Transforming System Views in Software Design

WALTER DOSCH
Institute of Software Technology and Programming Languages
University of Lübeck
Lübeck, Germany

`http://www.isp.uni-luebeck.de`

1. Introduction — IT Trends

- *Computing power* and *performance* increase.
- *Multi-functionality*
- *Interoperability*
- *Connectivity*
- *Dependability* (safety and security)
- *Reusabilty* of Solutions
- *(Quasi) Standards*
- *Computing Paradigms*
- *Software Engineering — Software Industry*

. . . Scientific foundations of software technology?

1. Introduction — Layers of Software Technology

Applications

$\updownarrow$

Tools

$\updownarrow$

System Development Process

$\updownarrow$

Description Techniques

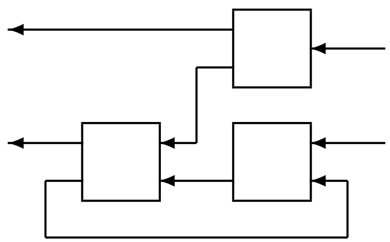
$\updownarrow$

$\rightarrow$ System Model $\leftarrow$

$\updownarrow$

System Model Theory

1. Introduction — System Model

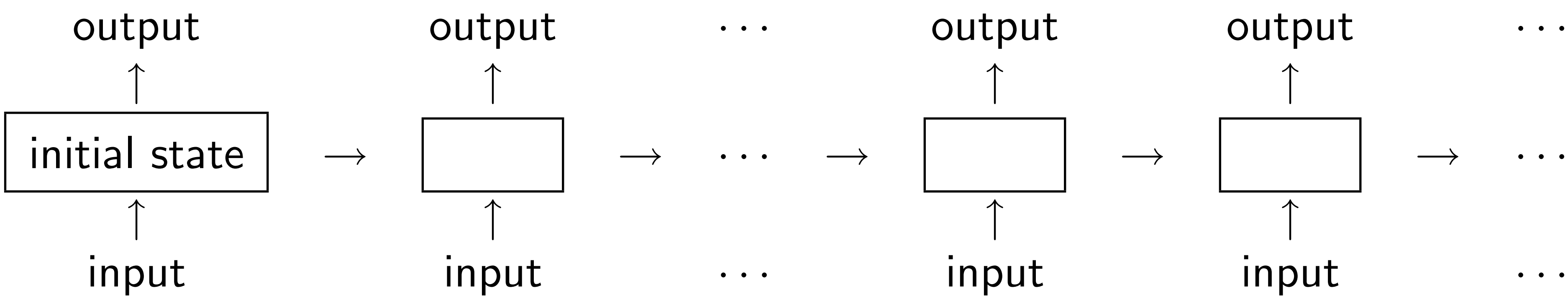


- **Distributed systems** are networks of *components*.
- Components **communicate asynchronously** via *unidirectional channels*.
- **Application areas**: processing — memory — transmission – control.
- **System model** uses *streams* to record *communication histories*.
- A **stream processing function** describes a component's *I/O behaviour*.
- A **state transition system** describes a component's **implementation.**

1. Introduction — Models of Computation

Classical Model of Computation

Model of Computation for Interactive Systems



1. Introduction — Components

Ingredients for a Component Engineering Framework

- (Uniform) notion of **component**
  - *Syntactic interface*
  - *Semantic interface* $\widehat{=}$ *(input/output) behaviour*
- **Specification**
- **Stepwise Refinement**
  - *Property refinement*
  - *Decomposition — Architecture*
  - *Implementation*
- **Compositionality** — Architecture
- **System views** on different **levels of abstraction**

... Graphical notation

1. Introduction — Views of a Component

Data Structure	Communication-Based Description	State-Based Description	Trace-Based Description
Data Model	*Black-box View*	*Glass-box View*	*Process View*

UML

- *Class diagrams* $\leftrightarrow$ data structure
- *State machines* $\leftrightarrow$ state-based description
- *Sequence diagrams* $\leftarrow$ communication-based description

2. Data Model

The **data model** describes the *data structure* underlying a component.

- **Signature** $\Sigma = (T, F)$
  - T is a set of *types*.
  - F is a family of *function symbols* $f : t_1 \star t_2 \star \ldots \star t_n \to t_{n+1} \quad (n \geq 0)$.
- $\Sigma-$**Equations** $\quad \forall x : t \;\; term_1 = term_2$
- **Abstract Data Type** (Σ, E)
- A **data model** $\mathcal{A}$ is a Σ-algebra providing
  - a *carrier set* $t^{\mathcal{A}}$ carrier set for each type $t \in T$,
  - a *function* $f^{\mathcal{A}} : t_1^{\mathcal{A}} \times t_2^{\mathcal{A}} \times \ldots \times t_n^{\mathcal{A}} \to t_{n+1}^{\mathcal{A}}$ for each function symbol $f \in F$.
- **Functional Programming**
- **Recursive Data Structures**

Stacks of Natural Numbers

$$
\begin{array}{l}
empty : \ \rightarrow stack \\
prefix : nat \star stack \rightarrow stack \\
first : stack \rightarrow nat \\
rest : stack \rightarrow stack
\end{array}
$$

$$
\begin{array}{rcl}
first(empty) & = & \textsf{undefined} \\
first(prefix(n, s)) & = & n \\
rest(empty) & = & \textsf{undefined} \\
rest(prefix(n, s)) & = & s
\end{array}
$$

Structural Induction

$$
\frac{P[empty] \ \wedge \ P[s] \Rightarrow P[prefix(n, s)]}{\forall \ s : stack \ P[s]}
$$

Data model $\mathcal{S}$ for *stacks of natural numbers*

$$
\begin{array}{rcl}
stack^{\mathcal{S}} & = & \mathbb{N}^{\star} \\
empty^{\mathcal{S}} & = & \langle\rangle \\
prefix^{\mathcal{S}}(n, S) & = & \langle n\rangle \,\&\, S \\
first^{\mathcal{S}}(\langle n_1, \ldots, n_k\rangle) & = & \begin{cases} \text{undefined} & \text{if} \quad k = 0 \\ n_1 & \text{if} \quad k \geq 1 \end{cases} \\
rest^{\mathcal{S}}(\langle n_1, \ldots, n_k\rangle) & = & \begin{cases} \text{undefined} & \text{if} \quad k = 0 \\ \langle n_2, \ldots, n_k\rangle & \text{if} \quad k \geq 1 \end{cases}
\end{array}
$$

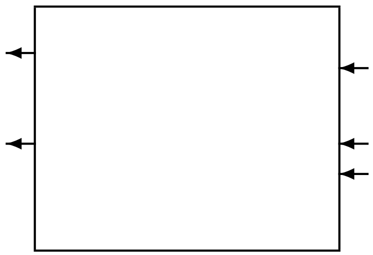
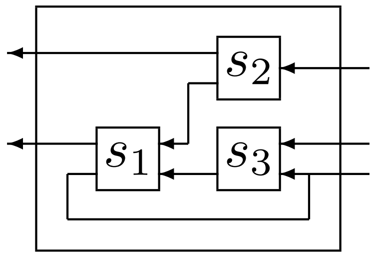
Vertical Modularization

Program
(Σ, E)
Data Model

No **selective updating**, **sharing**, **pointers**, . . .

3. Communication-Based Description: Black-box View

Top-down design of components

black box view		*glass box view*
		
stream processing function	$\Rightarrow$	**state transition machine**

Black box view versus **glass box view**

- *Input/output Behaviour — Architecture*
- *Composition — Implementation*
- *Correctness — Efficiency*

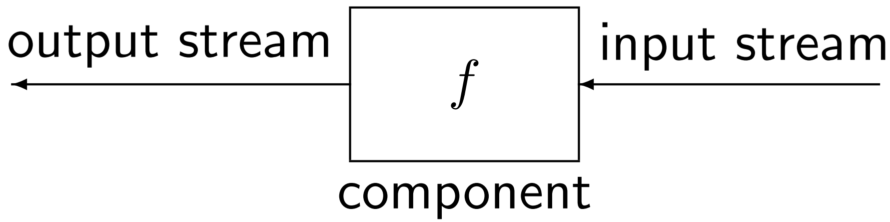
3.1 Streams and Stream Transformers

Streams model *communication histories* on *unidirectional channels.*

Finite streams $\mathcal{A}^\star = \{\langle x_0, \ldots, x_{m-1} \rangle \mid x_i \in \mathcal{A}\,,\, m \geq 0\}$

Concatenation $\langle x_0, ..., x_{m-1} \rangle \;\&\; \langle y_0, ..., y_{n-1} \rangle = \langle x_0, ..., x_{m-1}, y_0, ..., y_{n-1} \rangle$

Prefix relation $X \sqsubseteq Y$ iff $\exists R \in \mathcal{A}^\star : X \,\&\, R = Y$
describes *operational progress in time*.

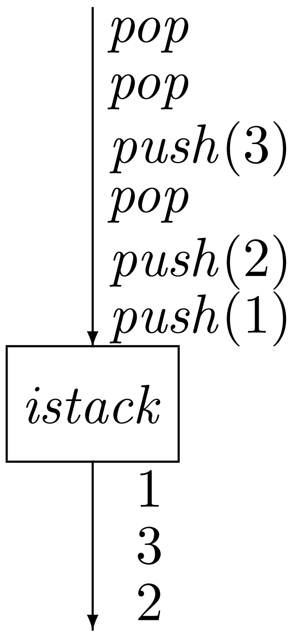


Stream transformer $f : \mathcal{A}^\star \to \mathcal{B}^\star$

Monotonicity $X \sqsubseteq Y \Rightarrow f(X) \sqsubseteq f(Y)$

3.2 Interactive Stack — Informal Description

An **interactive stack** stores an unbounded number of elements following a *last-in/first-out strategy*. The input consists of *push commands* entering a datum, and *pop commands* requesting the datum stored most recently.



3.3 Interactive Stack — Interface

Interaction Interface $(\mathcal{I}, \mathcal{O})$

type of input messages: $\mathcal{I} = \{pop, reset\} \cup push(nat)$

type of output messages: $\mathcal{O} = nat$

Transformation $Functional\ Signature \mapsto Interaction\ Interface$

$$\frac{prefix : nat \star stack \to stack}{push : nat \to [\underline{stack} \to \underline{stack}]}$$

$$\frac{\begin{array}{c} first : stack \to nat \\ rest : stack \to stack \end{array}}{pop : \underline{stack} \to nat \star \underline{stack}}$$

$$\frac{empty : \ \to stack}{reset : \underline{stack} \to \underline{stack}}$$

Design decisions about **encapsulation** $(\underline{stack})$ and interaction interface.

3.4 Interactive Stack — Behaviour

$$istack : \mathcal{I}^\star \to \mathcal{O}^\star$$

regular behaviour

$$\begin{aligned} istack(P) &= \langle\rangle \\ istack(P \mathbin{\&} \langle push(d), pop\rangle \mathbin{\&} X) &= d \triangleleft istack(P \mathbin{\&} X) \\ istack(P \mathbin{\&} \langle reset\rangle \mathbin{\&} X) &= istack(X) \end{aligned}$$

irregular behaviour

$$istack(pop \triangleleft X) = \ldots\ldots$$

- A sequence of $push$ commands generates no output. $(P \in push(\mathbb{N})^\star)$
- A pop command outputs the last datum not yet requested.
- After a $reset$ command, the stack is empty.
- How to react on an unexpected pop command?

3.5 Interactive Stack — Irregular Behaviour

A **fault tolerant stack** *ignores* an *illegal request* from the *environment* and *continues* to perform its *service* on future input commands:

$$istack(pop \triangleleft X) \;=\; istack(X)$$

A **fault sensitive stack** *breaks* after an illegal request and *provides no output* whatever further input arrives. The output stems from the *longest regular prefix* of the irregular input history.

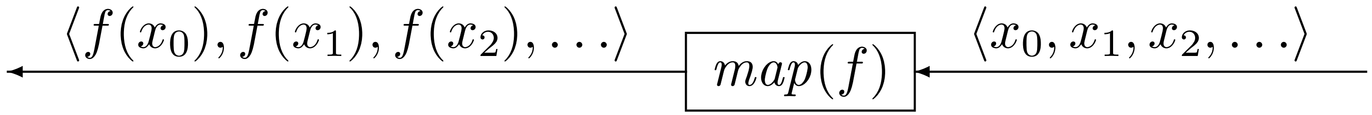
$$istack(pop \triangleleft X) \;=\; \langle\rangle$$

Further irregular behaviours: *Output a constant* or *suspend requests*.

Assumption/Guarantee Style

The *implementation* of the component *guarantees* the *specified behaviour* (service), only if the input streams *validate* the *assumptions* on the *environment*.

3.6 Iterator Component



An **iterator component** repeatedly applies a base function to all elements of the input stream.

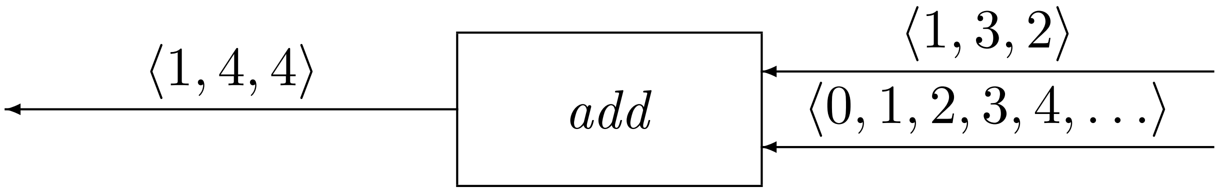
$$map : [\mathcal{A} \to \mathcal{B}] \to [\mathcal{A}^\star \to \mathcal{B}^\star]$$

$$\begin{aligned} map(f)(\langle\rangle) &= \langle\rangle \\ map(f)(x \lhd X) &= f(x) \lhd map(f)(X) \end{aligned}$$

The iterator component is *input/output synchronous* and *history insensitive*:

$$\begin{aligned} |map(f)(X)| &= |X| \\ map(f)(X)[i] &= f(X[i]) \qquad (0 \leq i \leq |X| - 1) \end{aligned}$$

3.7 Adder Component



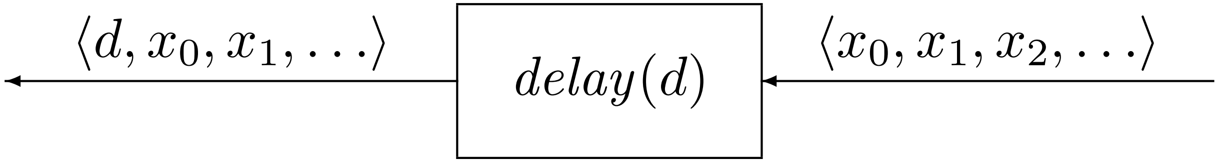
An **adder component** repeatedly calculates the sum of each two natural numbers arriving on the two input channels:

$add : \mathbb{N}^\star \times \mathbb{N}^\star \to \mathbb{N}^\star$			
$\lvert add(X,Y)\rvert$	$=$	$\min(\lvert X\rvert, \lvert Y\rvert)$	
$add(X,Y)[i]$	$=$	$X[i] + Y[i]$	$(0 \leq i \leq \lvert add(X,Y)\rvert - 1)$

The adder component is *strict* (reactive) in both arguments:

$$add(\langle\rangle, Y) \; = \; \langle\rangle \; = \; add(X, \langle\rangle)$$

3.8 Delay Component



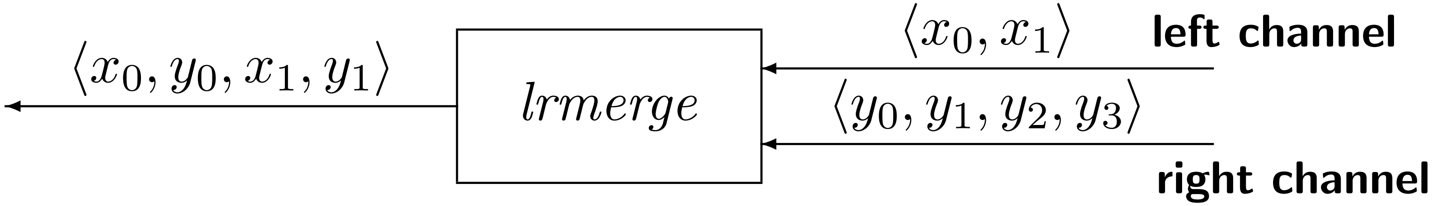
A **delay component** prefixes the input stream with an element:

$delay : \mathcal{A} \to [\mathcal{A}^\star \to \mathcal{A}^\star]$		
$delay(d)(\langle x_0, x_1, \ldots, x_m \rangle)$	$=$	$\langle d, x_0, x_1, \ldots, x_{m-1} \rangle$

The delay component is *input/output synchronous* and *history sensitive*:

$$\begin{aligned} |delay(d)(X)| &= |X| \\ delay(d)(X)[0] &= d \qquad (|X| \geq 1) \\ delay(d)(X)[i+1] &= X[i] \quad (0 \leq i < |X| - 1) \end{aligned}$$

3.9 Deterministic Merge Component



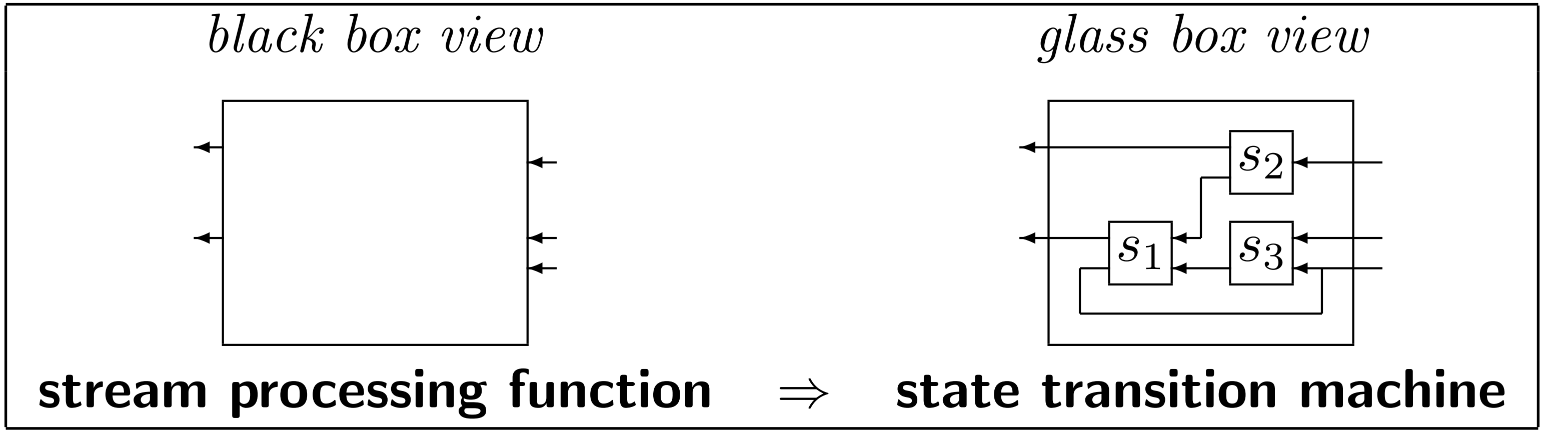
A **deterministic merge component** merges two communication streams in the following way: It first takes a message from the left channel, then a message from the right channel, and so on . . .

$$lrmerge : \mathcal{A}^\star \times \mathcal{B}^\star \to (\mathcal{A} \cup \mathcal{B})^\star$$

$$\begin{aligned} lrmerge(\langle\rangle, Y) &= \langle\rangle \\ lrmerge(x \triangleleft X, Y) &= x \triangleleft lrmerge(Y, X) \end{aligned}$$

The deterministic merge component is *history sensitive*. The *control state* must record the channel of the previous resp. the next message, the *data state* the elements stored.

4. State-Based Description: Glass-box View



The component is described by a *(collection of)* **communicating state transition machines with input and output**.

4.1 State Transition Machines

Constituents of the machine $M = (\mathcal{Q}, \mathcal{I}, \mathcal{O}, \delta, \varphi, q_0)$

set $\mathcal{Q}$ of *states*	one-step *state transition function*	$\delta : \mathcal{Q} \times \mathcal{I} \to \mathcal{Q}$
set $\mathcal{I}$ of *input data*	one-step *output function*	$\varphi : \mathcal{Q} \times \mathcal{I} \to \mathcal{O}^\star$
set $\mathcal{O}$ of *output data*	*initial state*	$q_0 \in \mathcal{Q}$

Processing input streams

multi-step *state transition function* $\delta^\star : \mathcal{Q} \to [\mathcal{I}^\star \to \mathcal{Q}]$

multi-step *output function* $\varphi^\star : \mathcal{Q} \to [\mathcal{I}^\star \to \mathcal{O}^\star]$

The multi-step output function $\varphi^\star(q)$ is a stream transformer!

4.2 Interactive Stack — State Transition Machine

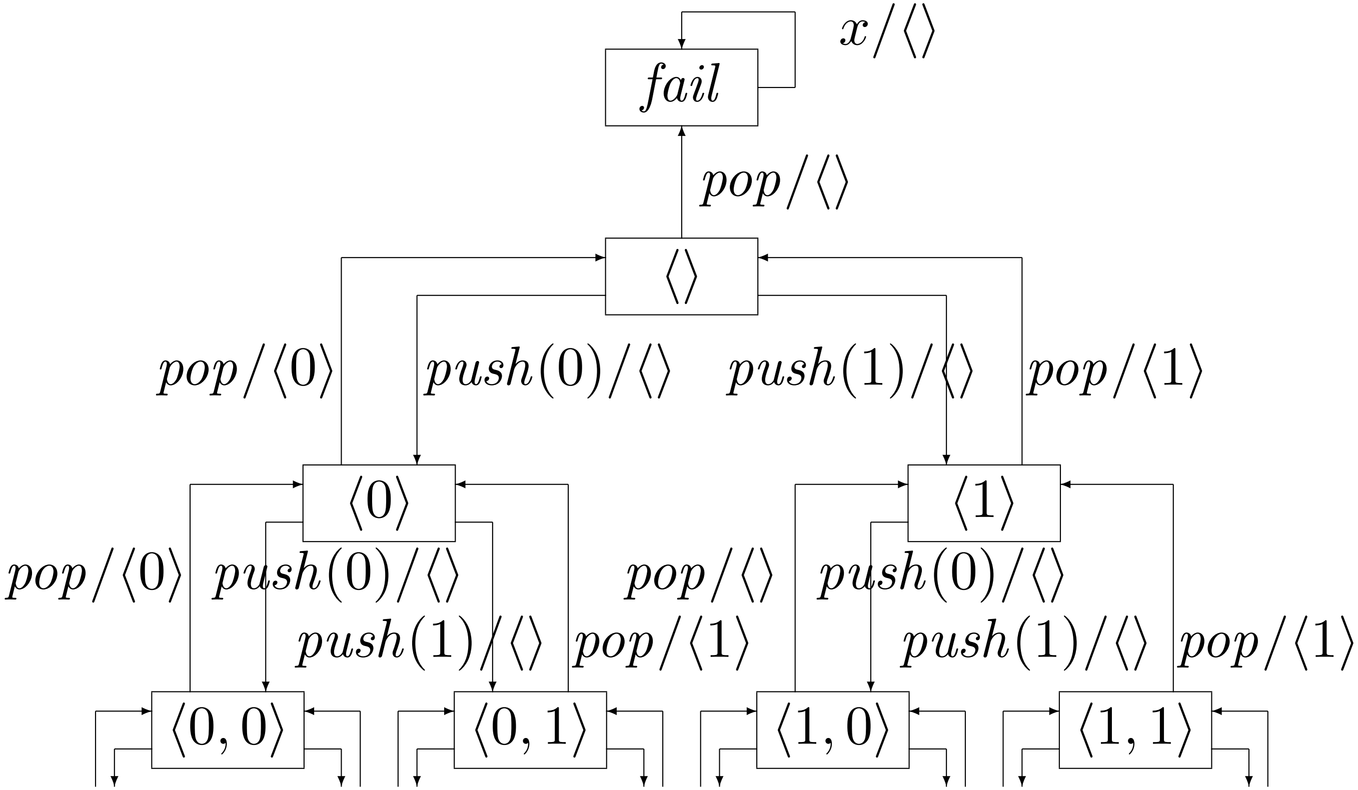
state space	$\mathcal{Q} = \mathbb{N}^\star \cup \{fail\}$
state transition function	$\delta(fail, x) = fail$ $\delta(Q, push(d)) = Q \,\&\, \langle d \rangle$ $\delta(Q, reset) = \langle\rangle$ $\delta(\langle\rangle, pop) = fail$ $\delta(Q \,\&\, \langle q \rangle, pop) = Q$
output function	$\varphi(S, push(d)) = \langle\rangle$ $\varphi(S, reset) = \langle\rangle$ $\varphi(fail, pop) = \langle\rangle$ $\varphi(\langle\rangle, pop) = \langle\rangle$ $\varphi(Q \,\&\, \langle q \rangle, pop) = \langle q \rangle$
initial state	$q_0 = \langle\rangle$

4.3 Interactive Stack — State Transition Table

Data State	Control State	Input	Data State'	Control State'	*Output*
	$fail$	x		$fail$	$\langle\rangle$
Q		$push(d)$	$Q \,\&\, \langle d\rangle$		$\langle\rangle$
Q		$reset$	$\langle\rangle$		$\langle\rangle$
$\langle\rangle$		pop		$fail$	$\langle\rangle$
$Q \,\&\, \langle q\rangle$		pop	Q		$\langle q\rangle$

A **state transition table** describes an *infinite state transition system* by a *finite number* of **transition rules**.

4.4 Interactive Stack — State Transition Diagram



Infinite double-linked tree with additional trap state (apart from $reset$ arcs)

5. Differentiation and Abstraction

black box view | *glass box view*

stream processing function $\Rightarrow$ **state transition machine**

Differentiation *summary description* $\mapsto$ *incremental description*.

$f(X)$	$\mathit{diff}(f)(X, x)$
$f(X \,\&\, \langle x \rangle)$	

Differentiation

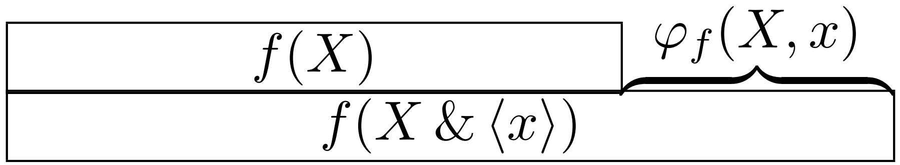
$$\mathit{diff} : [\mathcal{A}^\star \to \mathcal{B}^\star] \to [\mathcal{A}^\star \times \mathcal{A} \to \mathcal{B}^\star]$$

$$\mathit{diff}(f)(X, x) \quad = \quad f(X \,\&\, \langle x \rangle) \ominus f(X)$$

5.1 Canonical State Transition Machine

stream transformer	$\rightarrow$	**canonical state transition machine**
$f : \mathcal{A}^\star \rightarrow \mathcal{B}^\star$	$\rightarrow$	$M_f = (\mathcal{A}^\star, \mathcal{A}, \mathcal{B}, \delta, \varphi_f, \langle\rangle)$

states	input histories	$\mathcal{Q}$	$=$	$\mathcal{A}^\star$
state transition function	extend input history	$\delta(X, x)$	$=$	$X \,\&\, \langle x \rangle$
output function	incremental output	$\varphi_f(X, x)$	$=$	$\mathit{diff}(f)(X, x)$
initial state	empty input history	q_0	$=$	$\langle\rangle$



Only the *output function* depends on the particular *stream transformer* f .

Walter Dosch
Institute of Software Technology and Programming Languages
University of Lübeck, Germany

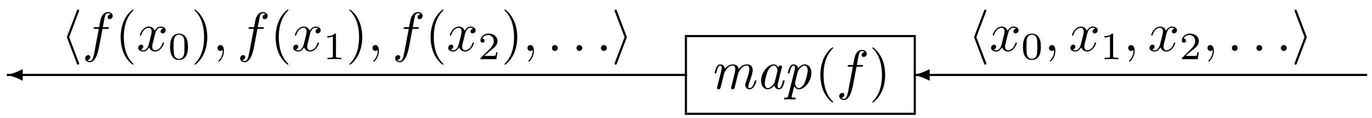
5.2. Abstraction

A **(state) abstraction function** *identifies* some states of state transition machine without changing the multi-step output function.

An *abstraction function* $abstr : \mathcal{Q} \to \mathcal{Q}'$ for a state transition machine $M = (\mathcal{Q}, \mathcal{I}, \mathcal{O}, \delta, \varphi, q_0)$ is **transition closed** and **output compatible**

reduced state transition machine $M' = (\mathcal{Q}', \mathcal{I}, \mathcal{O}, \delta', \varphi', q_0')$	
set of states	Q'
state transition function	$\delta'(abstr(q), x) = abstr(\delta(q, x))$
output function	$\varphi'(abstr(q), x) = \varphi(q, x)$
initial state	$q_0' = abstr(q_0)$

5.3 Iterator Component



An **iterator component** repeatedly applies a base function to all elements of the input stream.

Differential Description

$$\mathit{diff}(map(f))(X, x) \quad = \quad \langle f(x) \rangle$$

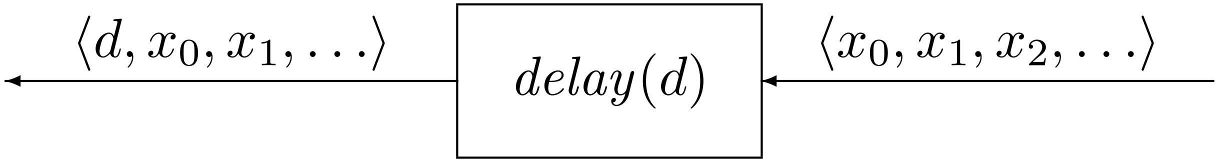
Abstraction

$$abstr : \mathcal{I}^{\star} \to \{q_0\}$$

$$abstr(X) \quad = \quad q_0$$

Component is *history insensitive.* ⇔ Implementation is *state free.*

5.4 Delay Component



A **delay component** prefixes the input stream with an element.

Differential Description

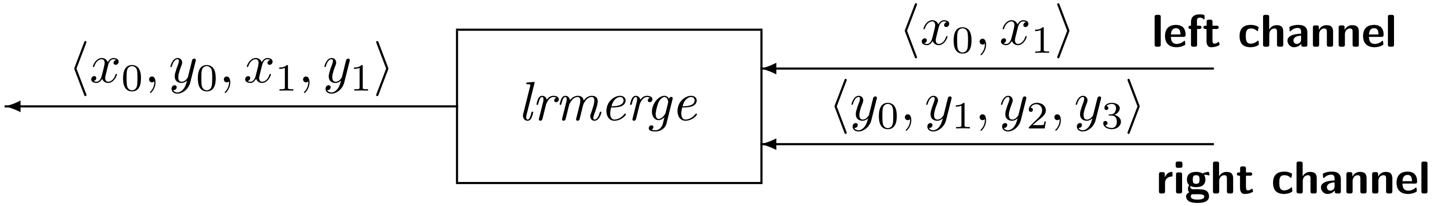
$$
\begin{array}{rcl}
\mathit{diff}\,(\mathit{delay}(d))(\langle\rangle, y) & = & \langle d\rangle \\
\mathit{diff}\,(\mathit{delay}(d))(X \,\&\, \langle x\rangle, y) & = & \langle x\rangle
\end{array}
$$

Abstraction

$$
\begin{array}{|rcl|}
\hline
\multicolumn{3}{|c|}{\mathit{abstr} : \mathcal{A}^{\star} \to \mathcal{A}} \\
\hline
\mathit{abstr}(\langle\rangle) & = & d \\
\mathit{abstr}(X \,\&\, \langle x\rangle) & = & x \\
\hline
\end{array}
$$

Component is *history sensitive* — the state records the *last input*.

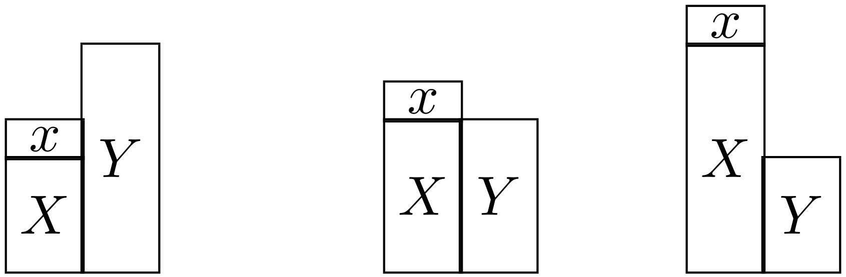
5.5 Deterministic Merge Component



The **deterministic merge component** merges two communication streams.

$$diff_1(lrmerge)(X,Y)(x) = \begin{cases} \langle x, Y[|X|] \rangle & \text{if} \quad |X| < |Y| \\ \langle x \rangle & \text{if} \quad |X| = |Y| \\ \langle \rangle & \text{if} \quad |X| > |Y| \end{cases}$$

Illustration



Walter Dosch
Institute of Software Technology and Programming Languages
University of Lübeck, Germany

State Abstraction

$$\mathcal{Q} \quad = \quad \underbrace{\{left, right\}}_{Control} \times \underbrace{(\mathcal{A}^\star \cup \mathcal{B}^\star)}_{Buffer}$$

State Transition Table

$Control$	$Buffer$	$Inleft$	$Inright$	$Control'$	$Buffer'$	Out
$left$	$\langle\rangle$	a	$-$	$right$	$\langle\rangle$	$\langle a\rangle$
$left$	$\langle r\rangle \,\&\, R$	a	$-$	$left$	R	$\langle a, r\rangle$
$right$	L	a	$-$	$right$	$L \,\&\, \langle a\rangle$	$\langle\rangle$
$left$	R	$-$	b	$left$	$R \,\&\, \langle b\rangle$	$\langle\rangle$
$right$	$\langle\rangle$	$-$	b	$left$	$\langle\rangle$	$\langle b\rangle$
$right$	$\langle l\rangle \,\&\, L$	$-$	b	$right$	L	$\langle b, l\rangle$

States are history abstractions.

5. Trace-Based Description: Process View

A **trace** records the *sequence* of *events* occurring during a *run* of the system.

Classification of Events

input event	$?i$	(receiving a command i on the input channel)
output event	$!o$	(sending a message o on the output channel)
internal event	$q \mapsto q'$	(updating the internal state from q to q')

Set of Events

$$\mathcal{F} = \underbrace{(?\mathcal{I})}_{\text{input events}} \cup \underbrace{(!\mathcal{O})}_{\text{output events}} \cup \underbrace{(\mathcal{Q} \mapsto \mathcal{Q}')}_{\text{internal events}}$$

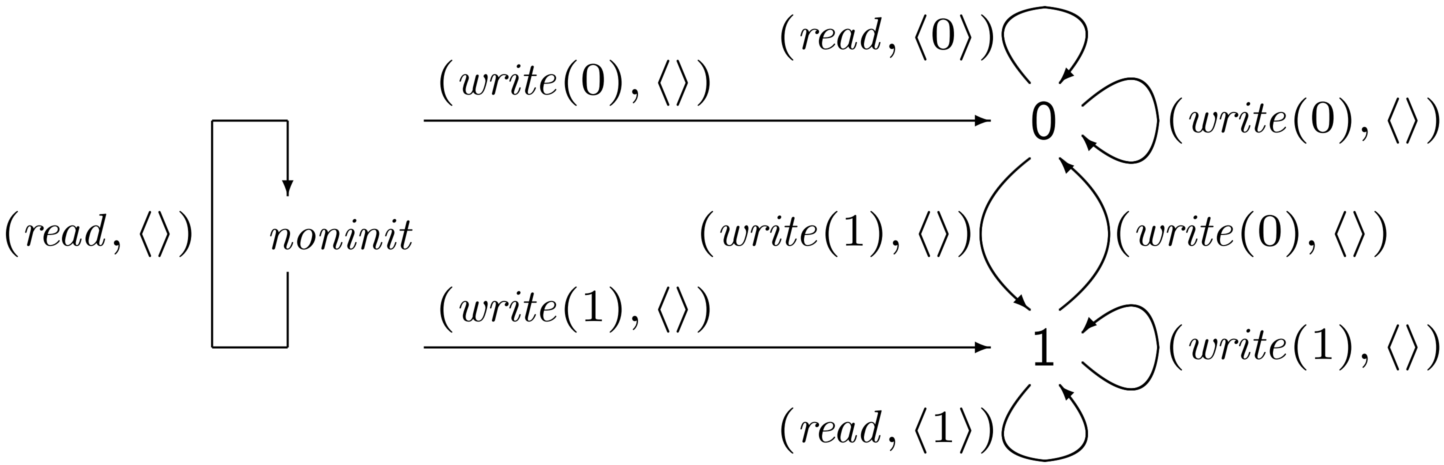
Input Traces $\quad trace : \mathcal{Q} \to [\mathcal{I}^\star \to \mathcal{F}^\star]$

Trace Behaviour

$traces : \mathcal{Q} \to \mathcal{P}(\mathcal{F}^\star)$
$trace(q) = \{trace(q)(X) \mid X \in \mathcal{I}^\star\}$

Example: Binary Memory Cell

State transition diagram for a memory cell storing a binary data type $\{0, 1\}$



The memory cell is described by the **set of possible traces**:

$$traces : \mathcal{Q} \to \mathcal{P}(\mathcal{F}^\star)$$

$$
\begin{aligned}
traces(noninit) \quad &= \quad \{\langle\rangle\} \\
&\cup \quad \{?read\} \lhd traces(noninit) \\
&\cup \quad \bigcup_{e \in \mathcal{D}} \{?write(e)\} \lhd \{(noninit \mapsto e)\} \lhd traces(e)
\end{aligned}
$$

$$
\begin{aligned}
traces(d) \quad &= \quad \{\langle\rangle\} \\
&\cup \quad \{?read\} \lhd \{!d\} \lhd traces(d) \\
&\cup \quad \bigcup_{e \in \mathcal{D}} \{?write(e)\} \lhd \{(d \mapsto e)\} \lhd traces(e)
\end{aligned}
$$

KLEENE's fixpoint theorem establishes an *approximating chain* $(d \in \mathcal{D})$:

$$
\begin{aligned}
traces^{(0)}(d) &= \emptyset \\
traces^{(1)}(d) &= \{\langle\rangle\} \\
traces^{(2)}(d) &= \{\langle\rangle, \langle ?read, !d\rangle\} \cup \\
&\quad \{\langle ?write(e), d \mapsto e\rangle \mid e \in \mathcal{D}\} \\
traces^{(3)}(d) &= \{\langle\rangle, \langle ?read, !d\rangle, \langle ?read, !d, ?read, !d\rangle\} \cup \\
&\quad \{\langle ?read, !d, ?write(e), d \mapsto e\rangle, \\
&\quad\ \langle ?write(e), d \mapsto e\rangle, \\
&\quad\ \langle ?write(e), d \mapsto e, ?read, !e\rangle, \\
&\quad\ \langle ?write(e), d \mapsto e, ?write(f), e \mapsto f\rangle \mid e, f \in \mathcal{D}\} \\
&\ \ \vdots
\end{aligned}
$$

7. Conclusion

Data Structure	+	Communication-Based Description
Data Model	Interaction interface Interaction behaviour	*Black-box View*

Communication-Based Description	+	State-Based Description
Black-box View	Differentiation Abstraction	*Glass-box View*

State-Based Description	+	Trace-Based Description
Glass-box View	Events Traces	*Process View*

Walter Dosch
Institute of Software Technology and Programming Languages
University of Lübeck, Germany

Applications

↕

Tools

↕

System Development Process

↕

Description Techniques

↕

→ System Model ←

Transforming System Views in Software Design

↕

System Model Theory

Component-Based Systems		
layout	*static*	dynamic
	topological	metric
communication	synchronous	*asynchronous*
	unidirectional	bidirectional
state	*state-full*	*state-less*
	continuous	*discrete*
	simple	*structured*
	shared	*distributed*
time	timed	*untimed*
	continuous	discrete
	sensitive	invariant
control	*(non)deterministic*	stochastic
	centralized	*distributed*
	event-driven	time-driven

Thanks for your Attention!

Any Questions, Amendments, Comments . . . ?