Concurrent models of computation for embedded software and hardware!

Jörn W. Janneck
Researcher
Xilinx Research Labs

Writing actors: the CAL actor language

overview

- CAL actor language
  - what it looks like
- semantics
  - what it means and how it relates
- designing an actor language
  - actor properties and how to represent them
- using the language
  - working with actor descriptions
- what Xilinx does with actors
simple actors

```plaintext
actor ID () In ==> Out :
  action In: [a] ==> Out: [a] end
end

actor Add () Input1, Input2 ==> Output:
  action [a], [b] ==> [a + b] end
end

actor AddSeq () Input ==> Output:
  action [a, b] ==> [a + b] end
end
```

parameters

```plaintext
actor Scale (k) Input ==> Output:
  action [a] ==> [k * a] end
end
```

parametric actor definitions represent a family of actors
actions ≈ firing rules + firing functions

```plaintext
actor Add () Input1, Input2 ==> Output:
  action [a], [b] ==> [a + b] end
end
```

actions like these describe firing rules and firing function:

\[ U = \{(a, b) : a, b \in \mathbb{Z}\} \]
\[ f : (a, b) \mapsto (a + b) \]

multiple actions result in multiple firing rules and functions:

\[ U_1 = \{(a, \bot) : a \in \mathbb{Z}\}, f_1 : (a, \bot) \mapsto (a) \]
\[ U_2 = \{\bot, (a) : a \in \mathbb{Z}\}, f_2 : \bot, (a) \mapsto (a) \]

such actors may be non-deterministic
nondeterminism

\begin{verbatim}
actor NDMerge () Input1, Input2 ==> Output:
    action Input1: [x] ==> [x] end
    action Input2: [x] ==> [x] end

actor NDSplit () Input ==> Output1, Output2:
    action [x] ==> Output1: [x] end
    action [x] ==> Output2: [x] end
\end{verbatim}

... but it gets even worse...

\begin{verbatim}
actor NDMerge () Input1, Input2 ==> Output:
    action Input1: [x] ==> [x] end
    action Input2: [x] ==> [x] end
\end{verbatim}

\begin{verbatim}
actor NDSplit () Input ==> Output1, Output2:
    action [x] ==> Output1: [x] end
    action [x] ==> Output2: [x] end
\end{verbatim}

different actions may result in overlapping (here: identical) sets of firing rules, and different firing functions: the combined firing function isn’t (a function)!!!

the resulting process is no longer functional

\begin{verbatim}
what could this possibly be useful for???
\end{verbatim}

guards may constrain the tokens accepted by an action:

\begin{verbatim}
actor Split () Input ==> Y, N:
    action [a] ==> Y: [a]
guard P(a) end
    action [a] ==> N: [a]
guard not P(a) end
\end{verbatim}
guaranteed actions

\textbf{actor} \text{Select} () S, A, B \Rightarrow \text{Output:}

\begin{verbatim}
action S: [sel], A: [v] \Rightarrow [v]
guard sel end

action S: [sel], B: [v] \Rightarrow [v]
guard not sel end
\end{verbatim}

\begin{align*}
U_1 &= \{(\text{true}), (v), \bot): v \in \mathbb{Z}, f_1 : (\text{true}), (v), \bot \rightarrow (v) \\
U_2 &= \{(\text{false}), \bot, (v)): v \in \mathbb{Z}, f_2 : (\text{false}), \bot, (v) \rightarrow (v) \}
\end{align*}

cf Lect. 15, slide 8

actors with state

\textbf{actor} \text{Sum} () \text{Input} \Rightarrow \text{Output:}

\begin{verbatim}
sum := 0;

action [a] \Rightarrow [\text{sum}]
do
\text{sum} := \text{sum} + a;
end
\end{verbatim}

\textit{refers to state at the end of the action execution}

state requires an extension of the actor model:
- the firing rules may depend on the state (will see this later)
- the firing function takes [state, input] to [state, output]

here, the state space is isomorphic to \(\mathbb{Z}\)--thus:
\begin{align*}
U &= \{[\sigma, (a)] : \sigma, a \in \mathbb{Z}\} \\
f : [\sigma, (a)] \rightarrow [a + \sigma, (a + \sigma)]
\end{align*}

\textit{note: we will omit singleton state for stateless actors}

\text{cf Lect. 15, slide 8}
actors with state: rationale

could state not be realized by direct feedback?

yes, but state is special in a number of practical ways...
• there is but one instance of it
• it is always directly fed back
• it is not shared with other actors

more fundamentally, state is what allows for actors to be compositional:

a dataflow network cannot in general be represented
by a single stateless actor (the queues contain state)

---

state-dependent guards

actor Select () S, A, B ==> Output:

```plaintext
state = 0;

action S: [sel] =>
guard state = 0 do
  if sel then
    state := 1;
  else
    state := 2;
end

action A: [v] => [v]
guard state = 1 do
  state := 0;
end

action B: [v] => [v]
guard state = 2 do
  state := 0;
end
```

\[ U_1 = \{0, \{true, 1\}\} \rightarrow \{0, \{false, 1\}\} \]
\[ f_1 : \{0, \{true, 1\}\} \rightarrow \{1, 0\} \]
\[ f_1 : \{0, \{false, 1\}\} \rightarrow \{2, 1\} \]

\[ U_2 = \{1, \{false, 1\}\} : a \in \mathbb{Z} \]
\[ f_2 : \{1, \{false, 1\}\} \rightarrow \{0, a\} \quad \text{for} \quad a \in \mathbb{Z} \]

\[ U_3 = \{2, \{false, 1\}\} : a \in \mathbb{Z} \]
\[ f_3 : \{2, \{false, 1\}\} \rightarrow \{0, a\} \quad \text{for} \quad a \in \mathbb{Z} \]

cf Lect. 15, slide 8
priorities (when order matters)

\[
\text{actor } \text{ProcessStream}() \text{ In, Config } \Rightarrow \text{ Out:} \\
\begin{align*}
\text{c} &: \text{ initialConfig();} \\
\text{action Config: } [\text{newC}] \text{ } &\Rightarrow \text{ do} \\
\text{c} &: \text{ newC;} \\
\text{end}
\end{align*}
\]

Intuition:
Among the enabled actions, one with maximal priority is fired.

how to enforce firing of one action over another?

\[
\text{actor } \text{ProcessStream}() \text{ In, Config } \Rightarrow \text{ Out:} \\
\begin{align*}
\text{c} &: \text{ initialConfig();} \\
\text{config: action Config: } [\text{newC}] \text{ } &\Rightarrow \text{ do} \\
\text{c} &: \text{ newC;} \\
\text{process: action In: } [\text{data}] \\
&\Rightarrow [\text{compute(data, c)}] \text{ end}
\end{align*}
\]

priorities (the harmless case)

\[
\text{actor Route()} \text{ A } \Rightarrow X, Y, Z: \\
\begin{align*}
\text{action } [v] &\Rightarrow X: [v] \\
\text{guard } P(v) \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{action } [v] &\Rightarrow Y: [v] \\
\text{guard } Q(v) \text{ and not } P(v) \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{action } [v] &\Rightarrow Z: [v] \\
\text{guard not Q(v) and not P(v)} \text{ end}
\end{align*}
\]

Intuition:
Among the enabled actions, one with maximal priority is fired.

\[
\text{actor Route()} \text{ A } \Rightarrow X, Y, Z: \\
\begin{align*}
toX: \text{ action } [v] &\Rightarrow X: [v] \\
\text{guard } P(v) \text{ end}
\end{align*}
\]

\[
\begin{align*}
toY: \text{ action } [v] &\Rightarrow Y: [v] \\
\text{guard } Q(v) \text{ end}
\end{align*}
\]

\[
\begin{align*}
toZ: \text{ action } [v] &\Rightarrow Z: [v] \text{ end}
\end{align*}
\]

Priority:
\[ toX > toY > toZ; \]
Xilinx Research Labs  
San Jose, CA

Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language

priorities (more merging)

actor BiasedMerge ()

Input1, Input2 ==> Output:

A: action Input1: [x] ==> [x] end
B: action Input2: [x] ==> [x] end

priority A > B end
end

actor PrettyFairMerge ()

Input1, Input2 ==> Output:

Both: action [x], [y] ==> [x, y] end
Both: action [x], [y] ==> [y, x] end
One: action Input1: [x] ==> [x] end
One: action Input2: [x] ==> [x] end

priority Both > One end
end

CAL actor language

formal version of a CAL actor (pre-semantics)

set of n action indices:  \( I = \{1, \ldots, n\} \)

state space:  \( \Sigma \)

initial state:  \( \sigma_0 \in \Sigma \)

non-reflexive partial order on I:  \( I \subseteq I \times I \)

each action i is defined as:  \( \langle U_i, f_i \rangle \)

(non-joinable) firing rules:  \( U_i \subseteq \Sigma \times S_{\text{fin}}^m \)

firing function:  \( f_i : U_i \rightarrow S_{\text{fin}}^n \times \Sigma \)

allowing for some handwaving about exactly how this information is derived from the actor source text
semantics
actor descriptions and actors

\begin{aligned}
\text{actor Split ( } & \text{ Input } = Y, N \text{):} \\
\text{ action } [a] & \Rightarrow Y: \{a\} \\
\text{ guard } P(a) \text{ end} \\
\text{ action } [a] & \Rightarrow N: \{a\} \\
\text{ guard } \neg P(a) \text{ end}
\end{aligned}

\[ L \]

\[ A \]

\[ \{ X \} \]

\[ ? \]

Writing actors: the CAL actor language
Jörn W. Janneck
Xilinx Research Labs
San Jose, CA

\[ \begin{aligned}
\text{semantics} \\
&\text{A: actor transition systems (untimed version)}
\end{aligned} \]

an actor transition system (ATS) in
a state space \( \Sigma \):
\[ \left( \sigma_0, \tau, \succ \right) \]

- initial state \( \sigma_0 \in \Sigma \)
- transition relation \( \tau \subseteq \Sigma \times \mathbb{S}_{\text{fin}}^m \times \mathbb{S}_{\text{fin}}^n \times \Sigma \)
- priority order \( \succ \) irreflexive partial order on \( \tau \)

\[ \begin{aligned}
\text{exercise for the reader:} \\
1. \text{ what could be a suitable fixed-point semantics for an ATS?} \\
(\text{i.e. what "process" corresponds to an ATS?}) \\
2. \text{ what would be a suitable semantics for a network of ATS?}
\end{aligned} \]

notation:
\[ (\sigma, s, s', \sigma') \in \tau \equiv \sigma \xrightarrow{\sigma \cdot s \cdot s'} \sigma' \]
semantics

\([\cdot]:\) semantic mapping

constructing an ATS from a formal CAL actor in state space \(\Sigma:\)

\[
\begin{align*}
\text{CAL actor} & \\
\langle \sigma_0, \{ U_i, f_i \}_{i \in I}, \succ \rangle & \\
I & = \{1, \ldots, n\} \\
\sigma_0 & \in \Sigma \\
\succ & \subseteq I \times I \\
U_i & \subseteq \Sigma \times S^m_{\phi_i} \\
f_i : U_i & \rightarrow S^m_{\phi_i} \times \Sigma
\end{align*}
\]

\[
\begin{align*}
\text{ATS} & \\
\langle \sigma_0, \tau, \succ \rangle & \\
\tau & = \bigcup_{i \in I} \tau_i \\
\tau_i & = \{ (\sigma, s, s', \sigma') : [\sigma, s] \in U_i \land f_i([\sigma, s]) = [\sigma', s'] \} \\
I & \succ \tau' \iff \\
& \exists i, j \in I : i \succ j \land t \in \tau_i \land t' \in \tau_j \land \\
& \neg \exists k \in I : k \succ i \land t' \in \tau_k
\end{align*}
\]

semantics

expressiveness

example 1:

\[
\tau = \{(a, b) \cdot) \cdot ; a, b \in \mathbb{Z}\}
\]

(nonfinite/unbounded nondeterminism)

example 2:

\[
\begin{align*}
& \begin{array}{c}
\text{with} \\
& (a_1, \ldots, a_k)
\end{array} \\
& \begin{array}{c}
\text{such that} \\
& a_i \in \mathbb{Z}, n = \min \left\{ k \in \mathbb{N} : k = \sum_{i=1}^{k} a_i \right\}
\end{array}
\end{align*}
\]

(prefix length depends on token values)
given a set of firing rules and a firing function

\[ \langle U, f \rangle \]

the corresponding ATS is

\[ \langle \bullet, \tau, \emptyset \rangle \]

with \( \tau = \{(\bullet, s, f(s)\bullet) : s \in U\} \)

note: this construction is injective up to isomorphism
what \textbf{CAL} actor results in FR+FF?

\begin{itemize}
  \item \textbf{actor} \textbf{C} () \textbf{X} \textbf{==> Y}:
  \begin{itemize}
    \item \textbf{action} \textbf{[a]} \textbf{==> \{f(a)\}}
    \item \textbf{guard} \textbf{P(a)}
    \item \textbf{action} \textbf{[a]} \textbf{==> \{g(a)\}}
    \item \textbf{guard} \textbf{Q(a)}
  \end{itemize}
\end{itemize}

\textit{what does it depend on?}

\begin{itemize}
  \item \textbf{actor} \textbf{E} () \textbf{X} \textbf{==> Y}:
  \begin{itemize}
    \item \textbf{A}: \textbf{action} \textbf{[a]} \textbf{==> \{f(a)\}}
    \item \textbf{guard} \textbf{P(a)}
    \item \textbf{B}: \textbf{action} \textbf{[a]} \textbf{==> \{g(a)\}}
    \item \textbf{guard} \textbf{Q(a)}
    \item \textbf{priority} \textbf{A > B;}
  \end{itemize}
\end{itemize}

\textit{(equivalence, isomorphism)}

\textbf{exercise:}
name some \textit{p} along with a representative \textit{p}\textsuperscript{CAL}

\begin{itemize}
  \item \textbf{actor} \textbf{B} () \textbf{X} \textbf{==> Y}:
  \begin{itemize}
    \item \textbf{action} \textbf{[a]} \textbf{==> \{1\}}
    \item \textbf{guard}
    \item \textbf{action} \textbf{[a]} \textbf{==> \{2\}}
  \end{itemize}
\end{itemize}

static properties

\[p^L \subseteq [p \cap [L]]^{-1}\]
language design

redundancy (non-orthogonality)

property p of this actor?
what’s the p^{CAL} that represents it?

working with actor descriptions

example: actor projection (remove outputs)

say the Diff output is unconnected...

remove port and all its output expressions
example: actor projection (remove inputs)

```plaintext
actor ProcessStream () In, Config ==> Out:
  c := initialConfig();
  config: action Config: [newC] ==>
     do
       c := newC;
     end
  process: action In: [data]
          ==> [compute(data, c)] end
  priority
  config > process;
end
```

assume the Config input is not connected...

```plaintext
actor ProcessStream' () In ==> Out:
  c := initialConfig();
  process: action In: [data]
        ==> [compute(data, c)] end
end
```

remove the port, and any action that reads from it

working with actor descriptions

discovering concurrency

```plaintext
actor B () a, b ==> x, y:
  s := <something>;
  action a: [v] ==>
    x: [f(v, s)] end
  action b: [v] ==>
    y: [g(v)]
    do
      s := h(v, s);
    end
end
```

actors to (programmable) hardware

- driver application
  - video encoding and decoding (MPEG4 et al.)
- challenges
  - fast hardware
  - small hardware
  - hardware & software
- actor machines
  - actor-specific configurable processor architectures
    » pipelined action firing
    » resource sharing
- shameless plug
  - Xilinx does take interns...

The end.

Thanks!

info:
  http://embedded.eecs.berkeley.edu/caltrop/

contact:
  jorn.janneck@xilinx.com