Concurrent models of computation for embedded software

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

CAL actor language
simple actors
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
Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language

## parameters

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

parametric actor definitions represent a family of actors
Writing actors: the CAL actor language
Jörn W. Janneck

## CAL actor language

## actions $\approx$ firing rules + firing functions

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

actions like these describe firing rules and firing function:

$$
\begin{aligned}
& U=\{\langle(a),(b)\rangle: a, b \in \mathbf{Z}\} \\
& f:\langle(a),(b)\rangle \mapsto(a+b)
\end{aligned}
$$

Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language

## multiple actions

```
actor NDMerge () Input1, Input2 ==> Output:
    action Input1: [x] ==> [x] end
    action Input2: [x] ==> [x] end
end
```

multiple actions result in multiple firing rules and functions:

$$
\begin{aligned}
& U_{1}=\{\langle(a), \perp\rangle: a \in \mathbf{Z}\}, f_{1}:\langle(a), \perp\rangle \mapsto(a) \\
& U_{2}=\{\langle\perp,(a)\rangle: a \in \mathbf{Z}\}, f_{2}:\langle\perp,(a)\rangle \mapsto(a)
\end{aligned}
$$

such actors may be non-deterministic

## CAL actor language

## nondeterminism

```
actor NDMerge () Input1, Input2 ==> Output:
    ction Input2: \([x]==>[x]\) end
\(U_{2}=\{\langle\perp,(a)\rangle: a \in \mathbf{Z}\}, f_{2}:\langle\perp,(a)\rangle \mapsto(a)\)
```

end
... but it gets even worse...

```
actor NDSplit () Input ==> Output1, Output2:
    action [x] ==> Output1: [x] end
    action [x] ==> Output2: [x] end
end
```

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
what could this possibly be useful for???
Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language

## guarded actions

```
actor Split () Input ==> Y, N:
    action [a] ==> Y: [a]
    guard P(a) end
    action [a] ==> N: [a]
    guard not P(a) end
end
```

guards may constrain the tokens accepted by an action:

$$
\begin{aligned}
& U_{1}=\{(a): P(a)\}, f_{1}:(a) \mapsto\langle(a), \perp\rangle \\
& U_{2}=\{(a): \neg P(a)\}, f_{2}:(a) \mapsto\langle\perp,(a)\rangle
\end{aligned}
$$

CAL actor language

## guarded actions

```
actor Select () S, A, B ==> Output:
    action S: [sel], A: [v] ==> [v]
    guard sel end
    action S: [sel], B: [v] ==> [v]
    guard not sel end
end
```

$U_{1}=\{\langle($ true $),(v), \perp\rangle: v \in \mathbf{Z}\}, f_{1}:\langle($ true $),(v), \perp\rangle \mapsto(v)$
$U_{2}=\{\langle($ false $), \perp,(v)\rangle: v \in \mathbf{Z}\}, f_{2}:\langle($ false $), \perp,(v)\rangle \mapsto(v)$
cf Lect. 15, slide 8
Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language
actors with state
actor Sum () Input ==> Output:

refers to state at the end
of the action execution end
end
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 Z---thus:

$$
\begin{aligned}
& U=\{[\sigma,(a)]: \sigma, a \in \mathbf{Z}\} \\
& f:[\sigma,(a)] \mapsto[a+\sigma,(a+\sigma)]
\end{aligned}
$$

## CAL actor language

## 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)
Writing actors: the CAL actor language
Jörn W. Janneck

CAL actor language

## state-dependent guards

```
actor Select () S, A, B ==> Output:
    state = 0;
    action S: [sel] ==>
    guard state = 0 do
            if sel then
                state := 1;
            else
                state := 2;
            end
        end
    action A: [v] ==> [v]
    guard state = 1 do
            state := 0;
        end
        action B: [v] ==> [v
        guard state = 2 do
            state := 0;
        end
end

CAL actor language
priorities (when order matters)


CAL actor language
Priorities (the harmless case)


CAL actor language
priorities (more merging)


CAL actor language
formal version of a CAL actor (presemantics)
\[
\left\langle\sigma_{0},\left\langle U_{i}, f_{i}\right\rangle_{i \in I}, \succ\right\rangle \begin{aligned}
& \text { set of } \mathrm{n} \text { action indices: } I=\{1, \ldots, n\} \\
& \text { state space: } \Sigma \\
& \text { initial state: } \sigma_{0} \in \Sigma \\
& \text { non-reflexive partial order on I: } \begin{array}{l}
I \\
\succ \\
\text { each action i is defined as: }
\end{array} \\
& \begin{aligned}
& \left\langle U_{i}, f_{i}\right\rangle
\end{aligned} \\
& \text { (non-joinable) firing rules: } U_{i} \subseteq \Sigma \times S_{\text {fin }}^{m} \\
& \text { firing function: } f_{i}: U_{i} \rightarrow S_{f i n}^{n} \times \Sigma
\end{aligned}
\]
allowing for some handwaving about exactly how this information is derived from the actor source text
Writing actors: the CAL actor language
Jörn W. Janneck

\section*{semantics}
actor descriptions and actors

semantics

\section*{\(A\) : actor transition systems (untimed version)}
an actor transition system (ATS) in
a state space \(\Sigma\) :
\[
\left\langle\sigma_{0}, \tau, \succ\right\rangle
\]
initial state \(\quad \sigma_{0} \in \Sigma\)
transition relation \(\quad \tau \subseteq \Sigma \times \mathbf{S}_{f i n}^{m} \times \mathbf{S}_{f i n}^{n} \times \Sigma\)
priority order \(\succ\) irreflexive partial order on \(\tau\)
exercise for the reader:
1. what could be a suitable fixed-point semantics for an ATS?
(i.e. what "process" corresponds to an ATS?)
2. what would be a suitable semantics for a network of ATS?
notation: \(\left(\sigma, s, s^{\prime}, \sigma^{\prime}\right) \in \tau \equiv \sigma \xrightarrow{s \mapsto s^{\prime}} \sigma^{\prime}\)
Writing actors: the CAL actor language
Jörn W. Janneck

\section*{semantics \\ [•]: semantic mapping}
constructing an ATS from a formal CAL actor in state space \(\Sigma\) :
\(\left|\right.\)\begin{tabular}{l||}
\hline \multicolumn{1}{c|}{CAL actor } \\
\(\left.\sigma_{0},\left\langle U_{i}, f_{i}\right\rangle_{i \in I},{ }^{I}\right\rangle\) \\
\hline
\end{tabular}\(|\)
\[
\begin{gathered}
\mathrm{ATS} \\
\left\langle\sigma_{0}, \tau, \succ\right\rangle
\end{gathered}
\]
\[
\begin{aligned}
& \tau= \bigcup_{i \in I} \tau_{i} \\
& \tau_{i}=\left\{\left(\sigma, s, s^{\prime}, \sigma^{\prime}\right):[\sigma, s] \in U_{i} \wedge f_{i}([\sigma, s])=\left[\sigma^{\prime}, s^{\prime}\right]\right\} \\
& t \succ t^{\prime} \Leftrightarrow
\end{aligned}
\]
\[
\exists \mathrm{i}, \mathrm{j} \in \mathrm{I}: \mathrm{i} \succ \mathrm{j} \mathrm{j} \wedge \mathrm{t} \in \tau_{\mathrm{i}} \wedge \mathrm{t}^{\prime} \in \tau_{\mathrm{j}} \wedge
\]
\[
\neg \exists \mathrm{k} \in \mathrm{I}: \mathrm{k} \succ \mathrm{i} \wedge \mathrm{t}^{\prime} \in \tau_{\mathrm{k}}
\]
Writing actors: the CAL actor language
Jörn W. Janneck
semantics
expressiveness


\section*{relation to firing rules/firing function}
given a set of firing rules and a firing function
\[
\langle U, f\rangle
\]
the corresponding ATS is
\[
\langle\bullet, \tau, \varnothing\rangle
\]
with \(\tau=\{(\bullet, s, f(s), \bullet): s \in U\}\)
note: this construction is injective up to isomorphism

semantics

\section*{relation to firing rules/firing function}

language design
what CAL actor results in FR+FF?

language design

\section*{static properties}

language design
redundancy (non-orthogonality)
```

actor AlmostFairMerge ()
Input1, Input2 ==> Output:
s := 0;
action Input1: [x] ==> [x]
guard s = 0
do
s := 1;
end
action Input2: [x] ==> [x]
guard s = 1
do
s := 0;
end
end

```
property p of this actor?
what's the \(p^{C A L}\) that represents it?
```

actor AlmostFairMerge ()
Input1, Input2 ==> Output:
A: action Input1: [x] ==> [x] end
B: action InputB: [x] ==> [x] end
schedule fsm s1:
s1 (A) --> s2;
s2 (B) --> s1;
end
end

```
rule of thumb:
if \(p\) is important, and \(p^{L}\) is nasty, it may be time for a new construct.
Writing actors: the CAL actor language
Jörn W. Janneck
working with actor descriptions
example: actor projection (remove outputs)
```

actor AddSub () A, B ==> Sum, Diff:
action [a], [b] ==> [a + b], [a - b] end
end

```
say the Diff output is unconnected...
remove port and all its output expressions
```

actor AddSub' () A, B ==> Sum:
action [a], [b] ==> [a + b] end
end

```
working with actor descriptions
example: actor projection (remove inpus)

working with actor descriptions
discovering concurrency


\section*{Cal @ Xilinx}

\section*{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...
Writing actors: the CAL actor language
Jörn W. Janneck

\section*{The end.}

\section*{Thanks!}
info:
http://embedded.eecs.berkeley.edu/caltrop/
contact:
jorn.janneck@xilinx.com```

