Model-Based Software and Systems Engineering

Elements of Seamless Development

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 Munich, Germany
When modeling software in systems we have capture following aspects:

- interaction – exchange of information/material
- distribution – structuring systems in architectures with elements related to locations
- context – operational system’s environment
- real time
- probability
- physicality
- ...

To do that we have to use concepts

- interfaces – scope and interaction
- state – state transition
- architecture – (de-)composition
What is seamless development?

- Development by a chain of models
  - High expressive power
  - Clear structure of role of models
  - All aspects in the development captured by models
  - Tight integration in the artifact models

- Development steps by well-defined relationship between modes
  - Refinement
  - Decomposition
  - Change of scope

- Extended tool support
  - High automation
  - All artifacts in tools and comprehensive data base (development back bone)
Basic System Modeling Concepts
System and its context
Basic System Notion: What is a discrete system (model)

A system has

• a system **boundary** that determines
  ◊ what is part of the systems and
  ◊ what lies outside (called its **context**)

• an **interface** (determined by the system boundary), which determines,
  ◊ what ways of interaction (actions) between the system und its context are possible (static or **syntactic interface**)
  ◊ which behavior the system shows from view of the context (**interface behavior**, dynamic interface, interaction view)

• a structure and distribution addressing internal structure, given
  ◊ by its structuring in sub-systems (**sub-system architecture**)
  ◊ by its states und state transitions (**state view**, state machines)

• **quality profile**

• the views use a **data model**

• the views may be documented by adequate models
Discrete systems: the modeling theory – interface behaviour as key concept

Sets of typed channels

\[ I = \{x_1 : T_1, x_2 : T_2, \ldots\} \]
\[ O = \{y_1 : T'_1, y_2 : T'_2, \ldots\} \]

syntactic interface

\((I \rightarrow O)\)

data stream of type \(T\)

\[ \text{STREAM}[T] = \{\text{IN}\{0\} \rightarrow T^*\} \]

valuation of channel set \(C\)

\[ \text{IH}[C] = \{C \rightarrow \text{STREAM}[T]\} \]

interface behaviour for syn. interface \((I \rightarrow O)\)

\[ [I \rightarrow O] = \{\text{IH}[I] \rightarrow \emptyset (\text{IH}[O])\} \]

interface specification

\[ p : I \cup O \rightarrow IB \]

represented as interface assertion \(S\)

logical formula with channel names as variables for streams

Towards a uniform model: Basic system model

**System class:** distributed, reactive systems

System consists of
- named components (with local state)
- named channels

driven by a global, discrete clock
Timed Streams: Semantic Model for Black-Box-Behavior

Message set:
\[ M = \{a, b, c, \ldots\} \]

Messages transmitted at time \( t \)

Infinite channel history
Modeling Interface Behavior
The Basic Behaviour Model: Timed Streams and Channels

\[ C \]  
set of channels

Type: \( C \rightarrow \text{TYPE} \)  
type assignment

\[ x : C \rightarrow (\mathbb{N}\{0\} \rightarrow M^*) \]  
channel history for messages of type \( M \)

\( \tilde{C} \) or \( \text{IH}[C] \)  
set of channel histories for channels in \( C \)
System interface model

Channel: Identifier of Type stream

\[ I = \{ x_1 : D_1, x_2 : D_2, \ldots \} \] set of typed input channels
\[ O = \{ y_1 : T_1, y_2 : T_2, \ldots \} \] set of typed output channels

Syntactic interface: \( (I \rhd O) \)

Interface behavior

\[ F : \bar{I} \rightarrow \wp(\bar{O}) \]

Set of interface behaviours with input channels \( I \) and output channels \( O \):

\[ \text{IF}[I \rhd O] \]

Set of all interface behaviours:

\[ \text{IF} \]
System interface behaviour - causality

(I \rightarrow O) \textit{syntactic interface} with set of input channels I and of output channels O

\[ F : \bar{I} \rightarrow \wp(\bar{O}) \textit{semantic interface} \] for (I \rightarrow O) with \textit{timing property addressing strong causality} (let \( x, z \in \bar{I}, y \in \bar{O}, t \in \text{IN} \)):

\[ x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t+1: y \in F(x)\} = \{y \downarrow t+1: y \in F(z)\} \]

\( x \downarrow t \) prefix of history \( x \) of length \( t \)

A system shows a \textit{total behavior}
Continuous systems: the model

Sets of typed channels

\[ I = \{x_1 : T_1, x_2 : T_2, \ldots \} \]
\[ O = \{y_1 : T'_1, y_2 : T'_2, \ldots \} \]

syntactic interface

\[(I \rightarrow O)\]

continuous data stream of continuous type \(M\)

\[ \text{ConSTREAM}[T] = \{IR_+ \rightarrow M\} \]

valuation of channel set \(C\)

\[ \text{CIH}[C] = \{C \rightarrow \text{ConSTREAM}[T]\} \]

interface behaviour for syn. interface \((I \rightarrow O)\)

\[ [I \rightarrow O] = \{\text{CIH}[I] \rightarrow \wp (\text{CIH}[O])\} \]

interface specification

\[ p: I \cup O \rightarrow IB \]

represented as interface assertion \(S\)

logical formula with channel names as variables for continuous streams

Sets of typed channels

\[ I = \{x_1 : T_1, x_2 : T_2, \ldots \} \]
\[ O = \{y_1 : T'_1, y_2 : T'_2, \ldots \} \]

syntactic interface

\[ (I \searrow O) \]

data stream of type \( T \)

\[ \text{STREAM}[T] = \{\text{IN}\setminus\{0\} \rightarrow T^*\} \]

valuation of channel set \( C \)

\[ \text{IH}[C] = \{C \rightarrow \text{STREAM}[T]\} \]

interface behaviour for syn. interface \((I \searrow O)\)

\[ [I \searrow O] = \{\text{IH}[I] \rightarrow \text{PD}[\varnothing (\text{IH}[O])] \} \]

interface specification

\[ p: I \cup O \rightarrow \text{IB} \]

represented as interface assertion \( S \)

logical formula with channel names as variables for streams
Extensions of the model

- Physical Aspects & Properties: Rich Models
  - Space
  - Geometry
  - Temperature
  - ...

Evolution and Development:
Specification, Refinement, Compatibility
Interface Specification

- An interface model describes in a particular abstraction the interface behavior of a system
  - interface behavior for syn. interface \((I \rightarrow O)\)
    \[[I \rightarrow O] = \{IH[I] \rightarrow \emptyset (IH[O])\}\]
  - interface specification by a predicate
    \[
p: I \cup O \rightarrow IB
    \]
  written by an interface assertion

Example: System interface specification

A transmission component TMC

Spec name

Input channel

TMC

in

x: T

out

y: T

x ≈ y

Output channel

Specifying interface assertion

x ≈ y \equiv (\forall m \in T: m\#x = m\#y)

TMC

x ≈ y

x:T

Verification: Proving properties about specified components

From the interface assertions we can prove

- Safety properties

\[ m\#y > 0 \land y \in TMC(x) \Rightarrow m\#x > 0 \]

- Liveness properties

\[ m\#x > 0 \land y \in TMC(x) \Rightarrow m\#y > 0 \]
Structure:
Composition and Decomposition
Composing Interfaces

• Composition is an operation on syntactically compatible system interfaces

\[ \otimes : [I_1 \rightarrow O_1] \times [I_2 \rightarrow O_2] \rightarrow [I \rightarrow O] \]

• The operation \( \otimes \) induces a composition operation on specifications

Modularity: Rules of compositions for interface specs

F1
- \textbf{in} x1, z12: T
- \textbf{out} y1, z12: T
- S1

F2
- \textbf{in} x2, z12: T
- \textbf{out} y2, z21: T
- S2

F1 \otimes F2
- \textbf{in} x1, x2: T
- \textbf{out} y1, y2: T
- \exists z12, z21: S1 \land S2
Refining Interfaces
Refinement

• The idea of system refinement is that systems are developed
  ◊ by a sequence of development steps
  ◊ each step produces a more refined system description
  ◊ there is a refinement relation between the current system description and the produced system description

• The refinement relation
  ◊ is a relation between systems descriptions

• The relation can be used as an idealized relationship between
  ◊ specifications to formalize the steps of gathering requirements in requirements engineering
  ◊ specifications and architectures to formalize the steps in design of going from requirements to architecture
  ◊ system specifications and implementations (e.g. by state machines)
  ◊ levels of abstraction
Refinement

• Given systems/specifications $S$ and $S'$ we write $S \text{ is\_refined\_to } S'$ (and also $S \approx> S'$) to express that $S'$ is a refinement of $S$.

• **Horizontal Refinement:** Property Refinement
  ◇ Adding properties to a specification – reducing the non-determinism of a system description

• **Vertical Refinement:** Interaction granularity refinement
  ◇ Changing the granularity of the interaction
  ◇ Interesting case: Refinements between continuous and discrete models


Remark: Both refinement notions can be applied not only to interface models but to all kinds of models
Horizontal Refinement

F: \( \tilde{I} \to \wp(\tilde{O}) \)
is refined by a behavior \( \hat{F}: \tilde{I} \to \wp(\tilde{O}) \)
if
\[ \forall x \in \tilde{I}: F.x \subseteq \hat{F}.x \]
we write
\[ F \approx>_{IF} \hat{F} \]

Compositionality of refinement:
Modularity

\[ \forall k: F_k \approx>_{IF} \hat{F}_k \]
\[ \otimes\{ F_k: k \in \mathbb{K} \} \approx>_{IF} \otimes\{ \hat{F}_k: k \in \mathbb{K} \} \]
Verification of refinement steps

• A system F with behaviour assertion Q is refined by a system F’ with behaviour assertion Q’ if and only if

\[ Q \iff Q’ \]

In other words: F’ is a refinement of F if all properties of F are also properties of F’

• The implication \( Q \iff Q’ \) shows also how to verify the refinement relation
Vertical refinement: Levels of abstraction

Theorems

- Property refinement implies interaction refinement
- Compositionality of interaction refinement
- Interaction refinement distributes over composition
- Abstractions of interaction refinements of implementations are interaction refinements of abstractions
- Time abstraction is interaction abstraction
- Interaction abstraction is a Galois connection
Compatibility

- A system $S'$ is called (replacement) compatible for system $S$ if $S'$ can be used instead of $S'$ in every system $M$ without violating the correctness of the system $M$
- **Compatibility** coincides with refinement in case of modular refinement
A flexible model of time

- Time is a key issue in embedded systems:
- Dealing with timing properties
  - Specification
  - Analysis
  - Verification
    - Analysis
    - Testing
    - Model checking
    - Deduction based verification
- Transforming time
- Dedicated models of time
  - Micro/Macro Step
  - Perfect synchrony
  - Scheduling
- Abstractions
Example: TMC with Timing Restrictions

TMC

<table>
<thead>
<tr>
<th>in</th>
<th>x: T</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>y: T</td>
</tr>
</tbody>
</table>

∀ t ∈ IN: ∀ m ∈ T:

m#(x ↓ t) ≥ m#(y ↓ t+delay)

m#(x ↓ t) ≤ m#(y ↓ t+delay+deadline)
Conclusion Refinement

• Refinement formalises development steps
• Going from
  ◊ an interface specification to an architecture (design step)
  ◊ an interface specification to a state machine (implementation)
  can be understood as special steps of refinement
• Compatibility is defined by refinement, too
• The change of time granularity is a refinement
Implementation: Systems as State Machines

The State View
System and States

• Systems have states
• A state is an element of a state space
• We characterize state spaces by
  ◊ a set of state attributes together with their types
• The behaviour of a system with states can be described by its state transitions
A system can be implemented by a state machine

\( \Sigma \) set of states, initial state \( \sigma \subseteq \Sigma \)

State transition function:

\[ \Delta: (\Sigma \times (I \rightarrow M^*)) \rightarrow \varnothing (\Sigma \times (O \rightarrow M^*)) \]

State transition diagram:

State Machines in general

A state machine \((\Delta, \Lambda)\) consists of

- a set \(\Sigma\) of states - the state space
- a set \(\Lambda \subseteq \Sigma\) of initial states
- a state transition function or relation \(\Delta\)
  - in case of a state machine with input/output:
    events (inputs \(E\)) trigger the transitions and events (outputs \(A\)) are
    produced by them respectively:
    \[
    \Delta : \Sigma \times E \rightarrow \Sigma \times A
    \]
    in the case of nondeterministic machines:
    \[
    \Delta : \Sigma \times E \rightarrow \emptyset (\Sigma \times A)
    \]
- Given a syntactic interface with sets \(I\) and \(O\) of input and output channels:
  \[
  E = I \rightarrow M^* \\
  A = O \rightarrow M^*
  \]
A state machine \((\Delta, \Lambda)\) defines for each initial state
\[
\sigma_0 \in \Lambda
\]
and each sequence of inputs
\[
e_1, e_2, e_3, \ldots \in E
\]
a sequence of states
\[
\sigma_1, \sigma_2, \sigma_3, \ldots \in \Sigma
\]
and a sequence of outputs
\[
a_1, a_2, a_3, \ldots \in A
\]
through
\[
(\sigma_{i+1}, a_{i+1}) \in \Delta(\sigma_i, e_{i+1})
\]
Computations of a State Machine with Input/Output

In this manner we obtain computations of the form

$$
s_0 \xrightarrow{a_1/b_1} s_1 \xrightarrow{a_2/b_2} s_2 \xrightarrow{a_3/b_3} s_3 \ldots
$$

For each initial state $s_0 \in \Sigma$ we define a function

$$
F_{s_0} : \overline{I} \rightarrow \wp(\overline{O})
$$

with

$$
F_{s_0}(x) = \{y : \exists \sigma_i : s_0 = \sigma_0 \land \forall i \in \mathbb{IN} : (\sigma_{i+1}, y_{i+1}) = \Delta(\sigma_i, x_{i+1})\}
$$

$F_{s_0}$ denotes the interface behavior of the transition function $\Delta$ for the initial state $s_0$.

Furthermore we define

$$
\text{Abs}((\Delta, \Lambda)) = F_{\Lambda}
$$

where:

$$
F_{\Lambda}(x) = \{y \in F_{s_0}(x) : y \in F_{s_0}(x) \land \sigma \in \Lambda\}
$$

$F_{\Lambda}$ is called the interface behavior of the state machine $(\Delta, \Lambda)$. 
Moore Machines

• A Mealy machine \((\Delta, \Lambda)\) with
  \[
  \Delta : \Sigma \times E \to \mathcal{P}(\Sigma \times A)
  \]
is called **Moore machine** if for all states \(\sigma \in \Sigma\) and inputs \(e \in E\) the set
  \[
  \text{out}(\sigma, e) = \{ a \in A : (\sigma, a) = \Delta(\sigma, e) \}
  \]
does not depend on the input \(e\) but only on state \(\sigma\).

• Formally: then for all \(e, e' \in E\) we have
  \[
  \text{out}(\sigma, e) = \text{out}(\sigma, e')
  \]

**Theorem:** If is \((\Delta, \Lambda)\) a Moore machine the \(F_\Lambda\) is strong causal.
Interface Abstraction for State Machines

• For a given state machine with input and output we define the interface through
  ◦ its syntactical interface (signature)
  ◦ its interface behavior

• We call the transition of the state machine to its interface the interface abstraction.

Verification/derivation of interface assertions for state machines
• similar to program verification (find an invariant)
• needs sophisticated techniques
Observable Equivalence

• Two systems modelled by state machines \((\Delta_1, \Lambda_1)\) and \((\Delta_2, \Lambda_2)\) are observably equivalent iff they fulfil the equation

\[
\text{Abs}((\Delta_1, \Lambda_1)) = \text{Abs}((\Delta_2, \Lambda_2))
\]
Composition of the two state machines

Consider Moore machines $M_k = (\Delta_k, \Lambda_k) (k = 1, 2)$:

$\Delta_k: \Sigma_k \times (I_k \to M^*) \to \wp(\Sigma_k \times (O_k \to M^*))$

We define the composed state machine

$\Delta: \Sigma \times (I \to M^*) \to \wp(\Sigma \times (O \to M^*))$

as follows

$\Sigma = \Sigma_1 \times \Sigma_2$

for $x \in I$ and $(s_1, s_2) \in \Sigma$ we define:

$\Delta((s_1, s_2), x) = \{(s'_1, s'_2), z|O): x = z|I \land \forall k: (s'_k, z|O_k) \in \Delta_k(s_k, z|I_k)\}$

This definition is based on the fact that we consider Moore machines.

We write

$\Delta = \Delta_1 \parallel \Delta_2$

$M = M_1 \parallel M_2 = (\Delta_1 \parallel \Delta_2, \Lambda_1 \times \Lambda_2)$
An example of an essential property ...

Interface abstraction distributes for state machines over composition

\[
\text{Abs}((\Delta_1, \sigma_1) \parallel (\Delta_2, \sigma_2)) = \text{Abs}((\Delta_1, \sigma_1)) \otimes \text{Abs}((\Delta_2, \sigma_2))
\]
Conclusion Systems as State Machines

• Each state machines defines an interface behaviour
• Each interface behaviour represents a state machine
• State machines can be described
  ◦ mathematically by their state transition function
  ◦ graphically by state machine diagrams
  ◦ structured by state transition tables
  ◦ by programs
• State machines define a kind of operational semantics
• Systems given by state machines can be simulated
• From state machines we can generate code
  ◦ state machines can represent implementations
• From state machines we can generate test cases
Functional View: Functional Decomposition
Combining Functions

Given two functions $F_1$ and $F_2$ in isolation

We want to combine them into a function $F_1 \otimes F_2$
Combining Functions

Their isolated combination

\[ F_1 \otimes F_2 \]
Combining Functions

If services $F_1$ and $F_2$ have feature interaction we get:

We explain the functional combination $F_1 \otimes F_2$ as a refinement step.
The steps of function combination

Given the isolated function $F_1$

We construct a refinement $F'_1$

And combine $F'_1$ with a refinement $F'_2$ of $F_2$

Functional architecture: functional decomposition

- The system interface behaviour $F$ as specified by the system requirements specification $A = \{A_i: 1 \leq i \leq n\}$ is structured
  - into a set of sub-interfaces for sub-functions $F_1, \ldots, F_k$
  - that are specified independently by introducing a number of mode channels to capture their feature interactions
  - each $F_i$ sub-function is described by
    - a syntactic interface and
    - an interface assertion $B_i$ such that
      $$\land \{B_i: 1 \leq i \leq k\} \Rightarrow A$$
Function Hierarchy

\[ F_{1,\ldots,n} \quad B_1 \quad B_2 \quad \cdots \quad B_n \]

\[ F_{1,2} \quad B_1 \quad B_2 \quad \cdots \]

\[ F_{k,k+1} \quad B_{k-1} \quad B_k \quad \cdots \]

\[ F_{n-1,n} \quad B_{n-1} \quad B_n \]

\[ F_1 \quad B_1 \quad B_2 \quad \cdots \]

\[ F_2 \quad B_2 \quad \cdots \]

\[ F_k \quad B_k \quad \cdots \]

\[ F_{k+1} \quad B_{k+1} \quad \cdots \]

\[ F_{n-1} \quad B_{n-1} \quad \cdots \]

\[ F_n \quad B_n \]

\( \leftarrow_{\text{sub}} \) subservice relation

\( \cdots \cdots \cdots \cdots \cdots \cdots \) channels of mode types
Model Integration

Technische Universität München
Institut für Informatik
D-80290 Munich, Germany
Integrating modeling concepts

- An architecture can be abstracted into an interface behavior
  ◊ Proof techniques for architecture verification
- A state machine can be abstracted into an interface behavior
  ◊ Proof techniques for implementation verification

Extensions of the model: Probability

• Probabilistic views
  ◊ Interface behavior: a probability distribution is given for the set of possible histories
  ◊ Architectural view: probability distributions for the sub-systems of the architecture
  ◊ State view: a probability distribution is given for the set of possible state transitions

• Then the model covers
  ◊ certain “non-functional properties” (safety, reliability, ...)
  ◊ Example: integrated fault trees

System development proceeds in working out a sequence of perspectives at several levels of abstraction that are related
• by refinement
• by decomposition
• by changing the scope/system boundary
• by embedding
Modular Model Based System Development
Comprehensive System Architecture
Levels of Abstraction
### Structuring Systems: levels of abstraction

<table>
<thead>
<tr>
<th>Anforderungen</th>
<th>Interface assertion</th>
<th>Safety</th>
<th>Priority</th>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R₁</td>
<td>Yes</td>
<td>high</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R₂</td>
<td>No</td>
<td>medium</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rₙ</td>
<td>no</td>
<td>low</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Goals and Requirements and Functional Specification
Relational view: Tracing – The power of logics

<table>
<thead>
<tr>
<th>Function</th>
<th>Safety</th>
<th>Priority</th>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>Yes</td>
<td>high</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₂</td>
<td>No</td>
<td>medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aₙ</td>
<td>no</td>
<td>low</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Glass Box Specification of a Car’s Architecture

Example: how A/P-specifications can be formulated

• The specification
  \[ \forall t: \neg \text{doors\_closed}(t) \Rightarrow \text{act\_speed}(t) = 0 \]
  can only be guaranteed if the two inner components work together. This requires
  \[ \forall t: \neg \text{ready}(t) \Rightarrow \text{act\_speed}(t) = 0 \]

• Then the system specification holds if
  \[ \forall t: \neg \text{doors\_closed}(t) \Rightarrow \neg \text{ready}(t) \]

• This is logically equivalent to the A/P-specification for the WatchDog assumption:
  \[ \forall t: \neg \text{ready}(t) \Rightarrow \text{act\_speed}(t) = 0 \]
  commitment: \[ \forall t: \neg \text{doors\_closed}(t) \Rightarrow \text{act\_speed}(t) = 0 \]

• In other words,
  ◊ the overall system specification can be guaranteed by the watchdog
  ◊ only if the assumption about the behaviour of the component motor holds.
Assumption/Promise to define Architectural Design Patterns

• A/P-specification

\textbf{assumption:} \( \forall t: \neg \text{ready}(t) \implies \text{act\_speed}(t) = 0 \)

\textbf{commitment:} \( \forall t: \neg \text{doors\_closed}(t) \implies \text{act\_speed}(t) = 0 \)

is logically guaranteed by the simple specification

\( \forall t: \neg \text{doors\_closed}(t) \implies \neg \text{ready}(t) \)

• This assertion no longer speaks about the specification of the environment, but is a pure interface specification.

• The example shows the simplification of an A/P-specification to a plain interface assertion.
Conclusion A/C

- A/C specs address the logic of the architecture rather than separated interface specifications
- From A/C specs we may derive simplified component specs
- This gives a methodology towards a modular decomposition
Key Principles in Engineering CPSs

- **Abstraction**
  - Interfaces
  - Changing levels of abstractions

- **Modularity**
  - Of composition
  - Of refinement

- **Semantic coherence**
  - From state machines to interfaces
  - From architectures to interfaces
  - From architectures of state machines to state machines of architectures
  - Probabilistic extension of logical (deterministic) models

- **Expressiveness**
  - Set theoretic and logical: discrete system models
  - Continuous models: control theory
  - Probabilistic: probability distributions on behavior
**The Triad of Modeling**

- **Denotational:**
  - mathematical models (of behavior)

- **Logical – system properties:**
  - specification
  - deduction
  - verification
  - transformation

- **Notational:**
  - Graphical: diagrams
  - Tables
  - Formulas
Concluding Remarks

• The modelling framework Focus
  ◇ originally worked out for model based development
  ◇ specification
  ◇ verification
  ◇ tool support

• Tool: Autofocus 3

• Also useful for semantic foundation following the same approach
  ◇ SDL
  ◇ Bus Systems: CAN, FLEXRAY
  ◇ UML/SysML
  ◇ SOA

See: https://af3.fortiss.org/projects/autofocus3

Springer Verlag 2012
Concluding Remarks

- Today software & systems engineering is too much orientated towards the technical architecture and solutions/implementation in the beginning
- We need a comprehensive “architectural” model-based view onto systems including requirements for dealing with complex multi-functional systems
- The models allow for
  - Separation of concerns
  - Separation of technical aspects from application aspects
- Technical architectures are modelled along the same theory
- Code and test cases can be generated from the models
The power of generalizing ideas, of drawing comprehensive conclusions from individual observations, is the only acquirement, for an immortal being, that really deserves the name of knowledge.

“Mary Wollstonecraft (1759–1797), British feminist. A Vindication of the Rights of Woman, ch. 4 (1792)