Engineering Dependable Software Systems

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 Munich, Germany
Dependability (Wikimedia)

**Attributes** - A way to assess the Dependability of a system

**Threats** - An understanding of the things that can affect the Dependability of a system

**Means** - Ways to increase the Dependability of a system
### Management of safety requirements

- Hierarchical structure
- Traceability
- Completeness
- External consistency
- ...

<table>
<thead>
<tr>
<th>Safety requirement 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>unambiguous</td>
</tr>
<tr>
<td>comprehensible</td>
</tr>
<tr>
<td>atomic</td>
</tr>
<tr>
<td>internally consistent</td>
</tr>
<tr>
<td>feasible</td>
</tr>
<tr>
<td>verifiable</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Safety requirement 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>unambiguous</td>
</tr>
<tr>
<td>comprehensible</td>
</tr>
<tr>
<td>atomic</td>
</tr>
<tr>
<td>internally consistent</td>
</tr>
<tr>
<td>feasible</td>
</tr>
<tr>
<td>verifiable</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Traceability Use Case: ISO 26262 – Functional Safety

• The management of safety requirements includes
  ◊ managing requirements,
  ◊ obtaining agreement on the requirements,
  ◊ obtaining commitments with those implementing the requirements, and
  ◊ maintaining traceability

• During the development of the software architectural design the following shall be considered:
  ◊ a) the verifiability of the software architectural design;
    NOTE This implies bi-directional traceability.

• The software unit design and implementation shall be verified in accordance with ISO 26262-8:
  ◊ b) the completeness regarding the software safety requirements and the software architecture through traceability;
A Logical Approach to Systems Engineering Artifacts and Traceability: From Requirements to Functional and Architectural Views

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 Munich, Germany
Content and Motivation

• Presentation of key artifacts in systems engineering in logic
  ◊ Assertions about the system

• System models and their representation in logic
  ◊ Interfaces
  ◊ Architectures

• Key artifacts in systems engineering
  ◊ System level requirements
  ◊ Functional specification
  ◊ Architecture

• Concepts relating assertions: logical dependence relations

• Concepts for relating artifacts
  ◊ Understanding the logical dependencies between artifacts
  ◊ Traceability: Intra- and inter-artifact links and traces
Assertions
## Assertions

- A logical predicate $p$ over a universe $D$ is a mapping
  \[ p: D \rightarrow \mathbb{I} \mathbb{B} \]
  where $D$ is a mathematical set also called the universe of discourse.
- Often the elements $d \in D$ can be characterized by a set of attributes
  \[ x_i: D \rightarrow T_i \text{ for } 1 \leq i \leq n \]
  where
  \[ T_i \] are the (data) types for these attributes and
  \[ n \] is the number of attributes.
Example: Assertions

- For a simple universe of discourse Car representing cars, consider attributes such as
  
  \[\text{length}: \text{Car} \rightarrow \text{IN} \]
  
  \[\text{number_of_seats}: \text{Car} \rightarrow \text{IN} \]
  
  \[\text{speed}: \text{Car} \rightarrow \text{IN} \]
  
  \[\text{situation}: \text{Car} \rightarrow \{\text{city, country, high_way}\} \]

- Based on the attributes, given \(d \in \text{Car}\), we write logical expressions such as
  
  \[\text{speed}(d) \geq 50 \land \text{situation}(d) = \text{city} \]

- This notation can be simplified for a fixed car \(d\):
  
  \[\text{speed} \geq 50 \land \text{situation} = \text{city} \]

- Such a logical expression referring to the attributes of the elements of the considered universe is called \textit{assertion}. 
Typing attributes

• In an assertion like
  \[ \text{speed} \geq 50 \land \text{situation} = \text{city} \]
  the attributes have types.

• Sometimes it is useful to indicate the types of attributes explicitly
  \[(\text{speed: IN, situation: \{city, country, high\_way\}}): \]
  \[ \text{speed} \geq 50 \land \text{situation} = \text{city} \]
Notation

• For assertions $Q$ the following shorthand notation is used:

\[ \forall X:Q \quad \text{for } \forall x_1, ..., x_n: Q \]

\[ \exists X:Q \quad \text{for } \exists x_1, ..., x_n: Q \]

where $X = \{x_1, ..., x_n\}$ are free variables in $Q$

\[ \forall Q \quad \text{iff } Q \equiv \text{true} \quad \text{e.g. } \forall x_1, ..., x_n: Q \]

where $x_1, ..., x_n$ are all the free variables in $Q$

\[ \exists Q \quad \text{iff } \neg \forall \neg Q \]
Language of assertions

• Given a signature $\Sigma$ of attributes by

$$\text{LA}(\Sigma)$$

we denote the assertion language over signature $\Sigma$ which is the set of assertions that can be formulated over signature $\Sigma$.

• Assertions are Boolean expresses and therefore all the logical operators can be applied to them.
Formalizing Domains
From the informal to the formal

• In the beginning, properties of the universe are formulated in natural language, in general
  “The airbag is activated within 200 msec whenever the crash sensor indicates a crash”

• The step to the formal means
  ◇ Derivation of a “data” model: Introducing a set of attributes
  ◇ Capturing properties by assertions in terms of these attributes

• This step into formalization has two aspects
  ◇ Abstraction: the attributes can only address a limited set of properties
  ◇ Precision: informal properties are made precise
    This includes
      • Decisions: there are usually several ways to make an informal property precise
Assertions about Systems

• Assertions and languages of assertions can be built for many different universes – problem domains

Examples:
- Airplanes
- Medical devices
- Cars
- Banking
- ...

• We are aiming at assertion languages for systems with emphasis on software systems and systems with embedded software
Remark: difference between assertions and propositions

- An assertion $P$ defines a property
  - By the attributes $P$ it formulates a property about a system
    $\text{situation} = \text{city} \Rightarrow \text{speed} \leq 50$
  - A car may have this property or not

- A proposition is either true or false
  - It either holds or not
    $\forall (\text{situation} = \text{city} \Rightarrow \text{speed} \leq 50)$
  - This proposition is true if the specified property is true for all cars
Remark: difference between axioms and specifications

• Using an assertion $P$ as a specification means that $P$ specifies a property that is required for the system under development
  ◊ By the attributes in $P$ it formulates a specification about systems
    \[ \text{situation} = \text{city} \Rightarrow \text{speed} \leq 50 \]
  ◊ A car may fulfill this specification or not

• An axiom is an assertion $P$ that states a property about all systems
  ◊ It holds for all systems
    \[ \forall (\text{speed} \leq 500) \]
  ◊ Then $P$ is a trivial specification

Note: The axioms describe the universe of systems under consideration, the assumptions about the considered universe of systems – they form the problem domain theory
Artifacts - Structure and Content
In systems development typically a large number of descriptions and statements about systems are worked out.

This information is captured in documents we call artifacts.

Examples of artifacts:
- List of requirements
- Architectures description
- Code listings
- Collection of test cases
- ...
Artifacts - Structure and Content

- An artifact is a development document
- An artifact has structure and content
- An artifact contains content that is structured into
  - (finite) sets of content chunks as well as
  - finite sets of finite sets of content chunks and so on.
- This way we get nested sets of content chunks forming content hierarchies.
- Typically content chunks are informal statements of assertions about the system under development (or more generally, its development process etc.)
Illustrating Examples: Content Chunks

- **System level requirements (functional requirements)**
  “the car must not increase its speed without user’s control”

- **System level functional specification**
  “the function acc (adaptive cruise control) accelerates the car up to the speed selected by the user, provided no obstacles are recognized in front”

- **Architecture specification**
  “the radar signal based sensor measures the distance to the car in front and sends it to the acc monitor every 100 ms”
From content chunks to assertions

• To go from content chunks such as
  “the car must not increase its speed without user’s control”
  “the function acc (adaptive cruise control) accelerates the car up to
the speed selected by the user, provided no obstacles are recognized
in front”

needs modeling and formalization.

This involves the following steps

• Formalizing the elements of the universe – elicitation of
the problem domain
  ◊ Selecting the attributes
  ◊ Defining basic propositions (called the problem domain theory)
    \( \forall (\text{speed} \leq 500) \)

• Expressing the informal statement by an assertion
Observation about the step of formalization

• The problem domain model has to be chosen in a way, that the informal statement can be captured
  ◦ “Expressiveness”
  ◦ This may require sophisticated models (talking about time, space, interaction, reaction, intension, ...)

• There might be several ways to formalize an informal statement
  ◦ Eliminating linguistic ambiguity

• Usually it is not a good idea that all content chunks are formalized
Given two assertions $P$ and $Q$; what does logical dependency mean?

Relating Assertions
Relating Assertions to Assertions - Implication

• Two assertions

\[ P, Q \]

are in an *implication* relation if

\[ \forall (P \Rightarrow Q) \]

or vice versa

\[ \forall (Q \Rightarrow P) \]

• Related relations are

\[ \forall (\neg Q \Rightarrow P) \]

or

\[ \forall (P \Rightarrow \neg Q) \]
Negating the independence conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Negation</th>
<th>Result</th>
<th>Result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
</tr>
<tr>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(Q ∨ P)</td>
</tr>
<tr>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
</tr>
<tr>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
<td>(P ∨ Q)</td>
</tr>
</tbody>
</table>

If every of the following four relations

\[
\begin{align*}
\exists (P \land Q) \\
\exists (\neg P \land Q) \\
\exists (P \land \neg Q) \\
\exists (\neg P \land \neg Q)
\end{align*}
\]

holds then we call assertions \( P \) and \( Q \) *logically independent.*
Example: Independence

- Consider the following assertions
  \[ P: \text{situation} = \text{city} \]
  \[ Q: \text{speed} \leq 50 \]
- Whether these assertion are independent depends on the problem domain theory
  ◦ If we assume (as part of the problem domain theory)
    \[ \forall (\text{situation} = \text{city} \Rightarrow \text{speed} \leq 50) \]
    P and Q are not independent
  ◦ If we assume no properties as part of the problem domain theory,
    P and Q are independent
The Cases of Dependence

**Tab.** Logical Consequences of Negations of the Conditions of Logical Independence

<table>
<thead>
<tr>
<th>$p \land q$</th>
<th>$p \land q$</th>
<th>$p \land q$</th>
<th>$p \land q$</th>
<th>Implies</th>
<th>consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>independence</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>$d$: $p(d) \lor q(d)$</td>
<td>unavoidance</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>$d$: $p(d) \iff q(d)$</td>
<td>implication</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>$d$: $p(d) \Rightarrow q(d)$</td>
<td>implication</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>$d$: $p(d)$</td>
<td>implication, unavoidance</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>$d$: $p(d) \iff q(d)$</td>
<td>equivalence</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>$d$: $p(d) \land q(d)$</td>
<td>$p$ and $q$ tautologies</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>$d$: $p(d) \lor q(d)$</td>
<td>mutual exclusion</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>$d$: $p(d) \iff q(d)$</td>
<td>antivalence</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>$d$: $q(d)$</td>
<td>implication, mutual exclusion</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>$d$: $p(d) \land q(d)$</td>
<td>implication, mutual exclusion, unavoidance</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>$d$: $p(d)$</td>
<td>implication, mutual exclusion</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>$d$: $p(d) \land q(d)$</td>
<td>implication,</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>$d$: $p(d) \land q(d)$</td>
<td>$p$ and $q$ tautologies</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>inconsistency</td>
</tr>
</tbody>
</table>
The Lattice of Dependence

True

p ∨ q

p ⇔ q

p ⇒ q

q ⇔ p

p ∨ q

p ∧ q

p

q

False
Inconsistency

• Assertions $P$ and $Q$ are called *inconsistent* if
  \[ \neg \exists (P \land Q) \]

• If assertions $P$ and $Q$ are inconsistent, then both propositions
  \[ \forall (P \Rightarrow \neg Q) \]
  \[ \forall (Q \Rightarrow \neg P) \]
  hold, i.e. they are logically dependent.
Logical Overlap

- Two assertions $P$ and $Q$ are called *logically overlapping* iff
  \[ \neg \forall (P \lor Q) \]
  which is equivalent to the statement,
  \[ \exists (\neg P \land \neg Q) \]

- Then there is a non-trivial property $R$
  ◊ (nontrivial means that $\neg \forall R$ holds)
  ◊ that is implied both by assertion $P$ and by assertion $Q$; i.e.
  \[ \forall (P \Rightarrow R) \text{ and } \forall (Q \Rightarrow R) \]

- We choose the strongest assertion $R$
  ◊ that is implied both by assertion $P$ and by assertion $Q$ as follows:
  \[ R = P \lor Q \]

- Property $R$ is not trivially true (i.e. $\exists \neg R$) iff assertions $P$ and $Q$ are overlapping.
Logical Overlap

• Not overlapping assertions are logically not independent, since

$$\forall (P \lor Q)$$

which transforms to

$$\neg \exists (\neg P \land \neg Q)$$

and to

$$\forall (\neg P \Rightarrow Q)$$
$$\forall (\neg Q \Rightarrow P)$$

• In other terms, independent assertions are always overlapping.
Example: overlapping assertions

• The assertions:
  
  P: speed ≤ 100
  Q: speed ≥ 50

  are not overlapping:
  \[ \forall (\text{speed} \leq 100 \lor \text{speed} \geq 50) \]

• The assertions:

  P: speed ≥ 100
  Q: speed ≤ 50

  are overlapping:
  \[ \neg \forall (\text{speed} \geq 100 \lor \text{speed} \leq 50) \]
  \[ \exists (\text{speed} \leq 100 \land \text{speed} \geq 50) \]
Negating the independence conditions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Negation</th>
<th>Result</th>
<th>Result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
</tr>
<tr>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(Q P)</td>
</tr>
<tr>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
</tr>
<tr>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
<td>(P Q)</td>
</tr>
</tbody>
</table>

System Properties at Different Levels of Abstractions:
Relating Views
Example: Relating Levels of Abstraction

Logical_level

... crash ⇒ air_bag
... Technical_level

... crash_sensor ⇒ activate_air_bag
...
Example: Relating Levels of Abstraction

Logical_level

... 

crash \Rightarrow air\_bag 

... 

Technical_level

... 

crash\_sensor \Rightarrow activate\_air\_bag 

... 

Translator

... 

crash \Leftrightarrow crash\_sensor 

air\_bag \Leftrightarrow activate\_air\_bag 

...
Translators between Levels of Abstractions

- A specification given by a set $S_1 \subseteq LA(\Sigma_1)$ of assertions over some attribute signature $\Sigma_1$ is translated into
- a specification $S_2 \subseteq LA(\Sigma_2)$ over some attribute signature $\Sigma_2$
  ◇ where signatures $\Sigma_1$ and $\Sigma_2$ only partially overlap or are disjoint
- by a set $T$ of assertions formulated over signatures $\Sigma_1$ and $\Sigma_2$. 
Translators between Levels of Abstractions

For a translation we require that for every assertion
\[ a_1 \in \text{LA}(\Sigma_1) \]
over signature \( \Sigma_1 \) there exists an assertion
\[ a_2 \in \text{LA}(\Sigma_2) \]
over \( \Sigma_2 \) such that the following formula is valid:
\[(\land T) \Rightarrow (a_1 \Leftrightarrow a_2)\]

• Then the set \( T \) is called a \textit{translator from signature} \( \Sigma_1 \) \textit{to signature} \( \Sigma_2 \).

• A set \( S_1 \) of assertions is called a \textit{refinement} of a set \( S_2 \) of assertions according to translator \( T \) if
\[(\land T) \land (\land S_1) \Rightarrow \land S_2\]
Translators between Levels of Abstractions

• If $T$ is free of contradictions $T$ is called **consistent translator**.

• If for every assertion $a_1 \in \text{LA}(\Sigma_1)$ and every set $S_1$ of assertions formulated over signature $\Sigma_1$ and for every assertion $a_2 \in \text{LA}(\Sigma_2)$ and every set $S_2$ of assertions formulated over signature $\Sigma_2$

  $$
  [(\wedge T) \land (\wedge S_1) \Rightarrow a_1] \iff [(\wedge S_1) \Rightarrow a_1] \\
  [(\wedge T) \land (\wedge S_2) \Rightarrow a_2] \iff [(\wedge S_2) \Rightarrow a_2]
  $$

  $T$ is called **unbiased translator between signatures $\Sigma_1$ and $\Sigma_2$**.
Why translators are useful?

- Translators relate logical assertions to technical/physical assertions.
- They force to make explicit assumptions behind physical/technical designs:
  - As part of specifications
  - To validate them – to discover invalid assumptions.
Goal:
Description of views of systems as captured by artifacts by sets of assertions

Logical Basis: Specifying Systems by Assertions
An industrial press system

What is a system?
An industrial press system

Here’s the problem world

A slide due to Michael Jackson
Here’s the machine

An industrial press system

Here’s the user interface

Here’s the problem world

Extending a slide due to Michael Jackson
System and its context

- Physical World
- Operational context
- Cyberspace Services & Data
- HMI
- System
- Context system

MOD SumScho 2012 Manfred Broy
A system has

- a system \textit{boundary} that determines
  - what is part of the systems and
  - what lies outside (called its context)
- an \textit{interface} (determined by the system boundary), which determines,
  - what ways of interaction (actions) between the system and its context are possible (static or \textit{syntactic interface})
  - which behavior the system shows from view of the context (\textit{interface behavior}, dynamic interface, interaction view)
- a structure and distribution with an internal structure, given
  - by its structuring in sub-systems (\textit{sub-system architecture})
  - by its states and state transitions (\textit{state view}, state machines)
- \textit{quality profile}
- the views use a \textit{data model}
- the views may be documented by adequate models
Interfaces
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 \((I \triangleright O)\)

data stream of type \(T\)

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

Valuation of channel set \(C\)

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

Interface behavior for syn. interface \((I \triangleright O)\)

\[ [I \triangleright O] = \{ \tilde{I} \rightarrow \varphi(\tilde{O}) \} \]

Interface specification

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

represented as interface assertion \(S\) - logical formulae with channel names as attributes of type stream
Interface Assertion

• Given a syntactic interface $(I \rightarrow O)$ with
  ◊ a set $I$ of typed input channels and
  ◊ a set $O$ of typed output channels,
  The channels form attributes in assertions.

• an interface assertion is a logical formula with the channel identifiers in $I$ and $O$ as free logical variables denoting streams of the respective types.
Example: Component interface specification

A transmission component TMC

TMC

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

x ~ y ≡ (∀ m ∈ T: {m} ⊗ x = {m} ⊗ y)

Input channel

Spec name

Output channel

y:T

x:T

TMC

x ~ y

Interface assertion
Example: Component interface specification – Airbag Controller

An air bag controller

\[
\begin{array}{l}
\text{AB}_\text{Cont} \\
\text{in} \quad x: T \\
\text{out} \quad y: T \\
\text{x} > 200 > y
\end{array}
\]

\[
x > 200 > y \equiv (t \in \text{Time} : \text{crash}_\text{sig} \in x(t) \iff \text{act}_\text{airbag} \in y(t+200))
\]
Can we give purely logical specifications of architecture?

Architectures
Specifying Architectures

Syntactic Architecture
Composition and Decomposition of Systems

\[ F_1 \quad [I_1 \Box O_1] \]
\[ F_2 \quad [I_2 \Box O_2] \]

\[ C_1 = O_1 \quad I_2 \]
\[ C_2 = O_2 \quad I_1 \]
\[ I = I_1 \setminus C_2 \quad I_2 \setminus C_1 \]
\[ O = O_1 \setminus C_1 \quad O_2 \setminus C_2 \]

\[ F_1 \quad F_2 \quad [I \Box O], \]

\[ (F_1 \Box F_2).x = \{ z|O: x = z|I \quad z|O_1 \quad F_1(z|I_1) \quad z|O_2 \quad F_2(z|I_2) \} \]
Interface specification composition rule

\[
\begin{align*}
\text{F1} & : \begin{array}{l}
\text{in} \quad x_1, z_{21}: T \\
\text{out} \quad y_1, z_{12}: T
\end{array} \\
\text{P1} \\
\text{F2} & : \begin{array}{l}
\text{in} \quad x_2, z_{12}: T \\
\text{out} \quad y_2, z_{21}: T
\end{array} \\
\text{P2} \\
\text{F1}_\otimes \text{F2} & : \begin{array}{l}
\text{in} \quad x_1, x_2: T \\
\text{out} \quad y_1, y_2: T \\
\exists \quad z_{12}, z_{21}: \text{P1} \land \text{P2}
\end{array}
\end{align*}
\]
Specifying Architectures

Syntactic Architecture
Set of Composable syntactic Interfaces

A set of component names $K$ with a finite set of interfaces $(I_k, O_k)$ for each identifier $k \in K$ is called *composable*, if the following propositions hold:

- sets of input channels $I_k, k \in K$, are pairwise disjoint,
- sets of output channels $O_k, k \in K$, are pairwise disjoint,
- the channels in $\{c \mid I_k: k \in K\}$ and $\{c \mid O_k: k \in K\}$ have consistent channel types in $\{c \mid I_k: k \in K\}$ and $\{c \mid O_k: k \in K\}$. 
Specifying Architectures

Syntactic Architecture
A syntactic architecture $A = (K, \cdot)$ with interface $(I_A, O_A)$ is given by a set $K$ of component names with composable syntactic interfaces $(k) = (I_k, O_k)$ for $k \in K$.

- $I_A = \{c \mid I_k: k \in K\} \cup \{c \mid O_k: k \in K\}$ set of input channels,
- $D_A = \{c \mid O_k: k \in K\}$ set of generated channels,
- $O_A = D_A \setminus \{c \mid I_k: k \in K\}$ set of output channels,
- $Z_A = D_A \setminus O_A$ set of internal channels
- $C_A = \{c \mid I_k: k \in K\} \cup \{c \mid O_k: k \in K\}$ set of all channels

By $(I_A, D_A)$ we denote the syntactic internal interface and by $(I_A, O_A)$ we denote the syntactic external interface of the architecture.
Definition. Interpreted Architecture
An interpreted architecture \((K, y)\) for syntactic architecture \((K, x)\) associates an interface behavior \((k) \ [I_k O_k]\) , where \((k) = (I_k O_k)\), with every component \(k \in K\).

Definition. Specified Architecture
A specified architecture \((K, z)\) for syntactic architecture \((K, x)\) associates an interface assertion \((k)\) with every syntactic interface \((k) = (I_k O_k)\) and every component \(k \in K\).
Interface Behavior of Interpreted Architectures: Glass Box View

For an interpreted architecture $A$, the glass box interface behavior $[\ ] A \ [I_A \sqcap D_A]$ is given by (let $(k) = F_k$):

$$([\ ] A)(x) = \{ y \in D_A : z \in C_A : x = z|I_A \quad y = z|D_A \quad k \in K : z|O_k \in F_k(z|I_k) \}$$

where the operator $|$ denotes the usual restriction operator.
In a black box view $A \ [I_A \rightleftharpoons O_A]$ we hide internal channels

$$A = \{ y \ \tilde{O}_A : z \ \tilde{C}_A : x = z|I_A \ y = z|O_A \ k \ K : z|O_k \ F_k(z|I_k) \}$$

$A$ describes the interface behavior of the architecture.
Specifying Architectures by Assertions

Given composable systems $k \in K$ with specifying interface assertions $C_k$ the specification of the architecture reads

$$\land \{C_k : k \in K\}$$

and the interface assertion of the composed is given by hiding the internal channels in set $Z$

$$\exists Z : \land \{C_k : k \in K\}$$

Syntactic Architecture
Where are we?

- Representing **artifacts** by **logical concepts** - **assertions**
- **Relating** assertions by logical concepts
  - Dependency, Overlap, Inconsistency
  - Translators for assertions at different levels of abstraction
- Representing systems by assertions
  - What is a system?
  - How to define interfaces?
  - What is an architecture?
  - How to compose sub-systems by assertions in a modular way?
How to structure system functionality?

- Typically systems offer a rich functionality structured into functional features
- A functional feature can be represented by some interface behavior \([I \rightarrow O]\)
- Interface behavior of functional features can be composed the same way as sub-systems are composed
What is a feature ...

• Is a feature just a name ... ?
  ◦ If yes – for what?
  ◦ What is the relation of a feature tree to system models?

• What are relation between features?
  ◦ Feature interactions?
  ◦ Requires?
  ◦ Excludes?

• Is there a way to model features?
  ◦ How can we find and identify features of a system?
  ◦ What is the semantic interpretation of a feature tree?

• Is there a way to interpret relations between features such as feature interactions?
Functional (Behavioral) Features

We concentrate on functional (behavioral) features!

◊ These are at the level of system level interface behavior!

• A (functional) feature is a sub-function of a multi-functional system

◊ that serves a certain purpose
Modeling functional (behavioral) features

- We give an interpretation of the notion of a (functional) feature in terms of the system interface model $F \in [I \rightharpoonup O]$.
- The functionality of a system is modeled by its interface behavior.
- A (functional) feature is modeled by the ◦ projection applied to $F$ to the sub-interface $(I' \rightharpoonup O')$ resulting in a sub-interface behavior $F' \in [I' \rightharpoonup O']$
  ◦ absence of feature interactions is modeled by faithful projections
  ◦ feature interactions are modeled by modes
Feature Specification – Constructive Approach
Combining Functions without Interference

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

We want to combine them into a function $F_1 \oplus F_2$
Combining Functions without Interference

Their isolated combination

\[ F_1 \oplus F_2 \]

Diagram:

- \( F_1 \)
- \( F_2 \)
- Input: \( I_1, I_2 \)
- Output: \( O_1, O_2 \)
Combining Functions with Feature Interaction

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

$$F_1 \otimes F_2$$

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 $F_2$
Feature Specification – Analytic Approach
The system interface behaviour $F$ as specified by the system requirements specification is structured
- into a set $H$ of sub-interfaces for sub-functions $F_1$, ..., $F_h$
- for which a set $M$ of mode channels is introduced
- such that the functions can be specified independently nevertheless capture their feature interactions
- each $F_i$ sub-function is described by
  - a syntactic interface (including mode channels) and
  - an interface assertion $B_i$ for each function
A typed channel set $C'$ is called a *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

$$C' \text{ subtype } C$$

Then we denote for the channel history $x \tilde{C}$ by

$$x|C' \tilde{C}$$

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

\[ I' \text{ subtype } I \quad \text{and} \quad O' \text{ subtype } O \]

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

\[ (I' \triangleright O') \text{ subtype } (I \triangleright O) \]
From overall syntactic system interfaces...
to ...

F: System

\[ x_2 : T_2 \]
\[ x_3 : T_3 \]
\[ y_1 : T'_1 \]
\[ x_1 : T_1 \]
\[ y_2 : T'_2 \]
\[ x_4 : T_4 \]
\[ y_3 : T'_3 \]
\[ y_4 : T'_4 \]
\[ x_5 : T_5 \]
sub-interfaces

F: System

$x_1 : T_1$

$y_2 : T'_2$

$x_2 : T_2$

$y_3 : T'_3$

$x_4 : T_4$
Given:

$$(I' \triangleright O') \text{ subtype } (I \triangleright O)$$

define for a behavior function $F$ [I☐ O] its projection

$$F^\dagger(I'☐ O') [I'☐ O']$$

to the syntactic interface $(I'☐ O')$ by (for all $x'$ $\bar{I}'$):

$$F^\dagger(I'☐ O')(x') = \{ y|O': \ x \ bar{I}' : x' = x|I' \ y \ F(x) \}$$

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

$$F(x)|O' = (F^\dagger(I'☐ O'))(x|I')$$
Example: Component interface specification – Airbag Controller

An air bag controller

\[
\begin{array}{c}
\text{AB}_\text{Cont} \\
\text{in} \quad x: T \\
\text{out} \quad y: T \\
x > 200 > y
\end{array}
\]

\[
x > 200 > y \equiv ( t \in \text{Time}: \quad \text{crash}_\text{sig} \in x(t) \iff \text{act}_\text{airbag} \in y(t+200))
\]
Example: Component interface specification – Airbag Controller

An air bag controller

<table>
<thead>
<tr>
<th>AB_Cont</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in</strong> x: T, m: {on, off}</td>
</tr>
<tr>
<td><strong>out</strong> y: T</td>
</tr>
<tr>
<td>x &gt;200&gt; y</td>
</tr>
</tbody>
</table>

x >200> y \equiv ( \quad t \in \text{Time}: \\
(ON(m, t+199) \land \text{crash\_sig} \in x(t)) \iff \text{act\_airbag} \in y(t+200) \\
ON(m, t) = \text{if } t = 0 \text{ then false } \text{elif } \text{on} \in m(t) \text{ then true } \text{elif } \text{off} \in m(t) \text{ then false } \text{else } ON(m, t-1) \text{ fi}
Given composable feature interface specifications $h \in H$ with specifying interface assertions $B_h$ the assertion of the functional specification reads

$$\land \{B_h : h \in H\}$$

and the interface assertion of the composed is given by hiding the mode channels in $M$

$$\exists M : \land \{B_h : k \in H\}$$
An interpreted feature tree

\[ F_1, \ldots, n \]

\[ F_{1,2} \quad \ldots \quad F_{k,k+1} \quad \ldots \quad F_{n-1,n} \]

\[ F_1 F_2 \quad \ldots \quad F_k F_{k+1} \quad \ldots \quad F_{n-1} F_n \]
Artifacts: Structure and Content
Artifacts: Structure and Content

- An artifact is a (perhaps virtual) document that
  ◦ has a structure
  ◦ provides some content
- This indicates that an artifact presents content in some structured way
- The content has
  ◦ a syntactic form
  ◦ a semantics (obtained by the interpretation of the syntactic form)
- Content can be represented
  ◦ informally (using natural language, diagrams etc.)
  ◦ formally (using formulas – in our case assertions)
- We call pieces of contents “content chunks”
Formally: Named Content Chunks

- An **elementary named content chunk** is a pair \((\text{id}, \text{ct})\) where
  - \text{id} is a unique identifier (which may be typed) and
  - \text{ct} is an elementary content chunk

- A **composed named content chunk** is a set of
  - elementary content chunks, or
  - composed named content chunks
Hierarchies of named content chunks

• A composed named content chunk can represent a hierarchy of named content chunks:

(Air_Bag: Spec,

{(Air_Bag_Safety: Req, (ABReq1, “if crash then ...”)),

...},

(Air_Bag_Reliability: Req: { ... },

...},

)
Formally: Artifacts

• An artifact is a hierarchy of named content chunks
Artifacts and their Named Content Chunks

• Given an artifact $E = (id, ct)$ its set $NCoCh(E)$ of named content chunks is:

  $NCoCh((id, ct)) = \{(id, ct)\}$

  if $ct$ is an elementary content chunk

  $NCoCh((id, ct)) = \{(id, ct)\} \cup (\bigcup\{NCoCh(t): t \in ct\})$

  if $ct$ is a set of named content chunks

• $NCoCh(E)$ denotes the set of named content chunks – that are unique since the identifiers are unique

  ◊ allows forming finite hierarchies of named content chunks by nested sets of properties.

  ◊ By construction, each content chunk in the hierarchy has a name and which each name content is associated.
Artifacts and their Content Chunks

• Given artifact $E = (id, ct)$ we define its set of content chunks as follows:

  $Co(NCoCh(E))$

  $Co(\{(id, ct)\}) = \{ct\}$

  if $ct$ is an elementary content chunk

  $Co(\{(id, ct)\}) = \bigcup\{Co(NCoCh(t)) : t \in ct\}$

  if $ct$ is a set of named content chunks

  $Co(S_1 \cup S_2) = Co(S_1) \cup Co(S_2)$

  if $S_1$ and $S_2$ are nonempty sets

• If the content chunks are assertions, then the meaning of the artifacts is given by

  $\land Co(CoCh(E))$
Artifacts as named hierarchies of content

With these concepts

- An artifact \((id, ct)\) is structured into a hierarchy of named content chunks
- \(NCoCh((id, ct))\) yields the set of all named content chunks
- \(Co(NCoCh((id, ct)))\) yields the set of all elementary (unnamed) contents of artifact \((id, ct)\)

```
Air_Bag: Spec

   Air_Bag_Safety: Req
   Air_Bag_Reliability: Req

   (ABReq1, "if crash then ...")

   ...
```

Manfred Broy
MOD SumScho 2012
Traceability in Software and System Development
A (bilateral) link $t$ defines a directed relation between two named content chunks $e$ and $e'$

$$(e, e') \in \text{NCoCh}(E) \times \text{NCoCh}(E')$$

$\Diamond$ $e$ is called the source of $t$ and

$\Diamond$ $e'$ is called the target of $t$

We write

$$\text{src}(t) = e \text{ and } \text{trg}(t) = e'$$
Linking, Tracing: Relating Artifacts and their Content Chunks

• A trace is a nonempty finite sequence of links 
  \( t_0, t_1, t_2, \ldots, t_n \)
  where the source of \( t_{i+1} \) is the target of \( t_i \):
  \[ \text{trg}(t_i) = \text{src}(t_{i+1}) \quad \text{for} \quad i = 0, 1, \ldots, n-1 \]

We distinguish between links and traces that
• relate the content chunks of one artifact, called \textit{intra-artifact links}, and links that
• relate the content chunks \( e \) of one artifact \( E \) to those of a different artifact \( E' \), called \textit{inter-artifact links}. 
Illustration: Tracing

- intra-artifact link
- inter-artifact link

Requirements Specifications Architecture Implementation

trace
Illustration: Backward Tracing

Requirements → Specification → Architecture → Implementation

- Requirements
- Specification
- Architecture
- Implementation
Meaning of Links and Traces

A link relates two syntactic named content chunks

• A link has a meaning that usually is related to the meaning of the content chunks it relates.

• A link states a proposition about the relationship between its source and its target.

• A link can be valid or invalid.
  ◊ It is called *valid*, if the proposition associated with the link is true.
  ◊ Otherwise it is called *invalid*. 

Syntax and Meaning of Links

- **Syntactically** a link is
  - a relationship between named content chunks of artifacts.

- **Semantically** a link expresses that
  - there is a particular property valid for the involved content chunks.

Example: link \( t \) with

\[
\begin{align*}
\text{trg}(t) &= \text{“Product\_Manager: Stakeholder”} \\
\text{src}(t) &= \text{“High\_Usability: Quality\_Attribute”}.
\end{align*}
\]

- Link is to express that the stakeholder \text{Product\_Manager} is the source of the quality requirement \text{High\_Usability}.

- In other terms, the link has the meaning
  - \((\text{Product\_Manager: Stakeholder, High\_Usability: Quality\_Attribute}) \in \text{Source\_of\_Requirement}\)
  - where \text{Source\_of\_Requirement} is a relation between stakeholders and requirements.
We distinguish the following concepts of links

- **supplemental** links: link $t$ relating $a$ and $z$ documents relationships between content chunks $a$ and $z$ providing additional information not explicitly contained in artifacts $E_k$ and $E_k'$;
  - Example: link between a stakeholder $a$ and a requirement $z$ that originates from that stakeholder.

- **derived** links: link $t$ relating $a$ and $z$ documents relationships between chunks $a$ and $z$ that can be derived from its logical meaning and justified logically (or even proved) from the assertions in artifacts $E_k$ and $E_k'$;
  - Example: specification of a functional property by assertion $a$ and its implementation or refinement by assertion $z$ such that $z \Rightarrow a$. 

Example: *supplemental* Link

List of stakeholders

... 
Product manager
Architect
... 

List of requirements

$R_1$: ...
... 
$R_k$: High usability
...
Example: derived Link

**Requirements**

<table>
<thead>
<tr>
<th>R₁: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>Rₖ: Thrust reversal can only be activated, if airplane is on the ground</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

**Specification**

<table>
<thead>
<tr>
<th>S₁: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>Sⱼ: Sensor AoG yields true if and only if airplane is on the ground</td>
</tr>
<tr>
<td>Sⱼ₊₁: ¬AoG ⇒ Thrust reversal cannot be activated</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Multilateral Links

• A multilateral link $t$ is a directed relation between two named sets of content chunks $e$ and $e'$ of artifacts $E$ and $E'$:

$$(e, e') \in \wp(\text{NCoCh}(E_k)) \times \wp(\text{NCoCh}(E_{k'}))$$

$\diamond e \subseteq \text{NCoCh}(E_k)$ is called the source of $t$ and
$\diamond e' \subseteq \text{NCoCh}(E_{k'})$ is called the target of $t$

• We write

$\text{src}(t) = e$ and $\text{trg}(t) = e'$
<table>
<thead>
<tr>
<th>Requirements</th>
<th>Specification</th>
<th>Architecture</th>
<th>Implementation</th>
</tr>
</thead>
</table>

Illustration: Multilateral Tracing
Content of Multilateral Links

- Given multilateral link $t$ relating between content chunks $e$ and $e'$ of artifacts $E$ and $E'$:

$$ (e, e') \in \emptyset (NCoCh(E_k)) \times \emptyset (NCoCh(E_k')) $$

in case the contents $Co(e)$ and $Co(e')$ are assertions the link relates two sets of assertions.
Representing Artifacts by Logic: System Requirements
System level functional requirements

- The system interface behaviour $F$ is specified by the system requirements specification

$$A = \{A_i : 1 \leq i \leq n\}$$

where the $A_i$ are interface assertions

<table>
<thead>
<tr>
<th>Function</th>
<th>Safety</th>
<th>Priority</th>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>...</td>
<td>Yes</td>
<td>high</td>
<td></td>
</tr>
<tr>
<td>$A_2$</td>
<td>...</td>
<td>No</td>
<td>medium</td>
<td></td>
</tr>
<tr>
<td>$A_n$</td>
<td>...</td>
<td>no</td>
<td>low</td>
<td></td>
</tr>
</tbody>
</table>
Representing Artifacts by Logic: Functional Specification
Function / Feature Hierarchy

- \( F_{1,\ldots,n} \)
- \( B_1 \)
- \( B_2 \)
- \( B_\ldots \)
- \( B_n \)

- \( F_{1,2} \)
- \( F_{k,k+1} \)
- \( F_{n-1,n} \)

- \( F_1 \)
- \( F_2 \)
- \( F_k \)
- \( F_{k+1} \)
- \( F_{n-1} \)
- \( F_n \)

\( \text{sub} \) subservice relation

channels of mode types
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 set \( M \) of mode channels to capture feature interactions
- each \( F_i \) sub-function is described by
  - a syntactic interface and
  - an interface assertion \( B_i \) such that
    \[ \land \{B_i: 1 \leq i \leq k\} \Rightarrow A \]
Representing Artifacts by Logic: Architecture
Architecture

- Composition $C_1 \land C_2 \land C_3$

F: $\exists Z: \wedge \{C_k: k \in K\}$
Specifying Architectures by Assertions

Given composable systems $k \in K$ with specifying interface assertions $C_k$, the specification of the architecture reads

$$\land \{C_k : k \in K\}$$

and the interface assertion of the composed is given by hiding the internal channels in set $Z$

$$\exists Z : \land \{C_k : k \in K\}$$
Three Artifacts
Three levels of Specification

• **Requirements** - system level
  ◦ List of requirements - functional system property
  ◦ Example: “The activation of safety relevant functions by the pilot is always double checked for plausibility by the system.”

• **Functional specification** - system level
  ◦ decomposition of system functionality in hierarchy of (sub-)functions
  ◦ Specification of (sub-)functions
  ◦ Specification of dependencies (feature interactions) between (sub-)functions based on a mode concept
  ◦ Example: “Thrust reversal may only be activated, if the plane is on the ground.”

• **Architecture specification** - component level
  ◦ decomposition a systems in sub-systems (component)
  ◦ relationship to data flow diagram
  ◦ interface specification of component
  ◦ Example: “The weight sensor indicates that the plane is on the ground.”
Three levels of system description in logic

• system level requirements
  \[ A = \bigwedge \{A_i: 1 \leq i \leq r\} \]

• functional specification at system level - functionality
  \[ B = \bigwedge \{B_i: 1 \leq i \leq n\} \]

• architecture specification
  \[ C = \bigwedge \{C_k: 1 \leq k \leq m\} \]

• Correctness
  ◊ functional specification correct w.r.t to requirements:
    \[ B \Rightarrow A \]
  ◊ architecture correct w.r.t to functional spec (let \( M \) be the set of mode channels):
    \[ C \Rightarrow \exists M: B \]
Relational view: Inter-artifact links and traces

<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>
Every assertion in the specification has to be guaranteed by the assertions of the architecture.

Can we find and identify them?
Illustration: Multilateral Tracing as refinement

Requirements | Specification | Architecture | Implementation

[Diagram showing the relationships between requirements, specification, architecture, and implementation with a focus on multilateral tracing as refinement.]
Guarantors and Guarantor Sets

- Let $P$ be an assertion and $R$ be a set of assertions.
- A subset $R' \subseteq R$ is called \textit{guarantor set for} assertion $P$ in set $R$ if
  \[
  \forall((\land R') \Rightarrow P)
  \]
  \[
  \diamond \text{ In this case the assertions in set } R' \text{ guarantee logically assertion } P.
  \]
- A guarantor set $R'$ for assertion $P$ in $R$ is called \textit{minimal}, if every strict subset of set $R'$ is not a guarantor set for assertion $P$.
- A minimal guarantor set $R' \subseteq R$ is called \textit{unique} in set $R$ if there do not exist different minimal guarantor sets in $R$. 
Guarantors and Guarantor Sets

- A assertion $Q$ is called *weak guarantor* for assertion $P \in R$ if it occurs in some minimal guarantor set for assertion $P$ in $R$.
- A assertion $Q$ is called *strong guarantor* for $P$ in $R$ if assertion $Q$ occurs in every guarantor set of assertion $P$ in $R$.
- Note that there is some relationship between guarantors and the so-called Primimplikanten a la Quine
Relationship: req spec to function spec - tracing

<table>
<thead>
<tr>
<th>sub-function reqs</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>A₆</th>
<th>A₇</th>
<th>A₈</th>
<th>A₉</th>
<th>.</th>
<th>.</th>
<th>.</th>
<th>Aₖ</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bₙ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Red:** Bᵢ is strong guarantor of Aⱼ

**Yellow:** Bᵢ is weak guarantor of Aⱼ

**Green:** Bᵢ is not a weak guarantor of Aⱼ
## Relationship: architecture to requirements

<table>
<thead>
<tr>
<th></th>
<th>System level reqs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A_1$</td>
</tr>
<tr>
<td>Sub-system reqs</td>
<td></td>
</tr>
<tr>
<td>$C_1$</td>
<td></td>
</tr>
<tr>
<td>$C_2$</td>
<td></td>
</tr>
<tr>
<td>$C_3$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_n$</td>
<td></td>
</tr>
</tbody>
</table>

**Red:** $C_i$ is strong guarantor of $A_j$

**Yellow:** $C_i$ is weak guarantor of $A_j$

**Green:** $C_i$ is not a weak guarantor of $A_j$
Tracing at a logical level

Requirements

\[ A = A_1 \land A_2 \land A_3 \land \ldots \]

Architecture

\[ C = C_1 \land C_2 \land C_3 \land \ldots \]

Correctness architecture

\[ C \Rightarrow A \]

Tracing requirement \( k \)

\[ C \Rightarrow A_1 \land A_2 \land A_3 \land \ldots \land A_k \land \ldots \]

Expanding \( C \)

\[ C_1 \land C_2 \land C_3 \land \ldots \Rightarrow A_k \]

Weakening \( C \)

\[ (C_1 \Rightarrow C'_1) \land (C_2 \Rightarrow C'_2) \land (C_3 \Rightarrow C'_3) \land \ldots \]

Such that

\[ C'_1 \land C'_2 \land C'_3 \land \ldots \Rightarrow A_k \]

Conclusion:

If the architecture spec \( C \) is correct with respect to a particular requirement \( A_k \) then there exist assertions \( C'_i \) contained in the specifications \( C_i \) of the sub-systems of the architectures that guarantee \( A_k \)
Analysis

• For every requirement $A_k$ its “guarantors” $C'_i$ are different, in general
  ◊ Conclusion: Syntactic tracing does not work
• For requirement $A_k$
  ◊ there are weakest “guarantors” $C'_i$
  ◊ its weakest “guarantors” $C'_i$ are not necessarily unique
  ◊ many of its “guarantors” $C'_i$ are not necessarily trivial (“true”)
    • There are many links!
Inter-artifact Links: Functional Specification to Requirements

• Relating Functional Specifications to System Level Requirements
  ◊ The trace concept as introduced above can be used to relate the functional specification $B$ to the requirement specification $A$.
  ◊ Due to the specific structure of set $B$ in terms of sub-functions this imposes a specific structure on the set $A$.

• A requirement $Q$ in $A$ is called *dedicated* functional feature $k$, if there exist only one strong trace to exactly one feature $h$ with $B_h \in B$. 
Inter-Artifact Traces: Relating Architecture to Requirements

- Traces relate content chunks of architectural specification \( C \) to the content chunks of system level requirements specification \( A \).

- A requirement \( Q \) in \( A \) is called \textit{sub-system requirement}, if there exist only one strong trace to exactly one assertion \( P \) in \( C \).

  ◊ Then the system level requirement does only affect one subsystem (this is a very special case).
Intra-artifact Links: System Level Requirements
Relating Content Chunks of Artifacts by Logic
Well-Formedness of Sets of System Assertions

A set $R$ of system requirements by assertions is called

- **consistent**, if the following proposition holds
  \[ \exists (\land R) \]

- **non-overlapping**, if (there is a relationship to case distinctions)
  \[ \forall (\lor R) \]

- **weakly independent**, if for every pair of non-empty subsets $R', R'' \subseteq R$ of disjoint non-empty sets of assertions with $Q = \land R'$, $P = \land R''$
  \[ \exists (P \land Q) \] $P$ and $Q$ are consistent
  \[ \exists (\neg P \land Q) \] $Q$ does not imply $P$
  \[ \exists (P \land \neg Q) \] $P$ does not imply $Q$
Non-overlapping: Sets of assertions forming case distinctions

• We consider a finite set of cases $Q_i$ and a finite set of consequences $P_i$, $1 \leq i \leq n$.

• We speak of a complete, disjoint case distinction if both the following two propositions hold

  $\forall \forall \{Q_i: 1 \leq i \leq n\}$  completeness

  $\forall (Q_i \Rightarrow \neg Q_j)$  for all $i \neq j$ - disjointness

• By these conditions following propositions are equivalent

  $\forall \{Q_i \land P_i: 1 \leq i \leq n\}$  disjunctive normal form

  $\forall \{Q_i \Rightarrow P_i: 1 \leq i \leq n\}$  implicative form

• The second form leads to a set $\{(Q_i \Rightarrow P_i): 1 \leq i \leq n\}$ of assertions that are non-overlapping
Consistency

- **Consistency** for sets $R$ of assertions
  \[ \exists \land R \]
- **Consistency** of two assertions $P$ and $Q$ means $\exists [P \land Q]$ which is equivalent to
  \[ \neg \forall [P \Rightarrow \neg Q] \]
  \[ \neg \forall [Q \Rightarrow \neg P] \]
  which is one of the conditions of logical independence.
- This shows that the fundamental requirement of consistency guarantees two conditions of logical independence.
There are many papers and even standards on the quality of requirements. The IEEE standard Std 830-1998 (see [IEEE 98]) requires the following quality attributes for system and software requirements:

- completeness
- consistency
- unambiguousness/precision
- correctness (more precisely validity)
- understandability/clarity
- traceability
- changeability
Notions from this list such as

◊ completeness
◊ correctness (more precisely validity – the requirement is what the stakeholder meant)
◊ understandability/clarity

cannot be explicitly addressed in our approach since they have to be analyzed on a different level.

They deal with properties of requirements that are not captured by our logical relations.
Formalization IEEE artifact quality attributes

- Clarity and understandability is not a formal notion.
  - depends on the skills and background of the people that read and write specifications.
  - This quality attribute is beyond our approach of formalization.
- Precision can be achieved by formalization.
- However, quality concepts such as
  - consistency
  - traceability
  - changeability (to some extend)
are captured by our approach.
Intra-artifact Links: Functional System Specification Relating Functional Features by Feature Interactions
Function Hierarchy

intra-artifact link: feature interaction

F₁, ..., n

F₁

F₂

Fₖ

Fₖ₊₁

Fₙ₋₁

Fₙ

B₁

B₂

Bₖ

Bₖ₊₁

Bₙ₋₁

Bₙ

←ₚ sub service relation

channels of mode types
Given (let be $I_1, O_1, I_2, O_2$ pairwise disjoint): $F \left[ I \sqcap O \right]$

$(I_1 \triangleright O_1)$ **subtype** $(I \triangleright O)$   
$(I_2 \triangleright O_2)$ **subtype** $(I \triangleright O)$

there is a **feature interaction** from the feature

$F^\dagger(I_2 \triangleright O_2) \left[ I_2 \triangleright O_2 \right]$ to $F^\dagger(I_1 \triangleright O_1) \left[ I_1 \triangleright O_1 \right]$

if

projection $F^\dagger(I \setminus I_2 \sqcup O_1)$ is not **faithful** in $F$
Intra-artifact links in functional feature specifications

- If there is a feature interaction from functional feature $k$ to feature $k'$
  ◊ There exists a mode channel from feature $k$ to feature $k'$
- If there is a mode channel $m$ from feature $k$ to feature $k'$ and there is no feature interaction from feature $k$ to feature $k'$ then
  ◊ $m$ can be eliminated in the specification of feature $k'$ and the mode channel can be dropped
Intra-artifact Links: Architecture
Analyzing Composition: Intra-Artifact Links and Relations

• Consider two realizable specifications $C_1$ and $C_2$ for composable systems with syntactic interfaces $(I_1 \rightarrow O_1)$ and $(I_2 \rightarrow O_2)$ into a system with syntactic interface $(I \rightarrow O)$.

• If specifications $C_1$ and $C_2$ are realizable, then $C_1 \land C_2$ is a specification for the syntactic interface $(I \rightarrow O)$ that is realizable.

• Realizability implies for $C_1$ and $C_2$:

$$\forall I_1 : \exists O_1 : C_1 \quad \forall I_2 : \exists O_2 : C_2$$

• By composition we derive the specification

$$\exists Z : C_1 \land C_2$$

where $Z$ is the set of internal channels.

• Note: realizability implies consistency
• If $O_1 \neq \emptyset$ and $O_2 \neq \emptyset$ then $C_1$ and $C_2$ are logically independent (if they are not trivial) since

\[
\forall [C_1 \Rightarrow C_2]
\]

\[
\forall [C_2 \Rightarrow C_1]
\]

cannot hold due to the fact that $C_1$ and $C_2$ talk about disjoint sets of output channels that cannot be constraint by the other assertion.
Intra-Artifact Links for Architecture Specifications

• From the syntactic architecture we conclude which components are connected by channels.
• Channels yield intra-artifact links for architectures.

Note that strictly speaking, there may be channels used as input channels in components that do not depend on that input.

• Then there is a syntactic dependency but not a behavioral dependency
• However, then the channel can be eliminated in the interface assertion
Logical Independence of Functional System Specifications

• The set $B$ consists of assertions being sub-function specifications
  ◊ Each assertion $B_k \in B$ specifies the interface behavior of a sub-function.

• Assume that these specifications are realizable
  ◊ As long as all interface assertions $B_k \in B$ for functional features in $B$ are consistent, the set $B$ is consistent, too.

• A simple analysis shows that as long as the interface specifications of the individual functions are not trivial and realizable, the assertions in set $B$ are pairwise
  ◊ logically independent
  ◊ consistent
Logical Independence of Functional System Specifications

• Given two interface specifications $B_k$ for syntactic interfaces $(I_k \rightarrow O_k)$ and disjoint output sets with $k = 1, 2$ that are realizable we get consistency

$$\exists(B_1 \land B_2)$$

for free.

• Actually, we should see a functional specification rather as a set of assertions about sub-functions.

• If interface assertions $Q_1$ and $Q_2$ for different features are not trivial, i.e. if

$$\neg \forall Q_1 \text{ and } \neg \forall Q_2$$

then we get weak independence of the assertions, since they refer to different input channels.
Change Management and Changeability: Impact Analysis for Change Requests
Changeability and Impact Analysis

• Typically, in requirements management we have to revise requirements.

• Requirements are
  ◊ changed and modified
  ◊ validated
  ◊ verified
  ◊ traced
  ◊ implemented

• One essential notion is the granularity of requirements.
• For validation, refinement, implementation, tracing, and verification a well-chosen granularity of assertions is useful.
  ◊ If the granularity is too coarse, a further decomposition is needed to address test cases.
  ◊ If it is too fine too many tests are needed to cover all requirements.

• Typically in requirements engineering we deal with lists of requirements or – more abstractly – with sets $R$ of requirements.
Changeability and Impact Analysis

• Actually, then the ultimate requirement given by the set \( R \) is

\[ \wedge R \]

◊ So for the ultimate requirement the granularity of the requirements is not actually relevant.

◊ However, it is relevant for the development activities related to requirements.

• What happens, if we change the granularity of requirements and go from set \( R \) with assertions \( P, Q \in R \) to \( R' = (R \setminus \{P, Q\}) \cup \{P \wedge Q\} \)?

Obviously then

\[ \wedge R \equiv \wedge R' \]

• Thus consistency and validity is not changed.
Concluding Remarks

- **Artifacts** represented by logic
  - Logical representation of the *content* by assertions

- **Dependencies** based on logic
  - Logical representation of dependencies

- **Formalization of traceability**
  - Intra- and inter-artifact links

- **Relating different levels of abstraction**

- **Engineering questions**
  - How many dependencies are there in systems today
  - What is the complexity of relations between
    - Requirements and functional specification
    - Functional specification and architecture
    - Requirements and architecture

**Next step:** variability