’ mailbox is guessed nondeterministically!

\[ \leftrightarrow \text{ Reflected by the transition label } \text{recv}(q, c) \]
Global states and the transition system $T_s$:

- $\tau$ transitions are autonomous evaluation steps.
  - can be lifted to the system layer semantics directly:

$$
\begin{align*}
  e \xrightarrow{\tau} e' \\
  S \cup \{(e_i, q_i, L_i, t_i)\} \xrightarrow{\tau_s} S \cup \{(e'_i, q_i, L_i, t_i)\}
\end{align*}
$$

  (SeqCore)

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

$$
\begin{align*}
  e_i \xrightarrow{jlc} e'_i \\
  S \cup \{(e_i, q_i, L_i, t_i), (e_j, q_j, L_j, t_j)\} \xrightarrow{\text{send}(i,j,c)} S \cup \{(e'_i, q_i, L_i, t_i), (e_j, q_j \cdot c, L_j, t_j)\}
\end{align*}
$$

  (Send_1)
Global states and the transition system $T_s$:

- $\tau$ transitions are autonomous evaluation steps.
  - can be lifted to the system layer semantics directly:

$$
\begin{align*}
  e & \xrightarrow{\tau} e' \\
  S \cup \{(e, i, q, L, t)\} & \xrightarrow{\tau} S \cup \{(e', i, q, L, t)\}
\end{align*}
$$

(\text{SeqCore})

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

$$
\begin{align*}
  e_i & \xrightarrow{j!c} e' \\
  S \cup \{(e_i, i, q_i, L_i, t_i), (e_j, j, q_j, L_j, t_j)\} & \xrightarrow{\text{send}(i,j,c)} S \cup \{(e'_i, i, q_i, L_i, t_i), (e_j, j, q_j \cdot c, L_j, t_j)\}
\end{align*}
$$

(Send$_1$)

- Message reception:

$$
\begin{align*}
  e & \xrightarrow{\text{recv}(q_1,c)} e' \\
  S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\} & \xrightarrow{\text{recv}(i,c)} S \cup \{(e', i, q_1 \cdot q_2, L, t)\}
\end{align*}
$$

(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'
The Transition System of the Mutual Exclusion Protocol

(call 'locker': 'start', 0, ε) τ

(let MyPid = call 'erlang': 'self', 0, ε)

self()

(let MyPid = 0, 0, ε)

(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)

T

(do 2 apply 'locker', 0, 0, ε)

(call 'locker': 'client', 0, 1, ε)

(call 'locker': 'client', 0, 2, ε)

T

spawn(0, 2)

(do 2 apply ' locker', 0, 0, ε)

(call ' locker': 'client', 0, 1, ε)

(call ' locker': 'client', 0, 2, ε)

T

T

T

T

T

...
Abstracting from $\tau$ evaluation steps

$T S_{\sim} := (S_{\sim}, \text{Act}, \rightarrow, [s_0]_{\sim})$, where

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

1. $\xrightarrow{\sim}_s^*$ denotes the reflexive, symmetric and transitive closure of $\xrightarrow{\tau}_s$.

2. Equivalence relation: $\sim := \xrightarrow{\tau}_s^*$
~ Equivalent States of the Mutual Exclusion Protocol

- Call 'locker': 'start'(0, 0, ε)
  - Let MyPid = call 'erlang': 'self'(0, 0, ε)
    - self()
      - (let MyPid = 0, 0, ε)
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

Maude preliminaries:

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

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
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
   - \(\phi : \Sigma \rightarrow 2^N\) defines frozen argument positions.
   - \(R\) denotes the set of rewriting rules
     - needs not to be confluent!

**Idea:** Normalize term wrt. \(ER \uplus A\) and then apply the rewriting rules from \(R\!\)!
  - Coherence properties between \(ER \uplus A\) and \(R\) must be fulfilled!
Representation of processes and process systems in MAUDE

- **Processes:**
  \[
  \text{op } <\square|\square|\square|\square|\square|\square|\square|\square|\square|\square|\square|\square|\square|\square> : \text{Label SysResult Expr Pid Mailbox ProcessLinks TrapExit ModEnv } \rightarrow \text{Process}.
  \]
  Label, SysResult and ModEnv are needed in order to operationalize the semantics.

- **Process systems:**
  \[
  \text{op empty-processes } : \rightarrow \text{Processes } [\text{ctor}] .
  \]
  \[
  \text{op } \square||\square : \text{Processes Processes } \rightarrow \text{Processes } [\text{ctor assoc comm id: empty-processes}] .
  \]
  \[
  \text{subsort relation: Process } \sqsubseteq_{\text{Spec}} \text{Processes}
  \]

- **Process environments:**
  \[
  \text{op } ((\square,\square,\square,\square)) : \text{SysLabel Processes ModEnv PidSequence } \rightarrow \text{ProcessEnvironment} .
  \]
  Process environments constitute the states of our transition system.
Specify the equivalence \( \sim \) using the equational theory \( (\Omega, \mathcal{E}) \):

**Example:**

\[
\text{do } \text{val } e \xrightarrow{\tau} e \quad (\text{Seq}_1)
\]

\[
e_1 \xrightarrow{\alpha} e'_1 \quad \text{do } e_1 \quad \text{do } e'_1 e_2 \quad (\text{Seq}_2)
\]
Specify the equivalence $\sim$ using the equational theory $(\Omega, \mathcal{E})$:

**Example:**

\[
\begin{align*}
\text{do } & \text{val } e \xrightarrow{\tau} e \quad \text{(Seq}_1) \\
\text{do } & e_1 \xrightarrow{\alpha_{e}} e_1' \\
\text{do } & e_2 \xrightarrow{\alpha_{e}} \text{do } e_1' e_2 \quad \text{(Seq}_2)
\end{align*}
\]

- Evaluation of the `do` operator itself:

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

- Evaluation of the first subexpression:

\[
\text{ceq } \text{[norm-do]} : \\
<\tau|\text{RES }|\text{do } \text{EX1 } \text{EX2 }|\text{PID }|\text{MBOX }|\text{LINKS }|\text{TRAP }|\text{ME}> = \\
<\#\text{filterExit (ESL) }|\text{RES1 }|\text{do } \text{EX1'} } \text{EX2 }|\text{PID }|\text{MBOX }|\text{LINKS }|\text{TRAP }|\text{ME}> \\
\text{if not (EX1 } :: \text{ Const} ) \\
\text{\textbackslash} <\text{ESL }|\text{RES1 }|\text{EX1'} |\text{PID }|\text{MBOX }|\text{LINKS }|\text{TRAP }|\text{ME}> := \\
<\tau|\text{RES }|\text{EX1 }|\text{PID }|\text{MBOX }|\text{LINKS }|\text{TRAP }|\text{ME}> .
\]
Rewriting rules define the transition relation $R \rightarrow$:

**Idea:** Specify $\rightarrow \subseteq S/\sim \times S/\sim$ 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:

$$
\frac{e \xrightarrow{\text{recv}(q_1,c)} e'}{S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\}} \quad \xrightarrow{\text{recv}(i,c)}_{s} \quad S \cup \{(e', i, q_1 \cdot q_2, L, t)\}
$$

(Recv)
Rewriting Rules Define the Transition Relation \( \rightarrow \)

Rewriting rules define the transition relation \( R \rightarrow \):

**Idea:** Specify \( \rightarrow \subseteq S/\sim \times S/\sim \) 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:

\[
\begin{align*}
  & e \xrightarrow{\text{recv}(q_1, c)} e' \\
  & S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\} \xrightarrow{\text{recv}(i, c)} S \cup \{(e', i, q_1 \cdot q_2, L, t)\}
\end{align*}
\]

(Recv)

The corresponding conditional rewrite rule:

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

**Remark:** Receivable messages are observed on expression layer but removed on system layer!
Soundness and completeness

- Semantic point of view:

\[ [s]_E = [s]_{\text{A} \cup \text{ER}} \xrightarrow{R_{/\text{A} \cup \text{ER}}} [s']_{\text{A} \cup \text{ER}} = [s']_E \]

- Operational point of view:

\[ [s]_A \xrightarrow{ER_{/A}*} \xrightarrow{R_{/A}} [s']_A \]

Do they coincide?
Soundness and completeness

- Semantic point of view:

\[ [s]_E = [s]_{A\cup ER} \xrightarrow{R_{A\cup ER}} [s']_{A\cup ER} = [s']_E \]

- Operational point of view:

\[ [s]_A \xrightarrow{ER/A \ast} R_A \xrightarrow{\ast} [s']_A \]

Do they coincide?

Yes, they do!
Defining predicates

States of $TS_{\sim}$ are represented by $(\Omega, \mathcal{E})$ normal forms.

$\leftrightarrow$ Associate predicates to these terms:

<table>
<thead>
<tr>
<th>$s \models send(i, j, c)$</th>
<th>“process $i$ just sent message $c$ to process $j$”</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \models receive(i, c)$</td>
<td>“process $i$ just received $c$”</td>
</tr>
</tbody>
</table>

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

States of $T S_{\sim}$ are represented by $(\Omega, \mathcal{E})$ normal forms.

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
  \[
  \varphi_1 = scheduler(0, 1, 2) \rightarrow \Box (send(0, 1, ”ok”) \rightarrow (\neg send(0, 2, ”ok”) \ U send(1, 0, \{”rel”, 1\})))
  \]

- Eventually, the second client enters the critical section:
  \[
  \varphi_2 = scheduler(0, 1) \rightarrow \Diamond (send(0, 2, ”ok”))
  \]
  \[
  \varphi_3 = scheduler(0, 1, 2) \rightarrow \Diamond (send(0, 2, ”ok”))
  \]

But: In general (unfair scheduling), $\varphi_2$ is not fulfilled:

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/