Communication-based Development of Systems Using Standard Programming Languages

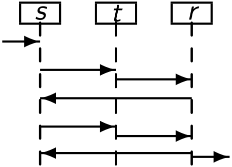
Dr. Annette Stümpel

Institute of Software Technology and Programming Languages
University of Lübeck

KPS'09

Motivation

Communication-based specification



message sequence charts
sequence diagrams
stream processing
...

?

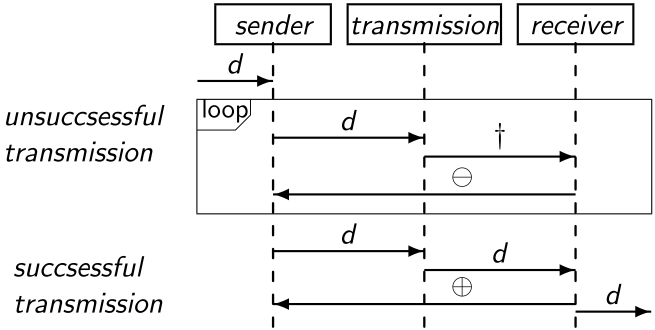
Program

C O D E

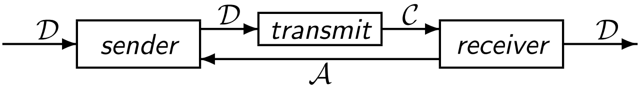
Java
Haskell
Erlang
...

Graphical Communication-based Specification

Example Transmission of a message *d* from a sender to a receiver via an unreliable transmission with acknowledgements



Architecture:



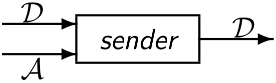
Stream Processing

Streams record the sequence of messages transmitted via a communication line.

Finite streams: $\mathcal{A}^\star = \underbrace{\{\langle\rangle\}}_{\text{empty stream}} \cup \underbrace{\mathcal{A} \times \mathcal{A}^\star}_{\text{non-empty streams } x \triangleleft X}$

Infinite streams arise as limit points.

A stream function $f : \mathcal{A}_1^\star \times \ldots \times \mathcal{A}_m^\star \to \mathcal{B}_1^\star \times \ldots \times \mathcal{B}_n^\star$ maps input streams to output streams:



monotonicity: Previous output messages can not be cancelled.

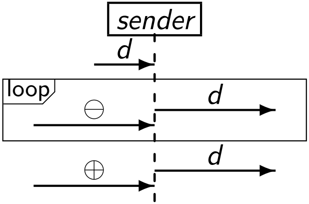
continuity: The behaviour on infinite input streams is approximated by the behaviour on finite streams.

Example Sender: Input / Output Behaviour

Interface $send : \mathcal{D}^\star \times \mathcal{A}^\star \rightarrow \mathcal{D}^\star$

Behaviour

$$
\begin{aligned}
send(\langle\rangle, A) &= \langle\rangle \\
send(d \triangleleft D, A) &= d \triangleleft wait(d)(D, A) \\
wait(d)(D, \langle\rangle) &= \langle\rangle \\
wait(d)(D, \oplus \triangleleft A) &= send(D, A) \\
wait(d)(D, \ominus \triangleleft A) &= d \triangleleft wait(d)(D, A)
\end{aligned}
$$

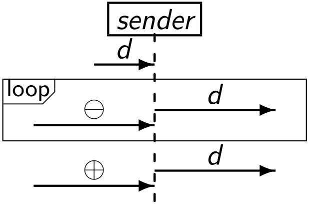


Example Sender: Input / Output Behaviour

Interface $send : \mathcal{D}^\star \times \mathcal{A}^\star \rightarrow \mathcal{D}^\star$

Behaviour

$$
\begin{aligned}
send(\langle\rangle, A) &= \langle\rangle \\
send(d \triangleleft D, A) &= d \triangleleft wait(d)(D, A) \\
wait(d)(D, \langle\rangle) &= \langle\rangle \\
wait(d)(D, \oplus \triangleleft A) &= send(D, A) \\
wait(d)(D, \ominus \triangleleft A) &= d \triangleleft wait(d)(D, A)
\end{aligned}
$$

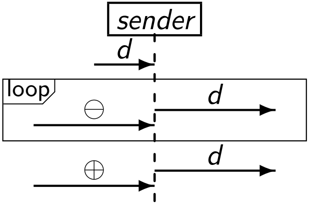


Example Sender: Input / Output Behaviour

Interface $send : \mathcal{D}^\star \times \mathcal{A}^\star \rightarrow \mathcal{D}^\star$

Behaviour

$$\begin{aligned} send(\langle\rangle, A) &= \langle\rangle \\ send(d \triangleleft D, A) &= d \triangleleft wait(d)(D, A) \\ wait(d)(D, \langle\rangle) &= \langle\rangle \\ wait(d)(D, \oplus \triangleleft A) &= send(D, A) \\ wait(d)(D, \ominus \triangleleft A) &= d \triangleleft wait(d)(D, A) \end{aligned}$$



Eliminating *wait***:**

$$\begin{aligned} send(\langle\rangle, A) &= \langle\rangle \\ send(d \triangleleft D, \langle\rangle) &= \langle d \rangle \\ send(d \triangleleft D, \oplus \triangleleft A) &= d \triangleleft send(D, A) \\ send(d \triangleleft D, \ominus \triangleleft A) &= d \triangleleft send(d \triangleleft D, A) \end{aligned}$$

Communication-based Programming in Haskell

Laziness makes Haskell an interesting candidate for communication-based programming.

channels lazy lists

$$\langle\rangle \;\widehat{=}\; \bot$$
$$x \triangleleft X \;\widehat{=}\; \mathtt{x : X}$$

components list functions

composition network of mutually recursive equations naming the channels

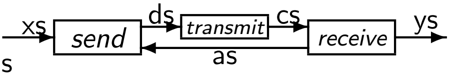
! caution with patterns for lazy lists

Example in Haskell

```
send :: [a] -> [Bool] -> [a]
send [] as = []
send (d:xs) as = d : wait d xs as
  where wait d xs [] = []
        wait d xs ( True:as) = send xs as
        wait d xs (False:as) = d : wait d xs as

transmit :: [Int] -> [a] -> [Maybe a] ...
receive :: [Maybe a] -> ([a],[Bool]) ...

network oracle xs = ys
   where ds = send xs as
         cs = transmit oracle ds
         (ys,as) = receive cs
```

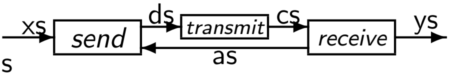


Example in Haskell

```
send :: [a] -> [Bool] -> [a]
send [] as = []
send (d:xs) as = d : wait d xs as
  where wait d xs [] = []
        wait d xs ( True:as) = send xs as
        wait d xs (False:as) = d : wait d xs as

transmit :: [Int] -> [a] -> [Maybe a] ...
receive :: [Maybe a] -> ([a],[Bool]) ...

network oracle xs = ys
   where ds = send xs as
         cs = transmit oracle ds
         (ys,as) = receive cs
```



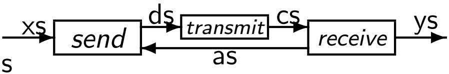
works fine

Example in Haskell without Auxiliary wait Function

```
send :: [a] -> [Bool] -> [a]
send [] as = []
send (d:xs) [] = [d]
send (d:xs) ( True:as) = d : send xs as
send (d:xs) (False:as) = d : send (d:xs) as
```

```
transmit :: [Int] -> [a] -> [Maybe a] ...
receive :: [Maybe a] -> ([a],[Bool]) ...
```

```
network oracle xs = ys
  where ds = send xs as
        cs = transmit oracle ds
        (ys,as) = receive cs
```

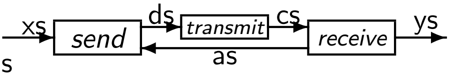


Example in Haskell without Auxiliary wait Function

```
send :: [a] -> [Bool] -> [a]
send [] as = []
send (d:xs) [] = [d]
send (d:xs) ( True:as) = d : send xs as
send (d:xs) (False:as) = d : send (d:xs) as
```

```
transmit :: [Int] -> [a] -> [Maybe a] ...
receive :: [Maybe a] -> ([a],[Bool]) ...
```

```
network oracle xs = ys
   where ds = send xs as
         cs = transmit oracle ds
         (ys,as) = receive cs
```



ERROR (control stack overflow)

Communication-based Programming in Java

Threads make Java suitable for communication-based programming.

channels	threads realizing FiFo queues
components	threads accessing input and output ports
composition	manual setting of channels as input and output ports
support	STREAMS! tool

State transition tables form the basis for the implementation of the components.

State transition table for the sender:

current state		input		next state		output
control	data	mess	ack	control	data	
send		d		*wait*	d	$\langle d \rangle$
wait	d		$\oplus$	*send*		$\langle\rangle$
wait	d		$\ominus$	*wait*	d	$\langle d \rangle$

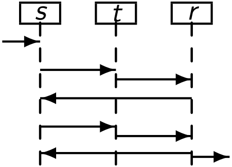
Example in Java

A component must contain a method which implements the transitions of the state transition table:

```
public void processStep() throws ... {
  switch (cState) {
    case SENDING: if (!isEmpty(0)) {
          cState = CState.WAITING;}
          dState = ((Integer)get(0)).intValue();
          set(0,dState);
          }
      break;
    case WAITING: ...
       break;
  }
}
```

Conclusion

Communication-based specification



message sequence charts
sequence diagrams
stream processing
...

State-based specification

state	input	state	output

state transition machines
state diagrams
state charts
...

Program

`C O D E`

Java
Haskell
Erlang
...