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**

- **Reusability 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 ... ↔ abstraction*
1.2 Layers of Software Technology

Applications

Tools

System Development Process

Description Techniques

→ System ←

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.

\[
\begin{array}{ccc}
\text{real world} & \downarrow & \text{abstraction} \\
\text{modeling} & \quad & \text{model}
\end{array}
\]

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

<table>
<thead>
<tr>
<th>Foundational Areas</th>
<th>Model-Driven Areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>foundations</td>
<td>logic-based models</td>
</tr>
<tr>
<td>ontologies</td>
<td>model transformation</td>
</tr>
<tr>
<td>model integration</td>
<td>pattern for modeling</td>
</tr>
<tr>
<td>generative models</td>
<td>model verification</td>
</tr>
<tr>
<td>model validation</td>
<td>model consistency</td>
</tr>
<tr>
<td>standardization</td>
<td>model engineering</td>
</tr>
<tr>
<td>model-driven architectures</td>
<td>aspect-oriented modeling</td>
</tr>
<tr>
<td>modeling nonfunctional requirements</td>
<td>model-based reengineering</td>
</tr>
<tr>
<td>model-based refactoring</td>
<td>business process model</td>
</tr>
<tr>
<td>enterprise application model</td>
<td>models for safety and security</td>
</tr>
<tr>
<td>models for distributed control</td>
<td>modeling tools</td>
</tr>
<tr>
<td>model animation</td>
<td>model simulation</td>
</tr>
<tr>
<td>modeling techniques</td>
<td>model quality</td>
</tr>
</tbody>
</table>

WALTER DOSCH
Institute of Software Technology and Programming Languages
University of Lübeck, Germany
2.3 Modeling Components — Views

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Communication-Based Description</th>
<th>State-Based Description</th>
<th>Trace-Based Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Model</td>
<td>Interaction Model</td>
<td>State Model</td>
<td>Process Model</td>
</tr>
</tbody>
</table>

**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

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
3.1 Data Model — Entity Relationship Modeling

- **Sets**
  - Reader, Book

- **Attributes**
  - name: Reader $\rightarrow$ String
  - address: Reader $\rightarrow$ String
  - reminder: Reader $\rightarrow$ Bool

- **Relations**
  - borrows $\subseteq$ Reader x Book
  - cites $\subseteq$ Book x Book
3.2 Algebraic Specification — Stacks

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

**STACK**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>empty</code></td>
<td>stack</td>
</tr>
<tr>
<td><code>prefix</code></td>
<td>data * stack</td>
</tr>
<tr>
<td><code>first</code></td>
<td>stack</td>
</tr>
<tr>
<td><code>rest</code></td>
<td>stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concrete Value</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first(empty)</code></td>
<td>undefined</td>
</tr>
<tr>
<td><code>first(prefix(d, s))</code></td>
<td><code>d</code></td>
</tr>
<tr>
<td><code>rest(empty)</code></td>
<td>undefined</td>
</tr>
<tr>
<td><code>rest(prefix(d, s))</code></td>
<td><code>s</code></td>
</tr>
</tbody>
</table>

abstract data type = \[
\frac{\text{signature}}{\text{interface}} + \frac{\text{axioms}}{\text{behaviour}}
\]
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 $S$

\[ \text{prefix}^S(n, s) \rightarrow s \]

\[ \begin{align*}
\text{first}^S(s) \quad & \text{rest}^S(s)
\end{align*} \]

- **Algebraic methods**
- **Vertical modularization**

**Equational & inductive reasoning**

**Program**

**Functional data model**
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.

```
T • A • R • T • U •
```

**Benefits for systems programming**
- **state** selective updating
- **pointer** sharing, manipulation
- **storage** cyclic structures
- **storage** create — destroy

**Object-oriented data model**
- **state encapsulation**
- **object identity**
- **life cycle**

---

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
4. Interaction Models

**Top-down design of components**

<table>
<thead>
<tr>
<th>Black box view</th>
<th>Glass box view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
<td>Internal state</td>
</tr>
</tbody>
</table>

**Black box view**

- Input/output behaviour
- Service, Composition
- Correctness, Properties

**Glass box view**

- Architecture
- Implementation
- Efficiency
4.1 Models of Computation

Sequential Model of Computation

\[ \text{initial state} \rightarrow \text{output} \rightarrow \cdots \rightarrow \text{final state} \]

Interactive Model of Computation

\[ \text{output} \rightarrow \text{output} \rightarrow \cdots \rightarrow \text{output} \rightarrow \text{input} \rightarrow \text{input} \rightarrow \cdots \]
4.2 Streams

Streams model communication histories on unidirectional channels.

Finite streams \( \mathcal{A}^* = \{ \langle x_0, \ldots, x_m \rangle \mid x_i \in \mathcal{A} \} \)

Concatenation \( \langle x_0, \ldots, x_m \rangle \& \langle y_0, \ldots, y_n \rangle = \langle x_0, \ldots, x_m, y_0, \ldots, y_n \rangle \)

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

Stream transformer \( f : \mathcal{A}^* \to \mathcal{B}^* \)

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.

```
push(a)
push(b)
push(c)
pop
pop
```

Diagram:
```
<table>
<thead>
<tr>
<th>push(c)</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(b)</td>
<td></td>
</tr>
<tr>
<td>push(a)</td>
<td></td>
</tr>
</tbody>
</table>

```
4.4 Interactive Stack — Regular Behaviour

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

- type of input messages: \(\mathcal{I} = \{\text{pop, reset}\} \cup \text{push}(\mathcal{D})\)
- type of output messages: \(\mathcal{O} = \mathcal{D}\)

\[
\text{istack} : \mathcal{I}^* \rightarrow \mathcal{O}^*
\]

\[
\text{istack}(\text{Push}) = \langle \rangle \\
\text{istack}(\text{Push} & \langle \text{push}(d), \text{pop} \rangle & X) = \langle d \rangle & \text{istack}(\text{Push} & X) \\
\text{istack}(\text{Push} & \langle \text{reset} \rangle & X) = \text{istack}(X)
\]

\(\text{Push} \in \text{push}(\mathcal{D})^*\)
### 4.5 Interactive Stack — Irregular Behaviour

<table>
<thead>
<tr>
<th>input history</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{push}(1))</td>
<td>pop</td>
</tr>
<tr>
<td>(\text{pop})</td>
<td>pop</td>
</tr>
<tr>
<td>(\text{pop})</td>
<td>(\text{push}(2))</td>
</tr>
</tbody>
</table>

fault-sensitive stack

<table>
<thead>
<tr>
<th>(\text{push}(1))</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\langle 1 \rangle)</td>
<td></td>
</tr>
</tbody>
</table>

fault-tolerant stack

<table>
<thead>
<tr>
<th>(\text{push}(1))</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{push}(2))</td>
<td>(\text{push}(3))</td>
</tr>
</tbody>
</table>

robust stack

| \(\text{push}(1)\) | pop | pop | pop | \(\text{push}(2)\) | \(\text{push}(3)\) | \(\text{push}(4)\) | pop | ... | \(\langle 1, u, u, 4 \ldots \rangle\) |

fault-correcting stack

| \(\text{push}(1)\) | pop | pop | pop | \(\text{push}(2)\) | \(\text{push}(3)\) | \(\text{push}(4)\) | pop | ... | \(\langle 1, 2, 3, 4 \ldots \rangle\) |

A component guarantees a service only for input from the service domain.
4.6 Interactive Stack — *Sequence Diagram* (Sample)

\[
\text{state}(P \& \langle \text{push}(d) \rangle) = \text{state}(P \& \langle \text{push}(d) \rangle \& Q) \\
\land \quad \text{istack}(P \& \langle \text{push}(d) \rangle \& Q) = R \\
\Rightarrow \quad \text{istack}(P \& \langle \text{push}(d) \rangle \& Q \& \langle \text{pop} \rangle) = R \& \langle d \rangle
\]
Outline

1. Introduction
2. Modeling
   3. Data Models
   4. Interaction Models
   5. State-Based 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**

- Communication

**Glass box view**

- Internal state

<table>
<thead>
<tr>
<th>Black box view</th>
<th>Glass box view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/output behaviour</td>
<td>Architecture</td>
</tr>
<tr>
<td>Service</td>
<td>Implementation</td>
</tr>
<tr>
<td>Correctness</td>
<td>Efficiency</td>
</tr>
</tbody>
</table>
5.1 State Transition Machines

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

- set $Q$ of states
- set $I$ of input data
- set $O$ of output data

one-step state transition function $\delta: Q \times I \rightarrow Q$

one-step output function $\varphi: Q \times I \rightarrow O$

initial state $q_0 \in Q$

Processing input streams

multi-step state transition function $\delta^*: Q \rightarrow [I^* \rightarrow Q]$

multi-step output function $\varphi^*: Q \rightarrow [I^* \rightarrow O^*]$

The multi-step output function $\varphi^*(q)$ is a stream transformer!
5.1 State Transition Machine (cdt.) — Interactive Stack

\[ M = (Q, I, O, \delta, \varphi, \langle \rangle) \]

\[
\begin{array}{c}
Q = D^* \cup \{fail\} \\
\delta(fail, x) = fail \\
\delta(Q, push(d)) = Q \& \langle d \rangle \\
\delta(\langle \rangle, pop) = fail \\
\delta(Q \& \langle d \rangle, pop) = Q \\
\varphi(q, push(d)) = \langle \rangle \\
\varphi(fail, pop) = \langle \rangle \\
\varphi(\langle \rangle, pop) = \langle \rangle \\
\varphi(Q \& \langle d \rangle, pop) = \langle d \rangle \\
\end{array}
\]

\( M \) implements the interactive stack: \( \varphi^*(\langle \rangle)(X) = istack(X) \)
5.2 State Transition Tables

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{state} & \text{input} & \text{state'} & \text{output} \\
\hline
q & x & \delta(q, x) & \varphi(q, x) \\
\hline
\end{array}
\]

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

The transition rules tabulate the transition functions \( \delta \) and \( \varphi \) for all possible combinations.

Interactive Stack

\[
\begin{array}{|c|c|c|c|}
\hline
\text{state} & \text{input} & \text{state'} & \text{output} \\
\hline
\text{fail} & x & \text{fail} & \langle \rangle \\
Q & \text{push}(d) & Q & \langle d \rangle \\
\langle \rangle & \text{pop} & \text{fail} & \langle \rangle \\
Q & \langle d \rangle & \text{pop} & Q \\
\end{array}
\]
5.3 State Transition Diagrams

State transition machines can be considered as labelled directed graphs:

<table>
<thead>
<tr>
<th>graph</th>
<th>state transition machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>( Q )</td>
</tr>
<tr>
<td>vertices</td>
<td>( q \to \delta(q, x) )</td>
</tr>
<tr>
<td>labels</td>
<td>( x / \varphi(q, x) )</td>
</tr>
<tr>
<td></td>
<td>states</td>
</tr>
<tr>
<td></td>
<td>transitions</td>
</tr>
<tr>
<td></td>
<td>(input, output)</td>
</tr>
</tbody>
</table>

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

\[ Q, Q' \in \mathcal{D}^* \quad d \in \mathcal{D} \quad x \in \mathcal{I} \]
State Transition Diagram (part) for $\mathcal{D} = \{0, 1\}$
5.4 Symbolic State Transition Diagrams

Stack of Data Elements from \( D \)

This \textit{representation} identifies all \textit{regular states} \((Q)\).
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
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

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.

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

\[
\begin{align*}
S(i_1, i_2, i_3) &= (o_1, o_2) \\
S_1(i_1, l_1) &= (o_1, l_3) \\
S_2(i_2, l_2, l_3) &= (l_1, l_4) \\
S_3(l_4, i_3) &= (l_2, o_2)
\end{align*}
\]
6.2 Protocol Architecture

Correctness
The equation $Z = X$ specifies that no messages are *lost, reordered, duplicated* or *faked.*
6.3 Layered Architecture
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: \(\mathcal{I} = \{\text{pop, reset}\} \cup \text{push}(\mathcal{D})\)
- type of output messages: \(\mathcal{O} = \mathcal{D}\)

**Transformation** Signature \(\mapsto\) Interaction Interface

\[
\text{prefix} : \text{data} \times \text{stack} \rightarrow \text{stack} \\
\text{push} : \text{data} \rightarrow [\text{stack} \rightarrow \text{stack}] \\
\text{first} : \text{stack} \rightarrow \text{data} \\
\text{rest} : \text{stack} \rightarrow \text{stack} \\
\text{pop} : \text{stack} \rightarrow \text{data} \times \text{stack} \\
\text{empty} : \rightarrow \text{stack} \\
\text{reset} : \text{stack} \rightarrow \text{stack}
\]

*Design Decisions:* **Encapsulation** \((\text{stack})\) and input/output parameters.
### 7.3 Data Models — From Streams to States

\[
\begin{align*}
\text{stream transformer } f : A^* & \to B^* \\
\text{history abstraction } abstr : A^* & \to Q \\
\to \quad \text{state transition machine } & \quad M[f, abstr] = (Q, A, B, \delta, \varphi, q_0)
\end{align*}
\]

<table>
<thead>
<tr>
<th>states</th>
<th>abstractions of input histories</th>
<th>( Q = abstr(A^*) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>state transition function</td>
<td>extend input history ( \delta(abstr(P), x) = abstr(P &amp; \langle x \rangle) )</td>
<td></td>
</tr>
<tr>
<td>output function</td>
<td>incremental output ( \varphi(abstr(P), x) = f'(P, x) )</td>
<td></td>
</tr>
<tr>
<td>initial state</td>
<td>empty input history ( q_0 = abstr(\langle \rangle) )</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
f(P) \\
\hline
f(P \& \langle x \rangle) \\
\hline
f'(P, x)
\end{array}
\]

States are history abstractions.
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

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
8.1 Models for Programming Languages — Disjunction

The denotational semantics provides models for programming language constructs.

<table>
<thead>
<tr>
<th>strict disjunction</th>
<th>left-right disjunction</th>
<th>parallel disjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{or}</td>
<td>\textit{lor}</td>
<td>\textit{por}</td>
</tr>
<tr>
<td>\top</td>
<td>\top</td>
<td>\top</td>
</tr>
<tr>
<td>\bot</td>
<td>\bot</td>
<td>\bot</td>
</tr>
<tr>
<td>\true</td>
<td>\true</td>
<td>\true</td>
</tr>
<tr>
<td>\false</td>
<td>\false</td>
<td>\false</td>
</tr>
<tr>
<td>\false</td>
<td>\false</td>
<td>\false</td>
</tr>
<tr>
<td>\true</td>
<td>\true</td>
<td>\true</td>
</tr>
</tbody>
</table>

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 expression \((0^+ \cdot 1 + 1^+ \cdot 0)^\omega\)

Grouping an infinite stream of Boolean values:

\[
\langle 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, \ldots \rangle
\]

<table>
<thead>
<tr>
<th>fairness</th>
<th>nondeterministic choice</th>
<th>fairness</th>
</tr>
</thead>
<tbody>
<tr>
<td>liveness</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Layout</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topological</td>
<td></td>
<td>Metric</td>
</tr>
<tr>
<td>Communication</td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td></td>
<td>Unidirectional</td>
<td>Bidirectional</td>
</tr>
<tr>
<td>State</td>
<td>State-full</td>
<td>State-less</td>
</tr>
<tr>
<td></td>
<td>Continuous</td>
<td>Discrete</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>Structured</td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td>Distributed</td>
</tr>
<tr>
<td>Time</td>
<td>Timed</td>
<td>Untimed</td>
</tr>
<tr>
<td></td>
<td>Continuous</td>
<td>Discrete</td>
</tr>
<tr>
<td></td>
<td>Sensitive</td>
<td>Invariant</td>
</tr>
<tr>
<td>Control</td>
<td>(Non)deterministic</td>
<td>Stochastic</td>
</tr>
<tr>
<td></td>
<td>Centralized</td>
<td>Distributed</td>
</tr>
<tr>
<td></td>
<td>Event-driven</td>
<td>Time-driven</td>
</tr>
</tbody>
</table>
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