Topics

• System modelling: goals
• System modelling theory
  ◊ System
    • Interface
    • Logical specification
  ◊ State machines
  ◊ Component decomposition
  ◊ Compositional verification
  ◊ Sub-function unbundling
• Refinement
• Architectures
• Comprehensive architecture modelling
• Architecture of OO systems
Part I
System Modelling

Motivation & Foundations
The role of modelling in software & systems engineering (S&SE)

Software & systems engineering means
• capturing the requirements
  ◦ domain specific
  ◦ functional, logical, technical, methodological
• specification of the system’s overall functionality
• design of a solution in terms of
  ◦ an architecture
  ◦ specifying the components
• implementing the components
• verification of the components and
• integrating them into the system and verifying the integration
• verification of the system
• further evolution

These are complex error prone tasks!
The role of modelling in software & systems engineering

Modelling helps for:

• expressing and documenting the requirements
• specifying the system
• describing the architecture
  ◊ specifying the components
  ◊ their composition and interaction
• modelling the components
• verifying of the components and
• integrating them into the system and verifying the system
The roots of modelling in S&SE

Graphical description:
- Early approaches: SADT, Structured Analysis (SA)
- Later: SDL, ADLs, OOA/D, ROOM
- Today: UML, SysML

Programming and programming languages
- Programming concepts:
  - types (as basis for data models)
  - modules (classes, interfaces, ...)
- Programming logics
- Object oriented programming concepts

Formal description techniques as modelling concepts
- Predicate Logic Based Specification
- Abstract Data Types
- State Machines (Mealy, ...)
- Temporal Logic
- Process Algebras (CCS, CSP, ...)
- Models of distributed concurrent systems (Unity, TLA, ...)
On models and modelling

What is a model?
◊ An abstraction!

Which representations for models?
◊ Informal: language, informal diagrams, ...
◊ Semiformal: formalized graphical or textual presentation languages
◊ Mathematical: in terms of mathematical theories
◊ Formal models: formalized syntax, semantics and logics

What do we use models for?
◊ for understanding - Gedankenmodell
◊ for communication
◊ for specification, design and documentation
◊ for analysis, validation, simulation, verification, certification
◊ for generation of implementations and tests
◊ for reuse

Modelling concepts provide methods for modelling!
What do we model?

- **Domain specific knowledge**
  - taxonomies, ontologies, data models, meta-models, ...
  - laws, rules, ...
  - ...

- **System specific knowledge**
  - data
  - interface behaviour
  - architecture
  - state
  - temporal
  - process
  - ...

- **Technical knowledge**
  - Protocols
  - CPUs
  - ...

We concentrate on digital (discrete) models in the following
The five areas of modelling

- Mathematical Models
- Logical Theories
- Methodology
- Description Techniques
- Tools
System Modelling, SAASE, San Diego  October 2009

Manfred Broy

Informal requirements

Formalized system requirements in terms of service taxonomies

Requirements Engineering

Validation

Component implementation

Verification

Integration

Architecture design

Architecture verification

S = S1 ⊗ S2 ⊗ S3 ⊗ S4
Ingredients for Integration

- Coherent Theory
  - Modelling (data/interface/state/interaction/architecture)
  - Refinement
  - Verification

- Consistent Terminology

- Tractable Description Techniques
  - Formulas/Logics
  - Diagrams/Graphics
  - Tables

- Comprehensive Architecture Structuring

- Flexible Development Process
  - Phases (requirements/design/implementation/test/integration)
  - Artefact Model (concept)
  - Process models
  - Methods

- Powerful Tools
  - Artefact Model (tool support)
  - Automation for documentation, analysis, verification, generation
What has to be modelled?

• Data
  ◦ states and their attributes
  ◦ messages, events, signals

• Requirements and specifications
  ◦ Functional
  ◦ Nonfunctional - Quality Models

• Systems
  • System Architecture
    ◦ Structure
    ◦ Components/interfaces
    ◦ Hierarchy
    ◦ Hardware/Software/Deployment

• Software Architecture
  ◦ Modules
  ◦ Tasks

• Test Cases

• Development Processes

• Development Steps
  ◦ Refactoring
  ◦ Code generation

• Quality attributes
• ...

...
A Comprehensive Mathematical System Model
Towards a comprehensive theory of system modelling: meta model

- **Feature model**
  - Composition
  - Refinement
  - Time

- **Interface model**: components
  - Input and output
  - Uses
  - Implementation

- **Process transition model**: Events, actions and causal relations
  - Composition
  - Refinement
  - Time

- **State transition model**: States and state machines
  - Composition
  - Refinement
  - Time

- **Data model**: Types/sorts and characteristic functions

- **Abstraction**
  - Hierarchy
  - and architecture
  - Is sub-feature

- **Composition**

- **Refinement**

- **Time**

- **Hierarchy and architecture**

- **Is sub-feature**

System Modelling, SAASE, San Diego October 2009

Manfred Broy
What is a (discrete) system?

A system

- has a scope (a boundary)
- a behaviour
  - black box view: interface
    - syntactic interface: defines the discrete events at the system boundary by input and output via ports, channels, messages (events, signals)
    - dynamic interface, interface behaviour: the processes of interaction in terms of discrete events at the system boundary
  - glass/white box view: an internal structure (state and/or distribution into sub-systems)
    - architecture in terms of sets of sub-systems and their relationships (communication connections)
    - state and state transition
- properties
  - quality profile (performance, ... )
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

Basic system model

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

Messages transmitted at time \( t \)

Infinite channel history

System Modelling, SAASE, San Diego  October 2009
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

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

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

Interface behavior

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

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

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

Set of all interface behaviours:

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

\[(I \rightarrow O)\] syntactic interface with set of input channels I and of output channels O

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

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

\[x \downarrow t\] prefix of history x of length t

A system shows a total behavior
Example: Component interface specification

A transmission component TMC

\[ x : T \]

TMC

Input channel

\[ \text{in } x : T \]

\[ \text{out } y : T \]

\[ x \sim y \]

Output channel

\[ \text{TMC } y : T \]

Specifying assertion

\[ x \sim y \equiv (\forall m \in T: \{m\}#x = \{m\}#y) \]

\{m\}#x denotes the number of m in stream x
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

\[ \{m\}#x > 0 \land y \in TMC(x) \Rightarrow \{m\}#y > 0 \]
### Remark: Timed and Untimed Streams

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s : \mathbb{N} \setminus {0} \rightarrow M )</td>
<td>untimed stream: infinite sequence of messages</td>
</tr>
<tr>
<td>( M^\infty )</td>
<td>set of infinite untimed streams</td>
</tr>
<tr>
<td>( M^* )</td>
<td>set of finite untimed streams</td>
</tr>
<tr>
<td>( M^{(\omega)} = M^* \cup M^\infty )</td>
<td>set of finite and infinite untimed streams</td>
</tr>
<tr>
<td>( s \hat{s}' )</td>
<td>concatenation of two streams</td>
</tr>
<tr>
<td>( s : \mathbb{N} \setminus {0} \rightarrow M^* )</td>
<td>timed stream</td>
</tr>
<tr>
<td>((M^*)^\infty )</td>
<td>set of infinite untimed streams</td>
</tr>
<tr>
<td>( - )</td>
<td>time abstraction for the timed stream ( x ), the result of concatenation of all sequences in ( x )</td>
</tr>
</tbody>
</table>
Time Abstraction

\[ x = \langle 1 \quad 4 \quad 7 \quad 8 \quad 9 \quad 5 \quad 3 \rangle \]

\[ \bar{x} = \langle 1 \quad 4 \quad 7 \quad 8 \quad 9 \quad 5 \quad 3 \rangle \]
An interface behaviour $F \in \text{IF}[I \rightarrow O]$ is called **time insensitive**, if for all $x, z \in \vec{I}$:

$$\overline{x} = \overline{z} \Rightarrow \overline{F(x)} = \overline{F(z)}$$

where

$$\overline{F(x)} = \{\overline{y} : y \in F(x)\}$$
A system

• has
  ◊ a scope (system boundary)
  ◊ a syntactic interface
  ◊ an interface behaviour in terms of its function mapping input onto output streams
    • in a time frame
    • supporting nondeterminism

• is specified by giving
  ◊ its syntactic interface (input and output channels and their types of messages (signals, events))
  ◊ its I/O-assertion specifying the interface behaviour in terms of logical formulas including
    • safety and liveness properties
    • real time properties
Systems as State Machines
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
Example: Memory Cell as State Model

Graphically (schematically) as state transition diagram (represents a finite state machine):

![State Transition Diagram]

- Empty
- Full

Actions:
- Write
- Read
- Delete
Example: Memory Cell as State Machine with Input/Output

Graphically (interpreted): state attribute $s : \text{Int} \mid \{\text{null}\}$

- **empty**: $s = \text{null}$
- **full**: $s \neq \text{null}$

Transitions:
- **write(n) / ackwrite**: $\{s := n\}$
- **delete / ackdel**: $\{s := \text{null}\}$
- **read / out(s)**

Diagram representation of the state machine.
Representation of the State Machine as a Table

<table>
<thead>
<tr>
<th>State s</th>
<th>Input</th>
<th>State s</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>write(n)</td>
<td>n</td>
<td>ackwrite</td>
</tr>
<tr>
<td>n</td>
<td>read</td>
<td>n</td>
<td>out(n)</td>
</tr>
<tr>
<td>n</td>
<td>delete</td>
<td>null</td>
<td>ackdel</td>
</tr>
</tbody>
</table>
**Representation as a Mathematical State Machine**

State space: \( \Sigma = \mathbb{Z} \cup \{ \text{null} \} \)

Input set: \( E = \{ \text{read, delete} \} \cup \{ \text{write}(z): z \in \mathbb{Z} \} \)

Output set: \( A = \{ \text{ackwrite, ackdel} \} \cup \{ \text{out}(z): z \in \mathbb{Z} \} \)

Equations for the state transition function:

\[
\Delta : \Sigma \times E \rightarrow \Sigma \times A
\]

\[
\Delta(\text{null, write}(z)) = (z, \text{ackwrite})
\]

\[
\Delta(z, \text{read}) = (z, \text{out}(z))
\]

\[
\Delta(z, \text{delete}) = (\text{null, ackdel})
\]
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 \(\Delta\)

\[\Delta : \Sigma \times \mathbb{E} \rightarrow \Sigma \times \mathbb{A}\]

in case of a state machine with input/output:

- events (inputs \(E\)) trigger the transitions and events (outputs \(A\)) are produced by them respectively:

\[\Delta : \Sigma \times \mathbb{E} \rightarrow \Sigma \times \mathbb{A}\]

in the case of nondeterministic machines:

\[\Delta : \Sigma \times \mathbb{E} \rightarrow \mathcal{P}(\Sigma \times \mathbb{A})\]

- 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 
\[
(\sigma_{i+1}, a_{i+1}) \in \Delta(\sigma_i, e_{i+1})
\]
In this manner we obtain computations of the form

$$\sigma_0 \xrightarrow{a_1/b_1} \sigma_1 \xrightarrow{a_2/b_2} \sigma_2 \xrightarrow{a_3/b_3} \sigma_3 \ldots$$

For each initial state $\sigma_0 \in \Sigma$ we define a function

$$F_{\sigma_0} : \tilde{I} \rightarrow \wp(\tilde{O})$$

with

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

$F_{\sigma_0}$ denotes the interface behavior of the transition function $\Delta$ for the initial state $\sigma_0$.

Furthermore we define

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

where:

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

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

• A Mealy machine \((\Delta, \Lambda)\) with
  \[\Delta : \Sigma \times E \rightarrow \wp(\Sigma \times A)\]
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 causal.
Interface Abstraction

• 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
The interface behavior of F can be exemplarily illustrated by an interaction diagram (sequence diagram) as follows:
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

\[
\text{Abs}((\Delta_1, \Lambda_1)) = \text{Abs}((\Delta_2, \Lambda_2))
\]
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
Sub-Services
Key questions

• What does it mean that
  ◊ a system (component) $S$ offers a number of services/functions $E$?
  ◊ The projection of the system $S$ to the syntactic interface of service $E$ is (a refinement of) the service $E$!
  ◊ A system can take the role of the service if it offers the service.

• Can we understand the behaviour of a multi-functional system as the hierarchy of the services it offers?

• How can we capture the dependencies between the services?
Services

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

F : \overline{I} \rightarrow \varnothing(\overline{O}) \textit{semantic interface} for (I \rightarrow O) with \textit{timing property addressing strong causality}

(let x, z \in \overline{I}, y \in \overline{O}, t \in IN, \text{dom}(F) = \{x: F(x) \neq \varnothing\}):

\begin{align*}
x, z \in \text{dom}(F) \land x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow t+1: y \in F(x)\} = \{y \downarrow t+1: y \in F(z)\}
\end{align*}

A service shows a partial behavior
Example: Service interface specification

A queue service

Queue

| in | x: Data ∪ {®} |
| out | y: Data |
| y ≤_{pre} x | Data ∧ {®} #x = #y |

x ≤_{pre} y ⇔ (∃ z: x^z = y)

M#x denotes the number of occurrences of elements of the set M in stream x
Syntactic sub-interfaces

A typed channel set $C'$ is called a \textit{sub-type} of a typed channel set $C$ if

- $C'$ is a subset of $C$
- The message types of the channels in $C'$ are subsets of the message sets of these channels in $C$

We write then

$C'$ \textit{subtype} $C$

Then we denote for the channel history $x \in IH[C]$ by

$x|C' \in IH[C']$

the restriction of $x$ to the channels and messages in $C'$
Sub-types between interfaces

For syntactic interfaces \((I \triangleright O)\) and \((I' \triangleright O')\) where

\(I'\) **subtype** \(I\) and \(O'\) **subtype** \(O\)

we call \((I' \triangleright O')\) a **sub-type** of \((I \triangleright O)\) and write:

\((I' \triangleright O')\) **subtype** \((I \triangleright O)\)

Then we define for a behavior function \(F \in IF[I \triangleright O]\) its **projection**

\(F^\dagger(I' \triangleright O') \in IF[I' \triangleright O']\)

to the syntactic interface \((I' \triangleright O')\) by (for all \(x' \in IH[I']\)):

\[
F^\dagger(I' \triangleright O')(x') = \{y|O': \exists x \in IH[I]: x' = x|I' \land y \in F(x)\}
\]

The projection is called **faithful**, if for all \(x \in \text{dom}(F)\)

\[
F(x)|O' = (F^\dagger(I' \triangleright O'))(x|I')
\]
Conclusion Services and Sub-Services

• Services and sub-services provide a structuring concept for multi-functional systems
• they allow to break down the functionality in a family of independent or only weakly dependent functions
  ◊ which can be specified and analysed independently
  ◊ their dependencies can be identified and analysed
Composing Systems
Composition and Decomposition of Systems

\[ F_1 \in \text{IF}[I_1 \triangleright O_1] \]
\[ F_2 \in \text{IF}[I_2 \triangleright O_2] \]
\[ C_1 = O_1 \cap I_2 \]
\[ C_2 = O_2 \cap I_1 \]
\[ I = I_1 \setminus C_2 \cup I_2 \setminus C_1 \]
\[ O = O_1 \setminus C_1 \cup O_2 \setminus C_2 \]

\[ F_1 \otimes F_2 \in \text{IF}[I \triangleright O], \]

\[ (F_1 \otimes F_2).x = \{ z \mid O: x = z \mid I \land z \mid O_1 \in F_1(z \mid I_1) \land z \mid O_2 \in F_2(z \mid I_2) \} \]
Interface specification composition rule

\[ \text{F1} \times \text{F2} \]

\begin{align*}
\text{in} & \quad x_1, z_{21}: T \\
\text{out} & \quad y_1, z_{12}: T \\
\text{P1} & \\
\hline
\text{F2} & \\
\text{in} & \quad x_2, z_{12}: T \\
\text{out} & \quad y_2, z_{21}: T \\
\text{P2} & \\
\hline
\end{align*}

\[ \exists z_{12}, z_{21}: \text{P1} \land \text{P2} \]
Composition of the two state machines

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

$$\Delta_k : \Sigma_k \times (I_k \rightarrow M^*) \rightarrow \emptyset (\Sigma_k \times (O_k \rightarrow M^*))$$

We define the composed state machine

$$\Delta : \Sigma \times (I \rightarrow M^*) \rightarrow \emptyset (\Sigma \times (O \rightarrow M^*))$$

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

$$\Delta = \Delta_1 || \Delta_2$$

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

Interface abstraction distributes for state machines over composition

\[
\text{Abs}(\langle \Delta_1, \sigma_1 \rangle \parallel \langle \Delta_2, \sigma_2 \rangle ) =
\text{Abs}(\langle \Delta_1, \sigma_1 \rangle) \otimes \text{Abs}(\langle \Delta_2, \sigma_2 \rangle)
\]
Conclusion Composition

• Given a set of components $K$ with $F_k \in IF$ we write

$$\otimes \{F_k : k \in K \}$$

for the interface behavior of the architecture formed by

$$F_1 \otimes F_2 \otimes F_3 \otimes \ldots$$

• The operator $\otimes$ is parallel composition including feedback

• The operator $\otimes$ is logically represented by logical “and” for the assertions and existential quantification for channel hiding

• Causality
  
  ◊ reflects the flow of time
  
  ◊ guarantees unique fixpoints of feedback loops in the case of deterministic systems
Strong and weak causality

Strong causality

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

Weak causality

\[ x \downarrow t = z \downarrow t \Rightarrow \{ y \downarrow t: y \in F(x) \} = \{ y \downarrow t: y \in F(z) \} \]
Conclusion on System Modelling

• A system
  ◊ has a syntactic interface
  ◊ an interface behaviour

• A system can be implemented by
  ◊ an architecture, where all its components are implemented
  ◊ a state machine

• An implemented system with a specified interface behaviour is correct, if
  ◊ the architecture has a black box behaviour that is a refinement of the specified behaviour
  ◊ the state machine has an interface behaviour that is a refinement of the specified behaviour

• A system implemented by an architecture can be refined by refining its components
Refining Systems
Refinement

- The idea of system refinement is that systems are developed
  - by a sequence of development steps
  - each step produces a refined system
  - there is a refinement relation between the current system and the produced system
- The refinement relation
  - is a relation between systems and their description
- 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
Horizontal Refinement

Compositionality of refinement

\[ \forall k : F_k \cong_{IF} \hat{F}_k \]

\[ \otimes \{ F_k : k \in IK \} \cong_{IF} \otimes \{ \hat{F}_k : k \in IK \} \]

\[ \forall x \in \bar{I} : \hat{F}.x \subseteq F.x \]

we write

\[ F \cong_{IF} \hat{F} \]
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

Refinement of State Machines

Given two state machines \((k, \kappa)\) for \(k := 1, 2\) where \(\kappa\) is a set of pairs \((\#_0, y_0)\) and \(!\kappa\) is a state transition function \(!\kappa: (\#_k \& M_{\kappa}) \& (\#_k \& M_{\kappa})\)

we call \(!\kappa_2\) a refinement of \(!\kappa_1\) if there exists a mapping \(\text{abs}: \#_2 \& \#_1\) such that

\[
\{(\text{abs.}\#_0, A_{\kappa_0}.y_0) : (\#_0, y_0) \subseteq \#_1\}
\]

and for each reachable state \# of the state machine \((!\kappa_2, \kappa_2)\) we have

\[
!\kappa_2(\#_2, y_0) \subseteq A_{\kappa_0}.y_0 \subseteq R_{\kappa_1}(\text{abs.}\#_0, A_{\kappa_0}.y_0)
\]
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
Informal requirements

Requirements Engineering
Validation

Formalized system requirements in terms of service taxonomies

Architecture design
Architecture verification
S \leftarrow S_1 \otimes S_2 \otimes S_3 \otimes S_4

Integration
R = R_1 \otimes R_2 \otimes R_3 \otimes R_4

Component implementation verification
R_1 \Rightarrow S_1
R_2 \Rightarrow S_2
R_3 \Rightarrow S_3
R_4 \Rightarrow S_4

System delivery
System verification
R \Rightarrow S

deliver

realization

architecture

integration

R = R_1 \otimes R_2 \otimes R_3 \otimes R_4

S

R_1
R_2
R_3
R_4

Manfred Broy
System Modelling, SAASE, San Diego October 2009
Layered Architectures
Characteristics of layered architectures

- A layered architecture consists of a family of layers
- A layer is a component with an interface split into two sub-interfaces
- Each layer has two interfaces:
  - one to the layer below
  - one to the layer above
Service layers

For service interfaces

$$(I \to O) \quad \text{export/ upward interface} \quad (O' \to I') \quad \text{import/ downward interface}$$

where $I \cap O' = \emptyset$ and $O \cap I' = \emptyset$;

service layer $L$

$$L \in IF[I \cup O' \to O \cup I']$$

$$(I \to O/O' \to I') \quad \text{syntactic service layer interface}$$

$IL[I \to O/O' \to I']$. \quad \text{set of layers}
Composition of Service Layers

\[ F' \in \text{IF}[I' \triangleright O'] \quad \text{import service} \]
\[ L \in \text{IL}[I \triangleright O/O' \triangleright I'] \quad \text{service layer} \]

\[ L[I'\leftrightarrow O']F' \quad \text{composition of layer with service} \]

\[ F \in \text{IF}[I \triangleright O] \quad \text{export service} \]

\[ F = L[I'\leftrightarrow O']F' \]

\[ L \in \text{IL}[I \triangleright O/O' \triangleright I'] \]
\[ L' \in \text{IL}[O' \triangleright I'/O'' \triangleright I''] \]

\[ L[I'\leftrightarrow O']L' \quad \text{composition of layers - a layer in IL}[I \triangleright O/O'' \triangleright I''] \].
Layered architectures

\[ F_j \in \text{IF}[I_j \mapsto O_j] \]  family of export services for \( 0 \leq j \leq n \).

\[ F_{j+1} \in \text{IF}[I_{j+1} \mapsto O_{j+1}] \]  export service

\[ F_{j+1} = L_{j+1}[I_j \leftrightarrow O_j]F_j \]

\[ F_j \in \text{IF}[I_j \mapsto O_j] \]  import service

\[ G_{j+1} \in \text{IF}[O_j \mapsto I_j] \]  downward service

\[ G_{j+1} = L_{j+1}^\dagger(O_j \mapsto I_j) \].
The comprehensive model

- Conceptional architecture
  - Usage function hierarchy
  - Service taxonomy
  - Logical architecture

- Technical architecture
  - Tasks
    - T1
    - T2
    - T3
    - T4
    - ...

- Software architecture
  - Deployment
    - T1
    - T2
    - T3
    - T4
    - ...

- Hardware architecture
State of Research on Service Taxonomies

• Theory worked out and stable
  ◊ System interface behaviour as combination of services
  ◊ Formal foundation of use cases

• Pragmatic description techniques for function hierarchies under development

• Method for development of function hierarchies
  ◊ Identify feature hierarchy
    • Names of services (use cases)
  ◊ Specify features in isolation
    • By logical specifications
    • By interaction diagrams
    • By partial state machines
  ◊ Identify dependencies
    • Use standard dependency relations
  ◊ Specify dependencies
    • Specify dependencies by logical messages

• Case studies: Application to mobile phones and cars
Specifying Services/Layered Architectures

Specify

\[ F_j \in IF[I_j \rightarrow O_j] \]
family of export services

\[ L_j \in IL[I_j \rightarrow O_j/O_{j+1} \rightarrow I_{j+1}] \]
family of layers

Prove

\[ F_{j+1} \Rightarrow L_{j+1}[I_j \leftrightarrow O_j]F_j \]

\[ L_{j+1} \text{ guarantees } F_{j+1} \text{ under import of } F_j \]
A communication connection is a special case of a service layer.
This way we can build communication layers
Refinement

- A layer refinement pair are two layers that form the time independent identity

Two layers $L \in IL[I \triangleright O/O' \triangleright I']$ and $L' \in IL[I' \triangleright O'/\triangleright I]$ are called a refinement pair for $(I \triangleright O / O \triangleright I)$ if

$$L[I' \leftrightarrow O']L' = Id(I \triangleright O)$$
Layered protocols

Remember

\[ I_k \rightarrow O_k \]
\[ \text{Service layer } L_k \]
\[ I_{k-1} \rightarrow O_{k-1} \]
\[ \text{Service layer } L_{k-1} \]
\[ I_{k-2} \rightarrow O_{k-2} \]
\[ \text{Service layer } L'_{k-1} \]
\[ I_{k-1} \rightarrow O_{k-1} \]
\[ \text{Service layer } L'_k \]
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
Open Issues

- Probability
- Non-functional properties
- Modelling of
  - hardware issues
  - mechanical aspects