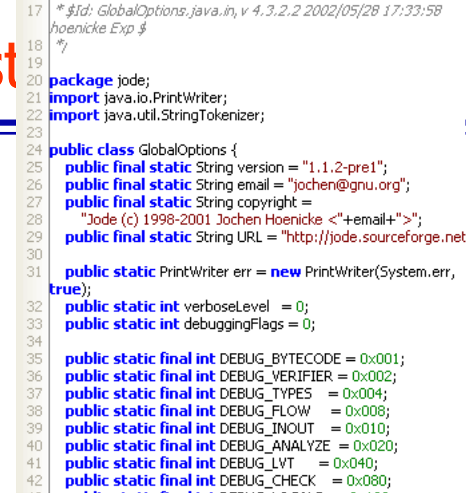
Requirements Engineering and Design Thinking:
From why and why ever to what and how

Manfred Broy

Technische Universität
München
Institut for Informatics

From programs to cyber-physical syst



- Programs:
  - mappings states to states or data to data,
  - supposed to terminate (exception OS),
  - time and interaction (often) not an issue,
  - concept of computation: Turing machines – algorithms
- Cyber-physical systems:
  - connected to the physical world,
  - need a coherent model of context, interface, interaction, time, architecture, state, probability, data and event flow, perhaps even space, geometry and movement
  - concept of computation: interaction, generalized timed Mealy machines
  - extensive requirements for dependability

The key role of requirements engineering (RE) in SSE

- RE is a key to software & systems engineering (SSE)
- Functionality
  - ◊ what is the needed functionality
  - ◊ do systems offer the needed functionality
  - ◊ are unneeded functions excluded
  - ◊ Functional quality
    - usability: is the functionality easy to access
    - safety and security
    - …
- Nonfunctional quality
  - ◊ Reusability
  - ◊ Changeability
  - ◊ Portability
  - ◊ …

Making requirements explicit

Do we distinguish between

- the system as required
- the system as implemented?

If yes, we need documentation!

When do we decide about requirements?

- Up front: Before we start implementation?
- Iterative and incremental: While we carry out implementation?
- After mortem: after implementation?
- Not at all: No distinction between implementation and requirements: It is the code that counts!

The challenge: machine learning

- Neuronal net learns from a training set
  - ◊ The training set is chosen according to the required functionality
- Result is an algorithm – being a black box
  - ◊ No spec
  - ◊ No verification

Agile development: finding requirements on the way

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Questions

- Where is the decision about the requirements?
- How are requirements documented?

The roles in RE: who decides what!

- Product manager
  - Decides about the goals and the key requirements
- Requirements engineer
  - Responsible for the methodology applied and
  - The quality of the requirements artifacts
- Requirements manager
  - Responsible for the requirements life cycle
- Architect
  - Responsible for reflecting the requirements in the overall system structure
- The verifier (tester)
  - Needs requirements to define test cases
- Stakeholders requirement and sources
  - Bring in their needs and expectations

Deficiencies in practice

- Wrong assumptions about functionality:
  - ◊ high discrepancies between expected use and effective use
- Role of a requirements engineer or a product manager missing
  - ◊ Product managers and architects responsible for requirements
- Requirements incomplete and description inadequate
  - ◊ missing structuring
- Requirements not reviewed and not validated
- Requirements finally not documented
  - ◊ Documentation not updated
  - ◊ In software evolution unclear what is required
- Verification starts too late
  - ◊ Only during test case engineering insufficient requirements identified

Crucial Aspects of Requirements Engineering

- Requirements – Quality in Use
  - Functionality
  - MMI
  - External Quality
- Architecture
  - Structuring
  - Modularity
  - Reusability
- Quality
  - External
  - Internal
- Evolution
  - Time to market

The challenges

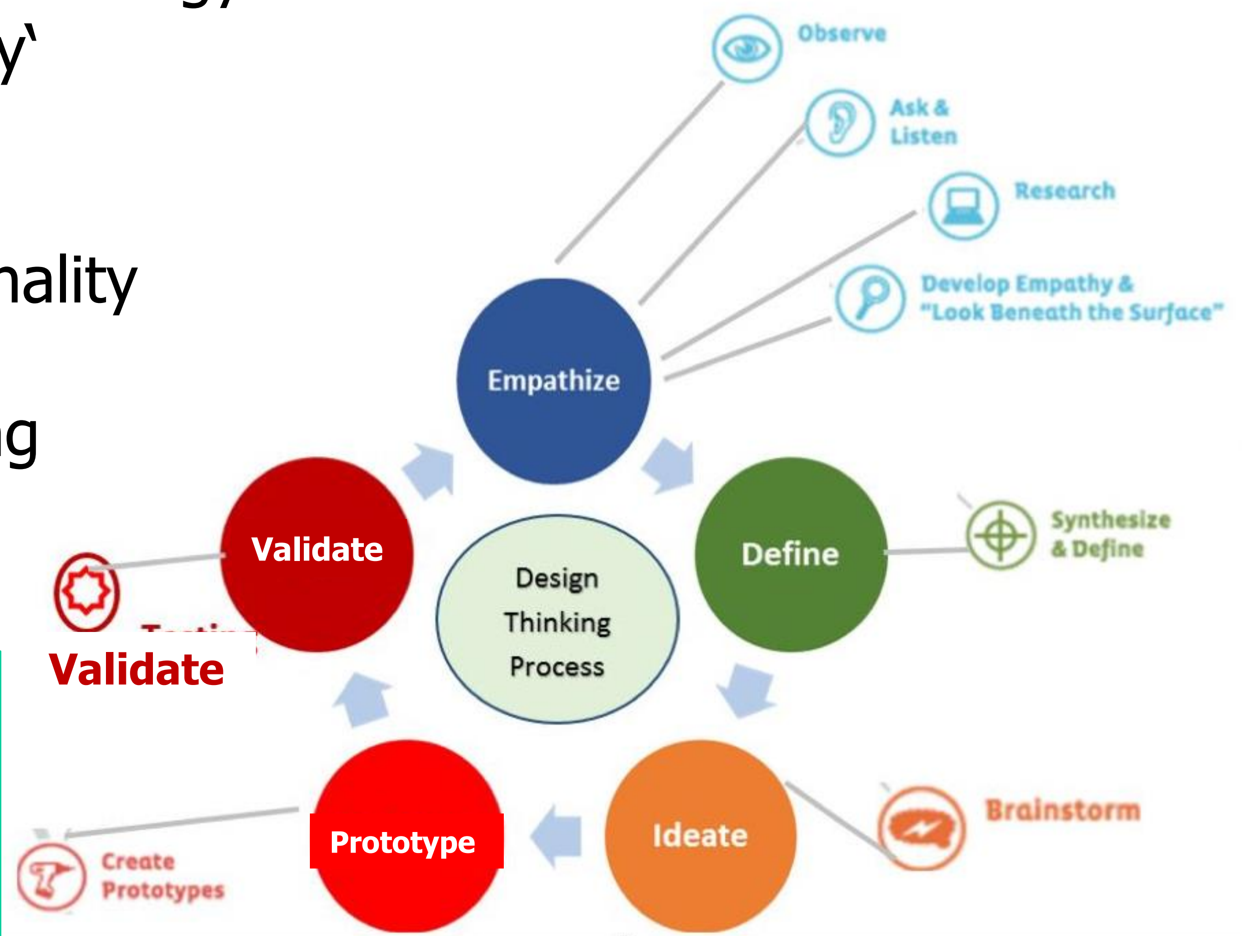
- To find out what is actually needed and what is feasible – innovation: who knows – design thinking
  - ◊ Which functionality
  - ◊ Usability
  - ◊ New ideas
- Having a prototype – how do we know what is essential
  - ◊ Identifying requirements – elicitation
- Achieving structured requirements
  - ◊ Functional architecture – use cases – detailed specification
    - Real time
  - ◊ Functional quality: safety, reliability, security, usability
    - Probabilities
  - ◊ Quality beyond functionality
- Managing requirements: implementation, verification, change

Design thinking – exploring options

Design thinking uses the designer's sensibility and methods to match people's needs with what is technologically feasible and what a viable business strategy can convert into customer value and market opportunity`

Design thinking –
find innovative functionality
and usability –
user centric engineering

The life cycle:



The missing link: How to extract from the prototype the requirements

Why: the business case

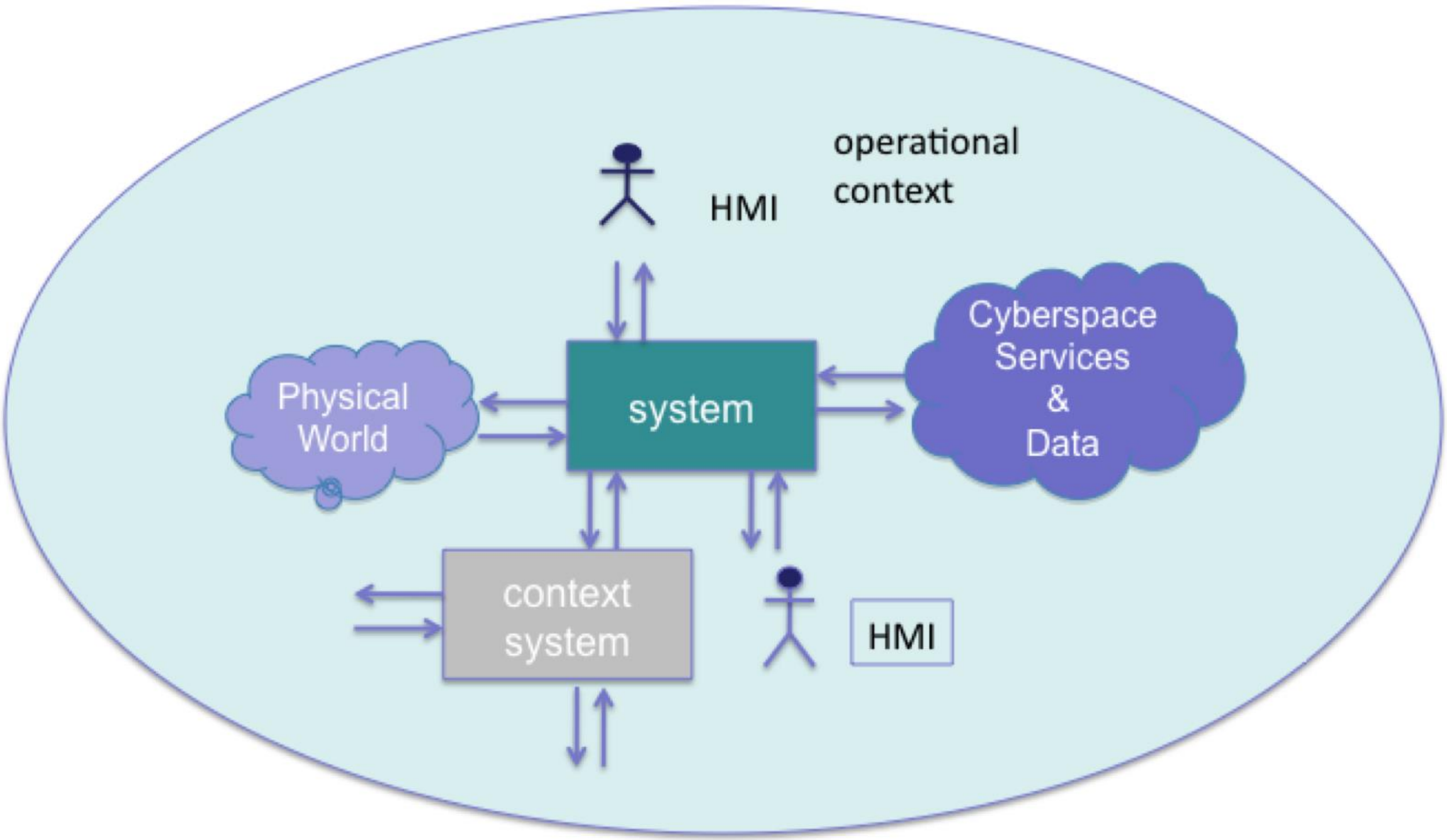
Why is it interesting to develop a specific functionality: the business case - innovation

- Individual solution
  - higher efficiency
  - higher quality
- Standard product
  - innovative functionality
- Embedded
  - better product

Aspects (attributes) of requirements

- Characteristic
  - Functional or quality
- Level of detail
  - From abstract to concrete
- Ways to express it
  - from informal to semiformal to formal
- Source
  - Where it came from
- Significance
  - Must or may
- Status
  - Accepted, implemented, verified

System and its context



From the informal to the formal

- In the beginning, properties of the universe of discourse 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 forming an ontology
  - ◊ 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

Example: Assertions

- For a simple universe of discourse Car representing cars, consider attributes such as

  length: Car → IN

  number_of_seats: Car → IN

  speed: Car → IN

  situation: Car → {city, country, high_way}

- Based on the attributes, given $d \in$ Car, we write logical expressions such as

$$speed(d) \geq 50 \wedge situation(d) = city$$

- This notation can be simplified for a fixed car d:

$$speed \geq 50 \wedge situation = city$$

- Such a logical expression referring to the attributes of the elements of the considered universe is called *assertion*.

Discrete systems: the system modelling theory

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

$$(I \blacktriangleright O)$$

data stream of type T

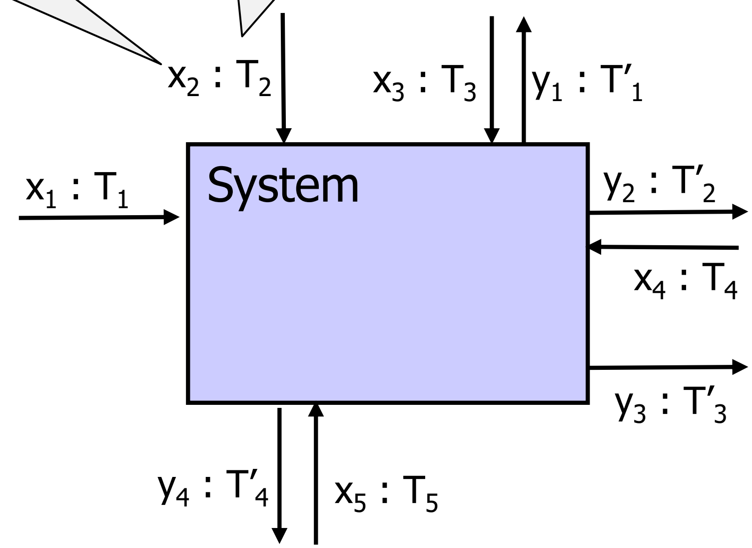
$$STREAM[T] = \{\mathbb{N}\backslash\{0\} \rightarrow T^*\}$$

valuation of channel set C

$$[C] = \{C \rightarrow STREAM[T]\}$$

interface behaviour for syn. interface $(I \blacktriangleright O)$

$$[I \blacktriangleright O] = \{[I] \rightarrow \wp([O])\}$$



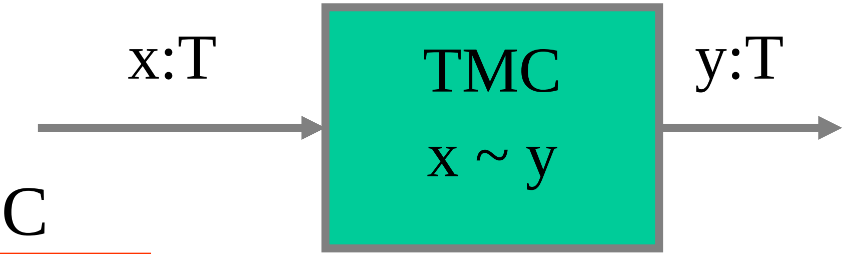
Interface Assertion

- Given a syntactic interface (I ▸ 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

Spec name



A transmission component TMC

Input channel

TMC

in x: T out y: T
x ~ y

Output channel

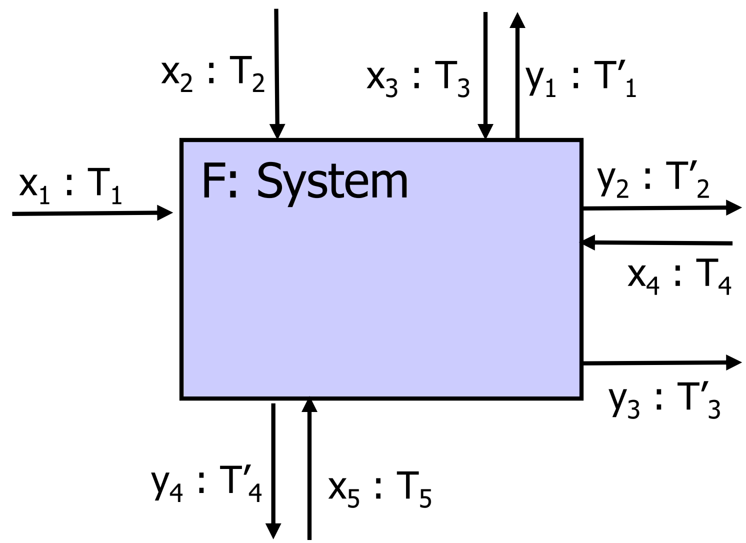
x ~ y º (" m Î T: m#x = m#y)

Interface assertion

Representing Artifacts by Assertions:
Functional Specification – Feature Specification

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▸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!

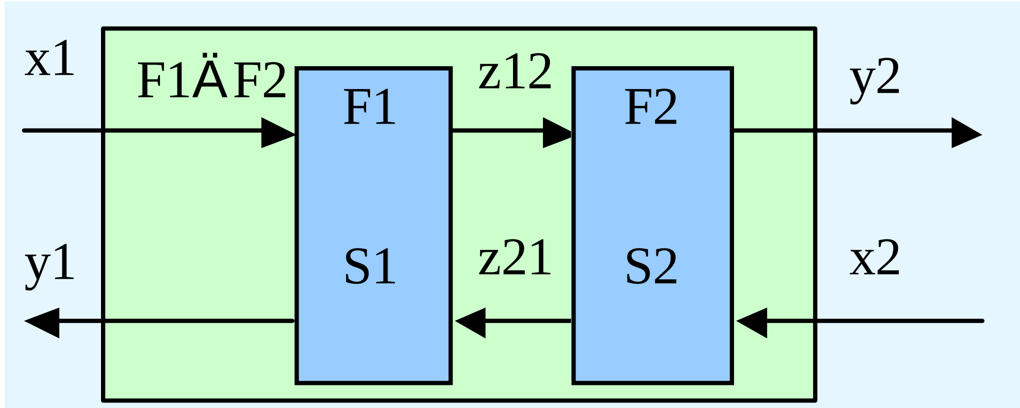
- These are at the level of system level interface behavior!

- A (functional) feature is a sub-function of a multi-functional system
  - that serves a certain purpose

Modeling functional (behavioral) features

- We give a interpretation of the notion of a (functional) feature in terms of the system interface model $F \in [I \blacktriangleright O]$
- 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 $(I' \blacktriangleright O')$ resulting in a sub-interface behavior $F' \in [I' \blacktriangleright O']$
  - ◊ absence of feature interactions is modeled by faithful projections
  - ◊ feature interactions are modeled by modes

Modularity: Rules of compositions for interface specs



F1

in	x1, z21: T
out	y1, z12: T
S1	

F2

in	x2, z12: T
out	y2, z21: T
S2	

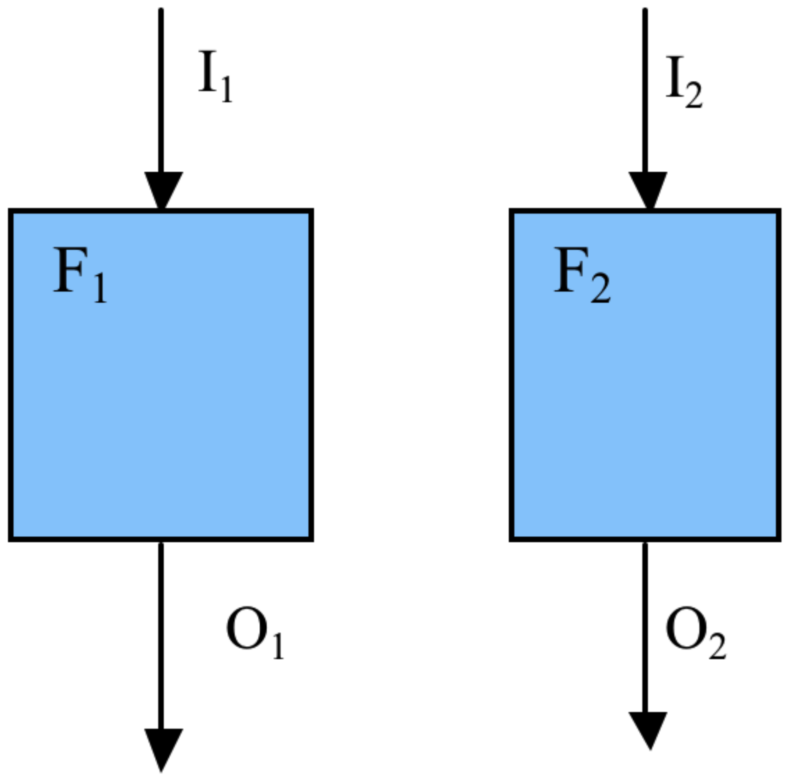
F1ÄF2

in	x1, x2: T
out	y1, y2: T
$ z12, z21: S1 Ù S2	

Feature Specification – Constructive Approach

Combining Functions without Interference

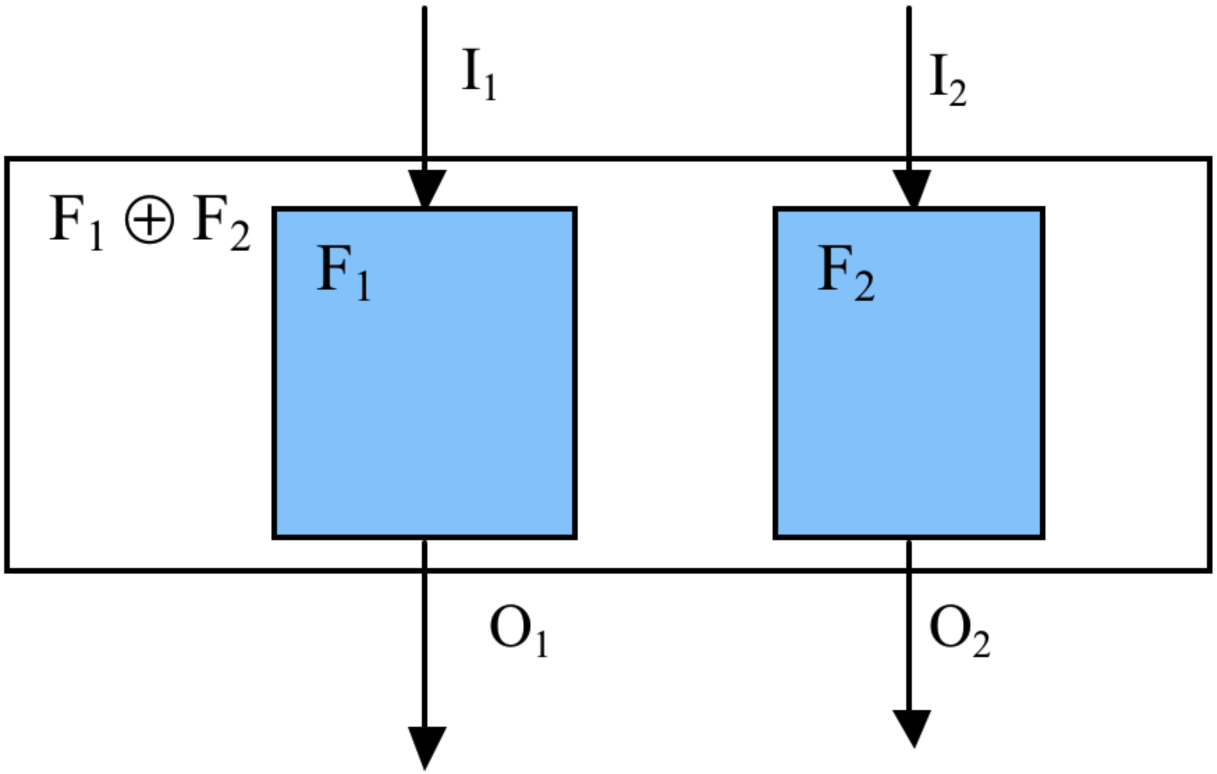
Given two functions F_1 and F_2 in isolation



We want to combine them into a function $F_1 \oplus F_2$

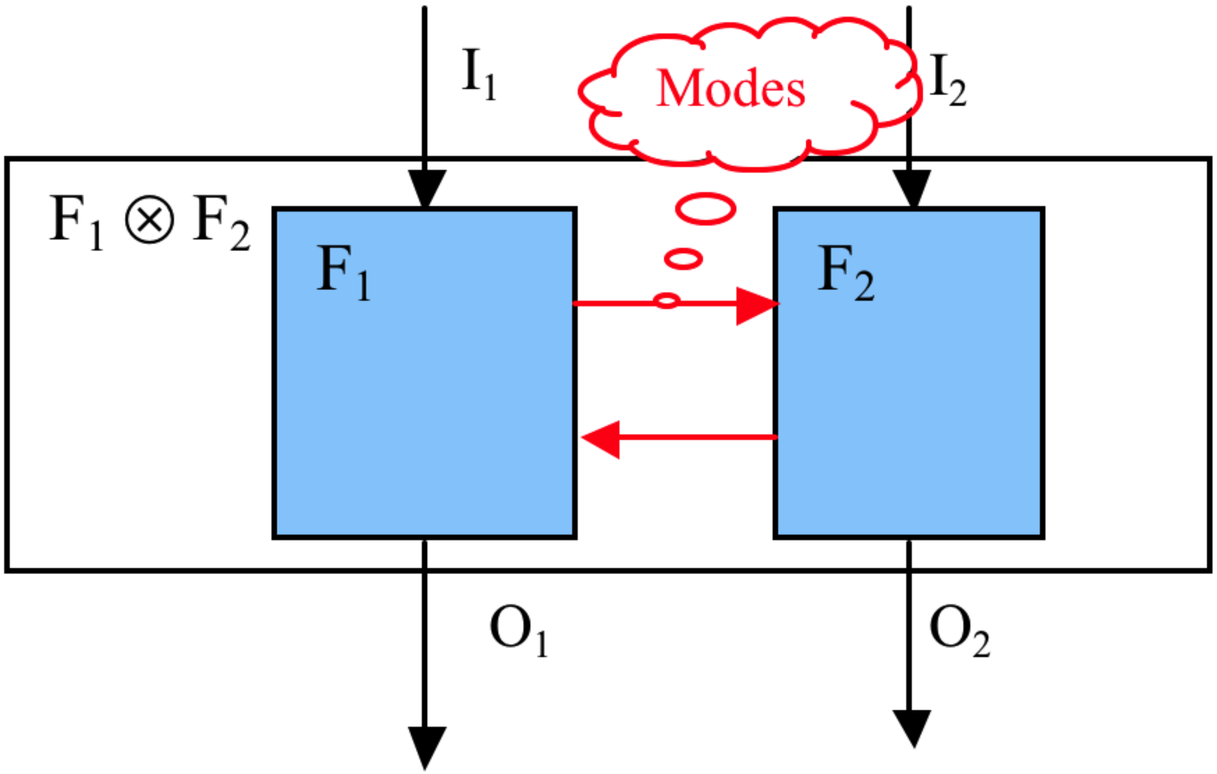
Combining Functions without Interference

Their isolated combination



Combining Functions with Feature Interaction

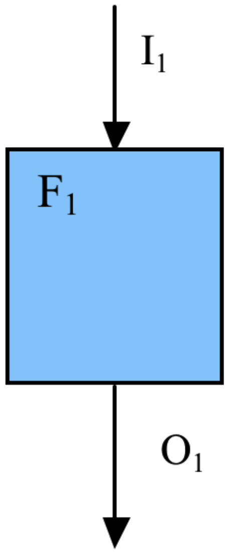
If services F_1 and F_2 have feature interaction we get:



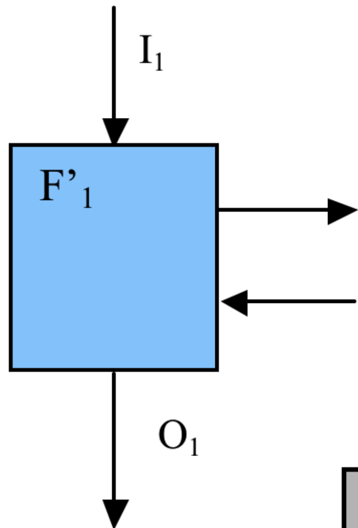
We explain the functional combination $F_1 \otimes F_2$ as a refinement step

The steps of function combination

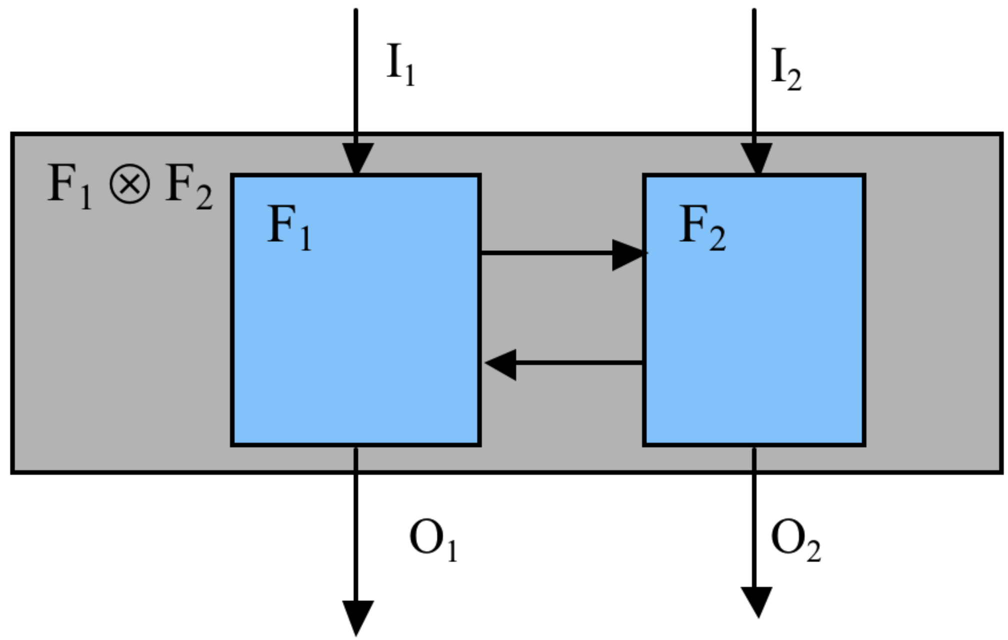
Given the isolated function F_1



We construct a refinement F'_1

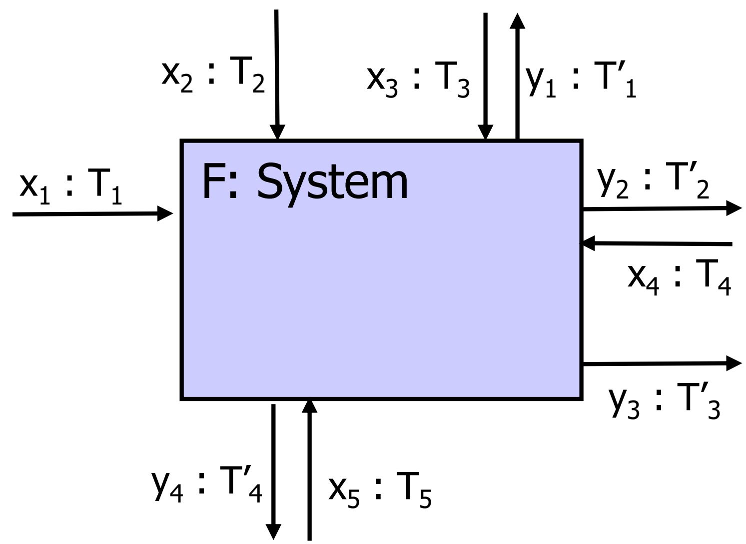


And combine F'_1 with a refinement F'_2 of F_2

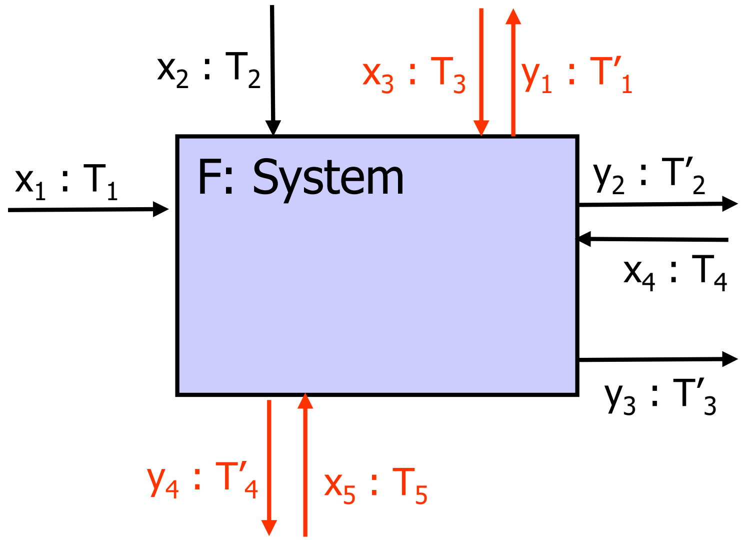


Feature Specification – Analytic Approach

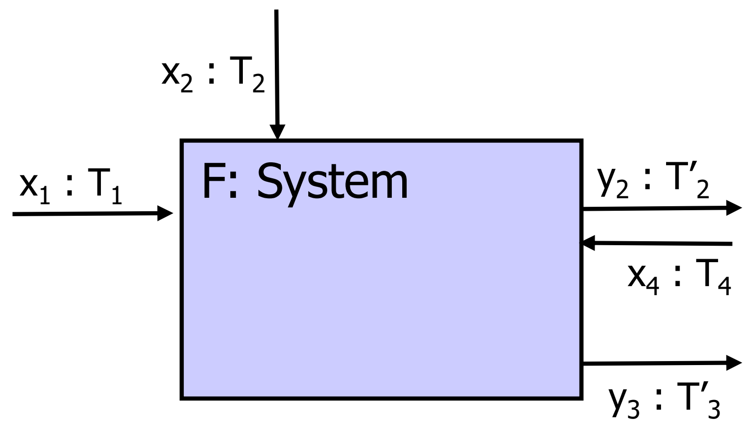
From overall syntactic system interfaces …



to …



sub-interfaces



Projection of assertions describing features

Given:

(I’▸ O’) subtype (I▸ O)

and interface assertion S for syntactic interface (I▸ O); we define its *projection* onto the feature with the syntactic interface (I’▸ O’) by

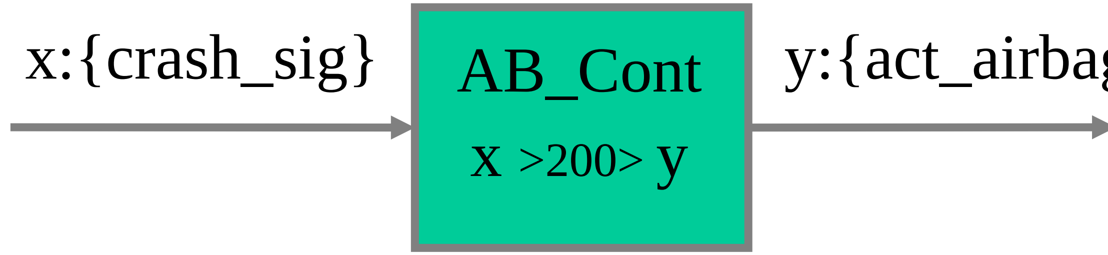
$ I\I’, O\O’: S

The projection is called *faithful*, if

($ I\I’, O\O’: S) ⇔ ($ O\O’: S)

Then the feature with syntactic interface (I’▸ O’) is free of feature interactions.

Example: Component interface specification – Airbag Controller



An air bag controller

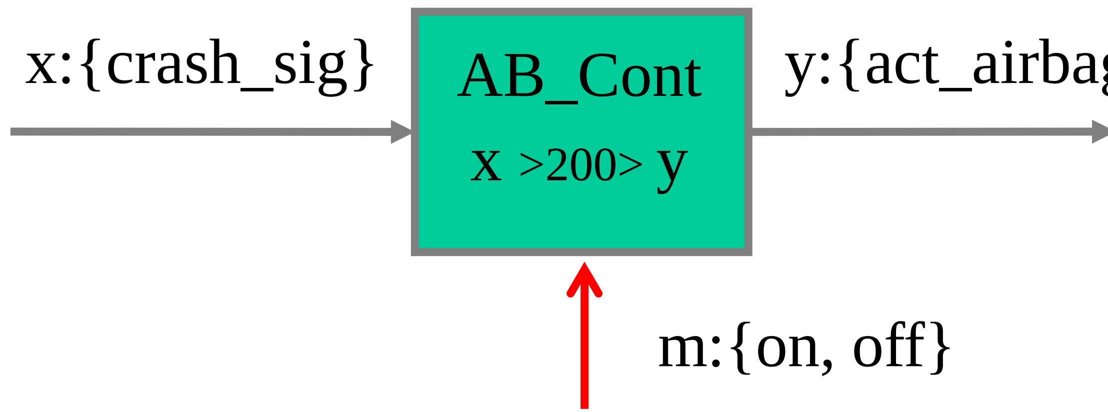
AB_Cont

in x: T out y: T
x >200> y

$$x >200> y \equiv (\forall\ t \in \text{Time}:$$

$$\text{crash\_sig} \in x(t) \Leftrightarrow \text{act\_airbag} \in y(t+200))$$

Example: Component interface specification – Airbag Controller



An air bag controller

AB_Cont

in	x: T, m: {on, off}
out	y: T
x >200> y	

x >200> y ≡ (" t ∈ Time:

(ON(m, t+199) ∧ crash_sig ∈ x(t)) ⇔ act_airbag ∈ y(t+200)

ON(m, t) = if t = 0 then false elif on ∈ m(t) then true
elif off ∈ m(t) then false else ON(m, t-1) fi

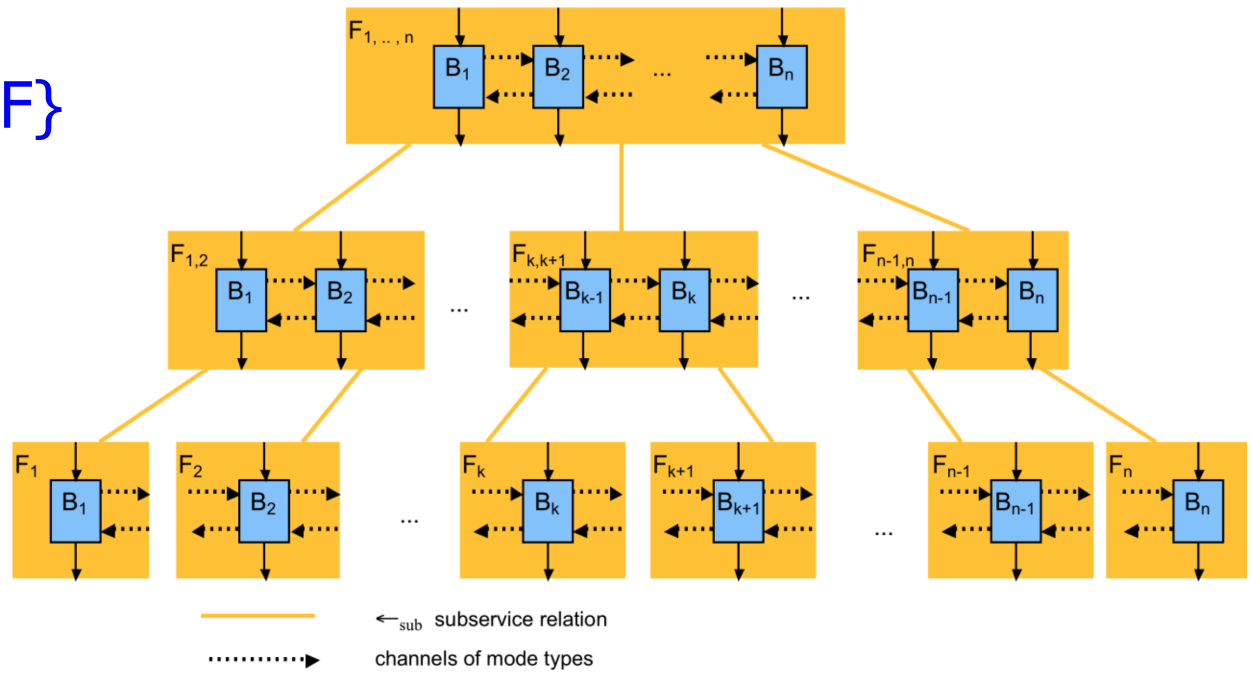
Specifying Functional Architectures by Assertions

Given composable features $f \in F$ with specified by interface assertions B_f the assertion of the functional specification reads

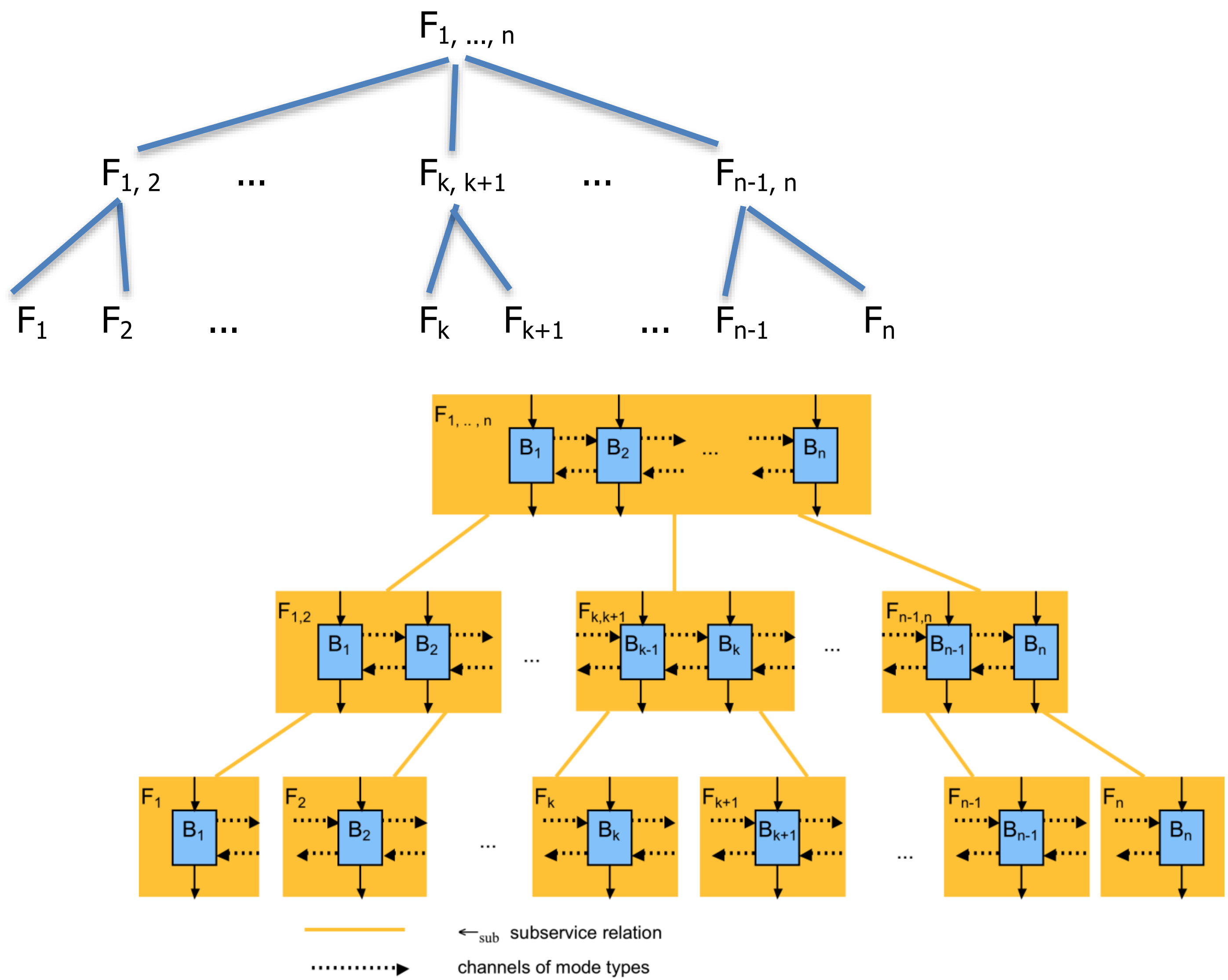
$$\wedge \{B_f : f \in F\}$$

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

$$\exists\, M: \wedge \{B_f : f \in F\}$$



An interpreted feature tree



Feature interaction in the architecture view

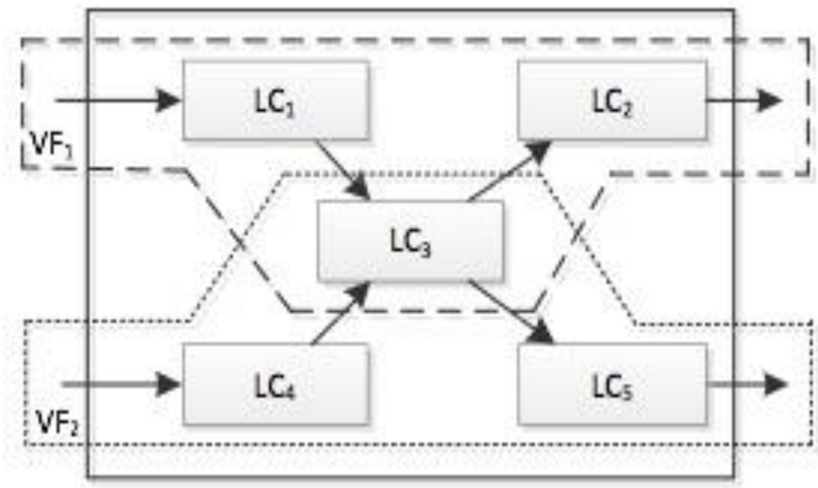


Table 4.2: Extent of dependencies in the vehicle function graph

	MAN System ($n = 55 \widehat{=} 100\%$)		BMW System ($n = 94 \widehat{=} 100\%$)	
Vehicle functions…	**Number**	**Ratio**	**Number**	**Ratio**
with incoming dependencies	36	65.5%	81	86.2%
with outgoing dependencies	29	52.7%	72	76.6%
with incoming and outgoing dependencies	27	49.1%	68	72.3%
without dependencies	17	31.0%	9	9.6%

Taken from:
A. Vogelsang: Model-based Requirements Engineering for Multifunctional Systems. PH. D. Dissertation, Technische Universität München, Fakultät für Informatik, 2014

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
…

Why translators are useful?

- Translators relate requirements in terms of assertions to technical/physical assertions
- They force us to make explicit assumptions behind physical/technical designs
  - ◊ As part of specifications
  - ◊ To validate them – to discover invalid assumptions

- Thrust reversal can only be activated, if airplane is on the ground
- Sensor AoG yields true if and only if airplane is on the ground
- $\neg$AoG $\Rightarrow$ Thrust reversal cannot be activated

Example: derived Link between two Ontologies

Requirements

R_1: …

…

R_k: Thrust reversal can only be activated, if airplane is on the ground

…

Specification

S_1: …

…

S_j: Sensor AoG yields true if and only if airplane is on the ground

S_{j+1}: $\neg$AoG $\Rightarrow$ Thrust reversal cannot be activated

…

Traceability in Software and System Development

Illustration: Tracing

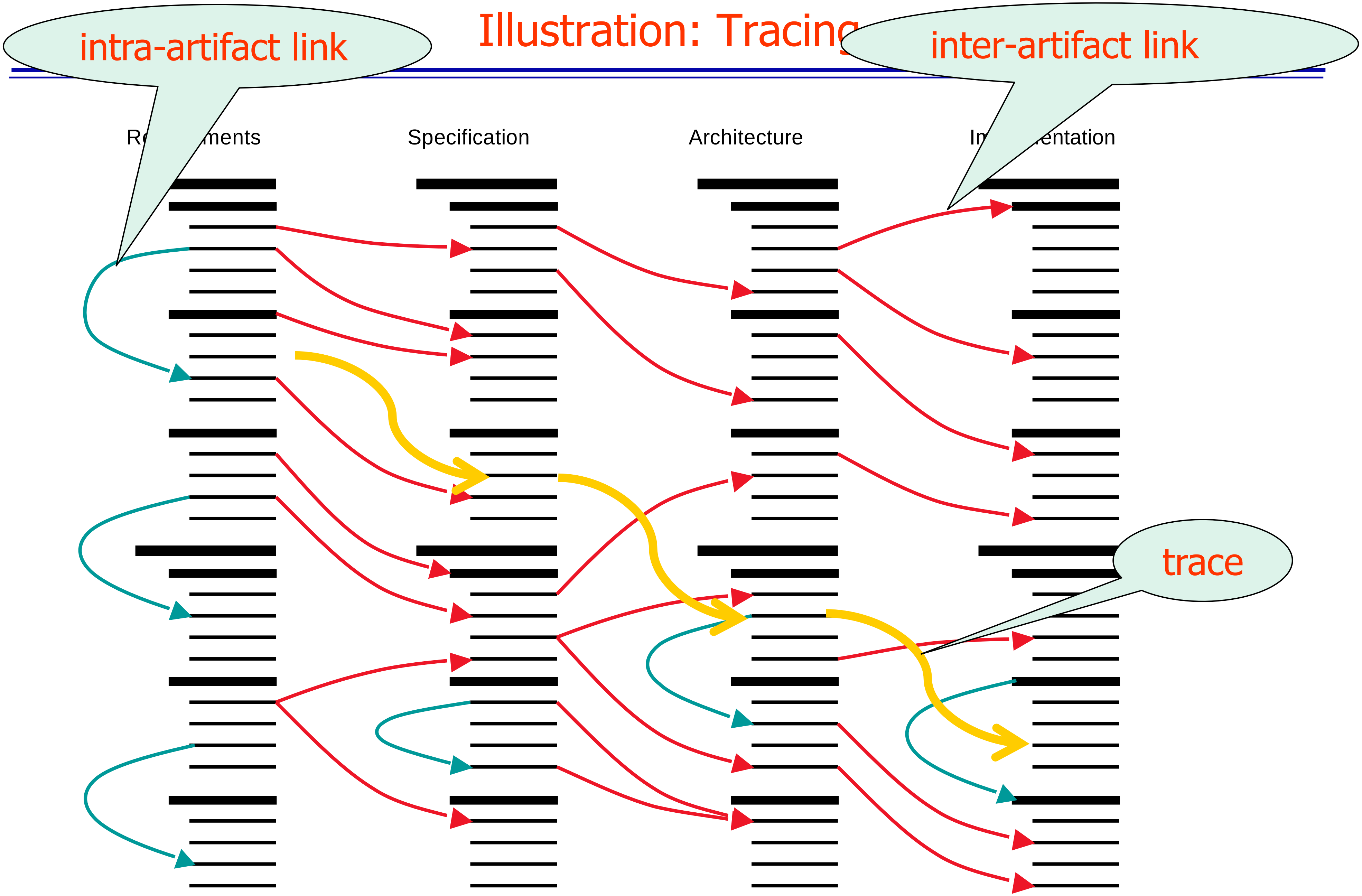


Illustration: Forward Tracing

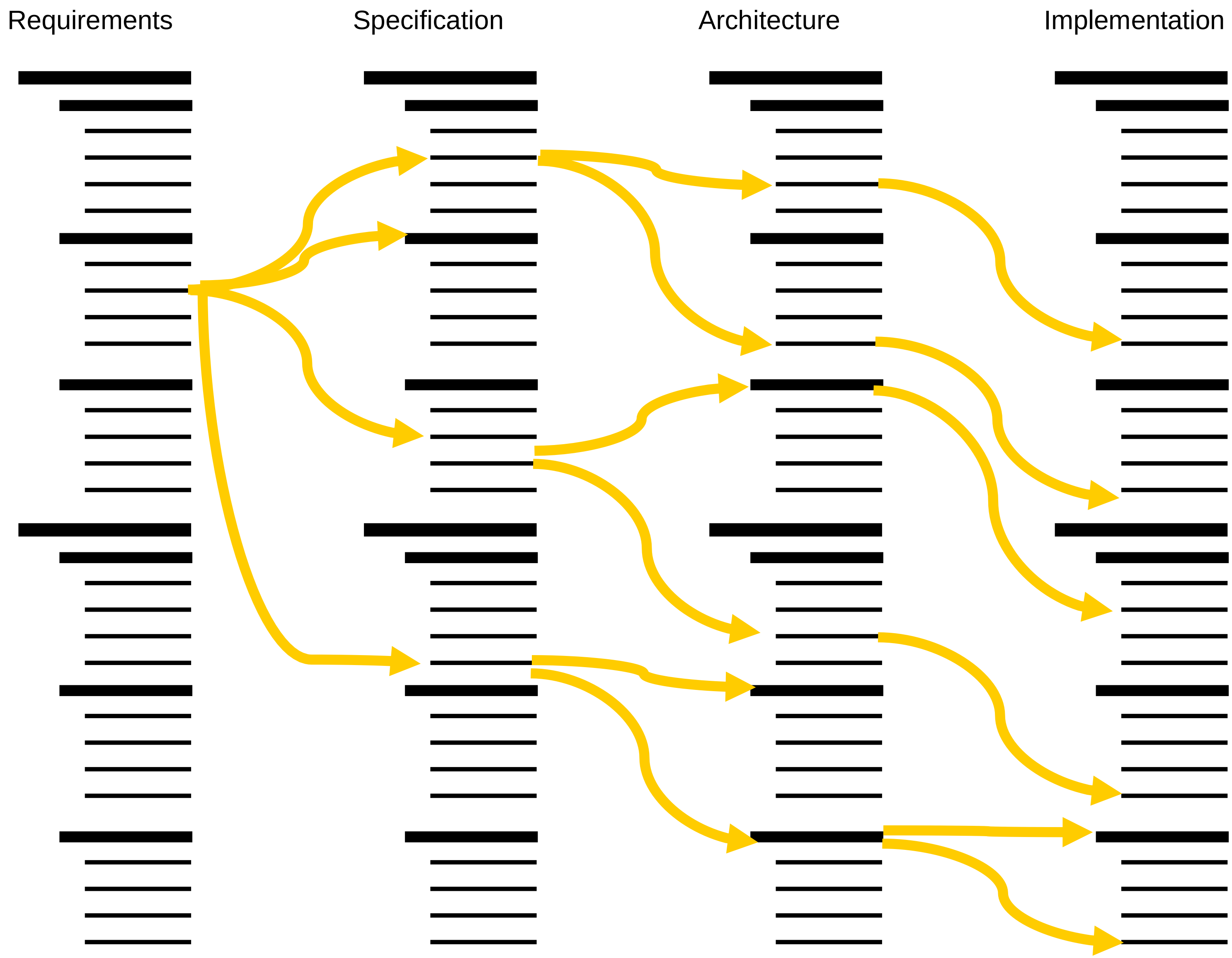
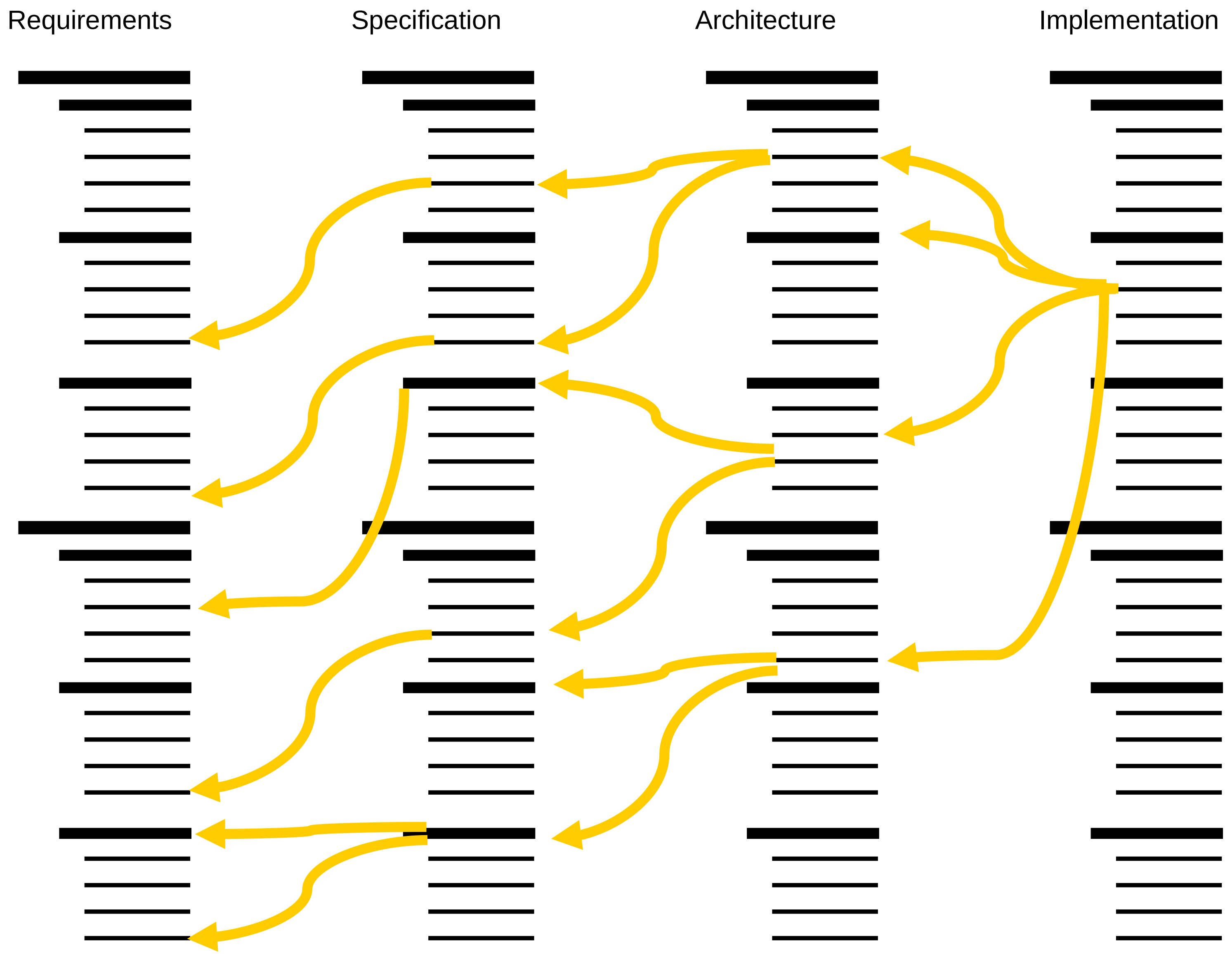


Illustration: Backward Tracing



Meaning of Links and Traces

A link relates two syntactic named content chunks – formalized as assertions

- A link has a meaning that usually is related to the meaning of the assertions 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*.

Three Artifacts

Three Levels of System 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."

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"

To formalize these statements by assertions we need appropriate ontologies on all three levels

In development we want to relate them by refinement or tracing

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

Three levels of system description in logic

- system level requirements

$$A = \wedge \{A_r : r \in R\}$$

- functional specification at system level - functionality

$$B = \wedge \{B_f : f \in F\}$$

- architecture specification

$$C = \wedge \{C_k : k \in K\}$$

- Correctness
  - ◊ functional specification correct w.r.t to requirements:

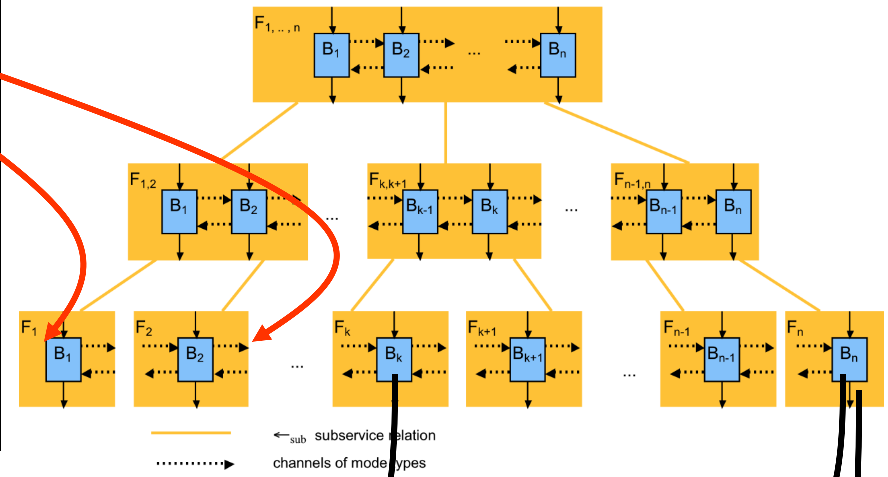
$$B \Rightarrow A$$

  - ◊ 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

	Functiona	Safety	Priority	Component	Function
A_1	...	Yes	high		
A_2	...	No	medium		
A_n	...	no	low		



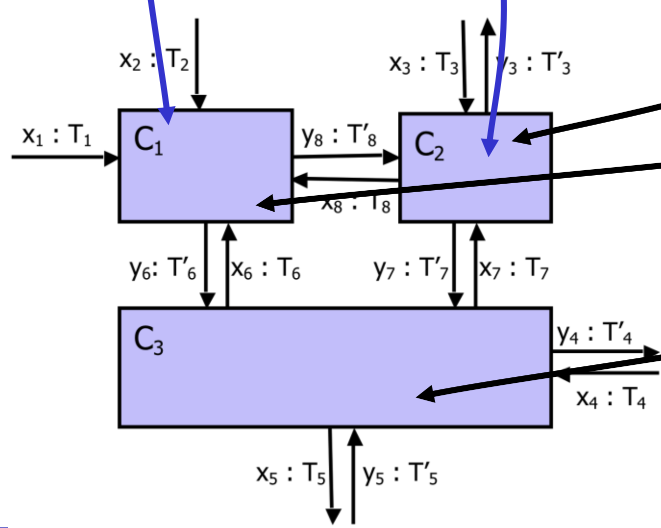
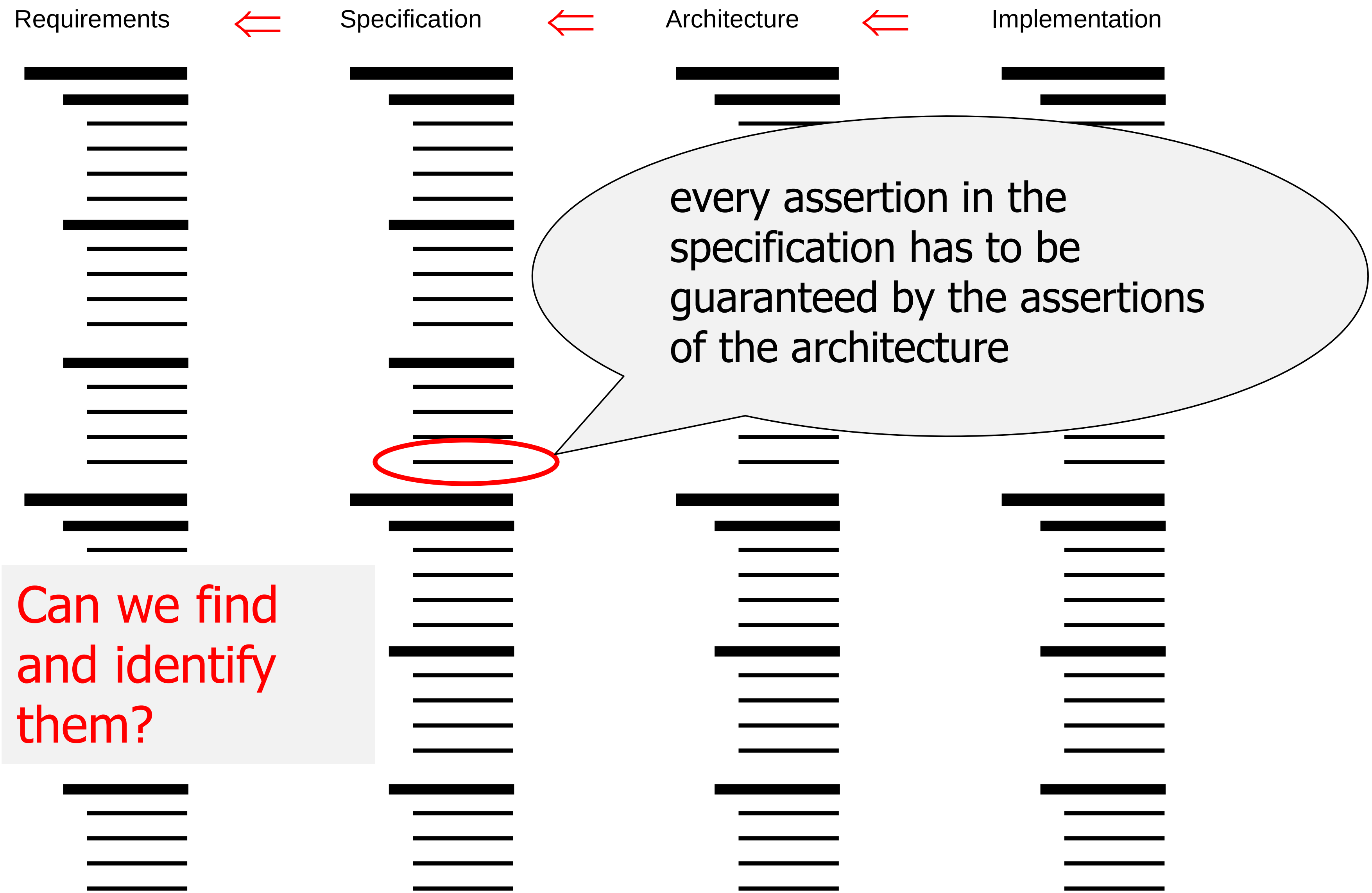
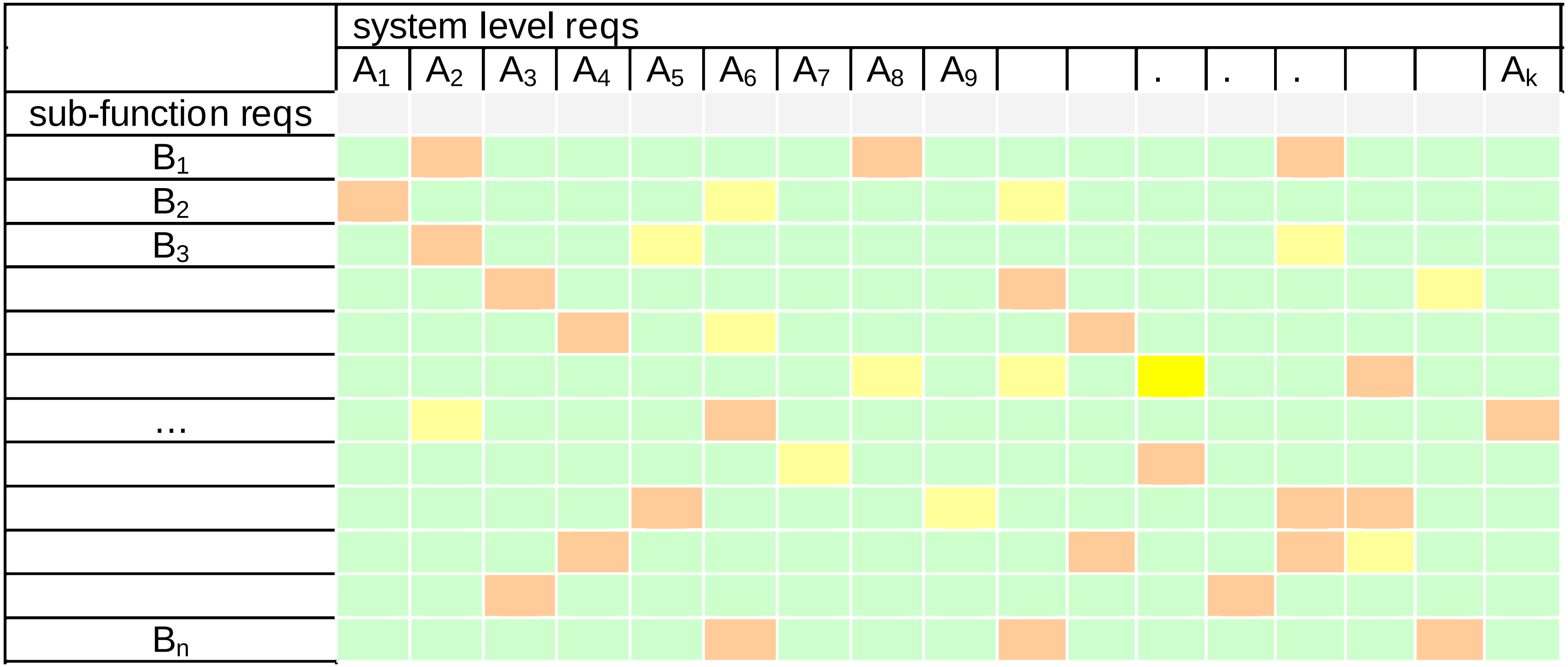


Illustration: correctness and refinement



Relationship: req spec to function spec - tracing

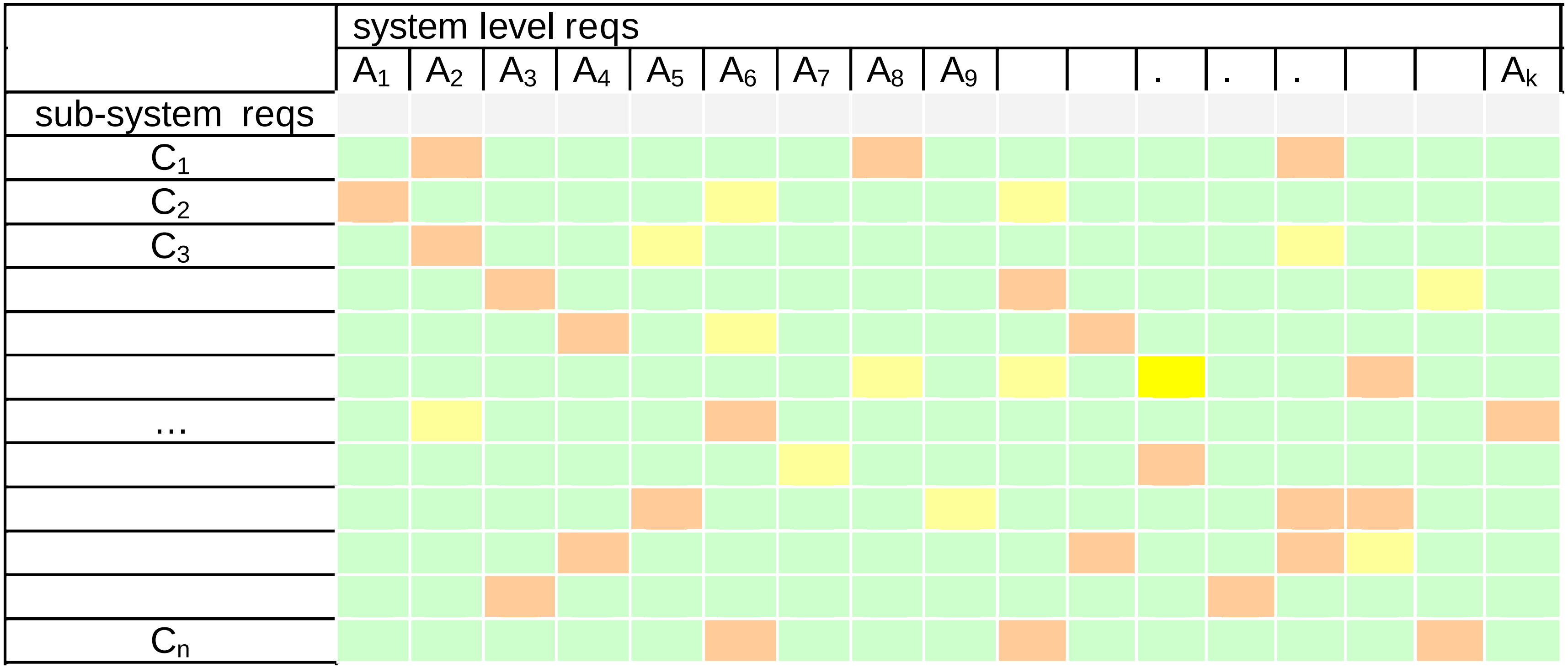


Red: B_i is strong guarantor of A_j

Yellow: B_i is weak guarantor of A_j

Green: B_i is not a weak guarantor of A_j

Relationship: architecture to requirements



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

Interfaces with assumptions

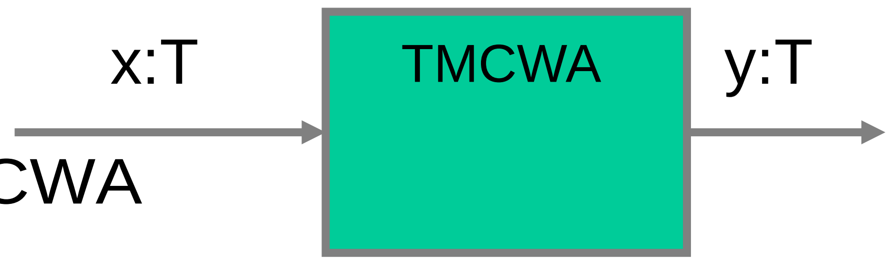
Often, in an interface specification for the syntactic interface (I ▸ O) we include

- an assumption asu(y, x) which is a specification of the *inverse* interface (O ▸ I) and defines properties of the context
- a commitment cmt(y, x) which is a specification of the behavior the syntactic interface (I ▸ O) as long as the assumption is fulfilled.

this leads to the specification

$$\text{asu}(y, x) \Rightarrow \text{cmt}(y, x)$$

Example: System interface specification

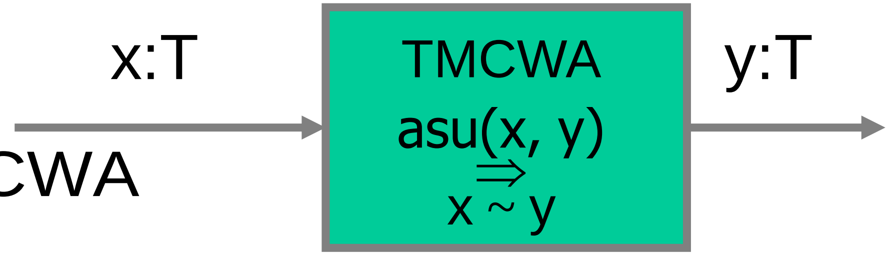


A transmission component TMCWA

TMCWA

in x: T out y: T
assume " $t \in \mathbb{N}: \#x\downarrow t \leq 1+\#y\downarrow t$
commit " $m \in T: m\#x = m\#y$

Example: System interface specification



A transmission component TMCWA

TMCWA

in x: T out y: T
$asu(x, y) \Rightarrow x \sim y$

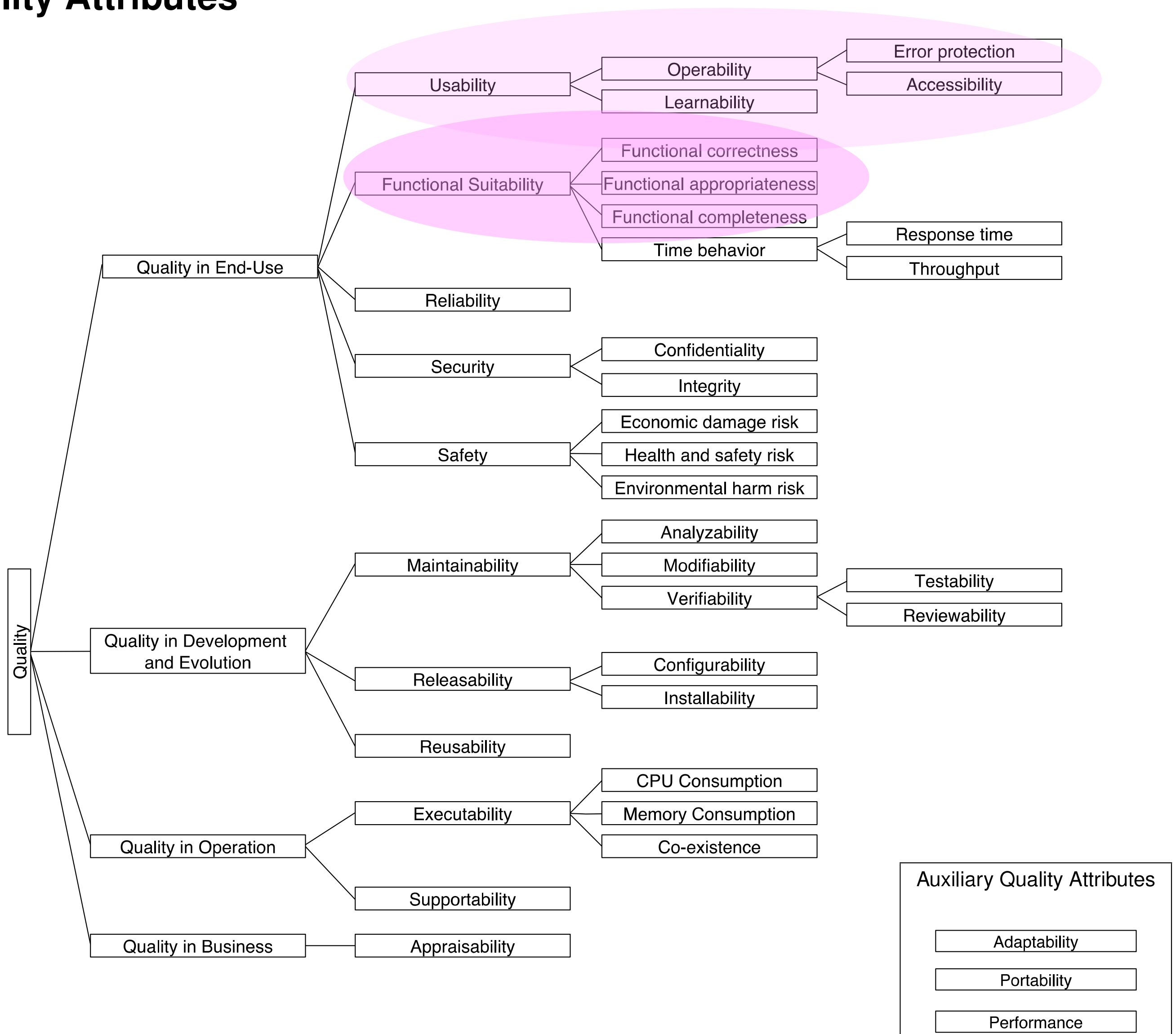
$x \sim y \equiv ("\ m \in T: m\#x = m\#y)$

$asu(x, y) \equiv ("\ t \in \mathbb{N}: \#x{\downarrow}t \le 1+\#y{\downarrow}t)$

We speak of a contract

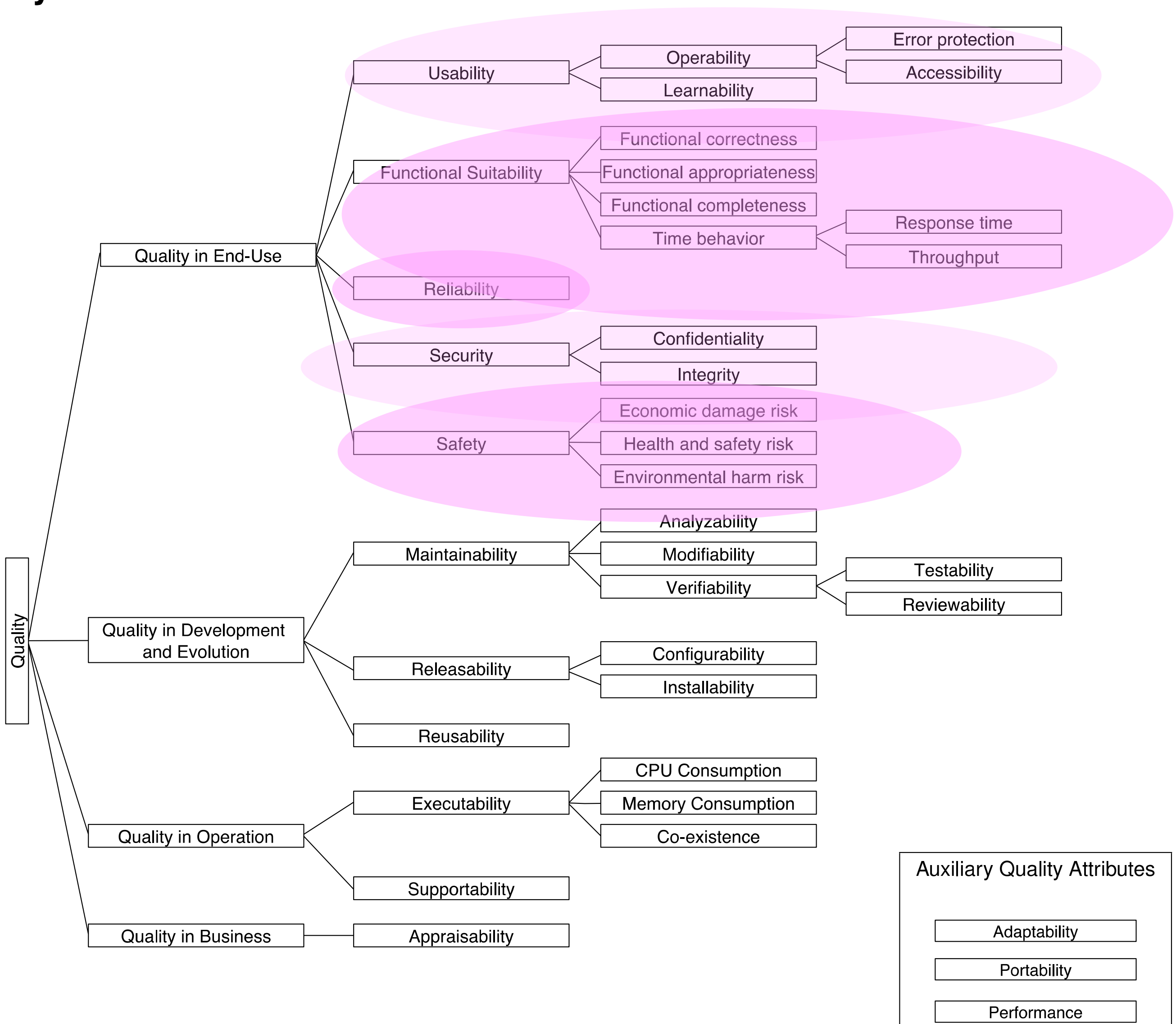
Quality model: functional requirements – conventional view

Main Quality Attributes



Quality model: functional requirements – novel view

Main Quality Attributes



Classification

- Functional requirements: logical and probabilistic interface behavior (including faults):
  - functional features
  - safety
  - reliability
  - …
- Architectural requirements: logical and probabilistic sub-system interface behavior (including faults)
  Quality requirements such as:
  - Performance
  - Security
- Requirements related to system context
  - Usability
  - Business - Return on investment

Conclusion

Probability

- Probabilistic system models and specifications are refinements of nondeterministic system models and specifications
- A rich set of so-called “non-functional” properties is captures by probabilistic interface specifications and thus become functional

Time

- Time-aware system models and specifications are refinements of non-time-aware system models and specifications
- For time critical systems so-called “non-functional” timing properties can be captured by by time-aware interface specifications and thus become functional

Performance

- There are two concepts of performance:
  - ◊ response time (in the case of non-time critical functionality)
  - ◊ efficient utilization of resources
- Response time is a functional property captured by time-aware probabilistic interface specifications