Modeling Software Systems

WALTER DOSCH
Institute of Software Technology and Programming Languages
University of Lübeck
Lübeck, Germany
http://www.isp.uni-luebeck.de

Outline

1. **Introduction**
2. Modeling
3. Data Models
4. Interaction Models
5. State-Based Models
6. Architectural Models
7. Model Transformation
[8. Programming Language Models]
9. Classification of System Models

1.1 Current Trends in Information Technology

- **Hardware performance increases:**
  computing power — storage capacity — transmission rate
- Growing **functionality** and **complexity**
- Worldwide **connectivity** — *networked infrastructures*
- **Interoperability** and **dependability** — *safety and security*
- **Industrial standards** — *scientific progress*
- **Reusabilty** of *products*, *artefacts* and *processes*
- Software is **embedded** into **engineering products**.
- Emerging **discipline of software engineering**
  specification — modeling — design — implementation — (re)use
- **Modeling software systems**
  structure — properties — behaviour . . . $\leftrightarrow$ abstraction

1.2 Layers of Software Technology

Applications

↕

Tools

↕

System Development Process

↕

Description Techniques

↕

$\rightarrow$ ***System*** $\leftarrow$

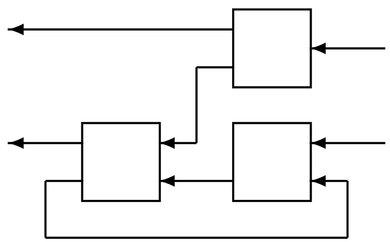
↕

System Theory

1.3 Modeling in Software Technology

Applications	**Domain Model**
↕	
Tools	
↕	
System Development Process	**Process Model**
↕	
Description Techniques	
↕	
System	**System Model**
↕	
System Theory	

1.4 System Model: Networks of Components



- **Complex systems** are networks of *software/hardware components*.
- Systems are **hierarchically composed** of (elementary) components.
- Components **cooperate** and **interact** by *exchanging information*.
- Components **communicate** sending *messages* on *directed channels*.
- **Specifications** define the *interface* and the *input/output behaviour*.

1.5 Application Areas

Service-oriented Classification

- Processing units
- Memory components
- Transmission components
- Synchronization and control components

Application-oriented Classification

- Process control
- Man-machine interaction
- Telecommunication
- Distributed computation

Outline

1. Introduction
2. **Modeling**
3. Data Models
4. Interaction Models
5. State-Based Models
6. Architectural Models
7. Model Transformation
8. Programming Language Models
9. Classification of System Models

2.1 Modeling — Overview

A **model** [lat. *modulus*] is an **artefact** which describes certain **aspects** of a **part** of the **world** by its **similarity** in **structure**, **function**, **behaviour** or **use**.

	real world	
modeling	$\downarrow$	*abstraction*
	model	

Four Areas	• **Pragmatics**	diagrams, plans, terminology, aspects
	• **Formalism**	preciseness, adequacy, expressiveness
	• **Methods**	modeling, validation, transformation
	• **Tools**	analysis, checking, visualisation, generation

2.2 Modeling — Areas

foundations
ontologies
model integration
generative models
model validation
standardization
model-driven architectures
modeling nonfunctional requirements
model-based refactoring
enterprise application model
models for distributed control
model animation
modeling techniques
logic-based models
model transformation
pattern for modeling
model verification
model consistency
model engineering
aspect-oriented modeling
model-based reengineering
business process model
models for safety and security
modeling tools
model simulation
model quality

2.3 Modeling Components — *Views*

Data Structure	Communication-Based Description	State-Based Description	Trace-Based Description
Data Model	*Interaction Model*	*State Model*	*Process Model*

UML

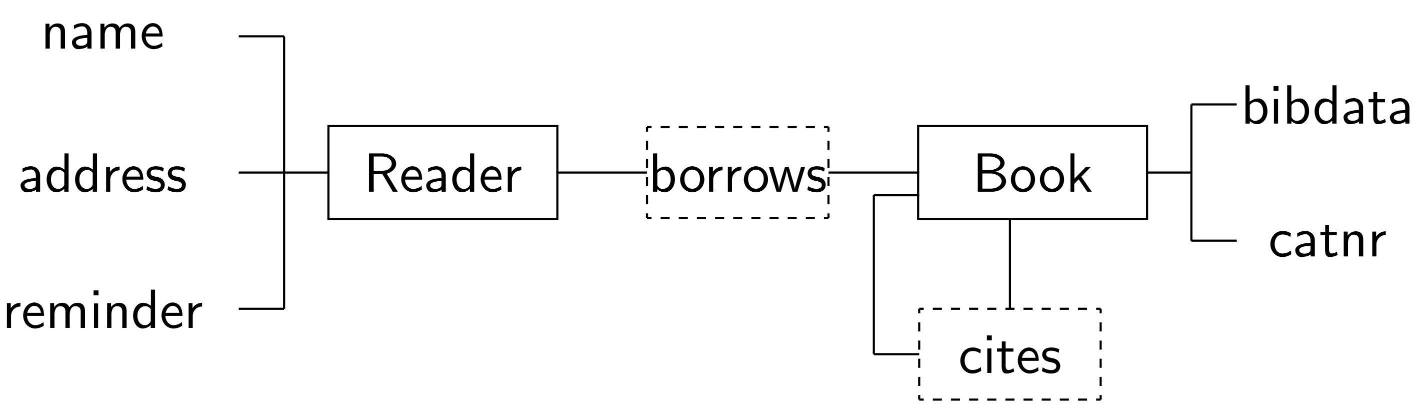
- *Class diagrams* ↔ data structure
- *State machines* ↔ state-based description
- *Sequence diagrams* ← communication-based description

Description Techniques

- Equational logic ↔ functional style
- Predicate logic ↔ relational style
- State transition systems ↔ tabular and graphic style

Outline

3.1 Data Model — Entity Relationship Modeling



- **Sets** `Reader`, `Book`
- **Attributes**

`name`	`Reader`	$\longrightarrow$	`String`
`address`:	`Reader`	$\longrightarrow$	`String`
`reminder`:	`Reader`	$\longrightarrow$	`Bool`
...			

- **Relations**

`borrows`	$\subseteq$	`Reader`$\times$`Book`
`cites`	$\subseteq$	`Book`$\times$`Book`

3.2 Algebraic Specification — *Stacks*

An **algebraic specification** defines a *data structure* by the **properties** of its basic **operations** in a *representation indepedent way*.

STACK

$$empty : \ \rightarrow stack$$
$$prefix : data \star stack \rightarrow stack$$
$$first : stack \rightarrow data$$
$$rest : stack \rightarrow stack$$

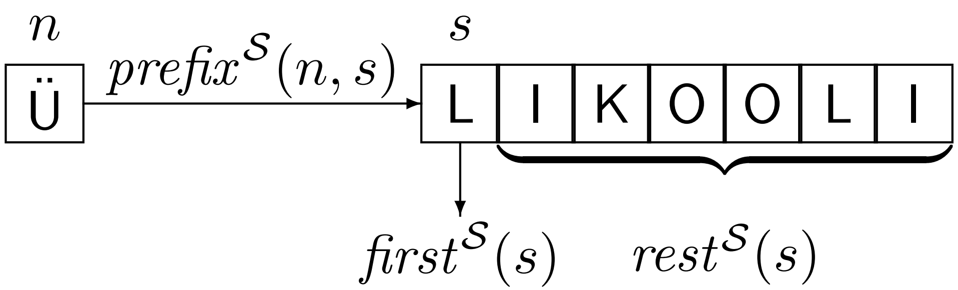
$$\begin{aligned} first(empty) &= \text{undefined} \\ first(prefix(d,s)) &= d \\ rest(empty) &= \text{undefined} \\ rest(prefix(d,s)) &= s \end{aligned}$$

abstract data type = [**signature** / *interface*] + [**axioms** / *behaviour*]

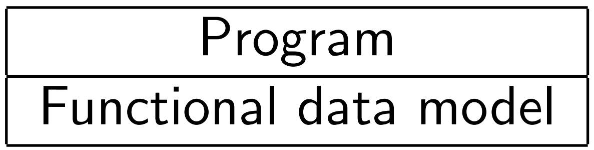
3.3 Functional Data Model — *Stacks based on Sequences*

A **functional data model** defines the **carrier sets**, **functions** and **relations** of a data structure.

Model $\mathcal{S}$

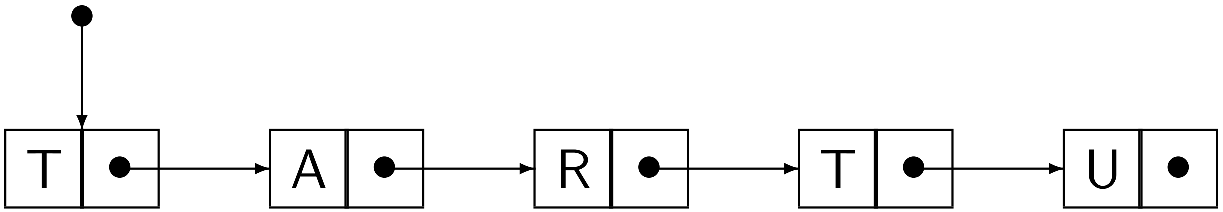


- **Algebraic methods** Equational & inductive reasoning
- **Vertical modularization**



3.4 Imperative Data Model — *Stacks as Linear Lists*

An **imperative data model** provides **variables** (*storage locations*) and **pointers** (*storage addresses*) to build **state-based implementations** along with **procedures** to manipulate them.



Benefits for systems programming

- **state** *selective updating*
- **pointer** *sharing, manipulation*
  cyclic structures
- **storage** *create — destroy*

Object-oriented data model

- **state encapsulation**
- **object identity**
- **life cycle**

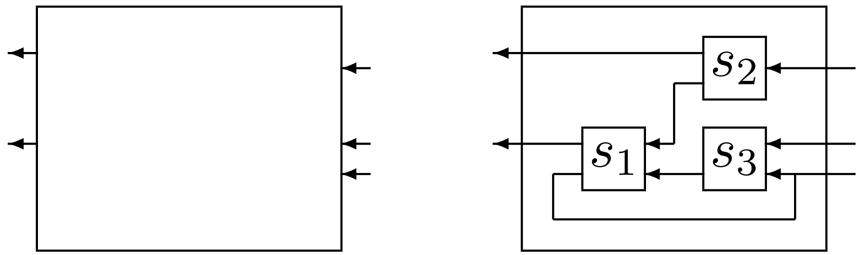
Outline

1. Introduction
2. Modeling
3. Data Models
4. **Interaction Models**
5. State-Based Models
6. Architectural Models
7. Model Transformation
8. Programming Language Models
9. Classification of System Models

4. Interaction Models

Top-down design of components

black box view	*glass box view*



communication	internal state

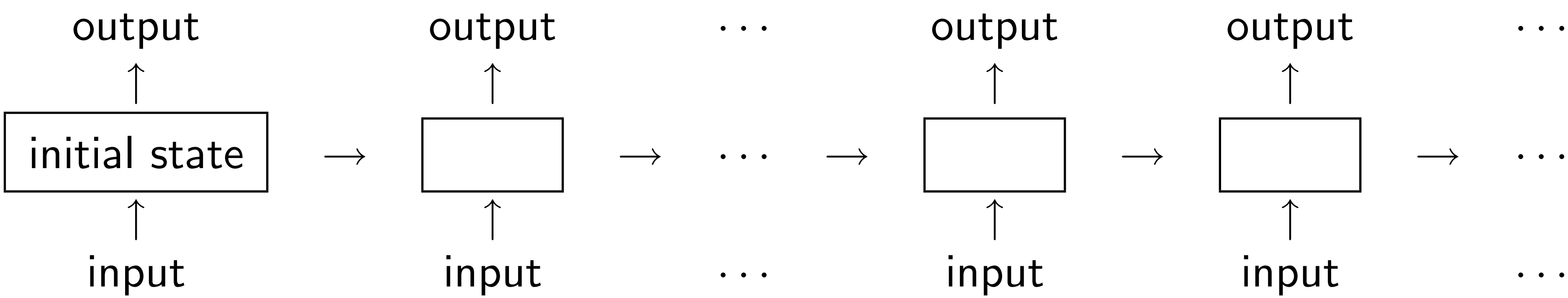
Black box view	Glass box view
Input/output behaviour	*Architecture*
Service, Composition	*Implementation*
Correctness, Properties	*Efficiency*

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

4.1 Models of Computation

Sequential Model of Computation

Interactive Model of Computation



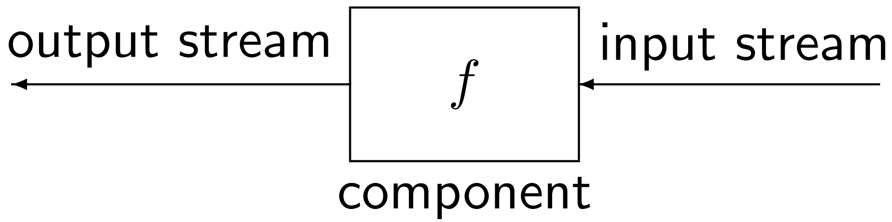
4.2 Streams

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

Finite streams $\mathcal{A}^\star = \{\langle x_0, \ldots, x_m \rangle \mid x_i \in \mathcal{A}\}$

Concatenation $\langle x_0, ..., x_m \rangle \;\&\; \langle y_0, ..., y_n \rangle = \langle x_0, ..., x_m, y_0, ..., y_n \rangle$

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

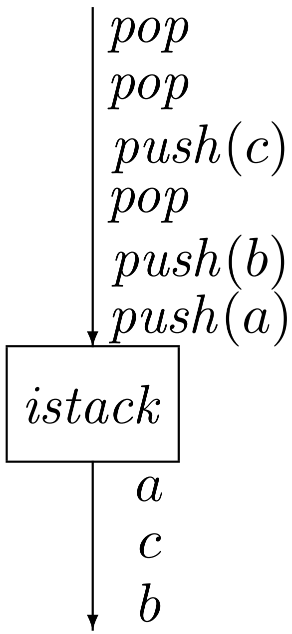


Stream transformer $f : \mathcal{A}^\star \rightarrow \mathcal{B}^\star$

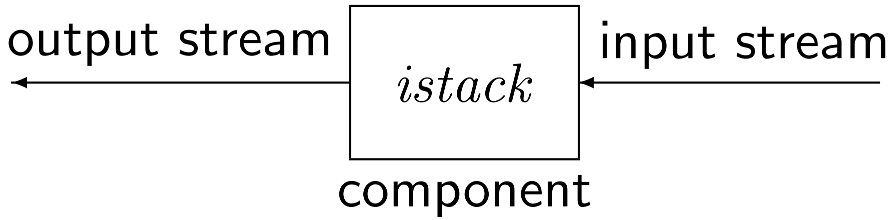
Monotonicity $X \sqsubseteq Y \Rightarrow f(X) \sqsubseteq f(Y)$

4.3 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.



4.4 Interactive Stack — *Regular Behaviour*



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

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

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

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

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

$$Push \in push(\mathcal{D})^\star$$

4.5 Interactive Stack — *Irregular Behaviour*

input history									output
$push(1)$	pop	**pop**	**pop**	$push(2)$	$push(3)$	$push(4)$	pop	$\ldots$	?

fault-sensitive stack

$push(1)$	pop								$\langle 1 \rangle$

fault-tolerant stack

$push(1)$	pop		$push(2)$	$push(3)$	$push(4)$	pop	$\ldots$	$\langle 1, 4 \ldots \rangle$

robust stack

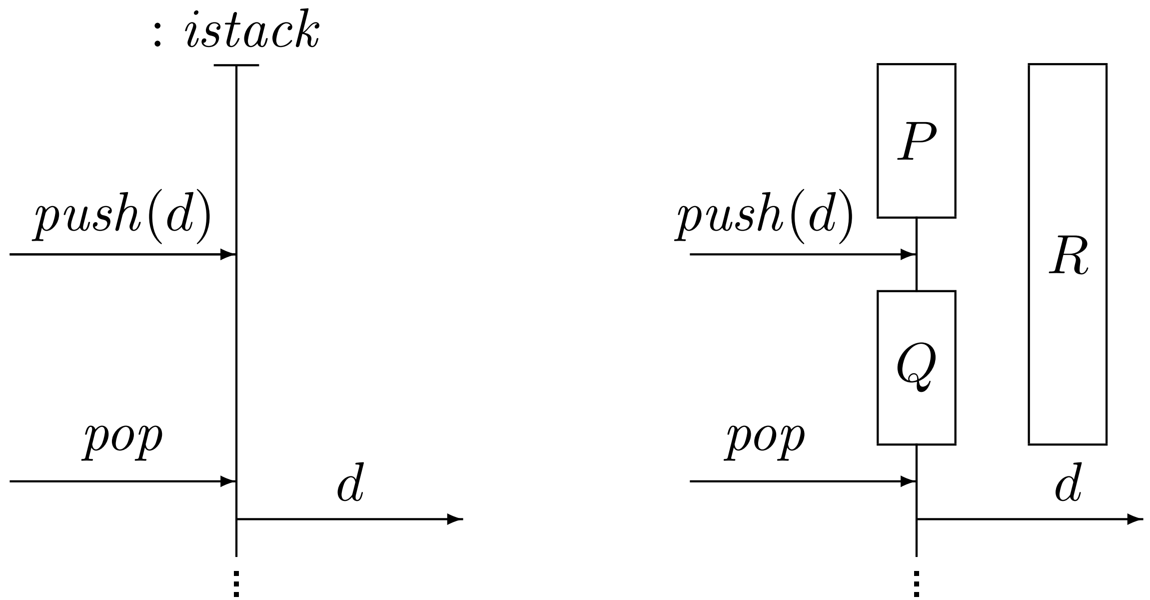
$push(1)$	pop	**pop**	**pop**	$push(2)$	$push(3)$	$push(4)$	pop	$\ldots$	$\langle 1, u, u, 4 \ldots \rangle$

fault-correcting stack

$push(1)$	pop	**pop**	**pop**	$push(2)$	$push(3)$	$push(4)$	pop	$\ldots$	$\langle 1, 2, 3, 4 \ldots \rangle$

A component *guarantees* a *service* only for input from the **service domain**.

4.6 Interactive Stack — *Sequence Diagram (Sample)*



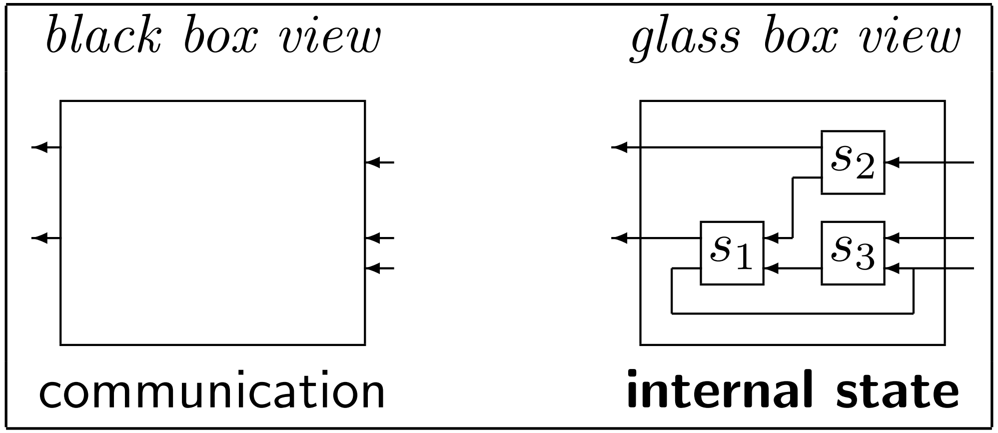
$$
\begin{array}{lrcl}
 & state(P \,\&\, \langle push(d) \rangle) & = & state(P \,\&\, \langle push(d) \rangle \,\&\, Q) \\
\wedge & istack(P \,\&\, \langle push(d) \rangle \,\&\, Q) & = & R \\
\Rightarrow & istack(P \,\&\, \langle push(d) \rangle \,\&\, Q \,\&\, \langle pop \rangle) & = & R \,\&\, \langle d \rangle
\end{array}
$$

diagram ↔ formalism

Outline

1. Introduction
2. Modeling
3. Data Models
4. Interaction Models
5. **State-Based Models**
6. Architectural Models
7. Model Transformation
8. Programming Language Models
9. Classification of System Models

5. State-Based Models

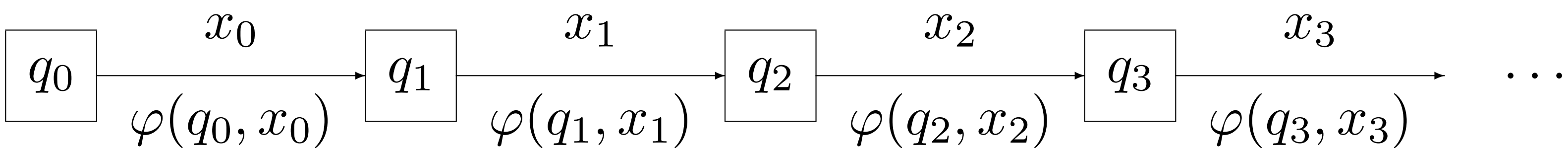


Black box view	**Glass box view**
Input/output behaviour	*Architecture*
Service	*Implementation*
Correctness	*Efficiency*

5.1 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!

5.1 State Transition Machine (cdt.) — *Interactive Stack*

$$M = (\mathcal{Q}, \mathcal{I}, \mathcal{O}, \delta, \varphi, \langle\rangle)$$

$$\begin{array}{rcl}
\mathcal{Q} & = & \mathcal{D}^\star \cup \{\mathit{fail}\} \\
\hline
\delta(\mathit{fail}, x) & = & \mathit{fail} \\
\delta(Q, \mathit{push}(d)) & = & Q \,\&\, \langle d \rangle \\
\delta(\langle\rangle, \mathit{pop}) & = & \mathit{fail} \\
\delta(Q \,\&\, \langle d \rangle, \mathit{pop}) & = & Q \\
\hline
\varphi(q, \mathit{push}(d)) & = & \langle\rangle \\
\varphi(\mathit{fail}, \mathit{pop}) & = & \langle\rangle \\
\varphi(\langle\rangle, \mathit{pop}) & = & \langle\rangle \\
\varphi(Q \,\&\, \langle d \rangle, \mathit{pop}) & = & \langle d \rangle
\end{array}$$

M **implements** the interactive stack: $\varphi^\star(\langle\rangle)(X) \;=\; \mathit{istack}(X)$

5.2 State Transition Tables

A **state transition table** displays the *different transition rules* in a clear way:

state	input	state'	output
q	x	$\delta(q,x)$	$\varphi(q,x)$

The **transition rules** relate *current states* and *inputs* to *successor states* (denoted by a prime) and *outputs*.

The *transition rules* **tabulate** the *transition functions* δ and φ for all *possible combinations*.

Interactive Stack

state	input	state'	output
$fail$	x	$fail$	$\langle\rangle$
Q	$push(d)$	$Q\,\&\,\langle d\rangle$	$\langle\rangle$
$\langle\rangle$	pop	$fail$	$\langle\rangle$
$Q\,\&\,\langle d\rangle$	pop	Q	$\langle d\rangle$

5.3 State Transition Diagrams

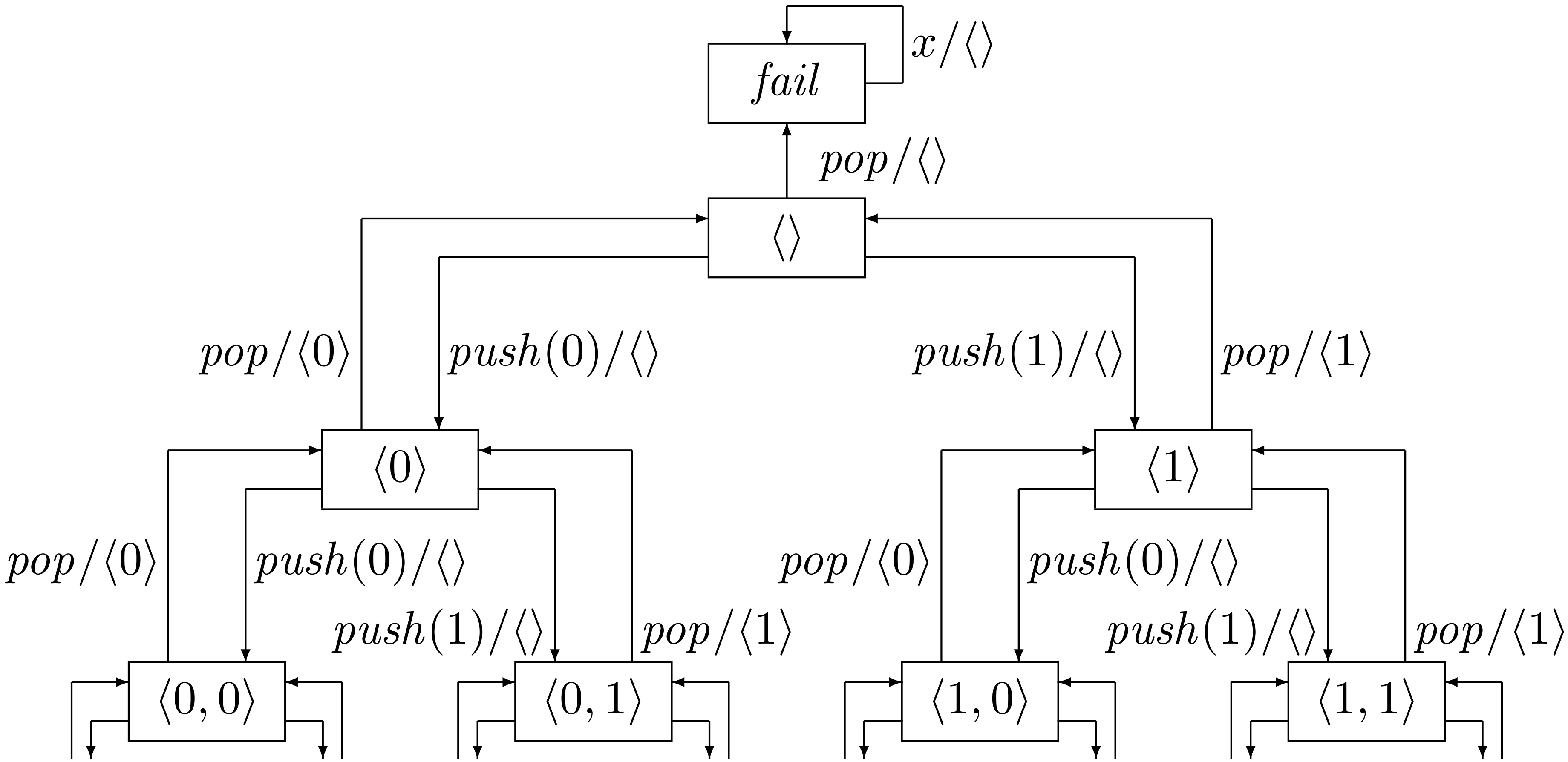
State transition machines can be considered as **labelled directed graphs**:

graph		state transition machine
nodes	$\mathcal{Q}$	states
vertices	$q \rightarrow \delta(q, x)$	transitions
labels	$x \,/\, \varphi(q, x)$	(input,output)

The finite *symbolic state transition diagram* can be **expanded** into an infinite *state transition diagram* by **instantiating** the four *variables*

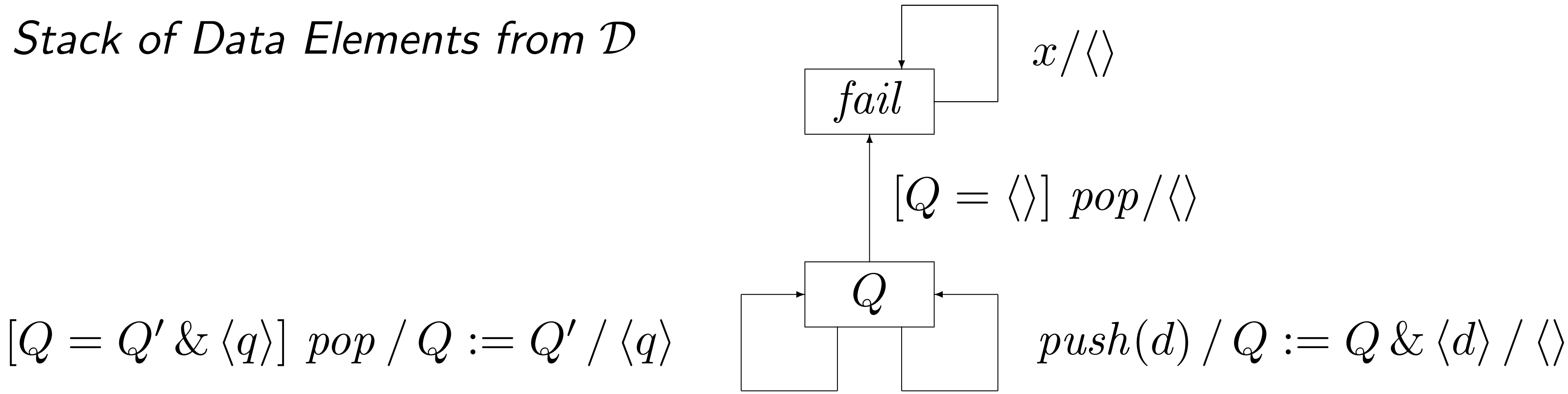
$$Q, Q' \in \mathcal{D}^\star \qquad d \in \mathcal{D} \qquad x \in \mathcal{I} \; .$$

State Transition Diagram (part) for $\mathcal{D} = \{0, 1\}$



5.4 Symbolic State Transition Diagrams

Stack of Data Elements from $\mathcal{D}$



This **representation** identifies all $regular\ states$ (Q) .

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

Outline

6. Architectural Models

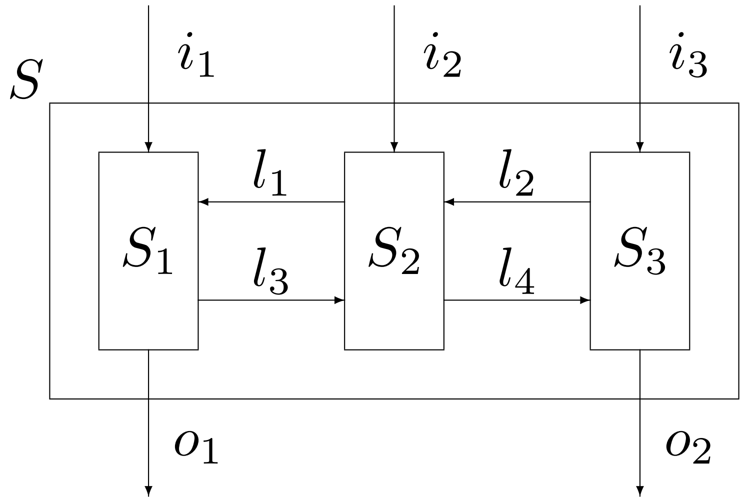
Composite systems are *hierarchically composed* of *elementary components* or *subcomponents*.

Composite systems can be constructed using a **graphical style** or by **composition operators**.

The **architecture** describes the *static structure* of *composite systems*.

The **behaviour** of a *composite system* must be inferred from the *behaviours* of its *components*.

6.1 Composition in Graphical Style

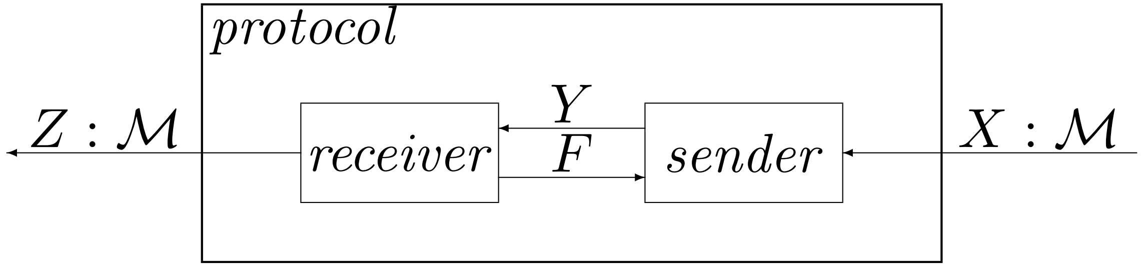


$$S : \mathcal{A}_1^\infty \times \mathcal{A}_2^\infty \times \mathcal{A}_3^\infty \to \mathcal{B}_1^\infty \times \mathcal{B}_2^\infty$$

$$\begin{aligned} S(i_1, i_2, i_3) &= (o_1, o_2) \\ S_1(i_1, l_1) &= (o_1, l_3) \\ S_2(i_2, l_2, l_3) &= (l_1, l_4) \\ S_3(l_4, i_3) &= (l_2, o_2) \end{aligned}$$

The composite system is described by a *collection of equations* (net list) using *named input*, *output* and *internal channels* based on the descriptions of the subcomponents.

6.2 Protocol Architecture

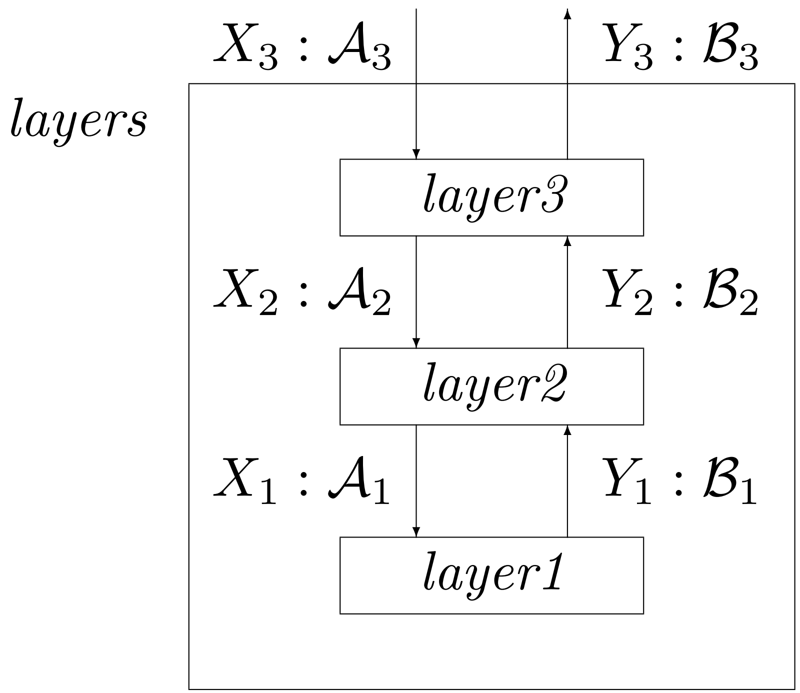


$protocol : \mathcal{M}^\star \to \mathcal{M}^\star$
$protocol(X) = Z$
$sender(F, X) = Y$
$receiver(Y) = (Z, F)$

Correctness

The equation $Z = X$ specifies that no messages are *lost*, *reordered*, *duplicated* or *faked*.

6.3 Layered Architecture



Outline

1. Introduction
2. Modeling
3. Data Models
4. Interaction Models
5. State-Based Models
6. Architectural Models
7. **Model Transformation**
8. Programming Language Models
9. Classification of System Models

7.1 Compilation — From Syntax to Code

	sequence of characters	
lexical analysis	↓	scanner
	sequence of symbols	
syntactic analysis	↓	parser
	syntax tree	
semantic analysis	↓	attribute evaluator
	attributed syntax tree	
code generation	↓	code generator
	sequence of commands	
code optimization	↓	code optimizer
	sequence of commands	

7.2 Data Models — From Sig's to Interaction Interfaces

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

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

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

Transformation *Signature* $\mapsto$ *Interaction Interface*

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

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

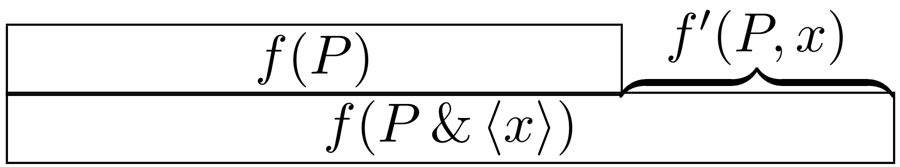
$$\frac{empty : \ \to stack}{reset : \underline{stack} \to \underline{stack}}$$

Design Decisions: **Encapsulation** $(\underline{stack})$ and **input/output parameters**.

7.3 Data Models — From Streams to States

stream transformer	+	*history abstraction*	$\rightarrow$	**state transition machine**
$f : \mathcal{A}^\star \rightarrow \mathcal{B}^\star$	+	$abstr : \mathcal{A}^\star \rightarrow \mathcal{Q}$	$\rightarrow$	$M[f, abstr] = (\mathcal{Q}, \mathcal{A}, \mathcal{B}, \delta, \varphi, q_0)$

states	abstractions of input histories	$\mathcal{Q}$	$=$	$abstr(\mathcal{A}^\star)$
state transition function	extend input history	$\delta(abstr(P), x)$	$=$	$abstr(P \,\&\, \langle x \rangle)$
output function	incremental output	$\varphi(abstr(P), x)$	$=$	$f'(P, x)$
initial state	empty input history	q_0	$=$	$abstr(\langle\rangle)$



States are history abstractions.

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

7.4 Model Refinement

Model refinement is a fundamental notion for *development* of *systems* from *abstract specifications* to *concrete implementations*.

The *refinement steps* may exploit different **development methods** (*transformation*, *generation*, *verification*) for the system under construction.

- **Behavioural Refinement**
- **Interface Refinement**
- **Architectural Refinement**
- **Communication Refinement**
- **State Refinement**
- **Data Refinement**

Outline

8.1 Models for Programming Languages — *Disjunction*

The **denotational semantics** provides **models** for *programming language constructs*.

strict disjunction			
or	$\perp$	T	F
$\perp$	$\perp$	$\perp$	$\perp$
T	$\perp$	T	T
F	$\perp$	T	F

left-right disjunction			
lor	$\perp$	T	F
$\perp$	$\perp$	$\perp$	$\perp$
T	**T**	T	T
F	$\perp$	T	F

parallel disjunction			
por	$\perp$	T	F
$\perp$	$\perp$	**T**	$\perp$
T	**T**	T	T
$\perp$	$\perp$	T	F

Programming languages implement a **three-valued logic**: the result of a Boolean expression is *true*, *false* or *undefined* .

8.2 Models for Schedules — *Fairness and Liveness*

Model the **schedule** for the **nondeterministic unboundedly fair interleaving** of two **infinite processes**.

Generalized regular expession $(0^{+} \cdot 1 + 1^{+} \cdot 0)^{\omega}$
Grouping an infinite stream of Boolean values:

$$\langle \underbrace{0,0,0,0,0,1}_{\in 0^{+} \cdot 1}, \underbrace{1,1,1,0}_{\in 1^{+} \cdot 0}, \underbrace{0,0,0,0,0,1}_{\in 0^{+} \cdot 1}, \underbrace{0,0,0,0,0,0,0,1}_{\in 0^{+} \cdot 1}, \underbrace{1,1,1,1,0}_{\in 1^{+} \cdot 0}, \ldots \rangle$$

<table>
<tr><td>$(0^{+} \cdot 1$</td><td>$+$</td><td>$1^{+} \cdot 0)$</td><td>ω</td></tr>
<tr><td>fairness</td><td>nondeterministic choice</td><td>fairness</td><td rowspan="2"></td></tr>
<tr><td colspan="3">liveness</td></tr>
</table>

Bounded Fairness $(0^{1..m} \cdot 1 + 1^{1..m} \cdot 0)^{\omega}$ $(m \geq 1)$

Walter Dosch
Institute of Software Technology and Programming Languages
University of Lübeck, Germany
Academica X
Tartu, Estonia
September 26, 2006

Outline

1. Introduction
2. Modeling
3. Data Models
4. Interaction Models
5. State-Based Models
6. Architectural Models
7. Model Transformation
8. Programming Language Models
9. **Classification of System Models**

9. Classification of System Models

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

programming = modeling on code level

What are **unifying higher-level models** for
software and system engineering
to cope with
⋮
architectures, *components*
interfaces, *services*
⋮
on different
levels of abstraction
integrating different
system views?

Acknowledgements ISP team, in particular A. Stümpel and B. Dölle