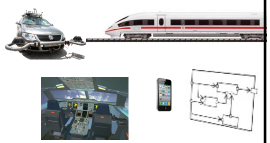
MBSE

Content of the lectures
Model-Based Systems Engineering (MBSysE) and
Model-Based Software Engineering (MBSE)

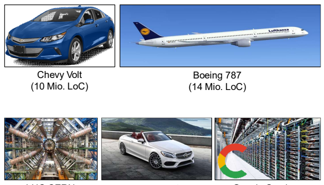


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

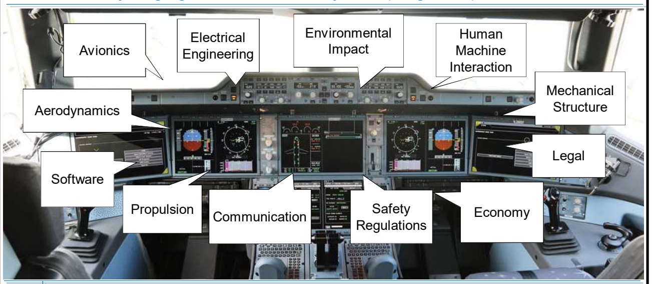
Why Modelling? We need to Compensate the Growth of Complexity

- **Society thrives on cyber-physical systems**
  - Communication, energy, home automation, manufacturing, medicine, transportation, …
- Added-value **mainly software**
- Software **complexity grows in magnitudes**
  - Distributed, self-adaptive, intelligent, …
- Modeling can **overcompensate the growth of complexity** in systems engineering
- Systems engineering is interdisciplinary
  - **Conceptual gap**: problem vs. solution domains



Chevy Volt (10 Mio. LoC)
Boeing 787 (14 Mio. LoC)
LHC CERN (50 Mio. LoC)
High-Value Car (100 Mio. LoC)
Google Services (2 Bio. LoC)

The Limits of my Language Mean the Limits of my World (Wittgenstein)

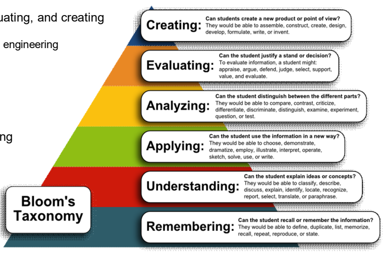


The Limits of my Language Mean the Limits of my World (Wittgenstein)

Modeling with domain experts requires **precise domain-specific modeling languages**

Learning Objectives

- Understanding, applying, analyzing, evaluating, and creating
  - Models by applying modeling methods
  - Functional modeling and models in systems engineering
  - Requirements modeling
  - Data modeling
  - Structure and behavior modeling
  - Systematic CPS engineering
- Syntax and semantics of selected modeling languages (including UML, SysML)
- Digital twins
- Quality assurance



Some First Literature

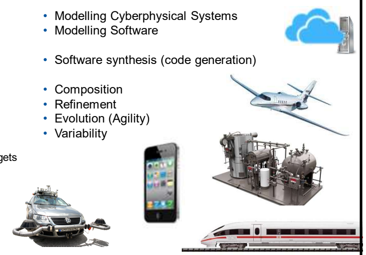
- [Rum16] Modeling with UML: Language, Concepts, Methods. Springer International, July 2016.
- [Rum17] Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International, 2017

- German versions are also available online: https://mbse.se-rwth.de/

Content of the Lecture

- Modelling in Development
- Software and Systems Engineering
- Development Methods (Agility, Scrum, V)
- Modelling Paradigms:
  - Data, Function, Structure, Behavior
- Modelling languages, e.g.
  - Class Diagrams for data and physical entities
  - StateCharts for state-based behavior
  - Architecture for function, component and gadgets

- Modelling Cyberphysical Systems
- Modelling Software
- Software synthesis (code generation)
- Composition
- Refinement
- Evolution (Agility)
- Variability



MBSE

1. Introduction and objectives
1.1. What is a Model?

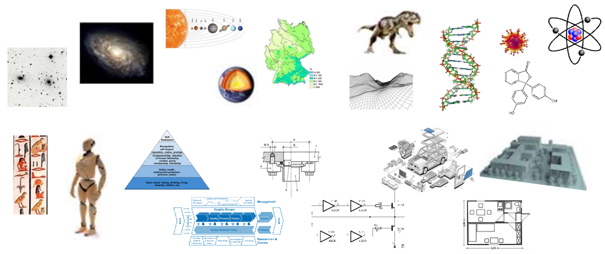
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

What is a model?

And what is it good for?

Models are Used in all Disciplines



Definition of the Term “model”

> A model is a reduced respectively abstracted representation of the original system in terms of size, detail, and/or functionality.

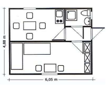
(Stachowiak 1973)

> Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems. (German original)

Consequences from the “model” Definition

> A model is a reduced respectively abstracted representation of the original system in terms of size, detail, and/or functionality.

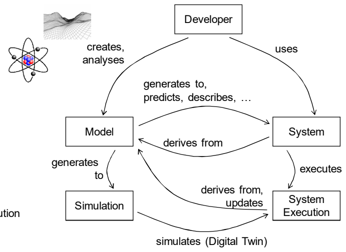
(Stachowiak 1973)



- There is an original
- Abstraction (reduction) is integral part of being a model
- Models have a purpose with respects to the original:
  - They are used to study the original
- Models can be prescriptive:
  - The model is designed first and the system then after the model
- Models can be descriptive:
  - The system exists and the model is used to study and understand the system

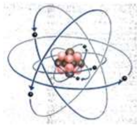
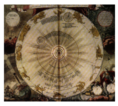
How Model and System relate

Forms of models:
- Mental model
  - exists in the head of a developer
- UML / SysML / DSL model
  - Engineering model for development: communication, generation, analysis, and simulation
- Implicit model (embedded in code)
  - Engineering model for simulation (no communication, no analysis, no generation)
  - May be usable as software part in a system
- Learned model (from data)
  - Derived from system or observed from system execution
  - (ML/NN: also in an implicit, "executable" form)



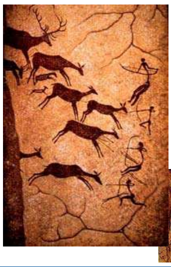
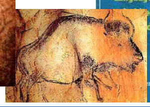
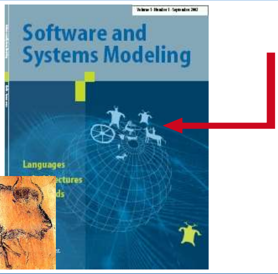
Example Physics: Validity of Models

- Rutherford's and Bohr's atomic models
- Einstein's theory of relativity
- Model of the Big Bang
- Physical laws are models, …
- Not all models are correct,
  - Geocentric model of Copernicus

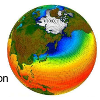
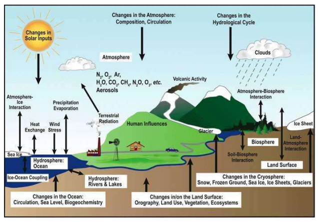

- Many models are only valid in certain boundaries

The First (Still Existing) Models: Ancient Cave Drawings

Model of Climate Change

- A model used for prediction
- There is exactly one original
- The model is complex:
  - Consisting of many sub-models for
  - Phenomena and geometric separation
- A bunch of integrated models
- The model is composed through various
  - Laws from theories (physics, etc.)
  - A large set of measurements
- The model is executable for simulation
  - And written in programming languages that (more or less directly) encode physical laws

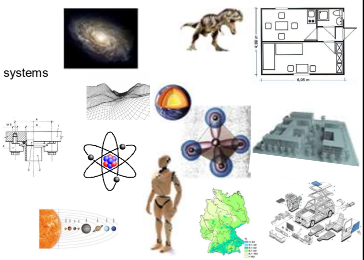



Modeling Needs a Sound Theory

- Models are composed
- Models are evolving
- Models are compared
- Models are analyzed
- Models are used to derive/generate/synthesize systems
- Models are defined using theorems and laws
- Consequently:

> Definition and use of models are based on an appropriate underlying theory

- Math is a well-known, very precise and rigorous foundation for such a theory.



MBSE

1. Models
1.2. An Example: Automata

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

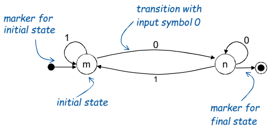
http://www.se-rwth.de/

Recognizing Automata

- Recognizing automaton (S,I,δ,s0,F) has
- (also: nondeterministic, alphabetical Rabin-Scott Machine (RSA))
  - Finite set of states S
  - Input alphabet I
  - Set of initial states $s0 \subseteq S$
  - Set of final states $F \subseteq S$
  - Transition relation $\delta \subseteq S \times I^{\varepsilon} \times S$

  where
  - ε represents the non-existent input characters in spontaneous transitions
    - $I^{\varepsilon} = I \cup \{\varepsilon\}$
  - All sets S, I, s0, F are non-empty and finite

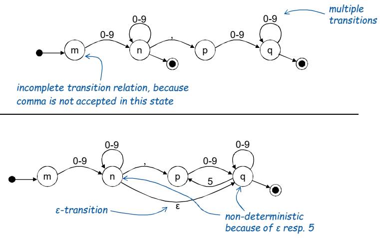
recognizing automaton



Examples of Recognizing Automata

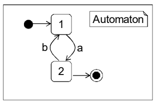
recognizing automaton

recognizing automaton



Example Automata Syntax: Model Representation by Graphics, Text and Math

- Graphical / diagrammatic:



- Textual in ASCII / UTF-8:

```
automaton Simple {
  state 1 <<initial>>;
  state 2 <<final>>;
  1 - a > 2;
  2 - b > 1;
}
```

- Mathematical:

Tuple $(S, I, 1, \{2\}, \delta)$
- Set of states $S = \{1,2\}$
- Set of inputs $I = \{a, b\}$
- Initial state $1 \in S$
- Final states $\{2\} \subseteq S$
- Transition function $\delta : S \times I \to S$
  - with $\delta(1, a) = 2;\ \delta(2, b) = 1$

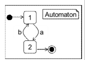
- Tabular:

source \ target	1	2 final
initial 1		a
2	b	

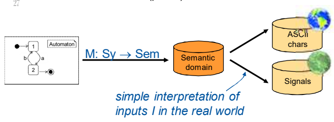
- typically restrictions apply (context conditions)
- more variants: XML/JSON-encoding, Java-encoding (State Pattern), …

Example Automata: Semantics Definition using Math

- Each representation of automata has its benefits
- However, math is efficient and optimal for constraints (context conditions), e.g.,:
  - All states are reachable: $\delta^*(s0, I^*) = S$
  - Automaton is total: $\forall s \in S,\ i \in I : \exists t \in S:\ \delta(s,i) = t$
    or short: $\mathrm{dom}(\delta) = S \times I$
- Math is also optimal for formal semantics definition:
  - Choice of the semantics domain:
    $Sem = I^*$
- Semantics mapping: the set of accepted words:
  $M(A) = \{\ w \in I^* \mid \delta^*(s0,\ w) \in F\ \}$
- (this definition assumes automaton is total)
- Various interpretations of domain I are possible:
  - ASCII characters (e.g., in parsing)
  - Signals (e.g., in communicating distributed systems)



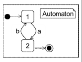
- Automaton syntax defined by tuple
  $Sy = (S, I, s0 \in S, F \subseteq S, \delta : S \times I \to S)$
  - Set of states S
  - Set of inputs I
  - Initial state $s0 \in S$
  - Final states $F \subseteq S$
  - Transition function (partial) $\delta : S \times I \to S$



PS: I^* is the set of words over alphabet I ;
δ^* the transitive closure over function δ

Example Automata: Nondeterministic Automata

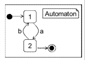
- First of all:
  - Nondeterminism and underspecification are related (almost the same)
- To introduce nondeterminism, we adapt the automaton syntax to:
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \to \wp(S))$
  - Set of initial states $S0$
  - Transition relation δ instead of function:
    δ can now offer multiple transitions (|δ(s,i)| > 1 allowed)
- Semantics domain uses again the set(!) of words over I:
  $Sem = I^*$
- Semantics mapping: the set of accepted words with a path to a final state:
  $M(A) = \{\ w \in I^* \mid \delta^*(S0,\ w) \cap F \neq \emptyset\ \}$



- Finite automata come with a rich theory and well-known techniques:
  - Powerset construction derives a deterministic automaton
  - Error completion
  - ϵ - Transition elimination
  - Equivalence checks (used e.g., by model checkers)
  - Mapping of regular expression to automata
    - Which includes various forms of automaton composition (∩, ∪, ¬, sequence .∘., Kleene closure .*)
- Theory helps to define semantics as well to efficiently map the automaton to an executable implementation

Example Automata: Automaton refinement and consistency

- Nondeterministic automata
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \to \wp(S))$
  - $Sem = I^*$
  - $M(A) = \{\ w \in I^* \mid \exists s0 \in S0:\ \delta^*(s0,\ w) \cap F \neq \emptyset\}$
- Automaton A is well defined: M(A) ≠ ∅
  - i.e. it accepts something
  - Syntactic sufficient criterion:
    - $\exists\ s \in S^*:\ \forall n:\ \exists i:\ s_{n+1} \in \delta(s_n, i) \wedge s_0 \in S0 \wedge s_n \in F$
  - Can effectively be checked using transitive closure
- Automaton A is refinement of B: M(A) ⊆ M(B)
  - i.e. A is more deterministic than B
  - Can effectively be checked using a simulation relation (see model checking)



- Automata A and B are consistent: M(A) ∩ M(B) ≠ ∅
  - i.e. the do not specify conflicting properties of a component
  - Can effectively be checked using an intersection automaton
- We recognize:
- Automaton theory demonstrates that:
  - M(A) ∩ M(B) = M(A ∩ B)
  - i.e. composition ∩ of automata is conform to individual mapping and composition of semantics
- Set theory is an excellent vehicle to understand consistency, underspecification and refinement

MBSE

1. Models
1.3. Theory

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

What is a theory?

And what is it good for?

Characterization of the Concept "Theory"

Definition:

A theory is an analytical tool for understanding, explaining, and making predictions about a given subject matter.

- This distinguishes from "theory" in some senses of common language:
  - A belief, policy, or procedure proposed
  - A hypothesis assumed for the sake of argument or investigation
  - An unproved assumption : conjecture
  - Abstract thought : speculation

"Nothing is more practical than a good theory."

Consequences from the Characterization of "Theory"

Definition:

A theory is an analytical tool for understanding, explaining, and making predictions about a given subject matter.

- A formal theory is syntactic in nature.
  - Thus a theory comes with an underlying language
- Theories are usually expressed mathematically, symbolically, or in natural language, but are generally expected to follow principles of rational thought or logic.
  - Consequence: A theory can always be constructed in the formal language of mathematical logic.

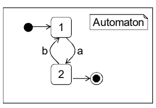
- Theory is constructed of a set of sentences that are entirely true statements about the subject under consideration.
  - Theories have underlying assumptions (axioms).
  - Theories have explanatory power, but also limitations.
- An axiomatic theory, consists of axioms and rules of inference.
  - A theorem is a statement that can be derived from those axioms by application of these rules of inference.

(partly adapted from Wikipedia)

Example Theory: Automata

The Automaton Theory has

- a foundational language / data structure / mathematical object:
  - usually a tuple: $(S, I, S0, F, \delta)$
- a rich body of algorithmic operations to
  - Transformation of automata
  - Derivation of relevant properties
- such as:
  - Removal of unreachable states
  - Transformation from nondeterministic to deterministic
  - Refinement
  - Composition of automata
  - Decision for language inclusion between automata, etc.
  - Slicing of relevant automaton parts
- and precise laws underpinning these algorithmic results



- Automata in practice also have many
  - concrete forms of syntaxes with
  - various extensions and
  - applications in various domains.

Example Theory: Math

- Math has been growing over hundreds of years.
  - math has become a big building consisting of a complex set of theories.
- Its basics:
  - Constructing math objects with: sets, functions, tuples, graphs and numbers
  - Laws as core mathematical property definition.
- Observations:
  - Math has tried hard to have only foundational axioms
  - To ensure consistency proofs are often re-done using different theorems
  - Proves can be seen as transformation of mathematical objects

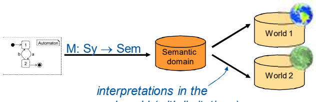
- Equation is the most important tool:
  - a+a = 2*a
- It has a very precise meaning (semantics):
  - It tells us what is equal,
- But also gives methodical assistance:
- the substitution property allows us to use it for transformation -- in both directions and with pattern matching, e.g.
  - a+a transforms to 2*a
  - 2*(x-1) transforms to (x-1)+(x-1)
  - a+a+a transforms to 2*a+a (or to a+2*a)
- Theory works best if the underlying objects of discourse are made explicit, a sound and consistent semantics is given, and operations on the objects are grounded on the semantics.

Modelling Language and its Theory is based on a Modelling Paradigm

For simplification and abstraction a modelling language theory is based on a modelling paradigm around which the theory is built. Highly relevant paradigms:

- Data structure
  - describing how data is organized (stored, transformed,...)
- Behavior
  - describing digital event and continuous processes
- Function
  - describing how (physical and software) components fulfill their duties
- Architecture
  - describes structure and relationships of components

- Care: Each paradigm induces an abstraction with underlying assumptions on the real world
  - these usually have limitations



interpretations in the real world (with limitations)

MBSE

1. Models
1.4. Language, Method

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Modelling Languages in the Software Domain

- Languages are a key for software development
  - Modeling languages like UML
  - Programming languages like Java
  - Markup languages like XML, HTML for various purposes
- Modelling languages enable to make models explicit and manageable ("first-class citizens").
- Explicit modelling languages enable to
  - Build automatic tools for model analysis, synthesis, code generation, etc.
  - Reuse models
  - Aggregate data to information in form of models
  - Models@Runtime enable adaptivity
- Computer science invented modeling languages
- Digitalization of other engineering domains enforces explicit languages for

Prolog Python Fortran Kolin Javascript Java Cobol Lisp Perl Sed, awk, bash Scala PHP Kotlin Statecharts Ada SQL C# Delphi Groovy ABAP MATLAB SIMULINK Groovy Scheme C TypeScript Haskell R Julia Eiffel C++ Rust Lustre Verilog XML

Modelling Languages in for Systems Engineering

- Digitalization of engineering domains demands explicit languages for as well
- Languages are a key for systems engineering
  - Physical modeling: Modelica, Simulink
  - CAD: STEP, NX CAD, ECAD
  - Simulation: Dymola
  - Knowledge: OWL, RDF
  - Integration: AutomationML
  - Circuits: VHDL
  - Building Information Models (BIM)

MODELICA CMSD DAML+OIL SysML4Mechatronics MATLAB SIMULINK ECAD STEP FFBD MODAF EXPRESS-G Statecharts HLA UML4IoT VHDL <AutomationML/> Dymola STEP-NC IDEF0 XML SysML4Modelica DiSpa BIM OPM RDF MechatronicUML UMM

Unified Modeling Language (UML)

~1990: OOD Booch; OOSE Jacobson; OMT Rumbaugh et al.; ...

1995: Booch / Rumbaugh / Jacobson

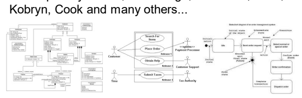
1997:	UML 1.1
1999:	UML 1.3
2001:	UML 1.4
2003:	UML 1.5
2005:	UML 2.0
2007:	UML 2.1.2
2009:	UML 2.2
2010:	UML 2.3
2011:	UML 2.4.1
2015:	UML 2.5
2017:	UML 2.5.1

- UML is a second-generation notation for object-oriented modeling

The Unified Modeling Language is a Graphical Modeling Language for Software Systems

Features

- Elements for specification, communication and documentation
  - among developers
  - developers with users
  - union of several previously existing methods
- Set of modeling concepts and concrete notations
- Standardized since September 1997 by OMG
- Developed by Booch, Rumbaugh, Jacobson, Selic, Kobryn, Cook and many others...

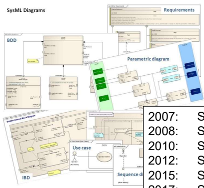


Goals

- Description of essential properties of the program like in a blueprint
- Structuring of problem and solution
- Abstraction of implementation details
- Definition of various views covering several paradigms
  - task assignment and workflows
  - software and system architecture
  - interaction between components
  - behavior of components
  - implementation
  - physical distribution

Systems Modeling Language (SysML)

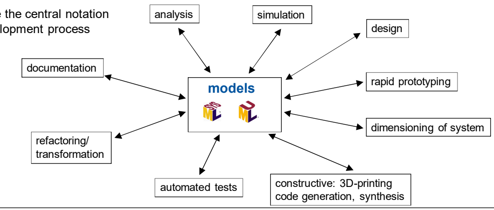
- SysML is dedicated to model the software part of (embedded) systems
- It started as variant of UML, but will probably become independent (with 2.0)
- SysML reuses 7 of UML's 14 diagrams, and adds 2 new diagrams
  - requirement and parametric diagrams



2007:	SysML 1.0
2008:	SysML 1.1
2010:	SysML 1.2
2012:	SysML 1.3
2015:	SysML 1.4
2017:	SysML 1.5
2019:	SysML 1.6
~2024:	SysML 2.0

Model-Based Development

- Models are the central notation in the development process



- Models can serve as central notation for systems development
- A good modeling language can be used for analysis and synthesis

Needs for modeling during development of systems and software

M1. Specifying systems in requirements engineering

M2. Formulating and evaluating design alternatives

M3. Describing system aspects or views for communication

M4. Designing system architectures

M5. Describing systems for validating desired system properties in simulations

M6. Collecting user feedback through visual simulations, prototypes, and mock-ups

M7. Modeling variants in product lines

M8. Defining reference models for the capture, design, or implementation of requirements

M9. Statically analyzing or verifying design decisions

M10. Efficiently evolving designs

M11. Understanding semantic differences between versions

M12. Describing detailed system behavior for generating software parts

M13. Implementing/realizing/synthesizing systems in general

From: [BR23] M. Broy, B. Rumpe: Development Use Cases for Semantics-Driven Modeling Languages. In: Communications of the ACM, Volume 66(5), pp. 62-71, ACM, May 2023.

Needs for modeling during system operation

M14. Customizing systems

M15. Monitoring running systems

M16. Capturing deviations between desired or even optimal (modeled) properties and observable (realized) system functionalities

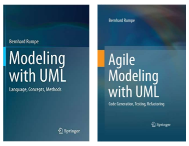
M17. Documenting systems

M18. Capturing system execution traces and labeling them with model elements, thus linking system and traces reliably

From: [BR23] M. Broy, B. Rumpe: Development Use Cases for Semantics-Driven Modeling Languages. In: Communications of the ACM, Volume 66(5), pp. 62-71, ACM, May 2023.

Models in Software Engineering

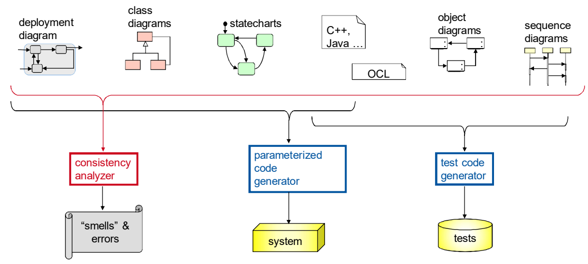
- Industry standard: Unified Modeling Language
  - 15 kinds of diagrams (class diagrams, Statecharts etc.)
- Industry standard: Systems Modeling Language
- But beyond the UML and SysML:
  - Petri Nets
  - Logic
  - Relations
  - Dataflow diagrams
  - SDL
  - Finite automata
  - Nassi-Schneidermann diagrams
  - etc.
  - Algebraic Specifications
  - Entity/Relationship-Models
  - Jackson Structured Diagrams
  - Control flow diagrams
  - Grammars
  - Regular expressions
  - BPMN



UML-P - UML-Profile for agile Modeling

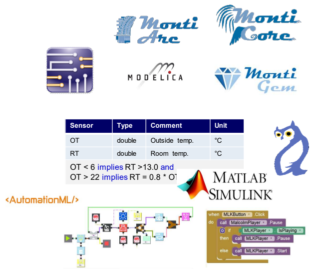
Tutorial on language, semantics, code generation, test cases, test pattern, refactoring, evolution
http://mbse.se-rwth.de/

Agile UML-based Software Development: Constructive Use of Models for Coding and Testing



Domain Specific Language (DSL)

- A domain specific language (DSL) is a software language specialized to a particular application domain.
  - A general-purpose language (GPL) in contrast is broadly applicable across domains and lacks specialized features for a particular domain (such as C++, Java, …).
- allows us to model application domains and systems in those domains like
  - business, telecommunication, traffic, …
- addresses the application domain instead of the technical solution
- usually built on one modelling paradigm
- is not necessarily "executable" or "complete"



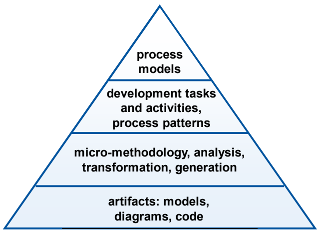
General Programming Language (GPL) vs SysML/UML vs Domain Specific Language (DSL)

- Explicit models written in UML/SysML → Systems Engineering
  - + Model can be analyzed (formal methods!)
  - + Relatively compact (and manageable)
  - - For execution: code generator needed
- Explicit models written in a DSL → Various domains with recurring problem structures
  - + Model can be analyzed (formal methods!)
  - + Very compact models
  - - Language tools need to be developed
  - - For execution: code generator needed
- Implicit "models" written in a programming language (GPL)
  - + Model can easily be executed
  - + very general programming techniques, Turing complete
  - - Relatively technical and awkward "models"
  - - Execution is the only purpose → Simulation in Systems Engineering
  - - No high-level analyses possible

Java
Python C#
C++
Kotlin
C

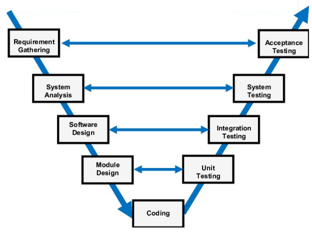
The Methodological Pyramid

- Process models, such as RUP, V-Model, define the overall development process .
- They are composed of an appropriate set of development tasks and activities, such as "elicit requirements", "review the architecture"
- To accomplish these tasks a large set of "micro methods", e.g. using a best practice, a design pattern, tools for analysis, generation or synthesis, tools for evolution and transformations, etc.
- All these tasks are finally executed on the set of artifacts, that contains all relevant development information, such as requirements, all kinds of models, tests, code.



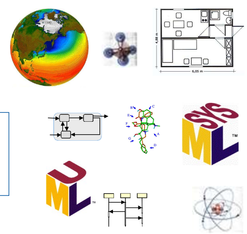
V-Model: A Standard Process to Develop Software

- The V-Model has
  - a constructive left wing:
    - from requirements, analysis, design, coding
  - and a quality assurance and testing right wing:
    - from unit tests to acceptance tests
- In the V-Model
  - each activity on the left corresponds to tests on the right
- The V-Model assumes manual work in all activities, it is agnostic to models and automation
- In practice: more than 2/3 of the work occur on the right side



Models are Most Useful if Grounded on a Theory of Modeling

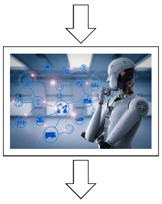
- In a software & systems development process:
  - Models are composed
  - Models evolve
  - Models are compared
  - Models are analyzed
  - Models are used to derive/generate/synthesize systems
- This works best if there is
  - A) a precise understanding of what a well-defined model is (syntax)
  - B) a precise definition of what a model means (semantics)
  - C) an elaborated underlying mathematical theory consisting of theorems and laws that give us tools at hand to analyze, transform, test, etc. the models
  - Syntax and semantics are covered by modeling language definitions
  - Theory is an underlying supplement for the semantics definition



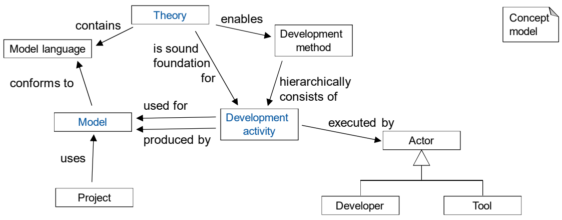
Automation Of Development Steps

- "Automation has proven to be the single most effective means of making dramatic improvements in both productivity and product quality"

  Bran Selic, 2019
- Automating :
  - Generation, synthesis
    - Constructive derivation of implementations or more detailed forms of models
    - Transformation
    - Correctness by construction
  - Analysis
    - Consistency checking, completeness, well-formedness rules, etc.
    - Applicability of transformations
    - Automatic verification
  - Testing
  - Simulation
    - As a technique for testing and dynamic analysis
  - … and many more



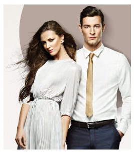
Summary of this Lecture in form of a Concept Model



- This conceptmodel illustrates relevant concepts/terms and their relationships of this lecture.

Excercise your understanding of Models: Who/What is (not) a Model?

Appendix

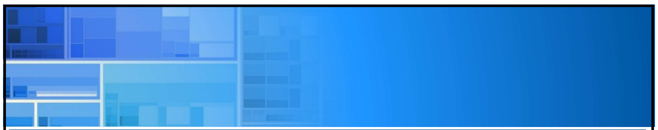
exercise

What is a Model, What is an Original?

Appendix

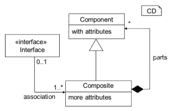
(Rene Magritte)

exercise



MBSE

2. Modeling Structures with Class Diagrams
2.1. Object-Oriented Structural Modeling in a Nutshell

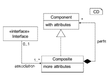


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Structural Modeling for CPS

- Modeling cyber-physical systems needs to describe the structure of relevant objects
  - physical components
  - software components
  - data
- Often many instances/copies of an object are needed
  - classes describe sets of objects (screws, engines, kinds of data values)
- A system is structurally decomposed in subsystems and components.
  - structural modelling is used throughout the development
- This chapter discusses class diagrams as the language for structural modelling using the object-oriented paradigm.

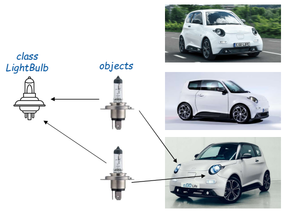
Object-Orientation is an Intuitive Modeling Paradigm Close to our Perception of Reality

- We perceive individuals (objects) that
  - have properties (attributes) and
  - act in their environment (through functions)
  - e.g.: Muhammad Ali, Albert Einstein, Ronaldo
- Objects belong to classes, which define their type
  - e.g.: Person, Boxer, Athlete, Scientist, Citizen, …
- Classes define properties and functions of all their objects
  - e.g.: Each Person has a name
  - e.g.: Each Boxer has a history of fights
  - e.g.: Each Scientist can conduct research and Scientists can act as Citizens
- Structuring systems and their parts via "classified" objects supports
  - grouping of properties and functions are modelled together
  - encapsulation: some of the properties are internal, access / connection through an explicit interface — e.g.: Current mood of a Person
  - inheritance: objects inheriting from a supertype inherit properties and functions — e.g.: each Scientist is a Person
  - polymorphism: an object (instance of a class) can occur as element of several "supertypes" and act as such — e.g.: Boxers and Scientists can act as Citizens

Objects in the Physical World (Systems Engineering)

Basics

- A system consists of a dynamically changing number of physical objects
- Objects represent entities of the domain and instances of exactly one class
- An object can be uniquely identified
- An object has a state as defined by its properties
  - result of an operation depends on the current state
- An object has a behavior modelled by the functions of its class
- Objects connect state with functional behavior.

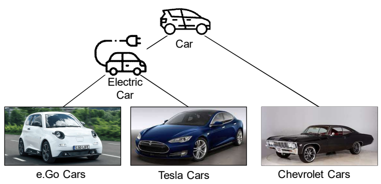


Concepts of Object Orientation

Class

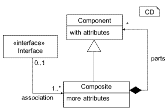
Basics

- An object belongs to exactly one class
  - a class defines the properties and the functional behavior of its objects.
- Classes are organized in a generalization tree, in which the properties and the functional behavior are refined in sub-classifications.



MBSE

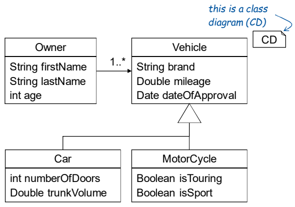
2. Modeling Structures with Class Diagrams
2.2. Modeling with UML Class Diagrams



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

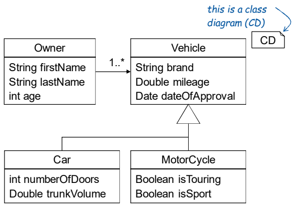
http://www.se-rwth.de/

Example of a Class Diagram



- Characterizes all relevant properties and associations of drivers and vehicles for a specific application or domain

Example of a Class Diagram

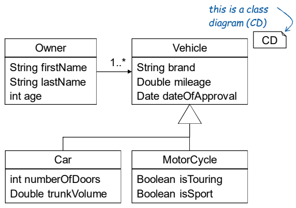


- Characterizes all relevant properties and associations of drivers and vehicles for a specific application or domain

Meaning of this Class Diagram (CD)

- It defines a class `Owner`
  - each instance of `Owner` (`Owner` object) has three attributes `firstName`, `lastName`, `age` (attributes are defining properties of owners)
  - each `Owner` object is related to 1 or more `Vehicle` objects
- a class `Vehicle`
  - each `Vehicle` object has three attributes
- a class `Car`
  - a `Car` is a special type of `Vehicle`
  - each `Car` is a `Vehicle`
  - each `Car` object can be used as a `Vehicle` object
  - each `Car` object has at least the attributes inherited from class `Vehicle` (and possible more)

Meaning Of Class Diagrams

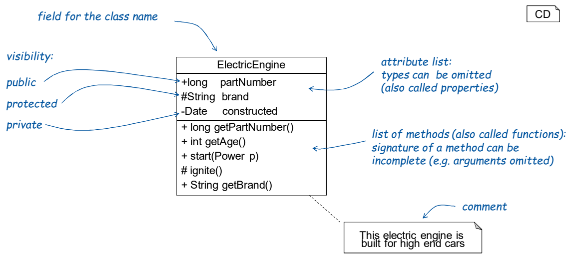


- Characterizes all relevant properties and associations of drivers and vehicles for a specific application or domain

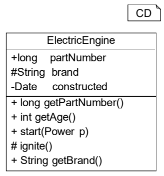
General Meaning Of Class Diagrams

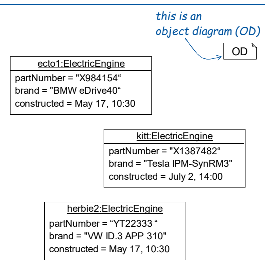
- A class diagram defines a set of possible object structures
- Classes define sets of objects with shared properties
- An object is instance of one class (this class is also the type)
- Classes can relate to other classes
  - governs their relations (associations) to other classes

Class with Attributes and Methods



Class and some Object Instances

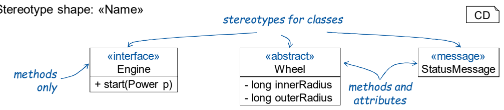




- A class has many objects as "instances"
- Objects have concrete attribute values

Stereotypes

- Stereotype classifies a model elements (e.g., class or attribute)
- Specializes the meaning of the model element
  - allows a special representation
  - target-specific code generation
  - etc.
- Stereotype shape: «Name»



Stereotype is

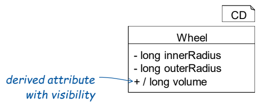
- Predefined
- Meant for external access
- No instances

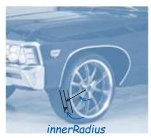
- Predefined
- No direct instances (but subclasses)

- Custom defined stereotype
- has application- or company-specific meaning

Derived Attributes

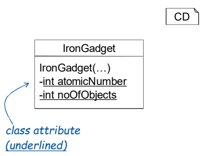
- If an attribute can be calculated (derived) from others, then it is labeled with "/".
- The relationship (calculation) of the attribute can be given using e.g. logic or math:
  - volume = 2 * π^2 * innerRadius² * outerRadius
- In OOP:
  - Derived attribute can be stored redundantly
  - Or can be calculated by a method each time





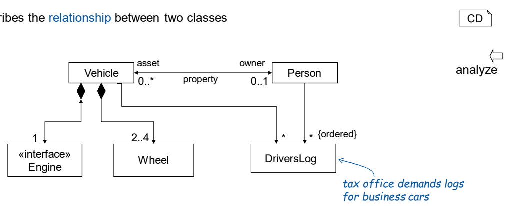
Class Attributes and Methods

- Certain attributes (and methods) are equal for all objects of a class: they do not belong to individual objects, but belong to the class
  - E.g. the atomic number of a material or the number of existing instances of a class
- In OOP (e.g. Java) "static" keyword is used; in the UML such elements are underlined.
- In OOP constructors are also static elements that belong to the class
- Methodical guideline: to be used as little as possible (inheritance issues: no access to super, no use with interfaces, …)

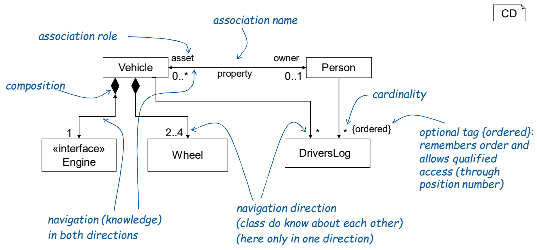


Associations

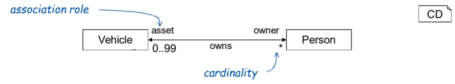
- An association describes the relationship between two classes



Associations -2



Associations - Roles, Names, Cardinalities



- Association as binary relation between classes
  - "Person owns Vehicle"
  - the person is navigating to the vehicle using
    - association role "asset" on the opposite side
    - a person may own of up to 99 cars (0 .99)
- Cardinalities
  - exactly one: 1
  - optional: 0..1
  - arbitrary: *
  - not null: 1..* (or +)
  - fixed intervals: 3..9, 17 , 21, 42..99 (but rarely used)
- {ordered} access can be used on cardinalities >=2

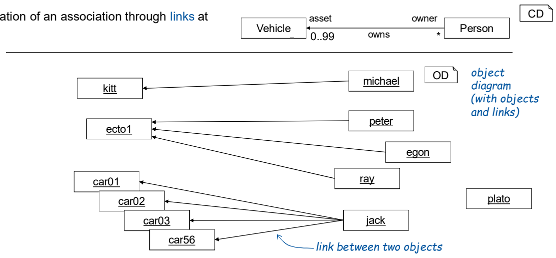
Role Names

- Role names are used for navigation (logical and/or in the implementation)

- What if the role name is missing?
- Alternatively usable, if unambiguous
  a) class name of the opposite class with small caps
  b) association name
- Example
  - given: Vehicle v
  - equivalent are:
    - v.owner, v.person, v.property

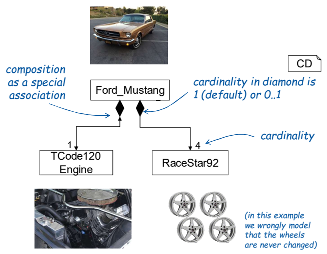
Associations relates Classes and Links relate Objects

- Manifestation of an association through links at runtime:

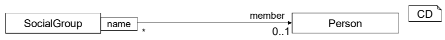


Composition

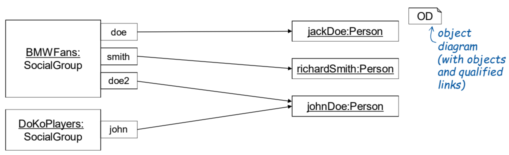
- Composition = special form of association
- Semantics of composition
  - composite is composed of parts
  - objects form a strongly coupled unit
  - parts depend on the composite
    - life cycle is combined (destroying the composite also destroys parts)
    - replacement of part is (normally) not possible
  - however: different interpretation in tools
- Consequence: part (object) can only be in one composite (structure)



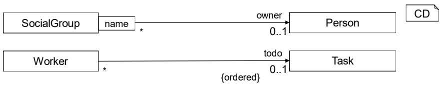
Qualified Association and Links



- Qualification "name" allows to select individual Person objects
- The same object can linked to different SocialGroup objects through different qualifications (names)

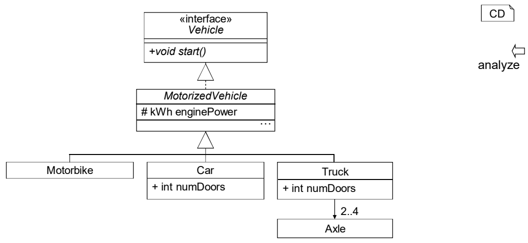


Qualified Associations

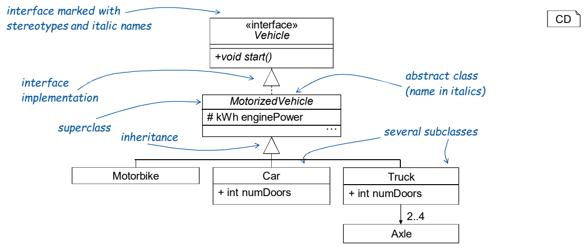


- Qualified associations allow selection of individual objects from a set using a qualifier
- Qualifiers can be:
  - integer interval (0 -..), if association is {ordered}
  - explicit identifier (attribute) of the target object (here: "name")
- Composition can also be qualified
- Qualification at both ends is possible
- Qualified association provides additional mechanisms for processing
  - selective access, selective modification

Generalization, Inheritance and Interface-Implementation

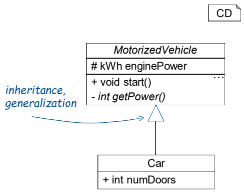


Generalization, Inheritance and Interface-Implementation



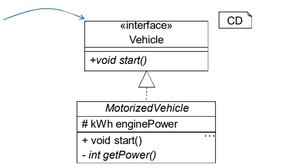
Generalization and Inheritance

- Generalization uses a semantic (classification) view
  - hierarchical structuring of classes
  - subclass is a subtype
  - subclass describes a subset of the objects of the superclass
  - substitution principle
    - instances of the subclass are usable, where instances of the superclass are allowed
- Inheritance is a technical mechanism used in OOP:
  - inheritance between pairs of classes
  - attributes and methods are transferred from superclass to subclass
  - further attributes and methods can be added
  - method overriding is allowed



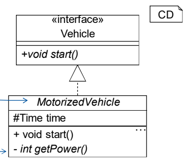
Interfaces

- Interface are used in OOP
  - an interface describes the signatures of a collection of methods, that belong together.
  - unlike classes
    - no attributes (only constants)
  - interfaces may also inherit from one another
- Classes implement interfaces
  - similar to inheritance
- Methodological use:
  - structuring access for classes
- Interfaces are mainly for software coding in OOP
  - Reason: e.g. implement multiple interfaces in a class

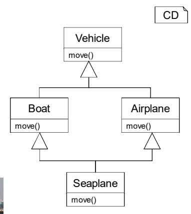


Abstract Class

- Abstract Class is used in OOP
  - representation: italic
    - or «abstract»
  - builds a hybrid form between an interface and a "normal" class
  - implementation in form of method bodies and attributes are partly available
  - abstract methods without implementation
  - but: no instances (objects)



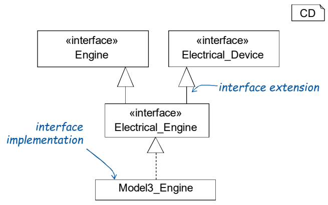
Multiple Inheritance



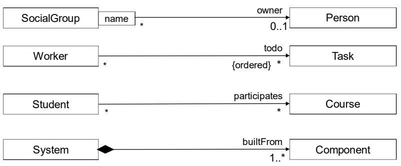
- When using generalization there may be "overlapping" classes
- Multiple classifications possible, e.g., the Seaplane
- UML modeling permits
  – a class inherits from multiple classes
- Java does not,
  – for technical reasons
  – but allows classes to implement multiple interfaces
- Kotlin allows this multiple inheritance
  – demands reimplementation of move() in subclass Seaplane

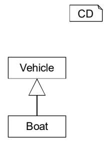
Interface Implementation

- Mainly applicable for OOP:
- An interface can extend multiple other interfaces
- A class can implement multiple interfaces
- UML: a class can inherit from multiple classes



Kinds of Associations

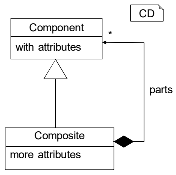




inheritance is not an association (between objects): Two classes, but only one Object is instantiated

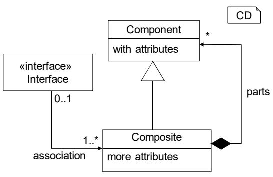
On the modeling power of CDs

- Associations can model arbitrary graph structures
  – Set: use *-associations
  – List: use *-associations with {ordered}
  – Map: use qualified association
  – Tree, Graph: use *-association with recursion
- Famous: the composite design pattern [Gamma et.al. 93]
  - Composite: manages sets of components
  - Components may also be composites
  - Additional "Leaf" classes build the atoms
  – Care: it is the responsibility of the software to ensure that its objects form a tree (and not a cyclic graph)



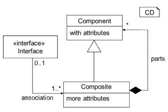
Summary: UML Class Diagram Syntax

- Class diagrams have
  – classes with attributes and methods
  – abstract classes, interfaces
  – inheritance, interface extension and implementation
  – associations with
    - names, roles, cardinalities, navigation directions
  – variants of associations
    - composition
    - qualified association
- Stereotypes and tags specialize individual model elements
- Not discussed extensions
  – aggregation, association with > 2 partners, …



MBSE

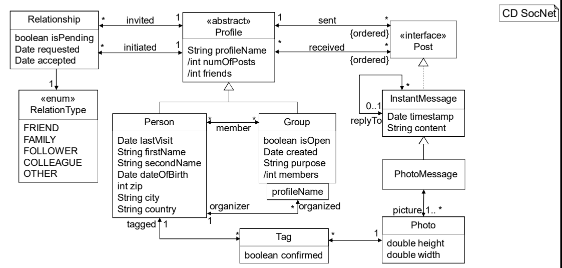
2. Modeling Structures with Class Diagrams
2.3. CD Modelling based on the UMLP-Tool



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Example "Social Network"



We Use a Textual Notation for CDs

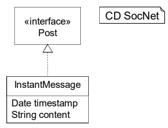
- A class diagram thus becomes a text-file of this form and is stored in `SocNet.cd`

CD

```
classdiagram SocNet {

  class {...}

  class {...}

  enum {...}

  association ... ;

  composition ... ;
}
```


Classes and Interfaces

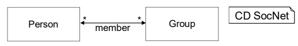
- Classes and interfaces look similar to Java code
- but neither contain methods nor visibilities.




```
classdiagram SocNet {

  interface Post;

  class InstantMessage implements Post {
    Date timestamp;
    String content;
  }
}
```


Associations (I)

- An association in the CD:




```
association member [*] Person <-> Group [*];
```

assoc. name

navigation direction,

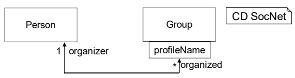
class name

multiplicity, can be [*], [1], [0..1], [1..]*

- Navigation directions: ->, <-, <-> and --
- Optional:
  - Association name, multiplicities

Associations (II)

- An association in the CD:




```
association [1] Person (organizer)
                <->
        (organized) [[profileName]] Group [*];
```

role names

qualifier

- Role names for navigation
- Qualifier is an
  - (1) a type (notation: `[Type]`) or
  - (2) attribute of the opposite class (notation: `[[attribute]]`)

Composition

- A composition in the CD
- Composed part (B) is part of the composition (A).
  - Composed object may only occur once in a composition
- Examples:

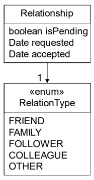
```
composition c1 A  -> B;
composition c2 A <-> B [*];
composition c3 A [[theQualfier]] -- B [1];
composition c4 A -- B [*] <<ordered>>;
composition c5 A -> B [1..*];
```

Enumerations

CD SocNet

- An enumeration (enum) can be used as type:

```
class Relationship {
  boolean pending;
  Date requested;
  Date accepted;
}

enum RelationType {
  FRIEND, FAMILY, FOLLOWER, COLLEAGUE, OTHER;
}

association Relationship -> RelationType [1];
```



Appendix

The full CD SocNet in textual form (I)

CD SocNet

```
package dex;
import java.util.Date;

classdiagram SocNet {
  abstract class Profile {
    String profileName;
    /int numOfPosts;
    /int friends;
  }
  class Person extends Profile {
    Date lastVisit;
    String firstName;
    String secondName;
    Date dateOfBirth;
    int zip;
    String city;
    String country;
  }
  class Group extends Profile {
    boolean isOpen;
    Date created;
    String purpose;
    /int members;
  }
  association member [*] Person <-> Group [*];
  association [1] Person (organizer) <-> (organized) [[profileName]] Group [*];
```


Appendix

The full CD SocNet in textual form (II)

CD SocNet

```
 class Relationship {
    boolean isPending;
    Date requested;
    Date accepted;
  }
  association invited   [*] Relationship <-> Profile [1];
  association initiated [*] Relationship <-> Profile [1];
  enum RelationType { FRIEND, FAMILY, FOLLOWER, COLLEAGUE, OTHER; }
  association Relationship -> RelationType [1];

  interface Post;
  association received [*] Profile <-> Post [*] <<ordered>>;
  association sent [1] Profile <-> Post [*] <<ordered>>;
  class InstantMessage implements Post {
    Date timestamp;
    String content;
  }
  association [*] InstantMessage <-> (replyTo) InstantMessage [0..1];
  class PhotoMessage extends InstantMessage;
  association [1..*] Photo (picture) <-> PhotoMessage;
  class Photo {
    double height;
    double width;
  }
  class Tag {
    boolean confirmed;
  }
  association [1] Person (tagged) <-> Tag [*];
  association [*] Tag <-> Photo [1];
}
```


CD4A Context Conditions - 1

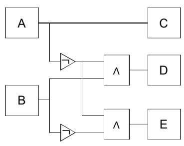
- Diagram Name
  - Must match file name
  - First character upper-case
- Keywords
  - May not be used for types, e.g., "class", "implements"
- Classes, Interfaces, Enumerations
  - Must be unique
  - First character upper-case
  - Enum constants must be unique within an enum
- Extends / Implements
  - No inheritance cycles
  - Classes may only extend classes, interfaces only interfaces
  - Only interfaces may be implemented
- Attributes
  - Start in lower-case and must be unique
  - Type must be resolvable and (optional) initialization must be type compatible
  - Overriding attribute in sub class must be of same type as the attribute in super class

CD4A Context Conditions - 2

- Associations
  - Source may not be an enumeration or external type
  - Ordered associations must have a cardinality greater than 1
  - Qualified
    - Attribute of attribute-qualifier must exist in referenced class
    - Type of type-qualifier must be resolvable
  - Names:
    - Association names and role names must start in lower-case
    - Association names, role names and implicit role names (lower-case name of target type) must not conflict with each other or attributes in the source class
  - Composition
    - Cardinality on side of the composite is [1] (default) or [0..1].
- Types
  - Generics may not be nested
  - Usage of generics must be parameterized with the correct number of type-parameters (e.g., invalid: Optional, Optional<>, Optional<A,B>, valid: Optional<A>)
  - Derived attributes may not be initialized

Graphical vs. Textual Models

- Models can have different representations:
  - e.g., graphical,
  - textual,
  - mathematical
- Best fitting form depends on the purpose of use.

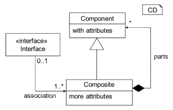


$$c = a$$
$$d = \neg a \wedge b$$
$$e = \neg a \wedge \neg b$$

```
C := A;
D := !A && B;
E := !A && !B;
```

MBSE

2. Modeling Structures with Class Diagrams
2.4. Software and Systems Modeling with CD

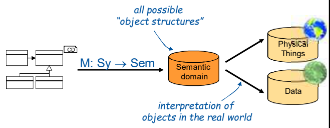


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

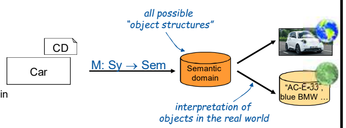
Class Diagrams, their Semantics and Interpretation in the Real World

- Class diagrams have
  - syntax: boxes, arrows, triangles, text, …
  - semantics (meaning) as intrinsic property:
    a class diagrams describes a set of object structures.
- To be discussed:
  - interpretation of the "objects" in the real world
- Systems engineering has different interpretations than software engineering
  - object can be a physical thing
  - vs. object can be data in a computer
- Examples
  - cyber-physical objects: car, plane, spacecraft, factory, screw, water, energy, …
  - data and events flowing in the system, …
  - beings: you, me, Alan Turing, Mickey Mouse, …
  - abstract objects: plan, recipe, …



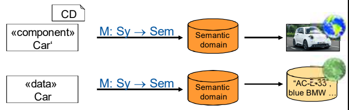
Class Diagrams, Semantics and their Interpretation in Real World -2

- Given a class "Car".
- What does it describe?
- Two standard interpretations:
  - the set of real cars (on the street)
  - data about cars, as e.g. stored in production, at EuropCar, in a tax office, or in Flensburg
- These are describing different, but related things
- During a development: the interpretation of a class diagram may change (from the real thing to the data about it)
- Data objects obviously describe physical objects
  - in the system development both interpretations may be used, because both (data and cars) are present in a CPS

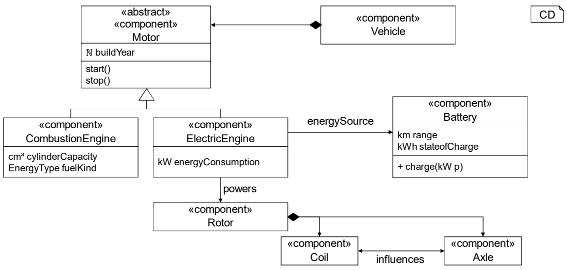


Stereotypes for Semantic Interpretation

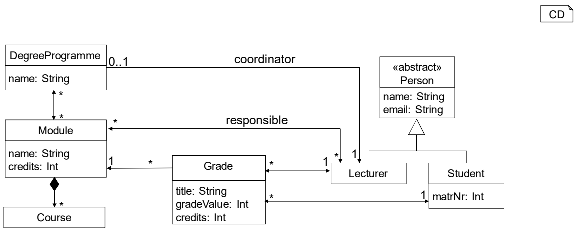
- Two standard interpretations:
  - the set of real cars (on the street)
  - data about cars, as e.g. stored in production
- The interpretation can be embodied in the model using stereotypes
  - «material»… elements, compounds, alloys, …
  - «component»… machinery, …
  - «energy»… types of energy
  - «being».. humans, animals, …
  - «data» … for object structures, and other forms of data
  - By default: no such stereotype = data
- More fine-grained stereotypes are possible, e.g:
  - «signal» … data sent around
  - «subsystem»
  - «item»



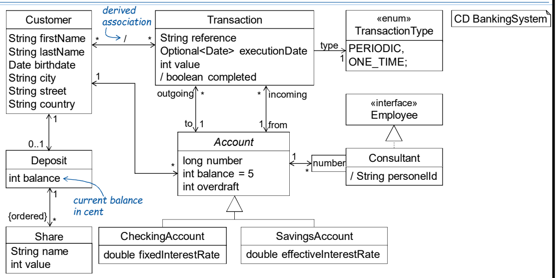
Structural Modeling of Components – Example



Data Structure of a Bachelor/Master Course – Example



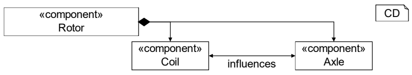
(Simplified) Data Structure of a Banking System -- Example



Associations and Composition in Software and System Structure

Modelling System Structure:
- Composition: a natural concept in the physical world
  - Widely used in physical CD's
- Associations in a physical world are of various forms
  - (but rarely used)
  - they can describe
    - physical connections
    - some form of interaction in the functional sense
    - or even a development dependency ("same height as")
- Generalization: as classification concept

Modelling Software:
- Composition: used to build/compose software sub-systems
- Associations used intensively
  - Object graphs know each other and interact along associations
  - Data structures realize associations (see Entity/Relationship models, SQL, …)
- Generalization/Inheritance: for classification and for reuse of methods and as extension principle



Modeling With Class Diagrams

A comparison between modelling of systems and coding in software (implementation):

UML CD Element	Interpretation in Systems Modelling	Interpretation in Software (Coding in OOP)
object	real physical item; and element of a defining class	instantiated from a class
class	used as classification resp. type for objects	software class, database table, acts as "blueprint" for its objects
attribute	property of an object	storage for a value
«abstract»	Usable in a generalization hierarchy to mark that all objects are elements of subclasses	abstract class (cannot be instantiated)
«interface»	N/A	software interface (like in Java, C++)
attribute visibilities +, -, #	N/A	defines access in software
method	denotes a function that a physical object can do	computational method

Modeling With Class Diagrams -2

UML CD Element	Interpretation in Systems Modelling	Interpretation in Software (Coding in OOP)
inheritance	generalization hierarchy: the elements of a subclass belong to the superclass too	generalization AND(!) reuse of code from superclasses
composition	geometric and functional composition of physical objects to higher components / systems	data objects composed to higher components (but OOP doesn't directly support composition)
associations	physical relations (glued together, interacting, etc.)	data connections and underlying infrastructure for interactions (method calls)
qualified associations	rarely used	models lists of objects and maps of (key,value) pairs
identity	"physical identity", e.g. the sum of its atoms	unique identifier to access an object from elsewhere

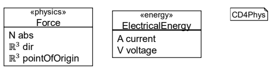
General Duties of Class Diagrams

Purpose of an UML CD	In Systems Modelling	In Software Modelling (Coding in OOP)
Encapsulation of attributes and methods into a conceptual unit	-	++
Instances as objects	+	++
Type specification of objects (~ all possible objects)	++	++
Extension (~ all objects that exist at a certain moment)	++	+
Characterization of the possible structures of a system (~ all objects that might exist at any time point in any system run)	++	++
Conceptual modeling of application field	++	++
Implementation description (~blueprint)	-	++
Class code (the translated and executable form of the implementation description)	-	++

Data in the Physical World

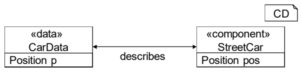
- Software knows types, such as
  - int, long, double, float
- Physical modelling prefers SI Units, such as
  - km, s (time), A (Ampere), etc.
- Typing a data value appropriately is helpful to avoid
  - Modelling errors, like:
  - conversion issues
  - illegal additions (e.g. 2 km + 3 Volt)
  - illegal assignments (e.g. v = 3 m2/s)

- Ideal solution 1:
  The Modelling language provides SI-Units
  - SI-Units as types, such as "km/h"
  - SI-Expressions, such as "3 m/s * 7min"
- Solution 2:
  Class diagram models "physical classes", e.g.



Connecting Data and Physical Classes

- In a CPS model, we sometimes need to model
  - the set of real cars (on the street), AND
  - data about cars, as e.g. stored in production
- Associations between classes of the physical and the data part of a system are allowed
  - but they neither model data, nor physical properties
- Such an association is used as intellectual modelling construct to allow bridging the worlds in specifications
  - E.g. to describe relations between their attributes
- Note: Type "Position" is a type like natural numbers (ℕ x, ℕ y) and can be used as data type as well as real property type in the physical world.

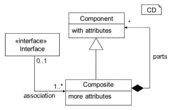


- We require the following:
  The data about a cars position and the real position only deviates 1m:
  - inv CarData d:
    | d.p - d.describes.pos | < 1m

(modelled in the Object Constraint Language (OCL))

MBSE

2. Modeling Structures with Class Diagrams
2.5. Further Aspects

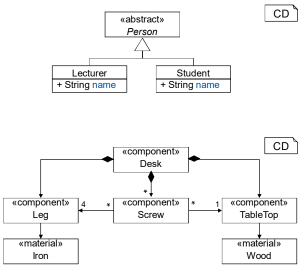


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

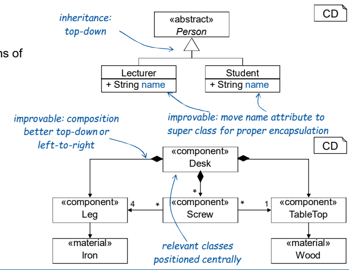
Quality of a Class Diagram Model

- Quality is essential for usability, correctness and other important model properties
- Quality is defined relative to its purpose:
  - Does the model fulfill its purpose?
- Few main categories of quality issues:
  - A. Is the model correct?
  - B. Is the level of abstraction good?
  - C. Is the model presented well? (Readable, understandable?)
- Techniques to ensure quality?
  - E.g. peer review, syntax checks, prototyping

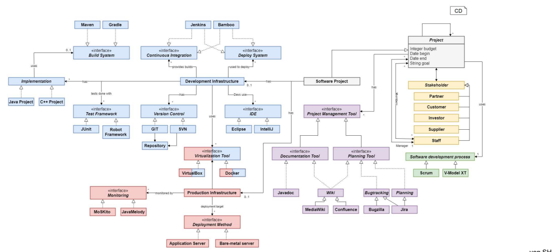


Quality of a Class Diagram Model

- Quality relevant questions should be refined:
-
- Does it model the correct properties and functions of all its object structures?
  - E.g. with respects to requirements
- Details:
  - 1. Is it sufficiently detailed?
  - 2. Is it not overly detailed?
  - 3. Shall it be complete or underspecified?
- Is it presented well?
  - 1. Readable?
    - comments are appropriate
  - 2. Well arranged?
    - inheritance: from top to bottom or left to right
    - composition: most relevant classes in the mid
    - unnecessary redundancy

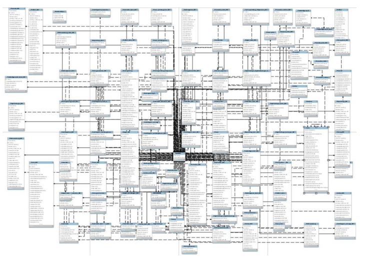


Datenstruktur des Artefakt Modells



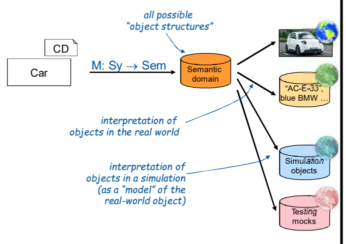
von SH

Example CD



Class Diagrams and their Interpretations

- A class "Car" has
- two typical interpretations:
  - the set of real cars (on the street)
  - data about cars, as e.g. stored in production
- Many additional interpretations possible:
- In a system simulation
  - the «material» "Car" class gets another interpretation:
  - it acts as surrogate for the real object
  - the «data» "Car" class remains unchanged
- In software testing
  - the «data» "Car" class may be mocked
- In data bases
  - class "Car" becomes a data table



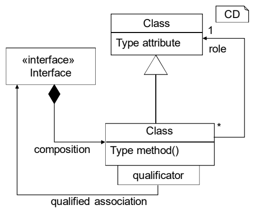
Example: MontiGem Code Generator from Class Diagrams

- Multi-user web-application for data management
- Developed using MBSE and lots of code generation
  - Generate full application stack
- Starting point:
  - Class diagram modelling the application data
  - (+ some GUI models)
  - + Application functions

Screenshot of MaCoCo (Management Cockpit for Controlling), developed by AGe, PH, JM, LN, SVa, GV, and others

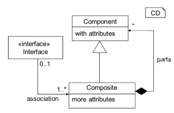
Summary: Class Diagrams for Structure and Data

- Object-orientation is paradigm of data modeling close to our perception of reality
  - classes model objects
  - associations their relations
  - inheritance allows for classification and for "code / property reuse"
- UML class diagrams can be applied to describe:
  - data in a system
  - physical objects (components) of the system using stereotypes «material» and «component»
  - energy types «energy»
  - types of fluids «material»
  - events in a system «event»
  - Context of a system, which then also may include humans
- Class diagrams are also used to describe meta-things, e.g. relevant for development tools, see SLE lecture



MBSE

3. Deriving Software from Class Diagrams
3.1. Code Generation from Classes

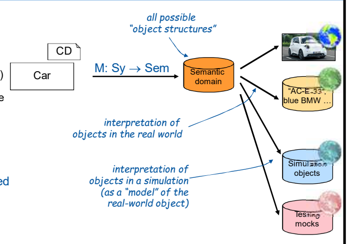


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

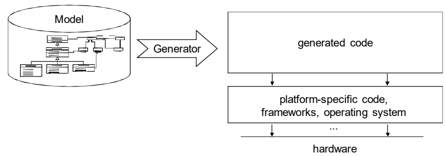
Using Class Diagrams

- A class has many interpretations
  - Physical: the set of real objects (car on the street)
  - Data: data about objects (car data stored in production)
  - Simulation: «material» objects acts as surrogate for the real object
  - Testing: «data» classes may be mocked
  - Data bases: class becomes a data table
- Dependent on interpretation a class diagram is used in different forms
- One prominent form is: derive code from a class diagram to be used in
  - Tests, simulations and/or the real product



Code Generation

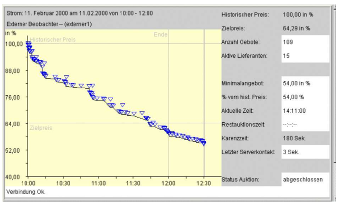
- Principle: mapping the model to a programming language artifact



⇨ Even if no "automated" generator is available, the following transformations of UML to code are useful

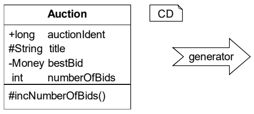
Running Example: Online Auction System

- A supplier online auction system
  (we developed for a company in the 90ties)
- Characteristics
  - One purchaser calls for the auction
  - Multiple bidders apply for a supply contract
  - Bidding downward (the cheapest gets the contract)
  - Real-time auction with duration of typically 2 h
- In the example:
  - Auction of the annual electricity needs of large bank in Frankfurt (value: millions)
  - Result: 46% cost reduction

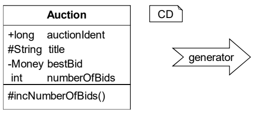


Snapshot of the web applet in the browser

Code Generation from a Class



Code Generation from a Class



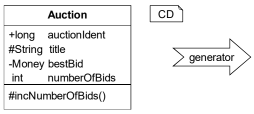
Java

```java
class Auction {
  public long      auctionIdent;
  protected String title;
  private Money    bestBid;
  public int       numberOfBids;

  protected void incNumberOfBids() { ... }
}
```

Challenges for Generators:

- Diagram may be inconsistent
  - invalid data type, attribute name twice, ...
- Class diagram does not contain method bodies?
  - How will these be completed?
- Diagram is incomplete
  - not all attributes,...
- Alternative forms of generation?

Alternative Code Generation?

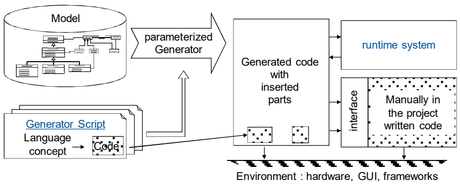


Java

```java
class Auction {
  public long      auctionIdent;
  protected String title;
  private Money    bestBid;
  public int       numberOfBids;

  protected void incNumberOfBids() { ... }
}
```

- Possible requirements or additional features
  - get/set methods for attributes
  - serializability of objects
  - storing objects in a database table
    - e.g., creation of the table as SQL statement
  - attribute access is secured by security manager
  - platform dependency of the code
- Different requirements lead to different generators
  - technique 1: parameterization of the generator
  - technique 2: generating against an abstract interface: providing a runtime system (similar to the Java Virtual Machine)

Code Generator: Parameterization + Runtime System

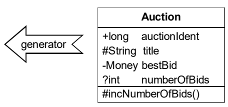


basic structure of the generated code with
- *parametrization*
- *handcrafted modules*
- *"runtime system"*

Code Generation using get/set-Methods

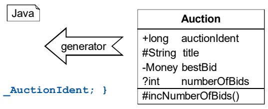
```java
class Auction {
```

Java



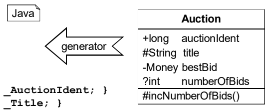
Code Generation using get/set-Methods

```
class Auction {
  private long   _AuctionIdent;




  synchronized public long getAuctionIdent() { return _AuctionIdent; }




  synchronized public    void setAuctionIdent(long x) { _AuctionIdent =x; }





  }
}
```



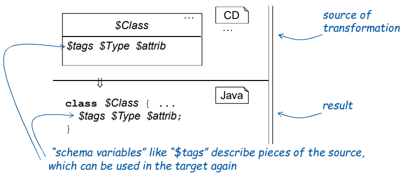
Code Generation using get/set-Methods

```
class Auction {
  private long   _AuctionIdent;
  private String _Title;
  private Money  _BestBid;
  private int    _NumberOfBids;

  synchronized public long getAuctionIdent() { return _AuctionIdent; }
  synchronized protected String getTitle()   { return _Title; }
  synchronized private Money getBestBid()    { return _BestBid; }
  synchronized public int getNumberOfBids()  { return _NumberOfBids; }

  synchronized public    void setAuctionIdent(long x) { _AuctionIdent =x; }
  synchronized protected void setTitle(String x)      { _Title =x; }
  synchronized private   void setBestBid(Money x)     { _BestBid =x; }
  synchronized public    void setNumberOfBids(int x)  { _NumberOfBids =x; }

  synchronized protected void incNumberOfBids() {
    setNumberOfBids(getNumberOfBids()+1);
  }
}
```



Script for Code Generation

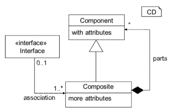
- Example:



- If necessary, further transformations describe the adaptation of individual parts
- Depending on visibility ("/", "+", etc.) , variants of these translation rule may be used
- Representation of these scripts in tools highly different!

MBSE

3. Deriving Software from Class Diagrams
3.2. Code generation for inheritance and associations

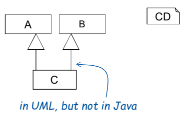


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Inheritance

- Inheritance, interface implementation and interface extension can be mapped directly to Java
- Problem: one class inherits from several superclasses:
- Three solutions:
  a) a super class is converted to an interface
  b) delegation instead of inheritance
  c) combination of both
- Selection of the solution depends on the context



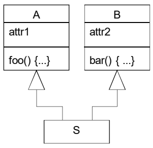
Handling of Multiple Inheritance

In the model:

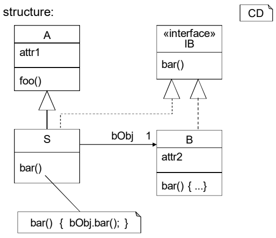
discuss

Handling of Multiple Inheritance

In the model:



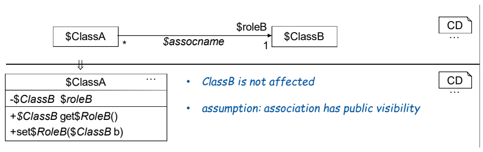
Code structure:



- But:
  - two objects contain the new distributed state: more complex

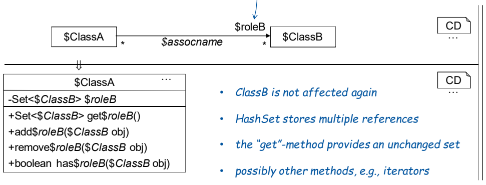
1-to-*-Association

- Simple association 1-to-1 or 1-to-*
  - has navigation in one direction only
- Code generation described by this transformation:



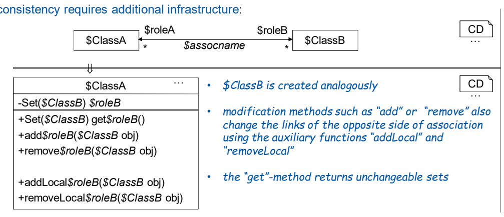
-to--Association

- Navigation only in one direction



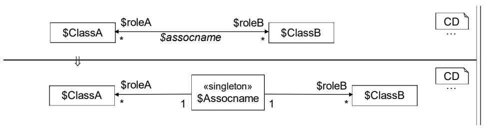
-to--Association with Navigation in both Directions

- Implementation in a decentralized variant
- In principle, management of the association on both sides as before
- But: consistency requires additional infrastructure:



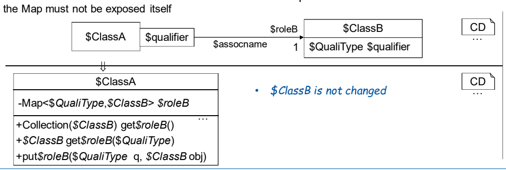
-to--Association with Navigation in both Directions

- Implementation in a centralized version with a Singleton
  - An *association class* manages links centrally
- Class $Assocname uses internally a relation navigable in both directions
- Access from $ClassA or $ClassB via one central object, i.e. the singleton
- But: more complex internal management structure



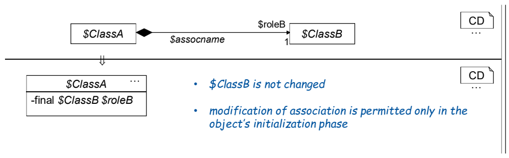
Qualified Association

- HashMap allows the realization of a qualifier
- But: redundant storage of the qualifier in a HashMap and the target class
  - may require that qualifier value cannot be changed in the target
- Access functions and modifiers can be offered in like the Map
  - but the Map must not be exposed itself



Composition

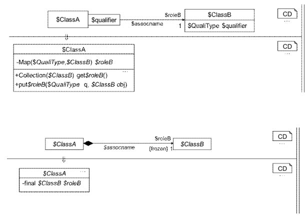
- Composition treated like an association
- But: (temporal) dependency of the sub-object is not realized
- Possible solutions:
  - A) developers must respect composition themselves
  - B) access signature is reduced, preventing sub-objects to be removed



- *$ClassB is not changed*
- *modification of association is permitted only in the object's initialization phase*

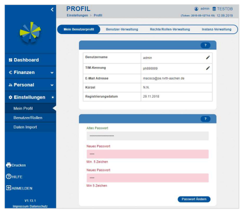
Summary Code Generation from CD

- This sections showed code generation from class diagrams for several constellations
  - variety of syntactic elements: many possible variants
  - some variants are optimal in various contexts
  - selection is not trivial!
- The transformations shown can be understood as guidelines for manual implementation
- But also: Code generation from class diagrams can be automated



Example: MontiGem Code Generator from Class Diagrams

- Multi-user web-application for data management
- Developed using MBSE and lots of code generation
  - Generate full application stack
- Starting point:
  - Class diagram modelling the application data
  - (+ some GUI models)
  - + Application functions



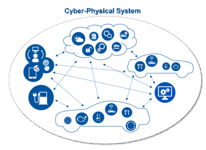
Screenshot of MaCoCo (Management Cockpit for Controlling), developed by AGe, PH, JM, LN, SVa, GV, and others

MBSE

4. System and System Engineering
4.1. System

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



What is a system?

And how to engineer it?

Definition System

A system is a set of entities, real or abstract, comprising a whole. (Wikipedia)

An engineered system is a technical or socio-technical system which is the subject of an SE life cycle.
It is a system designed or adapted to interact with an anticipated operational environment to achieve one or more intended purposes while complying with applicable constraints. (INCOSE)

Challenges in Systems Engineering

General observations:

- Mission complexity is growing faster than our ability to manage it […] increasing mission risk from inadequate specifications and incomplete verification.
- Knowledge and investment are lost between projects … increasing cost and risk: dampening the potential for true product lines. (Bradley Drake, et.al.)
- Knowledge and investment are lost at project life cycle phase boundaries … increasing development cost and risk of late discovery of design problems.
- System design emerges from pieces, rather than from architecture … resulting in systems that are brittle, difficult to test, and complex and expensive to operate.
- Most major disasters such as Challenger and Columbia have resulted from failure to recognize and deal with risks.

General System Theory Distinguishes Various Types of Systems

Boulding (1965):

1. Structures (Bridges)
2. Clock works (Solar system)
3. Controls (Thermostat)
4. Open (Biological cells)
5. Lower organisms (Plants)
6. Animals (Birds)
7. Man (Humans)
8. Social (Families)
9. Transcendental (Aliens)

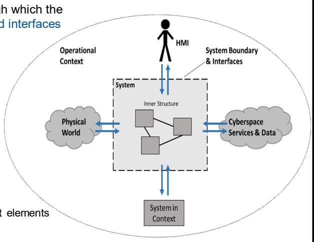
Checkland (1999):

1. Natural systems (Humans, Birds)
2. Designed physical systems (Car, Robot)
3. Designed abstract systems (Software)
   - do not contain any physical artifacts
   - designed by humans to serve some purpose
4. Human activity systems (Manufacturing, Politics)
   - Observable human activities
5. Transcendental systems (Aliens)
   - Systems beyond knowledge

In our lecture: natural systems, social systems, technological systems

A System in its Context

- Operational system context: all external elements through which the system of interest interacts with through its boundary and interfaces
- In the context of a system of interest:
  - Human actors (drivers, controllers, operators, …)
  - Parts of the physical world (roads, weather, …)
  - Parts of the cyberspace (data, services, …)
  - Related systems
- Closed systems: no interactions with environment
  →All aspects of the system are within the boundary
- Open systems: inputs and outputs with its environment
  →Boundary defines how system parts interact with environment elements



Cyber-Physical Systems

Cyber-physical systems are engineered systems where functionalities are emerging from the networked interaction of physical and computational processes. [BDS19]

CPSs are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical process, usually with feedback loops where physical processes affect computations and vice versa. [Lee08]

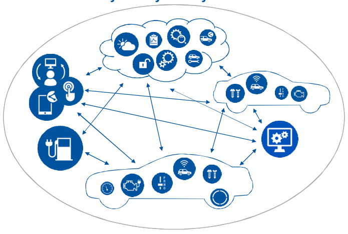
Cyber-physical systems combine computing and networking with physical dynamics. [Pto13]

- Comprise software parts and physical parts
- Often in networks or the Internet
- Popular applications:
  - Agriculture
  - Assistive (home) systems
  - Automated vehicles
  - Avionics
  - Manufacturing
  - Medical systems
  - Offshore systems
  - Oil drilling and mining systems
  - Robotics
  - Smart {home, garden, grid} components

Consequences from the Definition of System and Cyber-Physical System

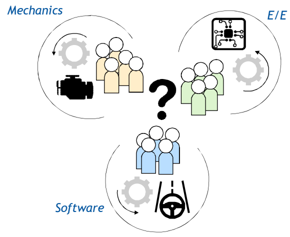
- CPSs consist of
  - (multiple) software sub-systems and
  - (multiple) physical sub-systems
  - Humans can be considered part of a CPS
- CPSs provide functionality through the interaction of
  - Software systems
  - Physical systems
  - Software with physical systems
  - Physical with software systems
  - Humans with other CPS components

Cyber-Physical System



Consequences for the Engineering of Cyber-Physical Systems

- Engineering CPSs requires expert knowledge from
  - Software Engineering (in several subdomains)
  - Mechanical Engineering (in several subdomains)
  - Electrical Engineering
- Domains use different models that need to be integrated in a holistic engineering approach
  - E.g., integrating concurrency models of computing with time abstractions in physical systems [BDS19]

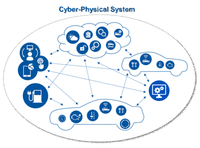


MBSE

4. System and Systems Engineering
4.2. Systems Engineering

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



What is systems engineering?

And how to manage SysE projects?

Systems Engineering is an Interdisciplinary Approach for the Realization of Systems

Definition (INCOSE 2016):

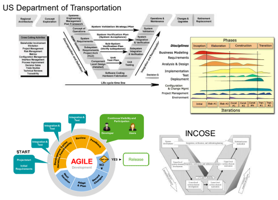
Systems Engineering (SysE) is an interdisciplinary approach and means to enable the realization of successful systems.

It focuses on

- holistically and concurrently understanding stakeholder needs;
- exploring opportunities;
- documenting requirements; and
- synthesizing,
- verifying,
- validating, and
- evolving solutions

while considering the complete problem, from system concept exploration through system disposal.

INCOSE



A method decomposes the big problem into a set smaller, manageable activities

---

Systems Engineering – Verification and Validation

- Validation:

The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers.

- Verification.

The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process.

(Wikipedia)

- *Verification* != Validation

Systems Engineering – a Characterization - To be read at home or at wikipedia

(Wikipedia)

Systems engineering is an interdisciplinary field of engineering and engineering management that focuses on how to design, integrate, and manage complex systems over their life cycles.

At its core, systems engineering utilizes systems thinking principles to organize this body of knowledge.

The individual outcome of such efforts, an engineered system, can be defined as a combination of components that work in synergy to collectively perform a useful function.

Issues such as requirements engineering, reliability, logistics, coordination of different teams, testing and evaluation, maintainability and many other disciplines necessary for successful system design, development, implementation, and ultimate decommission become more difficult when dealing with large or complex projects.

Systems engineering deals with work-processes, optimization methods, and risk management tools in such projects. It overlaps technical and human-centered disciplines …

Systems engineering ensures that all likely aspects of a project or system are considered, and integrated into a whole.

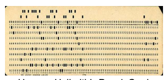
https://en.wikipedia.org/wiki/Systems_engineering

The Relationship between Software Engineering and Systems Engineering

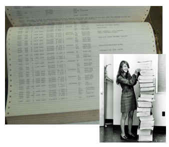
- In the past, software…
  - was embedded into machines
  - controlled the behavior of a single machine
  - engineering was part of systems engineering
- Consequently
  - Software was primarily developed by programmers
- Today smart software connects and coordinates
  - Heterogeneous Systems
  - Humans
  - Networks (Industrial Internet of Things)

→ Significant change of impact

→ Systems engineering tries to adopts software engineering methodologies



- Herman Hollerith's Punch Cards
- First use: 1890 U.S. census
- Last use: 2012 voting machines

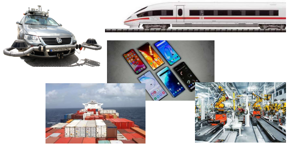


- Assembler code
- First use 1949
- Basically direct memory manipulation

- Modern general-purpose programming languages
- Object-oriented, functional, logical

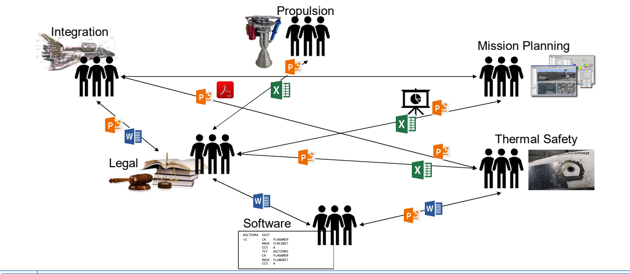
Systems Realize Functions Related to Their Purpose

- From the definition: A system has one or more purposes with respects to its operational context.
- Consequence: **Systems realize functions.**
- Functions of systems
  - Train → Mobility
  - Car → Individual mobility, also: storage
  - DaVinci medical robot → Remote operation
  - Freighter → Transport
  - Smart phone → Communication, photography, …
  - Manufacturing system → Produce goods
  - …



⇨ Systems thinking is based on functional specification, design and implementation. Geometry (i.e., physical shape): form follows function.

Traditional Systems Engineering is Document-Based



Traditional Systems Engineering Practice

- Stand alone domain models/designs related via
  - Documents (often Word, Excel, …)
    - Operations concepts
    - Natural language requirements
    - Bill of materials
    - Interface specifications (often tables)
    - Deployment plans
  - Manual reviews
  - Informal communication
    - White boards
    - Design team meeting presentations
    - Email

From Documents to Models

- Documents often use natural language (ambiguous), are not well-formalized (redundant), cannot be checked automatically (incomplete), cannot evolve automatically (static).
- Systematic and efficient engineering requires structural, behavioral, physics- and simulation-based models representing the technical designs which evolve throughout the life-cycle, supporting trade studies, design verification and system V&V.
- Current practice tends to rely on standalone (discipline-specific) models whose characteristics are shared primarily through static documents.
  - Models are there (implicitly, in engineers heads, in code)
- MBSE moves toward a shared system model with remaining discipline-specific models providing their characteristic information in a mathematically rigorous format.
  - Discipline models integrated by design; experts use views.

Model-Based Systems Engineering

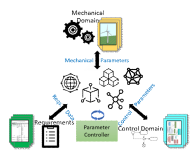
Model-Based Engineering (MBE): An approach to engineering that uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle.
– National Defense Industrial Association, 2011

Model-based systems engineering is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.
– INCOSE SE Vision 2020

INCOSE IW January 30th, 2016 * Fosse

Opportunities of Model-Based Systems Engineering

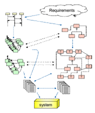
- A coherent set of consistent, related models ensure integrity and enable traceability throughout the development process
  - Enables top-down design decisions and drivers
  - Automated change propagation, ambiguity checking
  - Automated tracing of (changing) requirements to (changing) implementations
- These models provide the ability to codify institutional knowledge using formal methods, allowing for reuse and broad exposure
  - Model checking on subsystem and system level
  - Mitigation of loss of knowledge and investment
- They capture information in a durable, evolvable format
- They focus on information integration rather than document generation allows for decimation of artifact inconsistency/staleness



Model-Driven Systems Engineering

- Model-Based:
  – Models are used in some activities of development
- Model-Driven:
  – Models even drive and guide the process

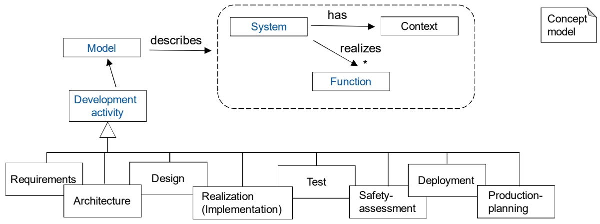
- Models are
  – Additional artifacts
  – Used in isolated forms
  – Employed for certain activities only
  – Often used for communication, documentation, … only
  – Rarely processed automatically
- Models are
  – primary development artifacts
  – used and reused across activities
  – enabler for a high degree of automation
  – drivers of reuse
  – enablers for agility



Literature

- [Wik-SysE] https://en.wikipedia.org/wiki/Systems_engineering
- [SeBoK] SEBoK Editorial Board. 2020. *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*, v. 2.2, R.J. Cloutier (Editor in Chief). Hoboken, NJ: The Trustees of the Stevens Institute of Technology. Accessed [20.07.2020]. www.sebokwiki.org.
- [BP07] Pahl, G., Beitz, W., Feldhusen, J., & Grote, K.-H. (2007). *Engineering Design - A Systematic Approach.* London: Springer.
- [BS08] Boardman, J. and B. Sauser. 2008. *Systems Thinking: Coping with 21st Century Problems.*
- [Alu15] Alur, R. (2015). *Principles of cyber-physical systems.*
- [KK98] Koller, R., & Kastrup, N. (1998). *Prinziplösungen zur Konstruktion technischer Produkte.*
- [Lee16] Lee, E. A. (2010). CPS foundations. *Proceedings - Design Automation Conference*, 737–742.
- [Lee08] Lee, E. A. (2008). Cyber physical systems: Design challenges. *Proceedings - 11th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing.*
- [FG13] Feldhusen, J., & Grote, K.-H. (Eds.). (2013). *Pahl/Beitz Konstruktionslehre.*
- [BDS19] Broy, M., Daembkes, · Heinrich, & Sztipanovits, J. (2019). Editorial to the theme section on model-based design of cyber-physical systems. *Software & Systems Modeling, 18*, 1575–1576.
- [BS01] Broy, M., & Stølen, K. (2001). Specification and development of interactive systems. In *Monographs in computer science*. New York: Springer.

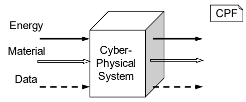
Summary formulated as a Concept Model



- Systems and their functions are described by models, which are part of various development activities

MBSE

5. Function as Modelling Paradigm
5.1. Functions Specifying Systems



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

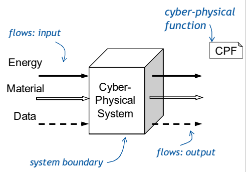
http://www.se-rwth.de/

What is a function?

And how to describe it?

System Specification through Functions

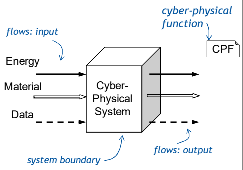
- A system defines a cyber-physical function
  – it encapsulates a physical and computational structure
  – performs data, energetic and physical transformations
  – and is connected to its context through its interfaces.
- A system function is described through its
- input and output signature
  - types and forms of the
    - signals / data
    - energy flow
    - material flow
- The functionality is mathematically described through the
  – relation between input and output



The concept of function is our first universal specification and construction principle

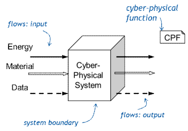
System Specification through Functions -2

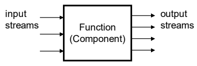
- A system defines a function
- Advantages of using the function principle:
- A) Mathematically very precise foundation exists
- B) Function composition exists
- C) Powerful modelling concepts
- Many possible forms of flow:
  - Continuous (e.g., current, fluids, sand) vs.
  - discrete (e.g., data, product items)
- Many forms of I/O relations:
  - May embody duration of the process
  - Internal state of the system
  - Delay of reaction
  - Etc.

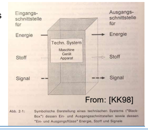


System Specification through Functions -3

- The function based construction principle was e.g. defined by Pahl/Beitz
  - function paradigm originally as a mental concept
-
- Later modelling of functions was added, e.g.:
  - Mathematical differential equations for continuous physical processes
  - Streams are well fitting a mathematical mechanism to describe these functions [BS01]
- Today, explicit modelling techniques are usable:
  - UML, SysML for discrete processes, data and physical structures, behavior of functions, …



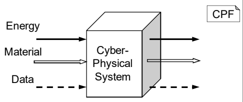




From: [KK98]

Models describing System Functions

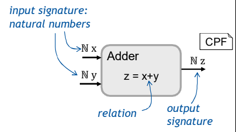
- A system defines a function
- Aspects to be defined in abstract, purpose fitting models:
  - Interface signature
  - Internal structure (architecture)
    - Logical structure
    - Geometrical shape
  - Behavior (over time)
  - Interactions
  - Assumptions about the context



⇨ Abstraction with dedicated models to master complexity is the second universal principle.

Example: Simple Adder as a Software Function

- A software system defines a function with behavior
  - Data are discrete numbers arriving pairwise and shall be added
  - This is a sufficient specification: $z = x+y$
    - It connects inputs directly with the output
  - Observation over time shows a stream of inputs being mapped to a stream of outputs
  - Here: we do not specify:
    - Timing details
    - Absence of values on x or y inputs
  - Here: Output is only dependent on current input (without any history)

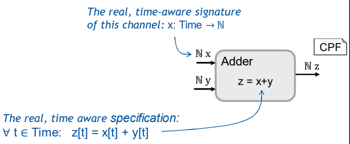


x:	2	3	0	6	1
y:	1	3	3	2	1
z:	3	6	3	8	2

time →

Timing in the Simple Adder

- Specification: $z = x+y$ only describes the „current“ behavior.
- Software function acts over time:
  - Discrete sequences of inputs and outputs
- Assuming we have "model of time" Time
- The real and complete interface:
  - $x, y : \text{Time} \rightarrow \mathbb{N}$
  - $z : \text{Time} \rightarrow \mathbb{N}$
- The complete, time dependent specification is:
  - $\forall\, t \in \text{Time}: \; z[t] = x[t] + y[t]$
- Which is only abbreviated by: $z = x+y$
- Time is so intrinsically present in all specifications: we often omit its explicit notion

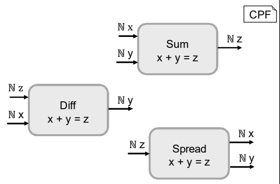


Time	0	1	2	3	4
x:	2	3	0	6	1
y:	1	3	3	2	1
z:	3	6	3	8	2

time →

Orientation of Inputs and Outputs is Relevant

- A software system defines a function with behavior
  - This specification: $F: \; z = x+y$ is an equality that does not distinguish input and output
  - Semantically equivalent alternatives:
    - $z = x + y$ $\qquad y + x = z$
    - $x = z - y$ $\qquad 2*y + x = 2*z - x$
- Flow direction of signals however distinguishes what is input and what is calculated/produced:
  - Sum of x and y (Function Sum)
  - Difference between z and x (Function Diff)
  - Or also: an underspecified spread of z to x and y (Function Spread)

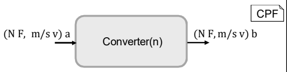


⇨ Input and output is distinguished in the signature of a function; not necessarily in the body of the specification

Forms of Denoting Formulae

- Unfortunately math and programming choose to use different forms of notations
  - (for a variety of reasons)
- We are aware that:
- Math uses indices e.g. $A_i = (Fi, vi)$
- Math uses SI-Units, like N, m/s (N = newton, force)
- PLs use types, records, classes, e.g. Newton
- We mix both at our convenience, e.g.:
  tuple type (N F, m/s v) and selectors, like a.F, a.v
- Math encodes "types" in variable names, e.g. "F" is always force.
- PLs separate types and variables e.g. "Newton f":
  - PLs often use capitals for types, lower cases for variables
  - Also common "f: Newton" and short "N f"
- Math uses single letter variables (incl. Greek letters)
- PLs use self-explanatory names and use ASCII

- Math style model of the function



- Programming style model of the same function

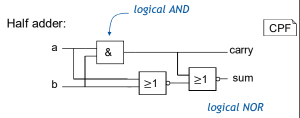
CPF

```
component Converter(double n) {
  in   (N force, m/s velocity) a;
  out  (N force, m/s velocity) b;

  laws:
     b.force = n * a.force;
     a.force*a.velocity = b.force*b.velocity
}
```

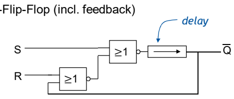

Example: Electrical Circuits

- Circuit and chip design relies on binary electrical current.
- Logical AND is specified as
  - o = $i_1 \wedge i_2$
- Half adder is specified as
  - carry = a ∧ b
  - 2*carry + sum = a + b
- and implemented using an AND and two NOR
- R-S-Flip-Flop includes a feedback loop:
  - this allows to store a state (a bit)

Half adder:

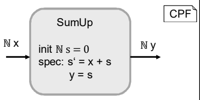


R-S-Flip-Flop (incl. feedback)



Example: SumUp as Software Function with State

- Building sum of arriving numbers:
  - Data are discrete numbers and shall be summed up
  - Internal state: the sum built so far
  - We use an internal variable
    - ℕ s initialized with 0
  - And as specification this invariant (also readable as update function):
    - s' = x + s ∧ y = s
  - It relates
    - input x
    - output y
    - the current internal state s
    - and the next internal state s'



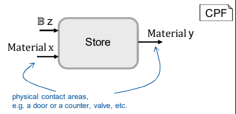
x:	2	3	0	6	1
s:	0	2	5	5	11
y:	0	2	5	5	11

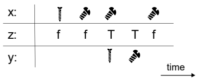
time

⇨ In software: The internal state contains all relevant information about the history that is relevant to fulfill the function.

Example: Store as a Function to Store Material

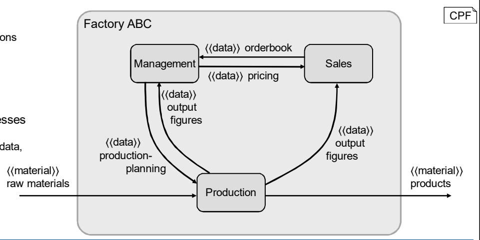
- Input:
  - Material arrives as discrete elements
  - Store releases arrived material on Boolean request
  - Internal state: the material stored so far
    - List<Material> s initialized with []
  - And as (not fully complete) specification the invariant
    - z=false ⇒ s' = s++x ∧ y = ε
    - z=true ∧ s'≠[] ⇒ s' = rest(s) ∧ y = first(s)
- Again next state s' is related to current state s
- Extensions:
  - Use part number to retrieve specific elements
  - or pickup times
  - or bags (multisets) instead of storage lists





Example: Dataflow and Material Flow in a Factory

- Companies, business processes, production processes:
  - Can be specified as functions
- Production takes time:
  - Material is processed,
  - Material is stored, etc.
- Company business processes
  - Use history for prediction
  - Use data to produce new data, work directives, etc.

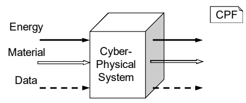


MBSE

5. Function as Modelling Paradigm
5.2. Underspecification

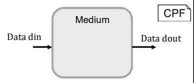
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Example: Underspecified Communication Medium

CPF



- Medium describes an unreliable communication device:
  - It may transport a signal (data) or may drop it
  - This behavior is nondeterministic in nature.
  - For simplicity: Medium does not replicate, alter or delay data, nor does it switch the order of data
  - Specification dout = din $\vee$ dout = ε
  - Remarks:
    - the fully transmitting medium is included
    - the disconnected medium is also included

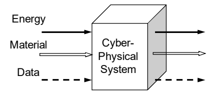
some alternatively possible output streams

din:	2	3	0	6	1
$dout_1$:		3	0		1
$dout_2$:				6	1
$dout_3$:	2	3	0	6	1
$dout_4$:	2				

time

The Underspecification Principle

- Deterministic and fully specified relations are normally not achievable
  - Delays happen
  - Energy fluctuates
  - Abstraction introduces lack of information
- Underspecification is the ability to describe the desired range of allowed behaviors (instead of a single, determined behavior)
- Advantages:
  - Easier to specify
  - Can be well combined with variant-building and methodical refinement

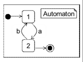


⇨ Controlled, explicit underspecification is the third universal specification principle

Example Automata: Nondeterministic Automata

Repetition

- First of all:
  - Nondeterminism and underspecification are related (almost the same)
- To introduce nondeterminism, we adapt the automaton syntax to:
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \to \wp(S))$
  - Set of initial states $S0$
  - Transition relation δ instead of function: δ can now offer multiple transitions ($|\delta(s,i)| > 1$ allowed)
- Semantics domain uses again the set(!) of words over I:
  $Sem = I^*$
- Semantics mapping: the set of accepted words with a path to a final state:
  $M(A) = \{\ w \in I^* \mid \delta^*(S0,\ w) \cap F \neq \emptyset\ \}$

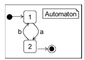


- Finite automata come with a rich theory and well-known techniques:
  - Powerset construction derives a deterministic automaton
  - Error completion
  - ϵ - Transition elimination
  - Equivalence checks (used e.g., by model checkers)
  - Mapping of regular expression to automata
    - Which includes various forms of automaton composition (∩, ∪, ¬, sequence .∘., Kleene closure .*)
- Theory helps to define semantics as well to efficiently map the automaton to an executable implementation

Example Automata: Automaton refinement and consistency

Repetition

- Nondeterministic automata
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \to \wp(S))$
  - $Sem = I^*$
  - $M(A) = \{\ w \in I^* \mid \delta^*(S0,\ w) \cap F \neq \emptyset\}$
- Automaton A is well defined: $M(A) \neq \emptyset$
  - i.e. it accepts something
  - Syntactic sufficient criterion:
    - $\exists\ s \in S^*\!:\ \forall n\!:\ \exists i\!:\ s_{n+1} \in \delta(s_n, i) \wedge s_0 \in S0 \wedge s_n \in F$
  - Can effectively be checked using transitive closure
- Automaton A is refinement of B: $M(A) \subseteq M(B)$
  - i.e. A is more deterministic than B
  - Can effectively be checked using a simulation relation (see model checking)



- Automata A and B are consistent: $M(A) \cap M(B) \neq \emptyset$
  - i.e. the do not specify conflicting properties of a component
  - Can effectively be checked using an intersection automaton
- We recognize:
- Automaton theory demonstrates that:
  - $M(A) \cap M(B) = M(A \cap B)$
  - i.e. composition ∩ of automata is conform to individual mapping and composition of semantics
- Set theory is an excellent vehicle to understand consistency, underspecification and refinement

Example: Underspecification in Math

- The mathematical equation is a perfectly deterministic tool
  - a = b
- Precisely determines a if b is given (and vice versa).
- Equation systems determine solutions …
- Non-injective mathematical operators allow several solutions:
  - $a^3 = b^2$
- Logical "or" introduces alternatives:
  - a = b $\vee$ a = 2*b
- Ranges can be specified:
  - b $\le$ a $\wedge$ a $\le$ 2*b
- Approximate equalities use "small" "unknowns" c:
  - a = b +c
- Functional dependencies can be extended with "small" "unknown" functions c:
  - f(x) = g(x) + c(x)
- Small unknowns are convenient, but the boundaries need to be clear, e.g.,
  - Which c, c(x) is "small" enough?
  - Or what properties does c(x) have? Continuous? What about its derivates (small changes only?)?
  - Does a stochastic distribution over c make sense?

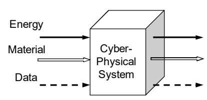
MBSE

5. Function as Modelling Paradigm
5.2. Streams to describe Functions

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

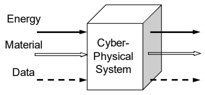
http://www.se-rwth.de/

CPF



Signature of Input and Output of a Function

- The signature of a function describes the forms of interactions of a system component with its environment.
- Interactions are broken down to streams of elements, which describe the time dependent flow and can be of the kinds
  - data,
  - energy or
  - material
- Interactions are organized through input and output channels.

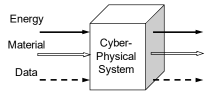


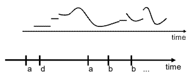
- The Interface of a Cyber-Physical System is defined through its function signature

⇨ The concept of stream is our fourth universal specification and construction principle

Interactions on Channels as Streams of Elements

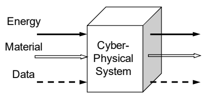
- The signature of a function describes the forms of interactions of a system component with its environment.
- The signature of a function is defined by
  - A set of channels
    - Channels are directed as inputs or outputs
  - A channel is of kind data, energy or material
  - A channel has a type (e.g., data type, energy or material type)
  - The streams of elements are time dependent, e.g.,
    - Continuous (or piecewise continuous) or
    - discrete: i.e. event based





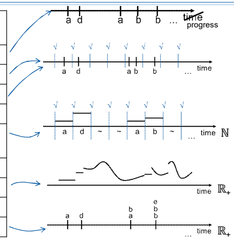
Behavior of a System

- The behavior of a system is defined as a function or relation on the streams of element according defined the system signature
  - Given the channels and their kinds a relation between input and output can be defined
  - Time is directed and time warp doesn't exist, i.e., output of a channel depends only on
    - the history of the input
    - and the (almost) current input
  - A function may have state to
    - remember its own history (i.e., data) or
    - to store products (i.e., material or energy).



All Variants of Communication Models (Streams) and their Timing

Underlying Topology	*Kind of Stream*	*Mathematical Definition*
discrete	event stream	M^ω
discrete	timed event stream	$M^\surd$
discrete	time slice stream	$\mathbb{N} \to M^*$ $(\cong M^\surd)$
discrete	time-synchronous stream	$\mathbb{N} \to M$
discrete	time-synchronous optional stream	$\mathbb{N} \to M^\sim = (M \cup \{\sim\})^\omega$
discrete	signal stream	$\mathbb{N} \to \wp(M)$
dense[1]	hybrid stream	$\mathbb{R}_+ \to M$
dense	dense signal stream	$\mathbb{R}_+ \to \wp(M)$
superdense	super dense stream	$\mathbb{R}_+ \to M^*$

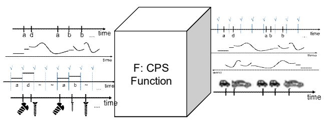


$\mathbb{N}$ = natural numbers; $\mathbb{R}_+$ = positive reals; M = messages; $\wp(M)$ = powerset over M; M^ω = discrete streams (i.e. finite and infinite lists) over M

[1] dense streams are typically continuous almost everywhere

Model Based System Specification - Functional View

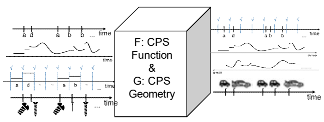
- A function F specifying a CPS component has a complex signature:
  - Many channels : ingoing and outgoing
  - Channels carry different kinds of elements
  - Flows may be discrete, dense, and even superdense
  - Time is relevant and it is directed:
    - Reaction is not really immediate (but can be very quick)
    - No time warp, no undo of emitted reactions
  - Component may have
    - delay in reaction and
    - store information, energy or material for future reactions: components have encapsulated state
  - Underspecification allows many possible behaviors



- Formally F relates its streams M^σ (i.e. one of M^ω , $M^\surd$, $\mathbb{N} \to M^*$, … $\mathbb{R}_+ \to M$) of
  - input channels $I_1, I_2, \ldots I_n$
  - and output channels $O_1, O_2, \ldots O_n$
- as set of mathematical functions:
  - $F \subseteq I_1^\sigma \times I_2^\sigma \times \ldots \times I_n^\sigma \to O_1^\sigma \times O_2^\sigma \times \ldots \times O_n^\sigma$ that obey timing restrictions

Model-Based System Specification – Function and Geometry View

- A CPS system (and equally a CPS component) are defined by a
  - functional view F and a
  - geometric view G
- A geometry defines an area in space that the CPS takes
  - Physical dynamics include a behavior over time, possibly dependent on input channels (called interactions)
  - Physical effective surface acts as interface of the geometry (Dt: "Wirkfläche")
  - Material properties are internal, only the surface is visible
- Underspecification as development principle

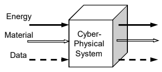


⇨ A CPS is described by a functional and a geometric view. Both share their CPS interface.

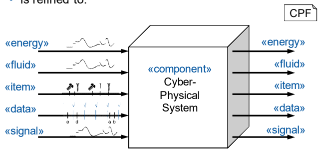
Stereotypes for Components and Interaction Channels

- We in this course define the following:
- for functions
  - «component» machinery, …
    - «system» machinery that is “complete”
  - «being» humans, …
- for function channels:
  - «material» elements, compounds, alloys, …
    - «fluid» continuously flowing material, typically not countable (water, gas, sand)
    - «item» discrete physical items, e.g. cars
  - «energy» types of energy
  - «data» for data objects, basic data (e.g. int)
    - «event» for discrete data that triggers behavior
  - «signal» for continuously flowing data

- Principle picture:



- is refined to:

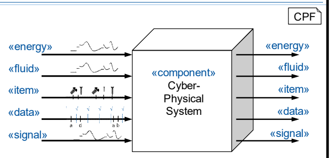


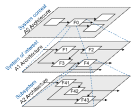
Summary: Universal Construction Principles 1-3 (more are coming up):

- 1: The function concept is a universal specification and construction principle
  - → Functions are a well-known mathematical construct that allow us to model system functionality precisely
  - Functions (and related math structures, such as continuous or discrete time, abstract data types) are the connection between systems thinking and mathematical foundations.
- 2: Abstraction with dedicated models to master complexity is the second universal principle.
- 3: Controlled, explicit underspecification is the third universal specification principle
  - → Underspecification allows us to model absence of information or uncertainty in analysis, variability of the products, degrees of freedom when customizing a component and also behavioral nondeterminism that occurs during system operation.
- 4: The concept of stream is our fourth universal specification and construction principle
  - → Streams allow to describe the “flow” of elements (material, data, data) through input and output interfaces over time. Dense, even continuous, or discrete streams allow to model all forms of possible behavior of a function.

Function-based Universal Specification and Construction Principles

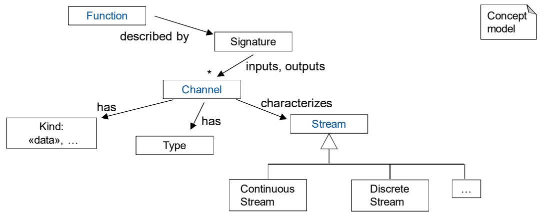
1. The function paradigm is the foundation
   - Clear boundaries, clear input/ouput signatures
2. Abstraction with dedicated models to master complexity is the second universal principle
   - Explicit, abstract models focusing on dedicated aspects
3. Controlled, explicit underspecification
   - abstraction, variability, ability to describe the desired range of allowed behaviors
4. The concept of stream
   - as mathematically precise, time dependent model of input/output behavior





… more coming

Summary defined in a Concept Model



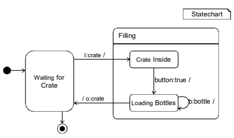
- Functions embody a signature consisting of typed channels

MBSE

6. Discrete Behavior Modeling with Statecharts
6.1. Examples

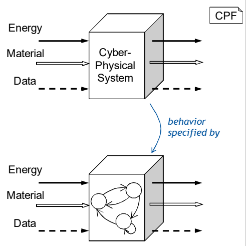
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



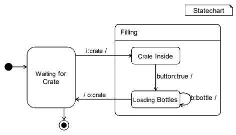
Specifying Function Behavior with Statecharts

- A cyber-physical function
  - needs a behavior specification
  - behavior maps input flows to output flows
- Statecharts describe discrete behavior
  - event-driven sequence
  - finite state space
  - discrete transitions caused by external events induce state changes and event emission
- Observations:
  - Data as well as materialized things are discrete
  - Energy and fluids are not
- How and when to use Statecharts?
- How to interpret a Statechart in CPS?



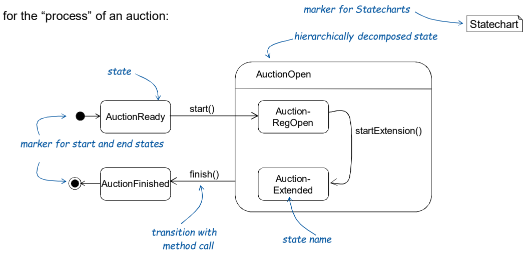
Statecharts

- Goal is the description of object or component behavior based on their internal state
- Statecharts extend automata theory:
  - hierarchical states,
  - actions in transitions and states,
  - explicit logic formulae as conditions, ...
- History of Statecharts
  - Statecharts introduced by David Harel in 1987
  - incorporated in many modeling languages
  - many variants developed
  - part of the UML from the beginning



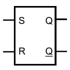
Example: Statecharts in Business

- Statechart for the "process" of an auction:

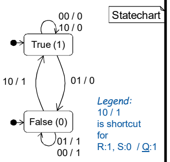


Example: RS-Flipflop as Statechart

- The functionality of an RS flip-flop circuit
  - Modelled by its internal state
- Internal state = the stored bit
- Observations:
  - Processes two (synchronous) inputs at once
  - Start state initially unknown
  - Output Q only depends on internal state (= Moore machine)
    - this introduces delay in its reaction
  - Statechart is incomplete:
    - R:1, S:1 is not considered

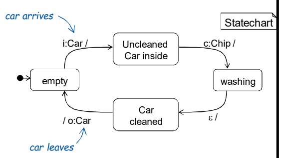


symbolic picture



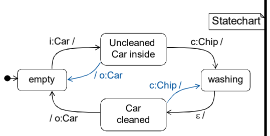
Example: Car Wash

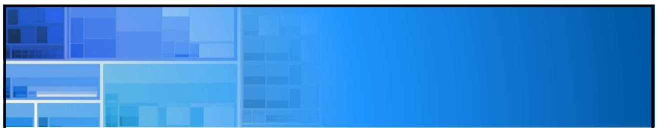
- A car wash
  - one car at a time
  - washing only if chip is entered
- Observations:
  - real physical things come in and out (cars, chips)
  - Functionality of "cleaning" is not actually modelled in its behavior (i.e. how to mathematically describe a car "is clean"), but the process around it
  - Abstracts away from many details, which?



Example: Car Wash – more complete

- A car wash
  - one car at a time
  - washing only if chip is entered
- Some additional behaviors:
  - Car leaves without wash
  - 2nd chip is used to wash same car twice
- Still missing:
  - Emergencies
  - Washing in steps …
  - Wrong chip
  - Chip arrives without a car
- Refinement: Adding behavior, where nothing was said before



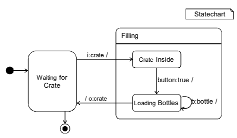


MBSE

6. Discrete Behavior Modeling with Statecharts
6.2. Underlying Automata Theory

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

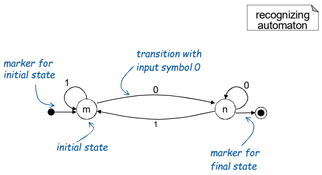
http://www.se-rwth.de/



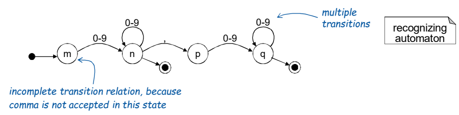
Recognizing Automata

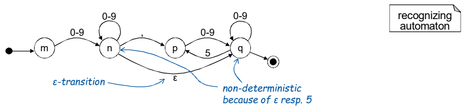
- Recognizing automaton (S,I,δ,s0,F) has
- (also: nondeterministic, alphabetical Rabin-Scott Machine (RSA))
  - Finite set of states S
  - Input alphabet I
  - Set of initial states $s0 \subseteq S$
  - Set of final states $F \subseteq S$
  - Transition relation $\delta \subseteq S \times I^{\varepsilon} \times S$

  where
  - ε represents the non-existent input characters in spontaneous transitions
    - $I^{\varepsilon} = I \cup \{\varepsilon\}$
  - All sets S, I, s0, F are non-empty and finite



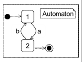
Examples of Recognizing Automata





Example Automata: Nondeterministic Automata

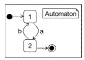
- First of all:
  - Nondeterminism and underspecification are related (almost the same)
- To introduce nondeterminism, we adapt the automaton syntax to:
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \rightarrow \wp(S))$
  - Set of initial states $S0$
  - Transition relation δ instead of function: δ can now offer multiple transitions (|δ(s,i)| > 1 allowed)
- Semantics domain uses again the set(!) of words over I:
  $Sem = I^*$
- Semantics mapping: the set of accepted words with a path to a final state:
  $M(A) = \{\ w \in I^* \mid \delta^*(S0,\ w) \cap F \neq \emptyset\ \}$



- Finite automata come with a rich theory and well-known techniques:
  - Powerset construction derives a deterministic automaton
  - Error completion
  - ϵ - Transition elimination
  - Equivalence checks (used e.g., by model checkers)
  - Mapping of regular expression to automata
    - Which includes various forms of automaton composition (∩, ∪, ¬, sequence .∘., Kleene closure .*)
- Theory helps to define semantics as well to efficiently map the automaton to an executable implementation

Example Automata: Automaton refinement and consistency

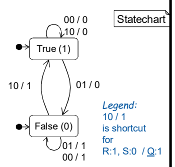
- Nondeterministic automata
  - $Sy = (S,\ I,\ S0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \rightarrow \wp(S))$
  - $Sem = I^*$
  - $M(A) = \{\ w \in I^* \mid \delta^*(S0,\ w) \cap F \neq ;\ \}$
- Automaton A is well defined: $M(A) \neq \emptyset$
  - I.e., it accepts something
  - Syntactic sufficient criterion:
    - $\exists\ s \in S^*\!:\ \forall n\!:\ \exists i\!:\ s_{n+1} \in \delta(s_n, i) \wedge s_0 \in S0 \wedge s_n \in F$
  - Can effectively be checked using transitive closure
- Automaton A is refinement of B: $M(A) \subseteq M(B)$
  - I.e. A is more deterministic than B
  - Can effectively be checked using a simulation relation (see model checking)



- Automata A and B are consistent: $M(A) \cap M(B) \neq \emptyset$
  - I.e. the do not specify conflicting properties of a component
  - Can effectively be checked using an intersection automaton
- We recognize:
- Automaton theory demonstrates that:
  - $M(A) \cap M(B) = M(A \cap B)$
  - I.e. composition ∩ of automata is conform to individual mapping and composition of semantics

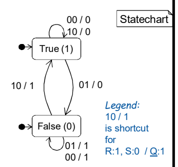
Readiness-to-Fire, Semantics

- A transition is ready to fire if the system is in the source state and
  - the input character arrived and or the
  - transition does not require an input character (i.e., it is spontaneous).
- The semantics of a recognizing automaton is the set of inputs (words over E), for which there exists a path from a start state to a final state
- But: pure recognition is too weak for behavioral description
  - therefore, extension of the machines to describe output
- Mealy machines have output on transitions
- Moore machines have output on states



Mealy Machine

- A Mealy machine
- $(S,\ I,\ O,\ s0 \subseteq S,\ F \subseteq S,\ \delta : S \times I \rightarrow \wp(S \times O))$
- includes recognition automaton $(S, I, s0, F, \delta)$
- and new:
  - output alphabet O
  - and transition relation δ is extended with output
- Semantics of Mealy machine is a relation between input and output words (I* × O*):
  - the “behavior” of the automaton that is exposed to the outside
- Mealy machines can describe functions on discrete in/ouputs

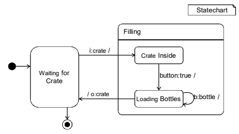


MBSE

6. Discrete Behavior Modeling with Statecharts
6.3. Application in Software and Systems

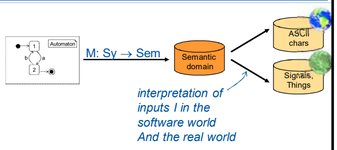
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



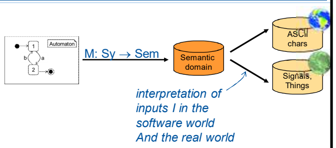
Example Automata: Semantics Definition using Math

- Mealy machines are a well-studied theory
  - Deterministic, completion, minimization, powerfulness, etc.
- Application in practice requires an interpretation in the real world
  - What is a state?
  - What is an input symbol?
  - What is output?
- Range of possible interpretations
  - ASCII characters (e.g., in parsing)
  - Method calls (in object-oriented software)
  - Signals (e.g., in communicating distributed systems)
  - Physical things
  - Electrical states (0, 1)



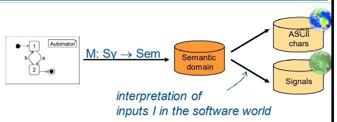
Application of Automata Theory in Object Oriented Modeling

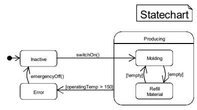
- Possible interpretations of states, transitions in OO?
  - State space defined by attributes of an object is in general infinite vs. finite set of states in the Mealy Automaton
  - State change through method call, asynchronous message via CORBA, timeout, ...?
  - What is the output?
  - What is a spontaneous transition in OO?
  - Initial and final states in OO?



UML Interpretation of Statecharts in Software

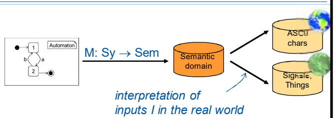
- State of the automaton = set of states of the object
- Initial state = set of object states that occur immediately after construction (new ...)
- Final state does not matter, because in Java garbage collection "terminates" objects
- Input characters = method call including arguments
- Output character = result of a method execution
  - Includes attribute changes, other method calls
- Transition = execution of a method body
- Distinction between diagram state and object state!

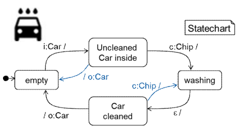




SysML Interpretation of Statecharts in Cyber-Physical Systems

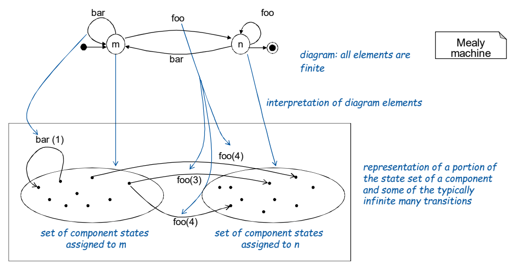
- State of the automaton = equivalence class of states of the component
- Initial state = component states that occur at start
- Final state normally not applicable to physical systems
- Input = incoming discrete things and signals in a machine
- Output = modified things as well as computed answers
- Transition = operation of the CPS transforming the input to an adequate output using and adapting internal states
- *Statecharts are useful in many domains. But the following the examples usually belong only to one domain. → Let's keep this in mind!*



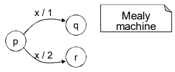


Relationship between Diagram and Component States

animated

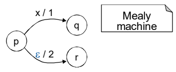


Non-Determinism (N.Det.) in the State Machine



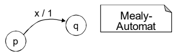
- If two transitions are ready to fire:
  Then the user of the system knows, that one of the transitions will be taken.
- Decision which one can
  a) Depend on missing details of the model states
     (= N. Det. by abstraction: underspecification)
  a) Left up to the developer
     (= N. Det. as a draft of freedom: underspecification)
  b) Determined at runtime (= N.Det. in the system)
- Decision may be left to the developer or system
  – This makes no difference to the user!
- Clarification of the interpretation:
  – N. Det. of the automaton as a concept for underspecification!
- Principle of underspecification: If no information is given, nothing is known!

Epsilon Transitions



- ε - transitions are "spontaneous" transitions
- Possible interpretations in the system/world
  1. timer has expired and causes transition
  2. automaton is incomplete: a message leading to this transition was not modeled, but effect is visible due to change of state
  3. the transition is a consequence of a previous transition and is executed automatically by the system.
- Notes on the variants
  1. Requires concepts to express this in an underlying programming language
  2. Allows for abstraction, but prevents code generation
  3. Allows to break down long actions in sequences, branches and even iteration of a method body into individual steps
     – notational comfort

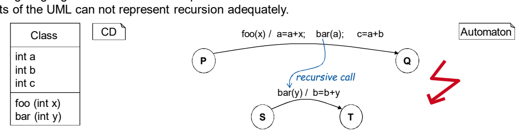
Incompleteness



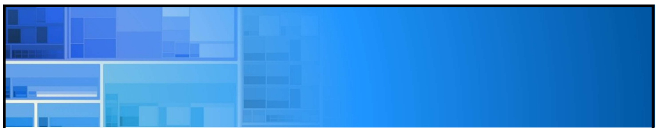
- In the current state "p" no transition is ready to fire for input "y"
- Possible interpretations
  1. ignore: do not execute any action, do not change any state
  2. chaos: arbitrary reaction allows change to arbitrary state and an arbitrary action
  3. error state is entered (and left only through return message)
  4. error message as the Smalltalk "message not understood", but no change of state
- 1, 3, and 4 are suitable for implementation: code generator
- Option 2 useful for Statecharts in a specification
  – Chaos allows robust implementation by subsequent design decisions (adding of transitions)
  – Chaos = no knowledge = underspecification

Expressiveness

- Implicit assumptions for data automata
  – Incoming stimuli are method calls and thus are processed sequentially
  – No parallelism in the individual component or the transition
- Programming languages such as Java allow parallelism and recursion on methods
- Statecharts of the UML can not represent recursion adequately.

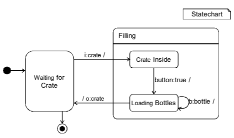


- To ensure correctness: Java code is synchronized and
- Assume that internally called methods (such as bar ()) are "helper methods", which do not use or affect the states



MBSE

6. Discrete Behavior Modeling with Statecharts
6.4. States

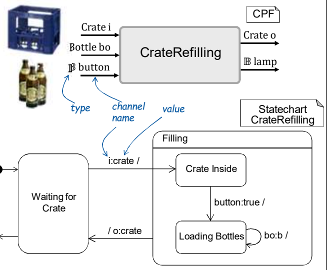


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

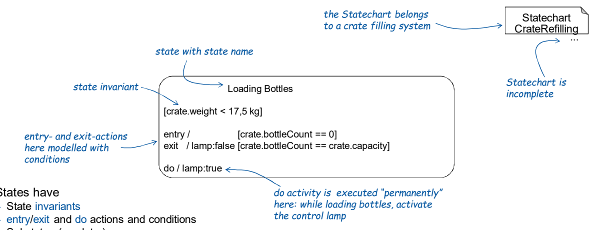
http://www.se-rwth.de/

Running Example: Function of a Crate Refilling System

- Function:
  – Load bottles into a crate
- Stimuli describe the external triggers that interact with the system via input channels
  – Crates through channel i
  – Bottles through channel bo
  – Binary signal of a pushed button in channel button
- Internal states of the inserted crate
  – ℕ bottleCount; // current bottle count
  – ℕ capacity; // max. bottle count
  – kg weight; // weight of the crate
- (Statechart is refined in forthcoming slides)



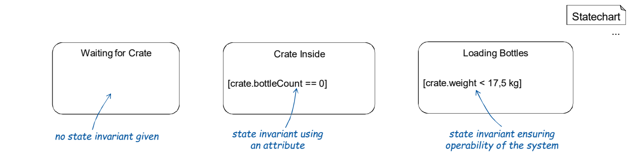
States



- States have
  - State invariants
  - entry/exit and do actions and conditions
  - Substates (see later)

State Invariants

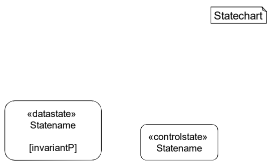
- State invariant is formulated over the attributes of the component (and potentially dependent components)
- Invariant connects the diagram state and component states



- Different states may have disjoint invariants, but is is not required in general!

Data States and Control States

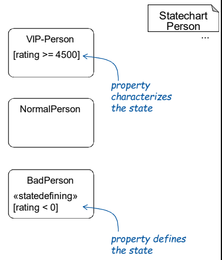
- If state invariants are not disjoint or even missing, then state machines will be "control states"
  - for implementation, an additional attribute is needed to distinguish them
- If invariants are disjoint, this is unnecessary: automaton states can be considered "data states"
  - state-defining invariants, "data" patterns
- Marking the states in the automaton by stereotypes:



- Usually not to be mixed in the same diagram!
- Control states can be transformed into data states, for example by introducing an attribute
  - This is a preparation step for code generation and can be automated

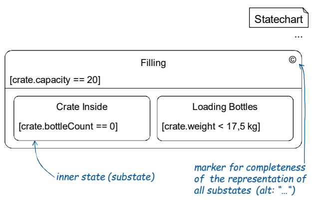
Invariant Defines the Data State

- Normally invariants characterize object states:
  - persons with rating >= 4500 can be VIP (but do not need to be!)
- «statedefining» state invariants define state:
  - persons are in state "BadPerson" if and only if rating <0
- «statedefining» state invariants must be disjoint.



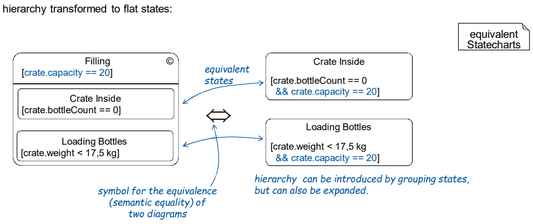
Hierarchical States

- Hierarchy for structured modelling of states
  - Substates share characteristics of enclosing state (invariants, actions, transitions)
- Remark: UML/P offers "or-decomposition" only
  - "or-decomposition" = component is exactly in one substate
  - "and-decomposition" would enforce a cross-product semantics; this can be achived using several components



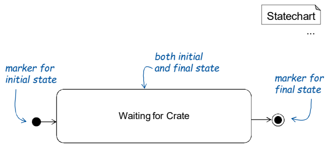
Semantics of Hierarchical States

- Explanation of semantics by mapping new concepts back on already familiar concepts:
  - E.g. states hierarchy transformed to flat states:



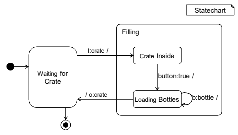
Initial and Final States

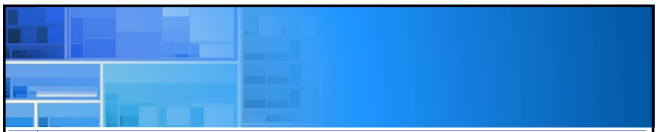
- Initial state: component starts in this state
- Final state: component may terminate activities/life here (may, not must!)
- A state can be initial and final
- Markers are not states
- Markers in substates have a different meaning (→ next section)



Summarizing Glossary for States

- State (syn. diagram state)
  - Represents a subset of the possible component states
- Initial state
  - Marks the beginning of the lifecycle
- Final state
  - Describes the component in this state has fulfilled its duty and is no longer needed.
- Nested state (syn. substate)
  - Is part of a hierarchically composed state
- State invariant
  - Is a condition (e.g., in OCL logic) that characterizes the states assigned to a state diagram
  - State invariants of different states may generally overlap (if not state-defining)



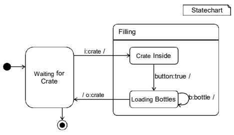


MBSE

6. Discrete Behavior Modeling with Statecharts
6.5. Transitions

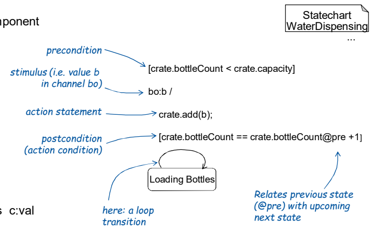
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



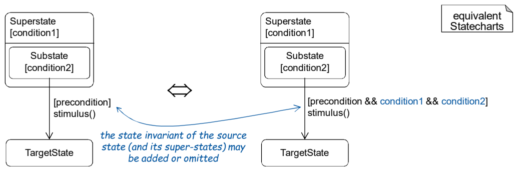
Transitions

- One transition describes a portion of the component behavior
- A transition has
  - Source state
  - Precondition
  - Stimulus
    - OO style: a method stimulus()
    - Signal style: a digital value, e.g., 3, "Theo"
    - "Thing" style: a real thing, e.g., the real car
  - Action (→ next section)
  - Postcondition
  - Target state
- Stimulus val over a channel "c" is denoted as c:val
  - The type of val is defined externally



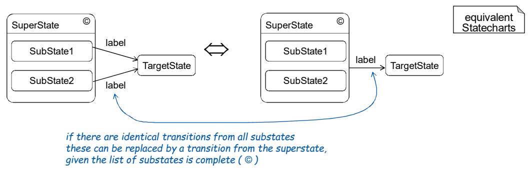
Preconditions in Transitions

- Readiness to fire is determined by (a) precondition of the transition and (b) state invariant
- State invariant can be explicated:



Superstate as a Source

- If the transition source is a superstate, the transition starts from each substate:



- Special cases: Substates that have initial/final markers

Types of Stimuli in Transitions

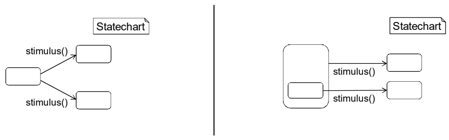
- General variants of stimuli:
  - Physical thing is received
  - Message is received (over a communication channel)
  - Method call is made
  - Result of a return statement is returned (=answer/solution of a method call)
  - Exception (i.e., an error) is caught, or
  - Transition occurs spontaneously.
- If several input channels are present, the channel name is added to the input stimulus
  - c:stimulus() c:3 c:person

- Stimuli types are denoted as shown:

item →	*thing is arriving over a physical channel*
methodname(arguments) →	*method call and asynchronous message - transmission are not distinguished here*
return(result) →	*reception of a result (due to a previously performed method call)*
Exception(arguments) →	*catching and manipulating occurred exception*
ε →	*spontaneous transition, e.g. as local continuation of an action*
value →	*a simple value*

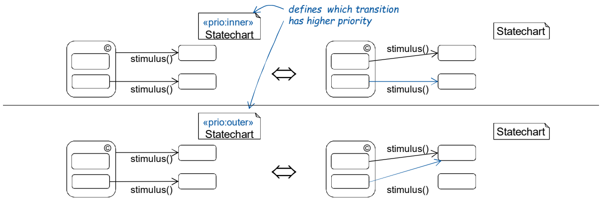
Overlapping Readiness-to-Fire

- shows up as non-determinism in the Statechart
- Is interpreted as underspecification
  - allows developers (during design) or physical device (during operation) to choose
- Examples



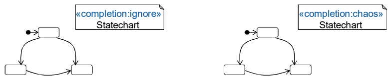
Prioritization of Transitions

- Overlapping can be resolved by prioritizing the transitions
- Statechart variants prioritize internal or external transitions
- UML/P allows to choose by means of stereotypes «prio:inner» and «prio:outer»



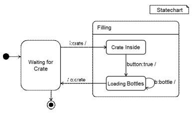
Incomplete Statechart

- i.e. there is no transition ready to fire:
- We remember the principle of underspecification: where no statement is given, nothing is known!
- However, stereotypes can be used
  - «error» marks a special "failure" state, which is then taken as target
  - «exception» marks a special state in which emerging exceptions are caught
  - «completion:ignore» means stimulus will be ignored
  - «completion:chaos» means stimulus can be handled arbitrarily (default, concurs with the full underspecification principle)



Definitions (related to Transitions)

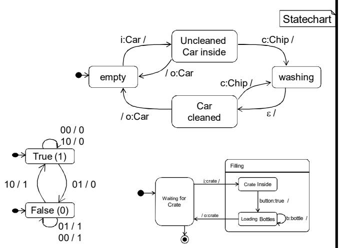
- Stimulus
  - Caused by other components, leads to firing a transition
  - Stimulus types: external call of functions, RPC, receiving asynchronously sent message, or timeout
- Transition
  - From source state to target state, contains a stimulus and an action as response
  - Logic constraints specify the transition more precisely
- Readiness to Fire
  - Transition is ready to fire if and only if the component in the source state of the transition and stimulus are correct and the precondition (of the transition) applies
  - If several transitions are ready to fire, the Statechart is nondeterministic and chosen transition is not determined



- Precondition of the transition
  - Logic condition that must hold for the attribute values and for the stimulus
- Postcondition of the transition (syn. action condition)
  - Logic condition describes properties of the reaction

Quality of a Statechart Model

- Quality is defined relative to a purpose:
  - Does it fulfill it's purpose?
- A) Does it model the correct behavior?
- B) Details:
  - B1) Is it sufficiently detailed?
  - B2) Is it not overly detailed / constraining?
  - B3) Complete, determined?
- C) Is it presented well?
  - C1) Readable?
  - C2) Well arranged?
  - C3) e.g. does it exhibit "main flows" well

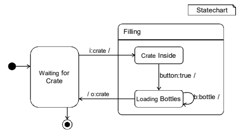


MBSE

6. Discrete Behavior Modeling with Statecharts
6.6. Actions

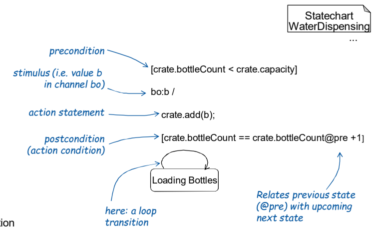
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

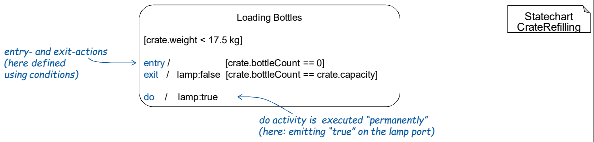


Action in a Transition

- Action in the Statechart
  - corresponds to output of the Mealy machine
- Effects:
  - change component states
  - Send messages / call methods / emit things
- Action representation in two forms:
  - Operational:
    - Specific instructions e.g., in Java
    - Emission of message / thing over a channel (similar to a programming statement): c:Message c:Person
  - Descriptive:
    - Action condition = post-condition of thetransition
    - Effect defined by math or a logic, e.g. OCL



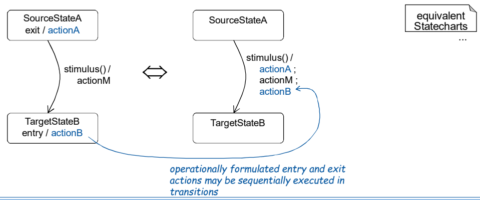
Actions and Activities in States



- entry action: executed when entering the state
  - the condition holds when entering the state (resp. after the execution of the entry action)
- exit action: when leaving
  - the condition holds when leaving the state (resp. after the execution of the exit action)
- do activity: executed permanently / regularly while in the state
- Entry and exit actions extend Statecharts to Moore machines with output related to states

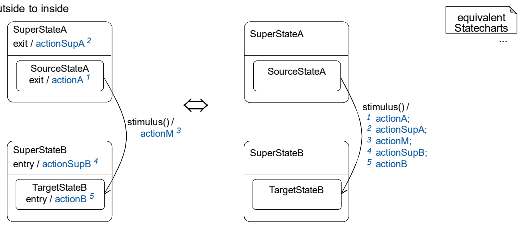
Semantics of Entry / Exit Actions

- Moore machine can be transformed into Mealy machine (a result from theory)
  - Through moving the state actions into adjacent transitions
- Simple case for operational activities:
  - Sequential composition (with " ; ")



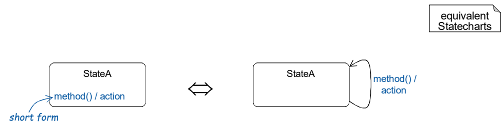
Interaction of the Entry / Exit Actions in the Hierarchy: Operational

- Operational entry and exit actions are executed in the order of leaving and entering states
  - exit: from inside to outside
  - entry: from outside to inside



Inner Transition

- A transition can be specified inside a state:
  - inner transitions form an alternative representation for a loop of this state:

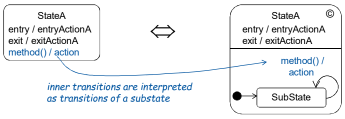


- Condition: state has no entry/exit actions, because
  - entry or exit actions of the state are not executed on the left side, but are executed on the right one
  - alternative?

Inner Transition

- Inner transitions can be transformed to transitions of the (only) substate, which is introduced for this purpose:
  - this ensures that entry/exit actions in inner transitions are not executed.

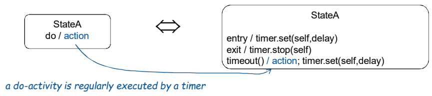
equivalent Statecharts



Do-Activity

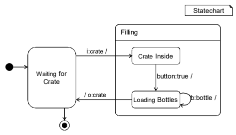
- Regular execution of the do-activity of a state means that
  - external time-driven mechanism triggers the contained action regularly
- A proposal(!) for an implementation using timer and internal transitions in software:
  - (or using a physical effect, e.g. bell ringing)

equivalent Statecharts …



Definitions for Actions

- Action
  - Is a change of the state of a component (and its dependent environment)
  - Often described by operational code (such as Java), or specifying signals and a sequence of emitted things by a logic condition
- Entry action
  - Belongs to a state and is executed (or evaluated in case of a condition) when the state is entered
- Exit action
  - Belongs to a state and is executed (or evaluated in case of a condition) when the state is left
- Do-activity
  - Is a permanently continuing activity of a state, or
  - is executed regularly (e.g., by means of a timer)



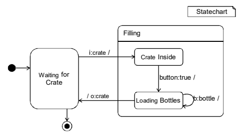
- Nondeterminism in a Statechart
  - If a situation occurs where several alternative transitions are ready-to-fire, then behavior of the component is underspecified
  - and also: if no explicit transition is given full underspecification is assumed

MBSE

6. Discrete Behavior Modeling with Statecharts
6.7. Further Issues

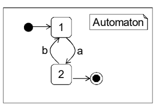
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Example Automata Syntax: Model Representation by Graphics, Text and Math

- Graphical / diagrammatic:



- Textual in ASCII / UTF-8:

```
automaton Simple {
  state 1 <<initial>>;
  state 2 <<final>>;
  1 - a > 2;
  2 - b > 1;
}
```

- Mathematical:

Tuple $(S, I, 1, \{2\}, \delta)$
- Set of states $S = \{1,2\}$
- Set of inputs $I = \{a, b\}$
- Initial state $1 \in S$
- Final states $\{2\} \subseteq S$
- Transition function $\delta : S \times I \rightarrow S$
  - with $\delta(1, a) = 2;\ \delta(2, b) = 1$

- Tabular:

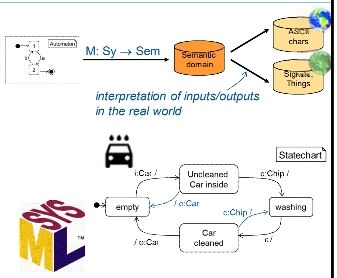
source \ target		1	2 final
initial	1		a
	2	b	

- typically restrictions apply (context conditions)
- more variants: XML/JSON-encoding, Java-encoding (State Pattern), …

Various Uses of Statecharts for Software and Systems

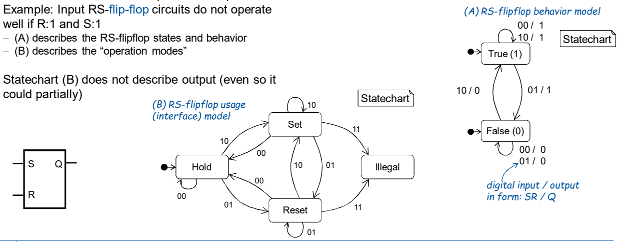
Statecharts can be used for different viewpoints:

1. Representation of the life cycle of a component
2. Implementation description of a method / operation (in software only)
3. Interface description of the useful operation modi
4. Abstract description of requirements on the state space
5. Representation of allowed sequences of stimuli occurrences (input signals/arriving things)
6. Characterization of the possible or allowed behaviors of a component
7. Connection between state and behavior of a component



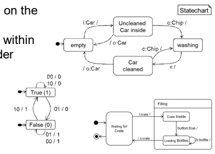
Statechart as Description of Allowed Inputs: Interface only

- Not all combinations and sequences of inputs allowed
- Example: Input RS-flip-flop circuits do not operate well if R:1 and S:1
  - (A) describes the RS-flipflop states and behavior
  - (B) describes the "operation modes"
- Statechart (B) does not describe output (even so it could partially)



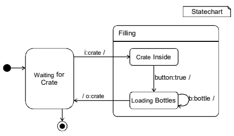
Summary Statecharts

- Statecharts are an extension of Mealy and Moore machines
  - for practical usability
- Statecharts build a powerful form to define discrete behavior based on a discrete state space
- The combination with pieces of code for actions, or with logic conditions makes Statecharts fully descriptive and comfortable
- Used in various phases of software and systems development: Analysis, design, implementation
- Extensions with differential equations integrate modelling styles from hybrid automata
- A number of variations for Statecharts allows different areas of application:
  - Machine behavior
  - Life cycles
  - Test sequences
- Much depends on the interpretation of a Statechart within the system under development



MBSE

6B. Executing Statecharts
6B.1. Semantics Revisited



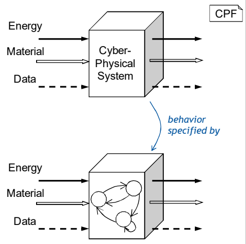
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Specifying Function Behavior with Statecharts

Rep.

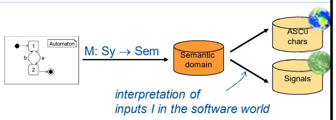
- A cyber-physical function
  - needs a behavior specification
  - behavior maps input flows to output flows
- Statecharts describe discrete behavior
  - event-driven sequence
  - finite state space
  - discrete transitions caused by external events induce state changes and event emission
- How and when to use Statecharts?
- How to interpret a Statechart in CPS?
- What to do with a Statechart?

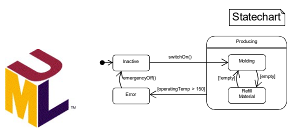


UML Interpretation of Statecharts in Software

Rep.

- State of the automaton = set of states of the object
- Initial state = set of object states that occur immediately after construction (new ...)
- Final state does not matter, because garbage collection in Java "terminates" objects
- Input characters = method call including arguments
- Output character = result of a method execution
  - Includes attribute changes, other method calls
- Transition = execution of a method body
- Distinction between state diagram and object state!

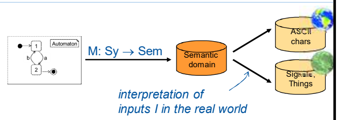


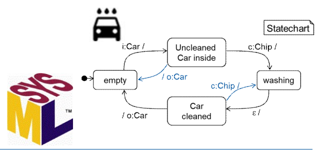


SysML Interpretation of Statecharts in Cyber-Physical Systems

Rep.

- State of the automaton = equivalence class of states of the component
- Initial state = component states that occur at start
- Final state normally not applicable to physical systems
- Input = incoming discrete things and signals in a machine
- Output = modified things as well as computed answers
- Transition = operation of the CPS transforming the input to an adequate output using and adapting internal states
- *Statecharts cover all domains. Even though in the following the examples usually belong only to one domain. → Let's keep this in mind!*



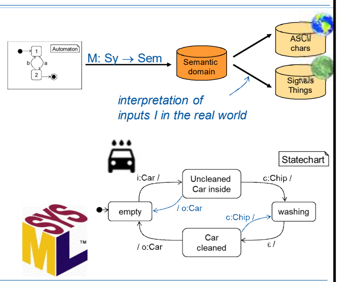


Various Uses of Statecharts

Rep.

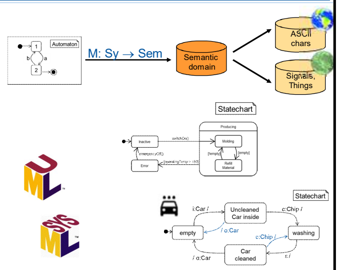
Statecharts can be used for different viewpoints:

1. Representation of the life cycle of a component
2. Implementation description of a method / operation (in software only)
3. Interface description of the allowed operation modi
4. Abstract description of requirements on the state space
5. Representation of allowed sequences of stimuli occurrences (input signals/arriving things)
6. Characterization of the possible or allowed behaviors of a component
7. Connection between state and behavior of a component



Use of Statecharts for Code Generation

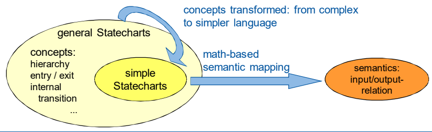
- A Statechart can be used for code generation
- This code can be:
  - A) part of the software product
  - B) part of a test driver
  - C) part of a simulation (especially, when modelling physical things)
- In a simulation, the
  - physical things are simulated through messages (data),
  - physical states are simulated through data states
- The principles of code generation are (almost) the same in both cases
- Further uses for Statecharts:
  - D) generator can derive test cases (along the paths)
  - E) run-time monitoring
  - F) visualization of executions
- These require other forms of code generators



Semantics – Revisited

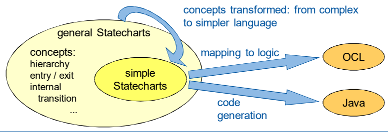
The semantics of a Statechart is defined in several levels:

1. Mealy Automaton (the core):
   - the semantics of a Mealy automaton is a relation of input and output sequences
   - interpretation: inputs are method calls, outputs are actions.
2. State invariants are added to refine the description.
   - connection between diagram and object states, allowing to cope with infinite states.
3. Additional concepts, such as hierarchy, entry/exit actions etc.
   - are transformed to a simpler sub-language of the Statechart language



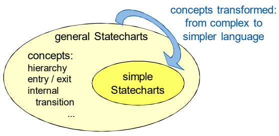
Transformations of Statecharts

- Transformations can map complex concepts to simple ones.
- Usage in
  - semantic definition (as shown before)
  - code generation
  - optimization of Statecharts (state minimization, ...)
  - mapping of Statecharts to logic constraints
- Transformation to code generation is analogous to the semantic definition, however, executability of transitions is important now:



Simplification of Statecharts by Transformation

- Collection of transformations has been presented on the previous slides
  - Applied in an intelligent order
- Most of the steps can be automated (i.e. by a tool)
  - design decisions in some cases necessary or advisable for an optimized implementation
  - decidability in logic constraints is not always given:
    - check manually or use verification tool?
- Only few optimization steps are missing and thus shown below to complete the transformations



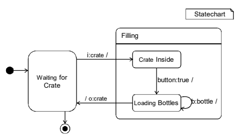
- Result of the transformation procedure: simplified Statechart without hierarchy (flat)

MBSE

6B. Executing Statecharts
6B.2. Transforming Statecharts

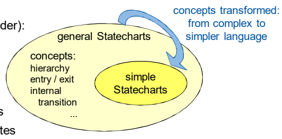
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Procedure to Simplify Statecharts: Steps 1-9: Remove Hierarchy

These steps are already known (and here is their application order):



1. Eliminate do-activities
2. Transform inner transitions to real transitions

3. Target states with substates: forward transitions to substates
4. Source state with substates: let transitions start from substates
5. Repeat 3.-4. at all levels of hierarchy until transitions have only atomic source and target states

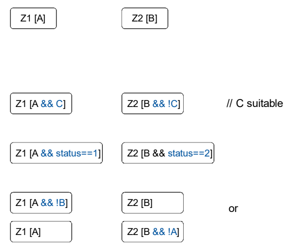
6. Move exit-actions of the state to the action of each outgoing transition
7. Move entry-actions to the incoming transitions analogously
8. Include state invariants of superstates explicitly in substates

9. Remove hierarchic states (only keep the atomic ones)

Procedure to Simplify Statecharts: Step 10: Refine State Invariants

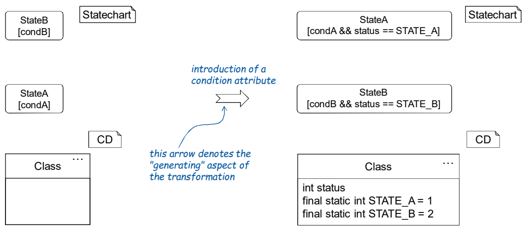
10. Refine state invariants
- Starting point: $A \wedge B \neq false$
- Objective: obtain data states
  by transferring into disjoint state invariants

- Alternatives:
  - conjugate invariants with other conditions, until disjoint
  - introduce state attribute ("status") and use it in invariant
  - remove overlapping invariant part from a state



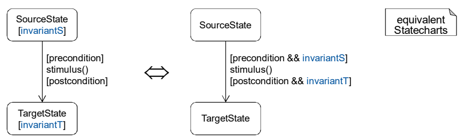
Example for Step 10:

- Can for example be used as implementation



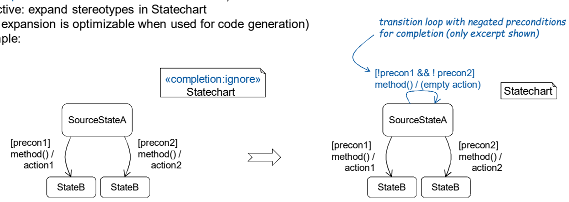
Procedure to simplify Statecharts: Steps 11-13: Remove State Invariants

11. Integrate state invariants into the preconditions
    - Objective: preconditions of transitions contain all information
12. Add state invariants to action conditions
    - Objective: action conditions of transitions contain all information
13. Remove state invariants



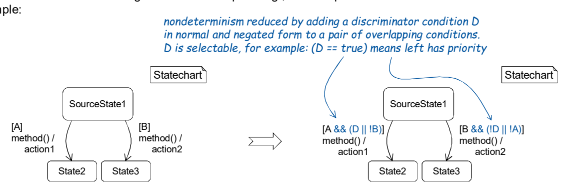
Procedure to simplify Statecharts: Step 14: Completion

14. Completion of the Statechart
    - depending on type: «error», «exception», «completion:ignore»
    - («completion:chaos» needs another transformation)
- Objective: expand stereotypes in Statechart
- (This expansion is optimizable when used for code generation)
- Example:



Procedure to simplify Statecharts: Step 15: Nondeterminism

15. Reduce the nondeterminism in overlapping transitions
    - introduction of a discriminator D
- Objective: deterministic Statechart
- There are more efficient code generation techniques: e.g., order of precondition checks
- Example:



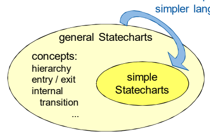
Procedure to simplify Statecharts: Step 16-17: Readiness to Fire, Reachability

16. Eliminate transitions that are not ready to fire
    - i.e. by precondition == false
- Objective: by the many transformations so far, many transitions have been duplicated and refined with additional conditions.
    - This may include empty fire conditions: These transitions are removable
    - ideal: test the firing conditions already during the transformation
    - undecidability issues if first-order-logic constraints are involved

17. Eliminate unreachable states
    - by building transitive closure over enabled transitions

- These steps 16, 17 are optimizations only.
- Final result: a substantially simplified flat form of Statecharts

concepts transformed: from complex to simpler language

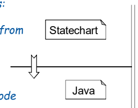


Code Generation

- Starting point:
    - simplified Statecharts:
        - state invariants not yet expanded
        - but states are flattened, etc.
        - i.e. transformations 1-17 have been applied
- Possible variants for representation of states:
    - explicit state attribute describes state (e.g. using "int state")
    - invariants of disjoint states as predicates, or
    - state pattern: each state is associated with an own object (this is a design pattern from Gamma et.al. 1994)

Representation for transformation rules:

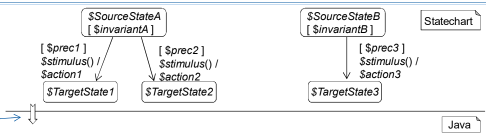
top = matching part from the model



bottom = resulting code

$elem describes a piece of the model to be matched (on top) and copied (to bottom)

Ruled defined by An Example



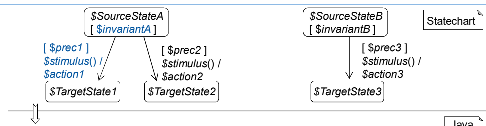
Java

transformation rule: from top to bottom

lets see … what happens with this example

Note:
- *Many more stimuli are possible in the same states*
- *Many more states*
- *The above covers a situation with may be overlapping firing conditions*

Variant 1: Disjoint Invariants for States



Java

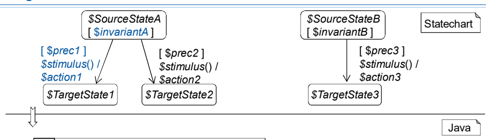
Result: double nested if-then-else cascade

state invariant and preconditions are used to distinguish the transitions

```
public ... $stimulus() {
  if($invariantA) {
    if($prec1) {
      $action1;
    } else if ($prec2) {
      $action2;
    } else {
      // error handling
    }
  } else if($invariantB) {
    ...
}}
```

disadvantage: code $invariantA is repeated in the resulting code (lots of code)

Variant 1 + Outsourcing State Invariants into own Predicates



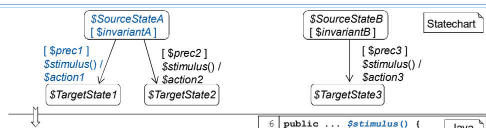
Java

each state is mapped to a predicate that evaluates the state invariant

```
public boolean inv$SourceStateA() {
    return $invariantA;
}
public boolean inv$SourceStateB() {
    return $invariantB;
}

public ... $stimulus() {
  if(inv$SourceStateA()) {
    ...
}
```

advantage: $invariantA generated only once.
disadvantage: $invariantA can be complex and time-consuming when executed
better: simple states attribute "remembers" current state

Introduction of a State Attribute



Result uses switch statement

state diagram is stored as enumeration

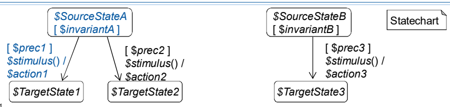
```
private int status;
final static int $SOURCE_STATE_A = 1;
final static int $SOURCE_STATE_B = 2;
final static int $TARGET_STATE_1 = 3;
...
```

Java

```
public ... $stimulus() {
  switch($status) {
    case $SOURCE_STATE_A :
      if($prec1) {
        $action1;
        $status = $TARGET_STATE_1;
      } else if ($prec2) {
        $action2;
        $status = $TARGET_STATE_2;
      } ...
    break;
    case $SOURCE_STATE_B :
      ...
}}
```

Advantage: efficient
Disadvantages: redundant storage + consistency not assured, i.e. always must hold (status==$SOURCE_STATE_A) implies $invariantA

Using Invariants for Tests



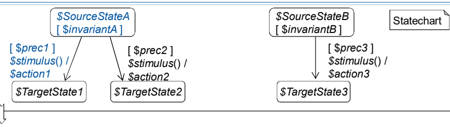

```
private int status;
final static int $SOURCE_STATE_A = 1;
final static int $SOURCE_STATE_B = 2;
final static int $TARGET_STATE_1 = 3;
...
```

Java

```
public ... $stimulus() {
  switch($status) {
    case $SOURCE_STATE_A :
      assert $invariantA;
      if($prec1) {
        $action1;
        $status = $TARGET_STATE_1;
      } else {
        assert $prec2;
        $action2;
        $status = $TARGET_STATE_2;
      } ...
    break; ...
}}
```

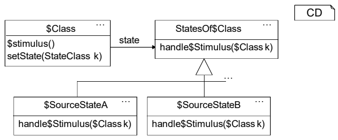
state invariants and some preconditions can be used as assertions for testing and simulation purposes and withed off in final code

Design Patterns: State Pattern (Gamma et.al., 1994)

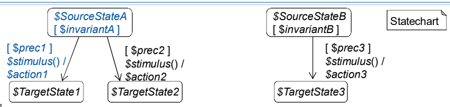


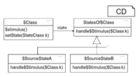
State pattern produces one individual subclass for each state. → complex structure, but also: easier to adapt by handwritten code.

Core idea: instead of a switch, let the OO dynamic lookup do the selection of the code efficiently



Design Patterns: State Pattern (Gamma et.al., 1994)





Advantage: additional flexibility.
Disadvantage: overhead due to classes and objects: one for each state.

Java

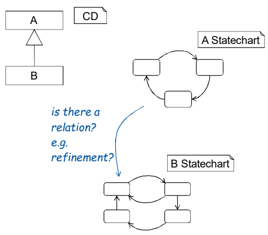
```
class $Class {

  $SourceStateA $sourceStateA = ...
  $TargetState1 $targetState1 = ...

  State$Class state;

  public ... $stimulus()
  { state.handle$Stimulus(this); }
  public setState(StatesOf$Class n)
  { state=n; }
}
```

Java

```
class $SourceStateA {

  public ...
  handle$Stimulus($Class k) {
    assert $invariantA';
    if($prec1) {
      $action1';
      k.setState(k.$targetState1);
    } else
    ...
}}
```

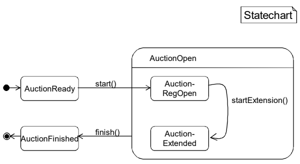

Inheritance of Statecharts

- Which requirements does a Statechart of the superclass impose for objects of the subclass?
  - different views (Harel, UML, Rumpe, ...)
- Formally:
  - subclass leads to behavior refinement
  - therefore: subclass must refine the behavior specified by Statechart
  - appropriate transformation rules ensuring this do exist
- Pragmatic view:
  - behavior refinement by transformation rules rather rigid
  - better: use of automata to test behavioral conformity



Summary Code Generation From Statecharts

- Statecharts are an extension of the Mealy machines.
- Step 1: Transforming Statecharts to a simpler form
- Step 2: Mapping flat Statecharts to Code. e.g. using the state pattern
- Code from Statecharts is usable in various phases of software development: analysis, design, implementation for
  - Test
  - Product code
  - Simulation

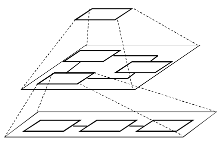


MBSE

7. Architectural Design
7.1. Architecture

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



System Architecture

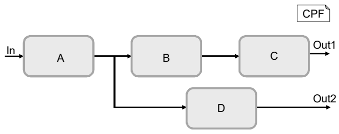
- System Architecture is the overall, macroscopic system structure: collection of physical and/or computational components together with connectors that describe their interaction.
- What is fundamental to understanding a system in its environment
- Things that people perceive as hard to change
- Architectural design decisions
  - not merely models or structures
  - include the decisions that lead to these structures, and the rationale behind them

Architectural Style

- In traditional building architecture: a specific method of construction, characterized by its notable features
- An architectural style defines:
  - a family of systems in terms of a pattern of structural organization
  - a vocabulary of components and connectors, with constraints on how they can be combined [SG96]
- Architectural styles provide design decisions and constraints to induce desirable qualities
- Through architectural styles design decision are documented upfront and pervasive through the system
- Facilitate reuse, understandability, interoperability

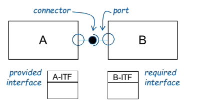
Architectural Styles: Pipe-and-Filter

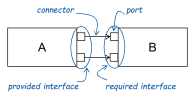
- Filters process input data and produce output data
- Pipes connect filters
  - often linear, but branching is possible
- Focus on data processing
  - (as opposed to, e.g., layers)
- A filter has several (often only one) 'in' streams and 'out' streams
  - syntactic compatibility: filters can be connected if compatible
- Heavy use in shells of Unix systems, e.g., in the command chain:
  - ls -l | grep "LOG" | sort -r



Component and Connector Pattern as a specific Architectural Style

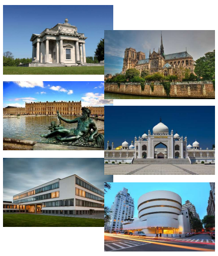
- Component & Connector Architecture Description Languages (C&C ADLs):
- Component: black-box performing functions behind an explicit interface
  - atomic components vs.
  - composed components have topologies of subcomponents,
  - component interface: set of (possibly directed) ports
- Connectors: connect components via their ports
- More generic patterns can be found denoted in other architecture description languages (ADLs)





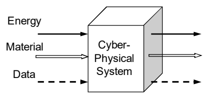
Other Architectural Styles

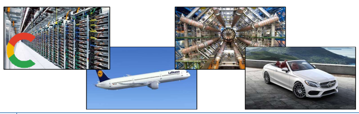
- Monolithic Architectures
- Layered Architectures
- Event-driven Architectures
- Publish-subscribe Architectures
- Client-Server Architectures
- Service-oriented Architectures
- Peer-to-Peer Architectures
- Microservice Architectures

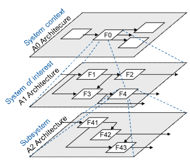


Modeling Architectural Structure for CPS

- Modeling cyber-physical systems needs to describe the structure of relevant things including
  - components – material – energy – data
- Functions of CPS use the data types for their channels & variables.
- A system is structurally decomposed in subsystems and components.
  - structural modelling is used throughout the complete development (design, validation & verification, deployment…)





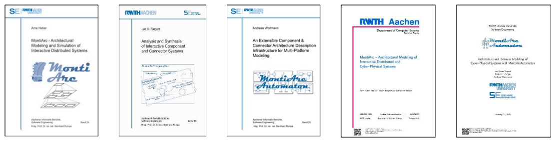


Literature

- [Wik-SysE] https://en.wikipedia.org/wiki/Systems_engineering
- [SeBoK] SEBoK Editorial Board. 2020. *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*, v. 2.2, R.J. Cloutier (Editor in Chief). Hoboken, NJ: The Trustees of the Stevens Institute of Technology. Accessed [20.07.2020]. www.sebokwiki.org.
- [BP07] Pahl, G., Beitz, W., Feldhusen, J., & Grote, K.-H. (2007). *Engineering Design - A Systematic Approach*. London: Springer.
- [BS08] Boardman, J. and B. Sauser. 2008. *Systems Thinking: Coping with 21st Century Problems.*
- [Alu15] Alur, R. (2015). *Principles of cyber-physical systems.*
- [KK98] Koller, R., & Kastrup, N. (1998). *Prinziplösungen zur Konstruktion technischer Produkte*.
- [Lee16] Lee, E. A. (2010). CPS foundations. *Proceedings - Design Automation Conference*, 737–742.
- [Lee08] Lee, E. A. (2008). Cyber physical systems: Design challenges. *Proceedings - 11th IEEE Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*.
- [FG13] Feldhusen, J., & Grote, K.-H. (Eds.). (2013). *Pahl/Beitz Konstruktionslehre*.
- [BDS19] Broy, M., Daembkes, · Heinrich, & Sztipanovits, J. (2019). Editorial to the theme section on model-based design of cyber-physical systems. *Software & Systems Modeling*, *18*, 1575–1576.
- [BS01] Broy, M., & Stølen, K. (2001). Specification and development of interactive systems. In *Monographs in computer science*. New York: Springer.

Literature 2

- EAST-AADT Language Specification: https://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf
- [HRR12] A. Haber, J. O. Ringert, B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. RWTH Aachen University, Technical Report. AIB-2012-03. February 2012.
- [RRW14a] J. O. Ringert, B. Rumpe, A. Wortmann: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. In: Aachener Informatik-Berichte, Software Engineering, Band 20. ISBN 978-3-8440-3120-1. Shaker Verlag, 2014.
- +

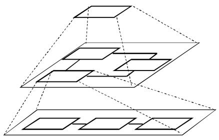


MBSE

7. Architectural Design
7.2. Function Composition Paradigm

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

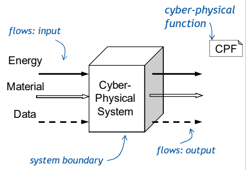
http://www.se-rwth.de/



System Specification through Functions

Repetition

- A system defines a cyber-physical function
  - it encapsulates a physical and computational structure
  - performs data, energetic and physical transformations
  - and is connected to its context through its interfaces.
- A system function is described through its
- input and output signature
  - types and forms of the
    - signals / data
    - energy flow
    - material flow
- The functionality is mathematically described through the
  - relation between input and output

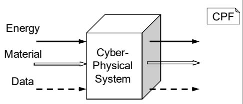


⇨ The concept of function is our first universal specification and construction principle

Models describing System Functions

Repetition

- A system defines a function
- Aspects to be defined in abstract, purpose fitting models:
  - Interface signature
  - Internal structure (architecture)
    - Logical structure
    - Geometrical shape
  - Behavior (over time)
  - Interactions
  - Assumptions about the context

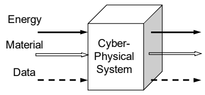


⇨ Abstraction with dedicated models to master complexity is the second universal principle.

The Underspecification Principle

Repetition

- Deterministic and fully specified relations are normally not achievable
  - Delays happen
  - Energy fluctuates
  - Abstraction introduces lack of information
- Underspecification is the ability to describe the desired range of allowed behaviors (instead of a single, determined behavior)
- Advantages:
  - Easier to specify
  - Can be well combined with variant-building and methodical refinement

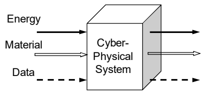


⇨ Controlled, explicit underspecification is the third universal specification principle

Signature of Input and Output of a Function

Repetition

- The signature of a function describes the forms of interactions of a system component with its environment.
- Interactions are broken down to streams of elements, which describe the flow and can be of the kinds
  - data,
  - energy or
  - material
- Interactions are organized through input and output channels.

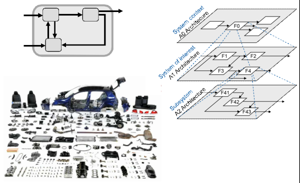


- The Interface of a Cyber-Physical System is defined through its function signature

The concept of stream is our fourth universal specification and construction principle

Composition

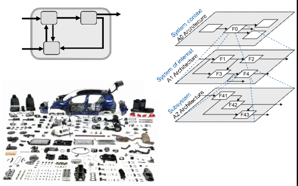
- Composition is an act or mechanism to combine simple elements to build more complicated ones
- Examples: function composition (math), product composition (mechanics), software composition (CS), …
- System is composed of components.
- Component is atomic or hierarchically composed of simpler components.
- Sub-system ~ non-atomic component



Composition is the 5th universal construction principle.
It helps to manage complexity.

Decomposition

- Decomposition is the act of deconstructing a specification into a structure of smaller sub-specifications
- Software is decomposed according to logical functions
- Physical systems are decomposed according to geometry
- Electronics is decomposed using electric devices and chips
- Decomposition and composition complement each other:
  - Decomposition structures the problem
  - Small sub-problems are solved into solution components
  - Composition integrates the components into a system

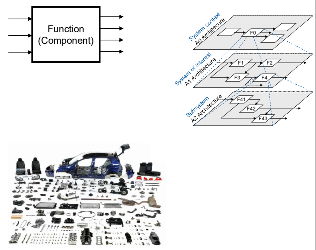


Interaction between the components enforces to cope with interfaces and their structures, e.g., by explicitly defined architectures (software) or composition plans (physical).

From Autobild and SEBoK

Decomposition in the Development Organization

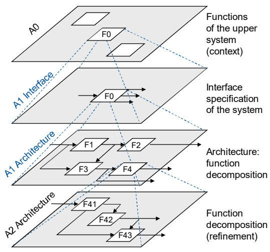
- Decomposition is paramount to manage complexity
- Software and physical systems decomposition are relatively orthogonal and largely incompatible:
  - Logical functions vs. physical geometry
- Interfaces easily become overly complex
- Thus, decisions need to be made:
  Who is the complexity and innovation driver?
- Consequence:
  Development divisions are structured like their products are decomposed.



From Autobild and SEBoK

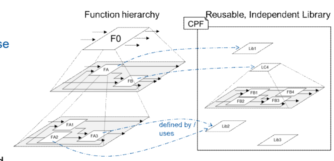
Decomposition introduces a Tree of Components

- Decomposition can be organized in layers, but finally forms a tree.
- This pattern can be repeatedly applied:
  - The level above (A0) describes the context of the system of interest
  - The System of Interest (SoI) (A1) describes the system as
    - A1 Interface: a black box function designing the interface
    - A1-Architecture: a decomposed structure consisting of component functions
  - The component functions are then described and modeled one level below (A2) again (using interfaces and decomposition)
- The hierarchy of decomposition may be imbalanced: do not use (numbered) layers on a global scale



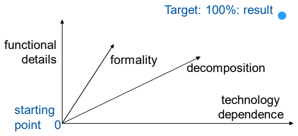
Decomposition supports Reuse … when done properly

- Functional decomposition leads to a hierarchy
  - A tree of subcomponents, each described as function
- Reuse of identical subcomponents enforces to distinguish component definition and component use
- Libraries define components in an independent reusable and adaptable form
- Reuse is black-box: no copy-pasting of models but referring to an existing artifact by name.
- Reuse is based on development for abstraction and encapsulation
  - E.g., technology dependent / product specific signal names disallow reuse



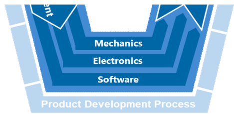
Four Dimensions in Development

- Degree of functional detail
  - from abstract principle, through signal types to differentials
- Degree of formality
  - from informal text, through models to logic formulae
- Decomposition hierarchy
  - from system, through subsystem to atomic component
- Degree of technology dependence
  - from abstract principle, through principle algorithm to HW specific machine code
- Main problem of process definition:
  Finding the optimal path from 0 to 100% result



Decomposition in the V-Model

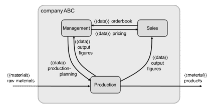
- V-Model suggests early decomposition in:
  - Mechanics, software and hardware?
- Potential merits:
  - Company structure according to domain
  - SW-Developers are less dependent on mechanics
    - own processes, own development culture, …
  - SW-SW interfaces are bigger than SW/Mechanics interfaces (shorter paths for discussion)
- Risks:
  - Not easy to optimize software/mechanics co-design
  - SW/mechanics interface to be defined early

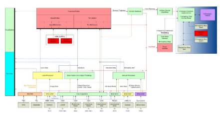


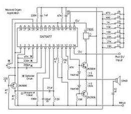
Bottom part of a Systems Engineering V-Model version

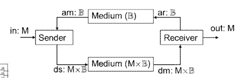
⇨ Decomposition structure should depend on innovation and complexity drivers

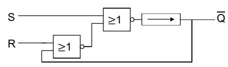
Examples: Composed Function Architectures







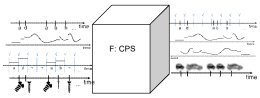


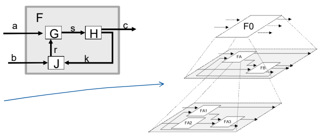


Decomposition and Behavior Semantics Using Streams

- Function F specifying a CPS has behavior semantics based on streams of messages/material flowing over channels.
- Mathematically F is a set of mathematical functions:
  - $F \subseteq I_1^\sigma \times I_2^\sigma \times \ldots \times I_n^\sigma \rightarrow O_1^\sigma \times O_2^\sigma \times \ldots \times O_n^\sigma$

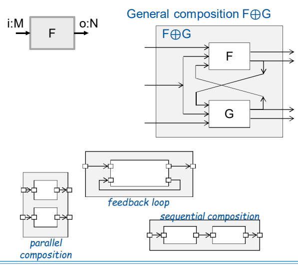
  that obey timing restrictions
- Decomposition of CPF into a CP Architecture is explained by mathematical function decomposition,
- Example:
  - $F = G \otimes H \otimes J$
- which is equivalent to explicit use of stream channels
  - $F(a,b) = c$ where
    $\exists$ k,r,s: $H(s)=(c,k) \wedge s=G(a,r) \wedge r=J(b,k)$
- Math also explains hierarchical CPF decomposition.
- Decomposition is compatible with underspecification and its refinement.



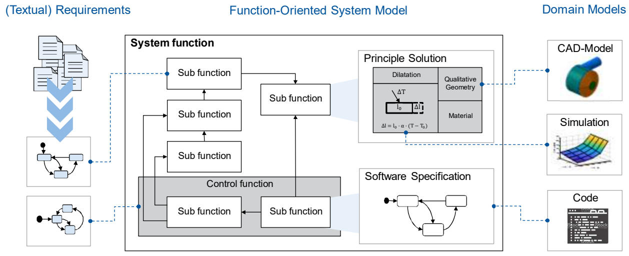


Forms of Composition

- Components are connected using typed channels
  - Component composition maps to mathematical function composition
- General composition F⊕G
  - allows feedback loops between components, with a sound mathematical foundation
  - is commutative and associative
- Thus composition ⊕ allows to compose arbitrary architectures of components → networks
- Special forms of composition can be derived
  - Most common: parallel c., sequential c., feedback



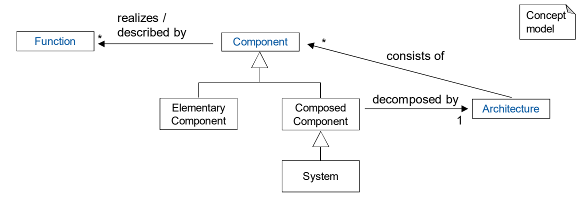
Systems Engineering: From Requirements through Functions to Components



Summary: Completed Set of Universal Construction Principles 1-5:

- 1: The function concept is a universal specification and construction principle
  - → Functions are a well-known mathematical construct that allow us to model system functionality precisely
  - Functions (and related math structures, such as continuous or discrete time, abstract data types) are the connection between systems thinking and mathematical foundations.
- 2: Abstraction with dedicated models to master complexity is the 2nd universal principle.
- 3: Controlled, explicit underspecification is the 3rd universal specification principle
  - → Underspecification allows us to model absence of information or uncertainty in analysis, variability of the products, degrees of freedom when customizing a component and also behavioral nondeterminism that occurs during system operation.
- 4: The concept of stream is our 4th universal specification and construction principle
  - → Streams allow to describe the "flow" of elements (material, data, data) through input and output interfaces over time. Dense, even continuous, or discrete streams allow to model all forms of possible behavior of a function.
- 5: Composition is the 5th universal construction principle.
  - → Composition and decomposition are essential to manage complexity.

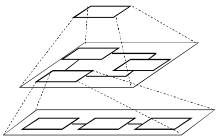
Summary as a Concept Model



- Decomposition is a main principle to master complexity.
- Streams / stream processing functions are the mathematical manifestation of CPF.

MBSE

7. Architectural Design
7.3. MontiArc for Function Architectures

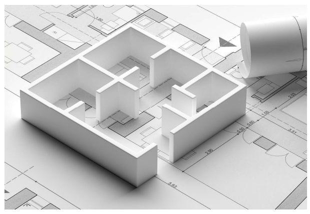


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Architecture Description Languages (ADLs)

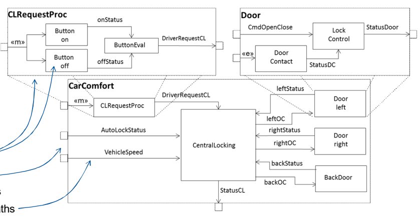
An ADL is a modeling language designed to describe systems in their decomposed structure and behavior



- Often with graphical or textual syntax
- Use
  - communicating architecture to all interested parties
  - exploring alternatives
  - support architecture creation, refinement, and validation
  - blueprint for further implementation
  - enable analysis and generation (mainly in software)
- Sometimes
  - domain-specific syntax (e.g., automotive, avionics, …)
  - formal foundations (i.e., well-defined semantics)

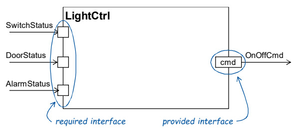
The MontiArc C&C ADL

- MontiArc is an ADL
  - developed using MontiCore,
  - based on the stream approach
  - for modeling software and system architectures
  - extensible with component behavior languages
- Most important MontiArc elements
  - component: unit of computation
  - interface: has typed, directed ports
  - hierarchy: topology of subcomponents
  - connectors: realize communication paths

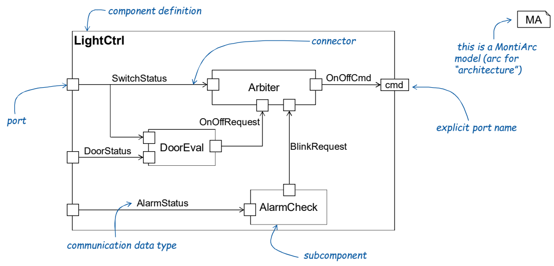


MontiArc: Initial Example

- The component LightCtrl
  - controls the interior light of a car
  - receives the status from the light switch, the alarm system and the car's doors
  - emits a command to turn the interior light on or off
  - the light is switched depending on the doors
    - doors are opened: the light is turned on
    - doors are closed: lights are turned off after a short delay
  - if the alarm system is active, the interior lights blink

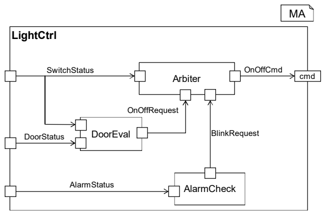


MontiArc: Initial Example



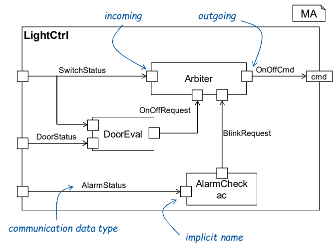
MontiArc: Components to Realize Functions

- The notion of component is central:
  - explicit interface definition
  - encapsulates its internals: can be used as black-box
- Two variants:
- Decomposed component definition
  - hierarchically decomposed to an architecture of sub-components
  - describes their communication
  - does not realize behavior itself, but usually has a black-box specification of its behavior
- Atomic component definition
  - usually not further decomposed
  - behavior is specified / implemented directly



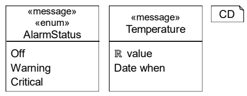
MontiArc: Ports and Connectors

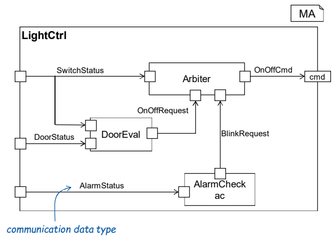
- Port
  - has a direction (unidirectional)
    - incoming port: receives messages
    - outgoing port: emits messages
  - has a name and a type (of its messages)
    - we may omit the name in the graphical representation if the port is uniquely identifiable by its type
- Connector
  - defines a connection between ports
    - connects one input with one or several output ports
  - forwards messages
  - can only connect ports of the compatible types



MontiArc: Data Types for Ports

- Communication data types
  - define the structure of messages
    - define the set of all possible messages exchanged in a connection
  - E.g., modeled in a class diagram
    - classes are possible communication types
    - attributes define their content
    - class instances represent possible messages



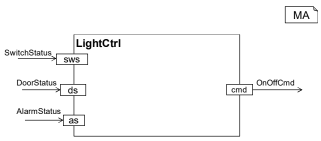


MontiArc: Textual Syntax - Ports

```
component LightCtrl {

  port in SwitchStatus sws,
       in DoorStatus ds,
       in AlarmStatus as,
       out OnOffCmd cmd;

}
```

ports

direction type name



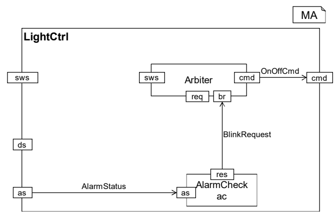
- Each port consist of
  - a direction (in/out)
  - a type (defined somewhere else, e.g., in a class diagram)
  - a name (must be unique in that component)
- Ports can be referenced via their name

MontiArc: Textual Syntax – Subcomponents and Connectors

```
component LightCtrl {
  port in SwitchStatus sws,
       in DoorStatus ds,
       in AlarmStatus as,
       out OnOffCmd cmd;

  Arbiter arbiter;
  AlarmCheck ac;


  as.res -> arbiter.br;
  as -> ac.as;
  arbiter.cmd -> cmd;

}
```

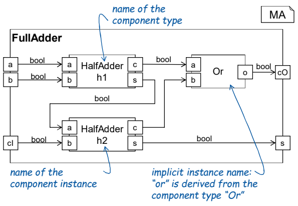
type name subcomponents

subcomponent port connectors



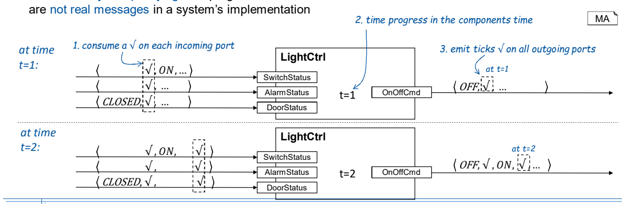
MontiArc: Component Instances

- A component instance belongs to a component type
- Distinguish between component types (definition) and component instance (usage) to foster reuse
- Libraries provide sets of reusable, black-box component types
- Component types can be instantiated multiple times in the same topology or across hierarchies

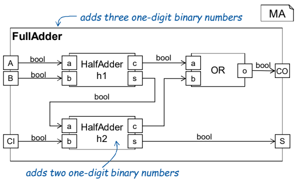


Time in MontiArc

- Virtual "tick-messages" √ describe timing progress in specification and simulation
  - each component emits a tick after consuming a tick
  - synchronizes consumption on incoming ports
- Ticks are only for specifying time progress and are not real messages in a system's implementation



MontiArc: Example FullAdder



Functional definition of components is given by truth tables:

HalfAdder

a	b	c	s
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

OR

a	b	o
0	0	0
1	0	1
0	1	1
1	1	1

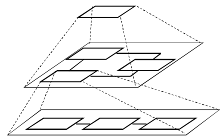
Derived FullAdder behavior over sequence of inputs:

	time				
A	1	1	0	0	1
B	0	1	1	0	1
CI	1	1	0	1	1
S	0	1	1	1	1
CO	1	1	0	0	1

```
S = sum
CO = carry over, out
CI = carry over, input
```


MBSE

7. Architectural Design
7.4. Simulation with Functional Architectures



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

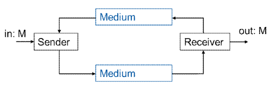
http://www.se-rwth.de/

Definition: Simulation

A simulation is an approximate imitation of the operation of a process or system that represents its operation over time.

- Simulation [...] of technology for performance tuning or optimizing, safety engineering, testing, training, education, and video games.
- Simulation is also used with scientific modelling of natural systems or human systems to gain insight into their functioning.
- Simulation can be used to show the eventual real effects of alternative conditions and courses of action.
- (all from Wikipedia)

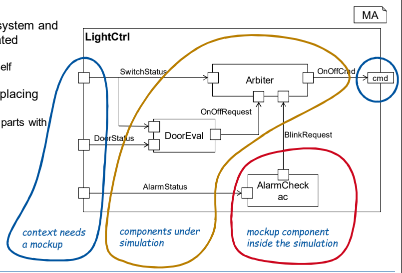
- Simulation always has a purpose!
  - Often, computer simulation experiments are used to analyze models (before systems are built).
- A simulation often uses mock-up models for certain components



- Simulation is NOT the same as Visualization!

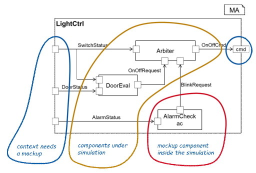
Simulation vs. System

- In software the difference between the system and the simulation of the system is complicated
  - software may simulate itself.
  - a mock may be the mocked component itself
- Simulation of physical components is replacing them by software
  - co-simulation combines some mechanical parts with some software mockups
- In a simulation setting:
  - identify the simulated elements and the mockups that drive a simulation
  - (like with normal software testing)
  - partial simulations
  - environment vs. system



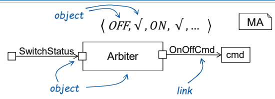
Simulation Infrastructure For MontiArc

- In the following we exemplarily examine a possible simulator infrastructure for MontiArc
- MontiArc provides a simulation infrastructure with the following key characteristics:
- 1) Simulation of the distributed system within a single Java machine
- 2) Each component is realized as a Java object
- 3) Time may optionally be used in the simulation
- 4) If so, time is simulated using Tick messages
- 5) Simulation consists of generated code and an runtime environment (RTE)
- 6) Efficiency is relevant
- These design decisions have consequences.
- In summary:
  - code, RTE and generators for simulation vs. simulation differ in various ways.



MontiArc Mapped to Java Simulation

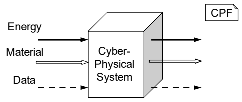
- MontiArc language constructs map to Java:
- Component → Object
- Port → Object
- Connection → Links (chain of links)
- Message → Object
- Tick → Special Object
- Buffer to store message queues → Queue, List
- Behavior → Method
- Scheduling → Management method in scheduling object



- Flexibility by design:
  - Local behavior implementation for atomic components
  - Local ports, local schedulers
- But also defaults:
  - For the buffers, ports and the scheduling
  - Only atomic behavior implementations need to be added

MontiArc Mapped to Java Simulation

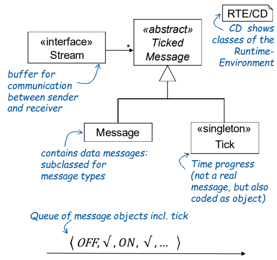
- MontiArc "typing" maps to Java typing:
- Component type → Class
- Port type → Class
- Connection → Association
- Message type → Class
- Tick → Singleton Class
- Buffer to store message queues → Queue, List class
- Scheduling → Management method in scheduling class



- Remember:
  A system defines a cyber-physical function
  - it encapsulates a physical and computational structure
  - performs data, energetic and physical transformations
- A system function is described through its
- input and output signature
  - signals / data, energy flow, and material flow
- All components, ports and channels are mapped, i.e.
  - material transport is mapped to communication

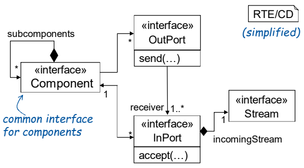
Realization of Streams of Messages with Ticks: Handling Simulation Time

- A stream stores a buffer of messages
- The tick-message simulates time progress
- Message class subsumes various types of messages
  +: allows a generic implementation of message buffer
  -: enforces marshalling of all kinds of types (e.g. int, String)
- Stream (+ implementation): the buffer
  - stores the buffered elements like a Queue
  - variant: allows to retrieve history (e.g. for testing)
- Explicit encoding of the tick as object decouples "simulated time" from simulator execution time, which is also called "wall-clock time" or "elapsed real time"



Realization of Ports

- Components consist of subcomponents and have in- and out-ports
- Ports forward and store messages in a stream
- Receiver association: realizes the connection
  - similar to a subscriber
- OutPort: accepts messages (via send-method) and sends it to the receiver
- InPort: stores messages that can be retrieved fromm the component (accept)
- ForwardPort acts a (efficient combination)
  - looks like a normal incoming port from outside but
  - forwards received messages to the ports of subcomponents



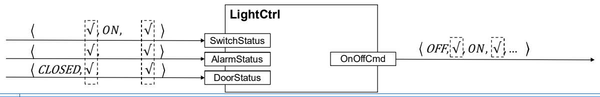
- *encapsulates incoming ports from contained sub-components*
- *looks like a normal incoming port from outside but*
- *forwards received messages to the encapsulated ports*

Scheduling in Simulation

- Simulated systems usually acts in parallel, but simulation may be single threaded → efficient and controllable!
  - But the execution of messages needs to be scheduled
  - Scheduler decides on the order of execution
- Design:
  - Each subsystem can have its own local scheduler
  - Messages are stored in streams at InPorts
    - InPort receives a message and notifies its scheduler
  - Scheduler
    - decides which messages are given to the components
    - default: first in, first out
    - but also handles time synchronization
- Time synchronization by scheduling the tick-message:
- Tick's are received individually in each port
- However, time is processed synchronous:
  The tick is given to the component exactly when:
  - All incoming port received the tick (i.e. have completed their time slice)
  - All messages prior to the tick are handled
- Consequence: component receives one tick all ports synchronously (= time progress)

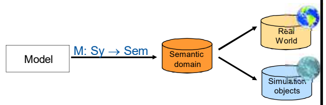
Simulating Time Progress

- Time synchronization by scheduling the tick-message:
- Tick's are received individually in each port (and "stored") at first
- Time is processed synchronous: The tick is given to the component exactly when:
  - All incoming port received the tick (i.e. have completed their time slice)
  - All messages prior to the tick are handled
- Consequence: component receives and emits only one tick synchronously (= time progress)



Simulation vs. Real World

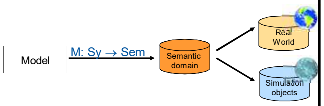
- The real world differs from simulations
  - Simulations execute models
    - models are abstractions
    - rely on assumptions about the real world
  - The models abstract from details: they may be too abstract
    - miss relevant phenomena of the real world?
    - certain technical details may be relevant
    - deficiencies of the real world may be relevant

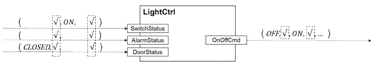


- What helps?
  - Underspecification captures not fully known real behavior,
  - Know the applicability conditions of a model (e.g. Newton's law does not hold in outer space)
  - Add more real world phenomena to the models if needed

Time in Simulation vs. Real World

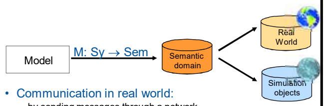
- The real world differs from simulations
- Time in Simulation:
  - The simulator can control the progress of time
  - (if no real hardware is involved)
  - Simulation can thus be much faster than the elapsed real time, e.g. in climate models that is very useful.
  - Simulation can also be much slower, e.g. in particle physics or the human-brain
- Time in real world:
  - Time cannot be influenced by the system
  - "Ticks do not exist in the real world!"



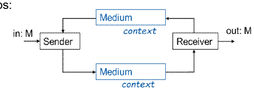


Communication in Simulation vs. Real World

- The real world differs from simulations
- In a simulation, transfer of physical gadgets and energy is mapped to message communication
- Communication in a simulation:
  - If the simulation is single threaded: Components exchange data by copying values in RAM
  - In HPC special operating system and middleware communication exists for transport, but this has different behavior than the targeted communication model
  - In HW/SW-co-simulations the real communication system resp. the physical item transport may be used
    - Rest-bus-simulations mock parts of the real components

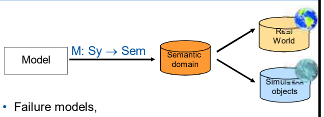


- Communication in real world:
  - by sending messages through a network
- Network may have deficiencies, such as
  - Varying latency; drop, repetition or altering of messages
- Explicit modelling the of the communication context helps:

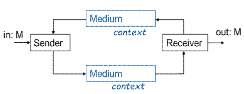


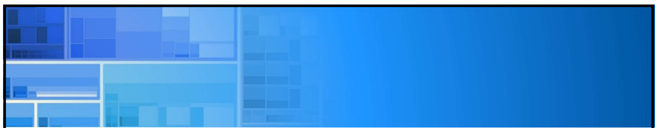
Assumptions about the Real World in a Simulation

- A simulation is based on a model (= abstraction of the real world)
- then there are underlying assumptions (that can be violated?).
- Fail Safe
  - Vibration, high/low temperature and other influences cause devices to fail
- Reliability
  - Sensors may provide inaccurate or wrong values; Actuators may fail to execute their tasks
- Lack of Isolation
  - External circumstances influence the system
- These can be modelled explicitly, by refining the models
  - Every form of (mis-)behavior can be modelled



- Failure models,
- Stochastic models, …
  - allow to understand the existing context,
  - but also to define the systems operability context



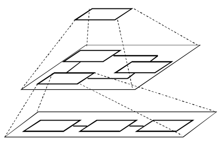


MBSE

7. Architectural Design
7.5. Architectures of CPS

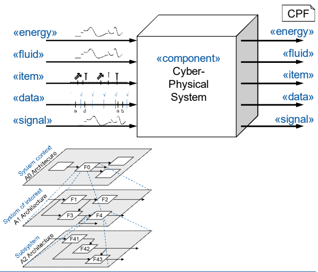
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



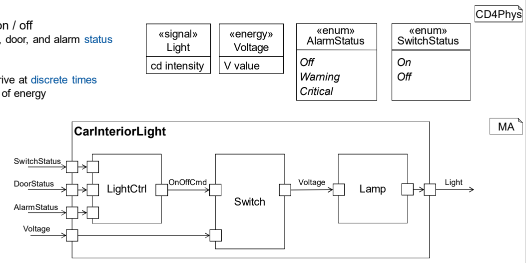
Components of a CPS

- A component is the realization of a cyber-physical function
  - defines the explicit interface of the function
  - provides the function via that interface
- Component decomposition is defined by functional decomposition
  - each component is described by a function
  - functions are composed like in math
- Ports define the interface of a function
  - explicitly typed
  - but energy, data, and materialized things don't mix



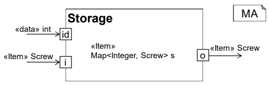
Example: Car Interior Light

- Switch car light on / off
  - based on switch, door, and alarm status
- Input:
  - Discrete data arrive at discrete times
  - Continuous flow of energy



Example: Storage of Screws

- Input:
  - Screws arrive as discrete items
  - Store releases arrived material based on identifier
- Internal state:
  - the received items are stored in dedicated racks, which is modeled by a
    - Map<Integer, Screw> s initialized with ε
    - The map looks like software data, but models a real storage with physical screws
- Output:
  - And as specification the processing of screws given as SpesML spec for messages arriving at the two channels

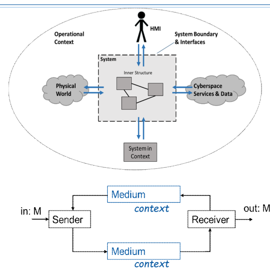


SpesML

```
spec Storage
  port in  Screw din;
           int id;
  port out Screw dout;
------------------------
Map<Integer, Screw> s = Map[];
------------------------
incoming screw:din
post: s.get(screw.id)==screw &&
      dout=epsilon
------------------------
incoming id
pre:  s.containsKey(id)
post: s==s@pre.remove(id) &&
      dout==s@pre.get(id)
```

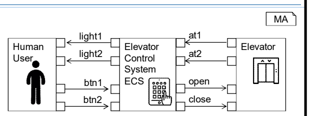

Modelling System and System Context

- MontiArc can be used to model the context of a system
  - context is also modelled by a set of components
  - but the intention is not to realize them, but to use them for specification, testing and simulation
- Example of context:
  The internet (communication medium) in a protocol:
  - goal: transmit messages
  - system to develop: Sender and Receiver
  - context: Medium, is defined to explicitly specify assumptions about the medium
- In general: the smarter a system is, the more assumptions about its context need to be modelled.
  - e.g. autonomous cars, connected airplanes, …



Humans as System Context

- Example: Elevator System (ECS)
  - defined by
    - Elevator (mainly the physical gadget)
    - ElevatorControlSystem (the software part)
    - HumanUser (part of the context)
  - interfaces are defined by a MontiArc model
- Untrained HumanUser behaves arbitrarily
  - E.g. pushing buttons repeatedly, not entering elevator, ...
- → ECS must be robust against demonic context i.e. demonic human behavior
- In an airplane, the pilot behavior can be constrained
  → a specification of pilot behavior finally is "implemented" via trainings, guidelines and handbooks.

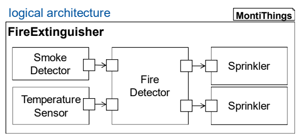


- Signature of the ECS:
  - btn1, btn2: buttons to request the elevator for a floor
  - light1, light2: indicators where the elevator will go to
  - at1, at2: signals that elevator has arrived
  - open, close actuators for the door

Logical and Technical Architecture e.g. in the Internet of Things

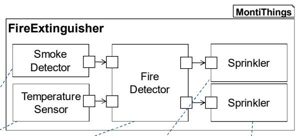
Internet of Things describes the network of physical objects that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data.

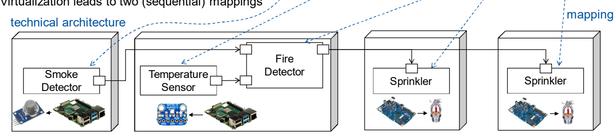
- Logical architecture:
  - components are logical computation units, independent of their later physical devices (threads / processes / processors / computers / clouds)
  - connectors are independent of the actual communication form (network, encoding, security, …)



Logical and Technical Architecture in the Internet of Things - 2

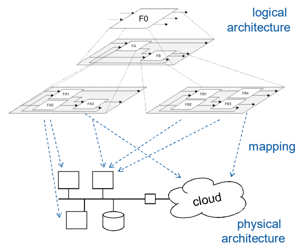
- Technical architecture
  - components are physical devices, CPUs, …
  - connectors are actual communication channels (Ethernet, Can-Bus, encoding, …)
- Mapping between logical and technical architecture
  - various criteria …
  - redundancy, robustness, load balance, security, …
- Virtualization leads to two (sequential) mappings





Logical and Technical Architecture

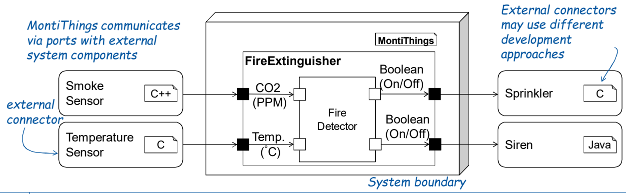
- Logical architecture:
  - components are logical computation units, independent of their later physical devices (threads / processes / processors / computers / clouds)
  - connectors are independent of the actual communication form (network, encoding, …)
- Technical architecture
  - components are physical devices, CPUs, …
  - connectors are actual communication channels (Ethernet, Can-Bus, encoding, …)
- Mapping between logical and technical architecture
  - various criteria …
  - redundancy, robustness, load balance, security, …
- When virtualization is used: mappings
  - (a) from logic to virtual architecture plus mapping
  - (b) from virtual to physical architecture plus mapping



Mapping Architectures into the Real World: Language & OS Barriers

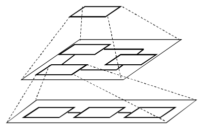
The MontiArc language has two generators:

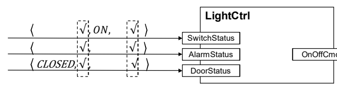
- MontiArc generates code to simulate distributed systems
- The MontiThings generator regards its components as software
  - maps models to code for execution on distributed systems
  - ports at system boundary act as connection to other systems, sensors, actuators



Summary Architectural Modelling

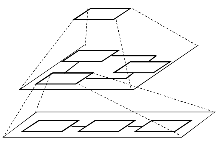
- MontiArc is a good example for modelling distributed systems
  - components as units of computation
  - interfaces of typed, directed ports
  - hierarchical decomposition
  - recursion loops
  - unidirectional connectors
  - explicit timing
- Underlying semantics:
  Focus: stream processing
- MontiArc also provides a simulation
  - flexible scheduling
  - extension with handwritten code
  - explicit simulation of time





MBSE

7. Architectural Design
7.6. Architecture and their Connection to other Models

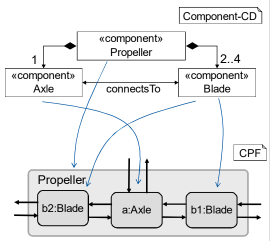


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

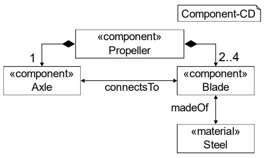
Physical Components as Class and as Function

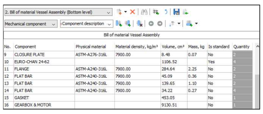
- A component realizes a function.
  - when modelling behavior, we attach functions to components
- We can use CDs to model the possible structure of CPF → stereotype «component»
  - A function decomposition must be compatible to a component decomposition to such a CD
  - Function composition connects instances
  - Functions include behavior (input and output), which is not defined in class diagrams
  - Class diagrams allow to model
    - a) several possible structures (here e.g. 2..4 blades)
    - b) possibly structural changes at runtime, (although functions have (mainly) a static structure)
  - Component-CD is also called "Meta-CD" and e.g. used in development tools



Bill of Material: BOM in Production

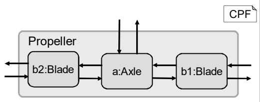
- In manufacturing and production:
- A bill of material defines the product structure
  - a list of the raw materials, sub-assemblies, parts, and the quantities of each needed to manufacture an end product.
- BOM is technically a subset of a class diagram
  - attributes describing properties and quantities
  - heavy use of composition
  - sometimes specialization (modelled by inheritance)
  - (but no associations)
- Variants of BOM (as defined by tools):
  - for design : engineering BOM,
  - as ordered : sales BOM,
  - as built : manufacturing BOM,
  - for maintenance : service BOM.
- Varying in detail, and whether they describe product structure or the raw materials to be transformed into the product.

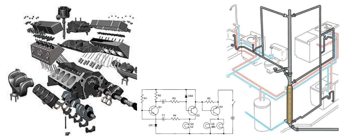




CFP Architectures in Maintenance, Construction, etc.

- CPF Architecture
  - for design : engineering architecture,
  - as built : manufacturing architecture,
  - for maintenance : service architecture.
- … for a concrete product is more detailed and thus potentially better suited to reveal failures, optimizations and other relevant aspects.
  - E.g. circuit diagrams for maintenance, exploded views, plumbing in buildings, etc.
- These are typically static diagrams of components and connections



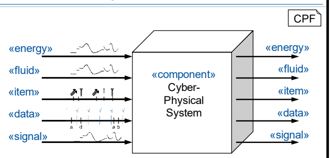


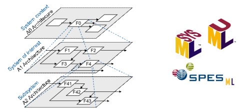
Literature

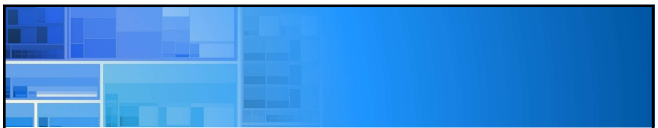
- https://en.wikipedia.org/wiki/Software_architecture
- https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns
- [MT00] Medvidovic , N. and Taylor, R.N.. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70-93. 2000.
- [MLM+13] Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., & Tang, A. What industry needs from architectural languages: A survey. IEEE Transactions on Software Engineering, 39(6), 869-891 2012. [HRR12] A. Haber, J. O. Ringert, B. Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. RWTH Aachen University, Technical Report. AIB-2012-03. February 2012.
- [RRW14a] J. O. Ringert, B. Rumpe, A. Wortmann: Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton. In: Aachener Informatik-Berichte, Software Engineering, Band 20. ISBN 978-3-8440-3120-1. Shaker Verlag, 2014.
- [KRW20] O. Kautz, B. Rumpe, A. Wortmann: Automated semantics-preserving parallel decomposition of finite component and connector architectures (Automated Software Engineering, Vol. 27, Apr. 2020)
- [Rin14] J. O. Ringert: In: Aachener Informatik-Berichte, Software Engineering, Band 19. ISBN 978-3-8440-3120-1. Shaker Verlag, 2014.
- [Hab16] A. Haber: In: Aachener Informatik-Berichte, Software Engineering, Band 24. ISBN 978-3-8440-4697-7. Shaker Verlag, Sept. 2016.

Function-based Universal Specification and Construction Principles

1. The function paradigm is the foundation
   - Clear boundaries, clear input/ouput signatures
2. Controlled, explicit underspecification
   - abstraction, variability, ability to describe the desired range of allowed behaviors
3. The concept of stream
   - as mathematically precise, time dependent model of input/output behavior
4. Composition / decomposition into hierarchies of function nets
5. Static dimensioning of parameterized functions
   - E.g. through simulations and optimization strategies
6. Adequate modelling techniques center around the function paradigm, e.g. SysML







MBSE

8. Specifying Constraints and Invariants with the OCL
8.1. Introduction to the Object Constraint Language

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

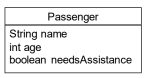
OCL

context Class inv:
invariant

context Method
pre: Precondition
post: Postcondition

OCL – Introductory Example

- Consider this class



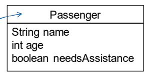
- Now apply the following constraints:
  - passengers are at least one year old
  - passengers older than 90 will automatically receive support

OCL

OCL – Introductory Example

CD

- Consider this class



context is
the class

- Now apply the following constraints:
  - passengers are at least one year old

OCL

```
context Passenger inv:
  age >= 1
```

invariant for the
attributes of the objects

  - passengers older than 90 will automatically receive support

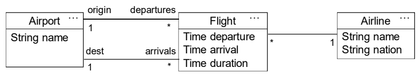
OCL

```
context Passenger inv:
  age >= 90 implies needsAssistance == true
```

operators from propositional logic
allows to combine expressions

OCL – Example 2 (#Landings)

CD

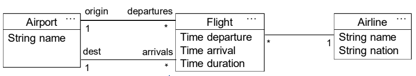


- Less than 300 arrivals at any airport:

OCL

OCL – Example 2 (#Landings)

CD



multiplicity 0..299 was an easy
alternative in this case

- Less than 300 arrivals at any airport:

explicit definition of object ap as context:
invariant applies **for all** objects of type Airport:

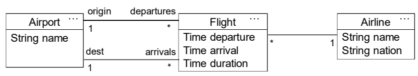
OCL

```
context Airport ap inv:
  ap.arrivals.size < 300
```

navigation along an association:
returns set of objects (type: Set<Flight>)

OCL – Example 3 (Schiphol)

CD

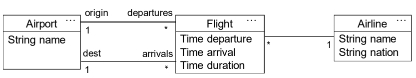


- All KLM flights start in Amsterdam (Schiphol):

OCL

OCL – Example 3 (Schiphol)

CD



- All KLM flights start in Amsterdam (Schiphol):

OCL

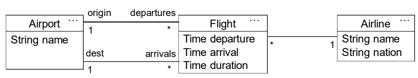
```
context Airline al inv:
  al.name == "KLM"  implies
    al.flight.origin.name == { "Schiphol" }
```

navigation along sequence of associations:
returns set of names (type: Set<String>)

set with only a single item

OCL – Example 4 (Start + Landing)

CD

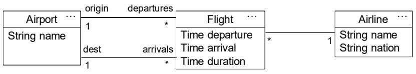


- All KLM flights start or land in Amsterdam (airport is called "Schiphol"):

OCL

OCL – Example 4 (Start + Landing)

CD



- All KLM flights start or land in Amsterdam (airport is called "Schiphol"):

OCL

```
context Airline al inv:
  al.name == "KLM"  implies
    forall fl in al.flight:
      fl.origin.name == "Schiphol" ||
      fl.dest.name   == "Schiphol"
```

quantifier over a set of flights

Object Constraint Language (OCL)

- OCL is a textual specification language
  - for properties that UML-diagrams do not cover
  - invariants, pre-/postconditions, guards, derived attributes
- OCL is similar to a First-Order Logic, but executable.
  - Boolean operators, quantifiers
- Basic data types
  - Boolean, Integer, Real, Char
  - sets and lists
- OCL is used in the context of UML diagrams
  - types and functions for OCL expressions are defined there
- In this lecture:
  - special version of the OCL that is aligned with Java

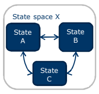
OCL

```
context Airport ap inv:
  ap.arrivals.size < 300

context Airline al inv:
  al.name == "KLM"  implies
  al.flight.origin.name == { "Schiphol" }

context Airline al inv:
  al.name == "KLM"  implies
    forall fl in al.flight:
      fl.origin.name == "Schiphol" ||
      fl.dest.name   == "Schiphol"
```


Modeling of Energy Efficient Buildings

- Model-based specification, analysis & energy optimization
- Methodology:
  - model + rule-based specification of technical facilities
  - automated data collection and -processing
- Example:
  - Checking adaptive heating circuits
  - Automated check of consumption data
  - Optimizing operations and correlation analysis

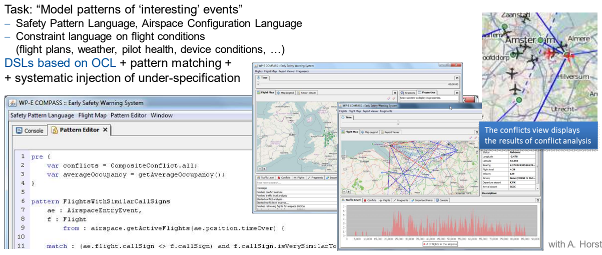
Sensor	Type	Comment	Unit
OT	double	Outside temp.	°C
RT	double	Room temp.	°C
OT < 6 implies RT > 13.0 and			
OT > 22 implies RT = 0.8 * OT			



Languages:
Facility modeling based on hierarchical function nets, OCL variant, Statecharts for condition monitoring

Sesar – Air Traffic Management (EU)

- Task: "Model patterns of 'interesting' events"
  - Safety Pattern Language, Airspace Configuration Language
  - Constraint language on flight conditions (flight plans, weather, pilot health, device conditions, …)
- DSLs based on OCL + pattern matching + + systematic injection of under-specification

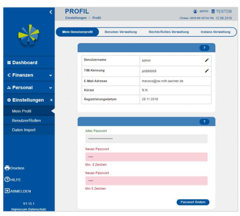


with A. Horst

Example: MontiGem Code Generator using OCL

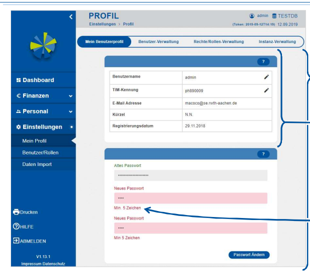
Rep.

- Multi-user web-application for data management
- Developed using MBSE and lots of code generation
  - Generate full application stack
- Starting point:
  - Class diagram modelling the application data
  - (+ some GUI models)
  - + Application functions



Screenshot of MaCoCo (Management Cockpit for Controlling), developed by AGe, PH, JM, LN, SVa, GV, and others

DSLs in MontiGem – MaCoCo



Data structure

CD4A

```
class User {
  String                    username;
  Optional<String> encodedPassword;
  ZonedDateTime             registrationDate;
  Optional<String> initials;
  String                    email;
  boolean                   authenticated;
  Optional<String> timID;
}
```

User interface

GUI-DSL

```
datatable "meinBenutzerInfoTabelle" {
  columns < it {
    row "Benutzername"          , <username (editable)
    row "TIM-Kennung"           , <tim (editable)
    row "E-Mail Adresse"        , <email
    row "Kürzel"                , <initials
    row "Registrierungsdatum"   , date(<registrationDate)
  }
}
```

Constraints

OCL/P

```
context User inv isPasswordValid:
  password.length() >= 5;
  shortError: "Min. 5 Zeichen";
  error: "Das Passwort muss aus mindestens 5
    Zeichen bestehen, hat aber nur " +
    passwort.length() + " Zeichen.";
```

Generator – OCL

OCL
- logic constraints + also error messages

- Generation
  - A) for the GUI: Validator methods, used in forms to immediately check input & give feedback, in JavaScript
  - B) for application server (backend): Validator methods, used to prevent erroneous inputs, in Java

Example model (shortened):

Domain.ocl
```
context User inv isPasswordValid:
  password.length() >= 5;
  shortError: "Min. 5 Zeichen";
  error: "Pwd zu kurz: " + password.length();
```

Application server:

UserValidator.java
```
public Result isPasswordValid(String password) {
  if (!(password.length() >= 5)) {
    return Result.error(
      "Pwd zu kurz: " + password.length()
    );
  }

  return Result.ok();
}
```

GUI frontend:

user.validator.ts
```
public static isPasswordValid(password : string | null): void {
  let constraintFailed = false;
  if (!((!(password !== null) || password.length >= 5))) {
    constraintFailed = true;
  }
  if (constraintFailed) {
    throw new ValidationError("Min. 5 Zeichen");
  }
}
```


MBSE

8. Specifying Constraints and Invariants with the OCL
8.2. Overview

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

OCL
```
context Class inv:
      invariant

context Method
   pre:  Precondition
   post: Postcondition
```

Concepts of OCL - 1

- Condition:
  - a condition is a logic formula about a system.
    It describes a property that a system or a result should have.
  - its results in a Boolean value, i.e. true or false.

- Consequences for their use in code and simulations:
  - a condition evaluation does not crash
    - Example: 1/0 == 7 has the Boolean value "false".
  - a condition is side-effect free
    - only result is the calculated value
  - invariants are only partially computable (details later)

- Invariant:
  - describes a property, that holds at each (observed) point of time.
  - observation points of time can be restricted.
  - temporary violations are permitted, e.g., while executing a method.
    - Example: invariant a==2*b is violated within method bodies like { a++; b=b+2 }

- Result:
  - invariants apply especially when the objects are "idle" and no methods operate on the objects.

Concepts of OCL - 2

- Context of a condition:
  - a condition is embedded in a context, that it constrains
  - context is defined by
    - one or several variable names and their
    - signatures, that can be used in the condition.
  - Context typically denotes
    - objects of given classes (like ap), which then allows access to their methods and attributes, or
    - methods of classes: describing the behavior of a method

- Conditions are usually meant to constrain this "context", i.e. the underlying data structures (classes) or method behaviors

OCL
```
context Airport ap inv:
   ap.arrivals.size < 300
```

- Evaluation of a condition is always based on a concrete object structure.
  - Evaluator assigns values / objects to the variables that are introduced in the context.

Context

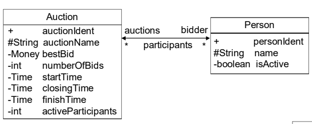
- Context defines variables a,b:

```
context Auction a,b inv:
  a.startTime < b.startTime  implies
  a.closingTime < b.closingTime
```

- Name for a condition:

```
context Auction a inv Bidders1:
  a.activeParticipants <= a.bidder.size
```

CD
...



OCL

Context without Explicit Names

- Context defines implicit new variable this:

```
context Auction inv:
  this.startTime.lessThan(this.closingTime)
```

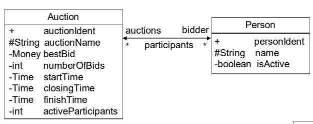
- equivalent to:

```
context Auction a inv:
  a.startTime.lessThan(a.closingTime)
```

- shortened form omitting "this" (like in Java):

```
context Auction inv:
  startTime.lessThan(closingTime)
```

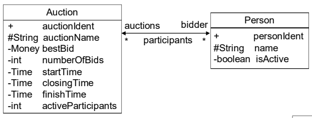
CD
...



OCL

Primitive Data Types and Collections

- As known from Java
  - **boolean, char, int, long, float, byte, short, double**
  - Corresponding operations are available (+,-, *..)
    - forbidden are --, ++ etc. because of side effects
- **String** is not a primitive data type, but a normal class
- Collection structures for sets, lists,…
  - **Set<int>, List<String>, Optional<.>, ...**
  - With special syntax assistance
- Systems Engineering also uses
  - physical types, such as **mol/m^2** and
  - physical expressions **3,3 km/h * 2,7 sek**

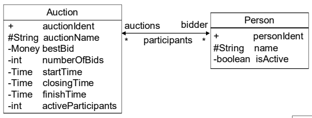


OCL

```
context Auction a inv:
  2 + 3,5 * a.numberOfBids > 1 - foo("text")
```


Set Comprehension

- Sets are similar to those in mathematics (like in Haskell)
- E.g.: the number of active participants is correct:



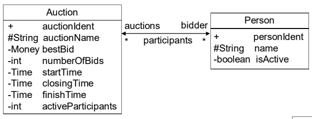
OCL

```
context Auction a inv:
    a.activeParticipants == { p in a.bidder | p.isActive }.size
```

p is in the set of bidders p introduced with the scope of the sets comprehension

characteristic of p: selects a subset

Local Variables in OCL: let-Construct



- Local variables for convenience:

OCL

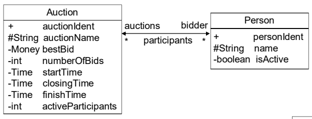
```
context Auction a inv:
    let min = startTime.lessThan(closingTime) ? startTime : closingTime
    in
      min == startTime
```

min is introduced as variable here

A ? B : C = If A Then B Else C

and can be used in the body

Local Operations in OCL: let-Construct



- Local operation:

OCL

```
context Auction a inv:
    let min(Time x, Time y) = x.lessThan(y) ? x : y
    in
      min(a.startTime, min(a.closingTime,a.finishTime)) == a.startTime
```

min is defined as operation with arguments here

Case Distinctions

- Case distinctions always evaluate to values (OCL has no statements)
- Variants:
  - `if condition then expression1 else expression2`
  - `condition ? expression1 : expression2`
  - `typeif variable instanceof type then expression1 else expression2`
- **typeif** is a type-safe version of the typecast for variables

OCL

```
context Supertype m inv:
  typeif m instanceof Subtype then (m known here as Subtype)
                              else (m here as only Supertype)
```


Appendix: List of OCL-Operations, Part 1

Priority	Operator	Associativity	Operands, Semantics
14	.@pre	left	value of the expression in precondition
	.**	left	transitive closure of an association
13	+., -., ~.	right	numbers
	!.	right	Boolean: negation
	(type).	right	type conversion (cast)
12	.*., ./., .%.	left	numbers
11	.+., .-.	left	numbers, string (+)
10	.<<., .>>., .>>>.	left	shifts
9	.<., .<=., .>., .>=.	left	comparisons
	.instanceof.	left	type comparison
	.in.	left	element of

Appendix: List of OCL-Operations, Part 2

Priority	Operator	Associativity	Operands, Semantics
8	.==., .!=.	left	comparisons
7	.&.	left	numbers, Boolean: strict and
6	.\.	left	numbers, Boolean: xor
5	.|.	left	numbers, Boolean: strict or
4	.&&.	left	Boolean logic: and
3	.||.	left	Boolean logic : or
2.7	.implies.	left	Boolean logic : implicit
2.3	.<=>.	left	Boolean logic : equivalent
2	. ? . : .	right	expression of choice (if-then-else)

MBSE

8. Specifying Constraints and Invariants with the OCL
8.3. Logic of the OCL

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

OCL
context Class inv:
invariant

context Method
pre: Precondition
post: Postcondition

Logic in OCL

- Boolean expressions about attributes and associations are combined with
  - logical operators: and, &&, or, ||, equivalence, <=>, implies, not
  - quantifiers: exists, forall
  - comparison: ==
- Boolean constraints are binary, either **true** or **false**
- In an implementation undefined values can occur
  - programs crash
  - no termination, e.g., infinite loop
  - invalid value (reference does not exist, enumeration out-of-range)
- Introduction of a pseudo value “undef” that is used only to explain the semantics

Binary vs. Three-valued Logic: Example Conjunction

- Binary logic uses true and false only:
- E.g. the truth-table for the and-operator:

A && B	true	false
true	true	f
false	f	f

- There are a number of laws

OCL

  - associativity (a && b) && c <=> a && (b && c)
  - commutativity a && b <=> b && a
  - involution a && a <=> a

- Truth-table for a three-valued conjunction:

A && B	true	false	undef
true	true	f	?
false	f	f	?
undef	?	?	?

- **undef** is a mathematical “pseudo value”
  - doesn't exist in the OCL language itself
  - Is used to describe semantics
- Q: Which laws can still hold?
  - Goal: all of them!

Variants of Three-valued Logic:

- (1) Strict Evaluation (Pascal, strict “&” in Java)
  + evaluation order doesn't matter because laws hold
  - always both arguments to evaluate
  - inconvenient for a logic, because of three cases
- (2) Sequential Execution (“&&” in C, C++, Java, ...)
  + easy to implement
  + efficient: if left is false, right will not be evaluated
  - not commutative (and thus not optimizable)
- (3) Kleene Logic (unusual in programming)
  + Boolean laws apply: associative, commutative, ...
  - both arguments need to be evaluated in parallel
- (4) Lifting of Undef (verification tool Isabelle)
  + simple laws and proofs
  + easy to formulate properties
  - not fully evaluatable

A && B	true	false	undef
true	true	f	undef
false	f	f	(1) undef (2),(3) false
undef	undef	(1),(2) undef (3) false	undef

A && B	true	false	undef
true	true	f	f
false	f	f	f
undef	f	f	f

Variant (4): undef and false in the lifted logic are treated as identical: so just binary logic!

Binary Semantics and Lifting

- Distinction of terms with Boolean values with three results and logic expressions with two only values
- Developers write ordinary terms, like “b==1/0”
- Implicit lifting of the “undef” value to “false” by an operator λ with
  - λ **true** == **true**
  - λ **false** == **false**
  - λ **undef** == **false**
- Logic expressions implicitly use the lifter λ:
  - λ(a==5) implies λ(isOpen())
  - λ(b==1/0)
- Lifter λ can be implicitly added in the OCL
- Challenge: λ **undef** == **false** cannot be implemented
- Practice in Java shows “undef” mostly occurs by
  1. abnormal error (exception)
  2. infinite recursion
  3. infinite loops occurs rather rarely
- Case 1&2 result in exceptions (e.g., stack overflow). So lifting of an expression $x is partially implementable:

Java

```java
boolean res;
try {
  res = $x;     // evaluate expression $x
} catch(Exception e){
  res = false;
}
```

Special Operator “defined”:

- In rare cases it is helpful to talk about the definedness of a value resp. an expression.
- Special operator
  - `defined( ... )`
- allows to clarify the definedness of a value
- For example, it holds:
  - `inv:`
    `!defined(1/0)`
- Operator **`defined`** is not fully computable, but with tricks similar to the lifter λ it can be used e.g. in tests and simulations

OCL
```
context Auction a inv:

  let Message mess = a.message[0]
  in
    defined(mess.foo()) implies ...
```

OCL
```
inv:
  let int a = 3,
      int b = 0
  in
     defined(a/b) ? a/b : a+b
```


Comparisons using ==

- Operators that may return defined values for undefined arguments can be called “non-strict”
- Boolean operators, case distinction, and the let-construct are not strict:
  - if true then a else undef equiv.: a
  - let a=1/0 in 3+7 equiv.: 3+7
- The comparative operator == (as well as != and equals()) are strict according to convention:
  - (undef == undef) equiv.: undef
- Please note that
  - undef == undef equiv.: λ(undef == undef) equiv.: false
  - λ(undef) == λ(undef) equiv.: true
  - undef <=> undef equiv.: λ(undef) == λ(undef) equiv.: true
  - i.e. <=> uses lifting inside arguments, while == uses lifting outside

MBSE

8. Specifying Constraints and Invariants with the OCL
8.4. Collections in the OCL

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

OCL
```
context Class inv:
                invariant

context Method
    pre:  Precondition
    post: Postcondition
```

Collections in OCL

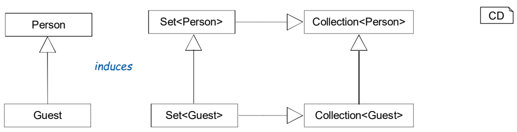
- Particularly important for navigation along associations
- The OCL uses generic types, like Java
- Set<X> represents sets of type X
- List<X> represents lists:
  - elements of index 0..(length-1) accessible
  - multiple occurrences possible
- Collection<X> is super type of Set<X> and List<X>
  - common interface of the two collections
- Optional<X> describes possible absence of an element
- Nesting is possible, e.g. Set<List<X>>
- Example expressions
  - `Set{}, Set{ 2,3,5 }, Set{ "text", "part" }`
  - `{}, { 2,3,5 }, {2}, {{2}}`
  - `[2,3,3],  List{2,3,3}`
  - `Person` *// in OCL a class name represents its extension;*
    // i.e., the set of all currently existing objects
- All datatypes have operators, such as: add, first, last, … similar to the Set, List, Optional types from Java
- additional forms of expressions, such as
  - Set comprehension { … | … }
  - Elvis operators for optionals . ? .
  - Quantifiers forall, exists

Collections: Nesting, Subtype Hierarchy

- Collection types can be nested:

OCL
```
inv:
  let Set<int>       si  = { 1, 3, 5 };
      Set<Set<int>>  ssi = { {}, {1}, {1, 2}, si };
      List<Set<int>> lsi = List{ {1}, si, {}, si }
  in ...
```

- Subtype hierarchy applies for collection and element types (as opposed to Java):



Comparison for Collections

- Comparison == for collections requires comparison of the elements:
  - == is used for primitive data types (int, km/h, …)
  - equals() for object types (Person, String, …)
- Collections themselves do not have an “object identity” in OCL
  - A==B compares contents of both collections elementwise
- Care: implementations sometimes redefine method equals()
- Comparison of lists is implemented analogously
- If X is a primitive data type or a collection, we have for Set<X>:

OCL
```
context Set<X> sa, Set<X> sb inv:
  sa==sb <=>
   (forall a in sa: exists b in sb: a==b)
   &&
   (forall b in sb: exists a in sa: a==b)
```

- For object type X it holds:

OCL
```
context Set<X> sa, Set<X> sb inv:
  sa==sb <=>
   (forall a in sa: exists b in sb: a.equals(b))
   &&
   (forall b in sb: exists a in sa: a.equals(b))
```

Set and List Comprehension

OCL

- Abbreviation for collections of integers and characters
  - `Set{'a'..'c'}    == {'a', 'b', 'c'}`
  - `List{-1..1,3..7,14} ==`
    `List{-1,0,1,  3,4,5,6,7,  14}`
- General form of comprehensions with $expr and $description as placeholder for appropriate terms
  - `{ $expr | $description }`
  - `List{ $expr | $description }`
- Comprehension forms `$description`
  - 1: the Generator v in *List/Set*
    new variable v, which iterates over the list
  - 2: the Filter: a Boolean *condition*
    becomes powerful through combination with generator
  - 3: Auxiliary result introduces local variable: *v = expr*
- Examples for the Generator
  - `List{ x*x | x in List{1..6} } ==`
    `List{1,4,9,16,25,36}`
  - `List{ m.time.asMsec() |`
    `Message m in a.message }`
- Filter:
  - `List{ x*x | x in List{1..8}, !even(x) } ==`
    `List{1,9,25,49}`
- Auxiliary result in y:
  - `List{ y | x in List{1..8},`
    `int y = x*x,  !even(y) } ==`
    `List{1,9,25,49}`
- Comprehension elements can rely on previously defined elements

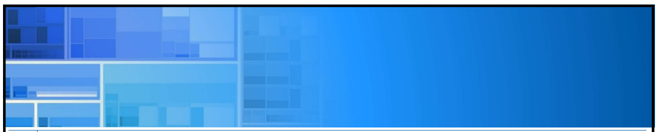
Comprehension: Excercise

- Quite good expressiveness due to combination of the three forms
  - unfortunately, the UML 2 standard does not offer these forms
  - these have been borrowed from functional languages (Gofer, Haskell)

OCL

```
List{ z+"?" | x  in  List{"Spiel", "Feuer", "Flug"},
              y  in  List{"zeug", "platz"},
              String  z = x+y,
              z  !=  "Feuerplatz"                }
  ==
```

write



MBSE

8. Specifying Constraints and Invariants with the OCL
8.5. Associations in the OCL

OCL

context Class inv:
invariant

context Method
pre: Precondition
post: Postcondition

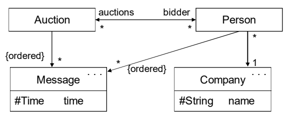
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Navigation along Collections and Associations in OCL

CD

Lets try:
- 1) Set of bidders of auction a:
- 2) Set of names of companies involved in a:
-
- 3) Set of messages from auction a:
- 4) Set of persons of a company c:
- 5) Set of auctions in which the company c is involved:



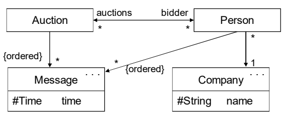
OCL

write

Flattening Operator in Navigation

CD

- Set of auctions in which company c is involved:
  - `c.persons.auctions`
  - `c` has type Company
  - `c.person` has type Set<Person>
  - `c.person.auctions`
    would now have type Set<Set<Auction>>
    but is flattened and has Set<Auction>
- Automatic flattening removes a level of hierarchy:
  - operator "flatten" is inserted implicitly by navigation, wherever a collection is available as output



OCL

```
c.person.auctions ==
      { p.auctions | p in c.person }.flatten
```


Qualified Navigation

CD

- Normal navigation: `ad.auction` evaluates to `Set<Auction>`
- Qualified navigation: `ad.auction[ident]` evaluates to `Auction`
- Examples:

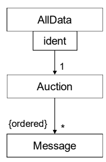
OCL

```
context AllData ad, Auction a inv:
  ad.auction[a.auctionIdent] == a &&
  ad.auction[a.auctionIdent] in ad.auction;
```

- {ordered}-associations have natural numbers from 0 as index.

OCL

```
a.message[1].content != a.message[0].content
```



Quantifiers

- `forall` and `exists` are the quantifiers known from mathematics
- Context definition behaves like a universal quantifier. Logically equivalent are:
  - `context $Class $var inv:     $P($var);`
  - `inv: forall $var in $Class: $P($var);`
- Computability: Quantifiers belong to first-order logic (FOL) and may have infinite "quantification space"
- But: object-valued quantifiers in OCL are interpreted on the sets of the currently-existing objects.
  - these sets are finite: hence the OCL quantifiers are "computable".

- Examples:

```
forall a in Auction, m in a.message:
                   a.startTime <= m.time;

exists a in p.auctions: a.category == "Clock"

inv:
  exists int a, b, c, n: n>2 && a^n == b^n + c^n
```
OCL

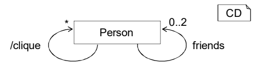
- Some properties:

```
(forall x in Set{}: false) <=> true

(exists $var in $collExpr: $expr) <=>
         !(forall $var in 4collExpr: !$expr)
```
OCL

Transitive Closure of Associations

- OCL is a subset of first-order logic and thus cannot describe induction of natural numbers or transitive closure of a recursive association.
- `TClosureTry` tries to describe the derived association that `clique` represents the transitive closure of `friends`, but fails -- why?

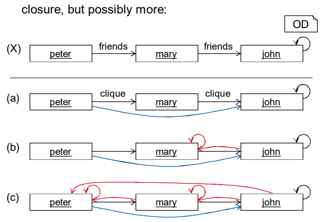
- Example:

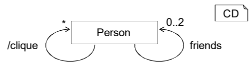



```
context Person inv TClosureTry:
  clique ==
      friends.addAll(friends.clique)
```
OCL

Transitive Closure of Associations

- `TClosureTry` is a logic formula with several solutions: each solution contains the transitive closure, but possibly more:





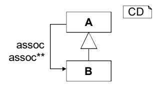

```
context Person inv TClosureTry:
  clique ==
      friends.addAll(friends.clique)
```
OCL

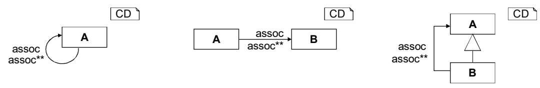
- Based on the OD in (X) at least three solutions occur:
  - (a) the intended transitive closure
  - (b) another transitive solution
  - (c) a 'maximal' solution
  - … and there are some more

Transitive Closure of Associations - 3

- Math has proven: FOL cannot specify transitive closure
- Solution: use a predefined explicit operator **, which builds the transitive closure of an association
  - `friends**`
- The typing of the transitive closure is exactly like that of the underlying association (in all 4 cases).

```
context Person inv TClosure:
  clique == friends**
```
OCL

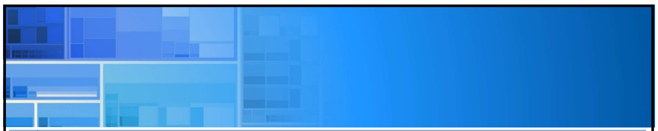




Object Constraint Language (OCL) Summary

- OCL is a textual specification language
  - for properties that UML-diagrams do not cover
  - invariants, guards, derived attributes
- OCL is similar to a First-Order Logic, but executable.
  - Boolean operators, quantifiers
- Basic data types
  - Boolean, Integer, Real, Char
  - sets and lists
- OCL is used in the context of UML diagrams
  - types and functions for OCL expressions are defined there
- OCL provides good navigations along CD associations

```
context Airport ap inv:
  ap.arrivals.size < 300

context Airline al inv:
  al.name == "KLM" implies
  al.flight.origin.name == ( "Schiphol" )

context Airline al inv:
  al.name == "KLM"  implies
    forall fl in al.flight:
      fl.origin.name == "Schiphol" ||
      fl.dest.name   == "Schiphol"
```
OCL



MBSE

9. Specifying Behavior with the OCL
9.1. Queries

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

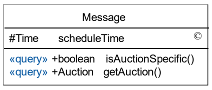
OCL
context Class inv:
    invariant

context Method
  pre: Precondition
  post: Postcondition

Queries

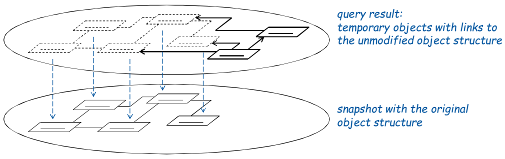
CD

- A query is a method of the underlying object model
- A query is free of side effects
  - attributes may not be changed!
- In CDs: use stereotype «query»
- Java code: undecidable, what is a query, but common style to begin a query with `get`, `is` or `has`
- The stereotype «query» is a promise to the users and a commitment for the developer:
  - queries may only call other queries
- Implementation in Java:
  a) pragmatic: "hoping" on absence of side effects
  b) conservative: analyze the methods for query property
  Also use the try-catch approach for lifting when using Java methods in the OCL constraints



Queries and Object Creation

- A query may create new objects and manipulate them
  - example: building a collection as the result of a query
- Old object structure has no knowledge of (links to) the new objects :

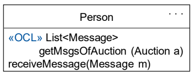


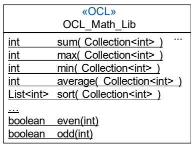
- Checking whether a method is a query requires a data flow analysis

Library of «OCL» Methods

CD

- The definition of reusable queries for OCL is often useful
  - queries are part of the underlying object model
  - OCL does not need the definition of methods (except in let-constructs)
- Methods marked with the stereotype «OCL» are like queries, but only used for specification and simulation, not part of the product code
  - they can only be used in OCL constraints
- Useful for a library of «OCL» methods
  - Like math functions, collection operations, etc.





these (static) methods: can be used in OCL when the library is imported. Example:

```
min(Auction.bidder.age) >= 18
```
OCL

MBSE

9. Specifying Behavior with the OCL
9.2. Function Specifications: Methods

OCL

context Class inv:
invariant

context Method
pre: Precondition
post: Postcondition

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Method Specification

- OCL for the specification of the effect of a method:
  - precondition describes what must be considered for the method to work correctly
  - postcondition describes the effect of the method
- Meaning:
  - "If the caller fulfills a condition, then the called method fulfills the postcondition"
  - in short: "Pre implies Post"
- Precondition is an obligation to the caller
- Postcondition is a implementation requirement
- See also Bertrand Meyer: Eiffel programming language
  - contract (contract between caller and environment)

context

OCL

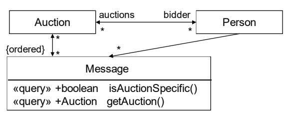
```
context boolean BidMessage.isAuctionSpecific()
  pre:  true
  post: result == true
```

Means: no restrictions on the pre-state at the time of call

Means: result is simply always the value true in subclass Bidmessage

- Context now is a method defined by method signature (incl. its class)
- `result` is a special variable available in the postcondition, denoting the methods result

Example: Method getAuction()



- Example:
  The method Message.getAuction() returns the associated auction of a message:
  - `context Auction Message.getAuction()`

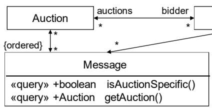
OCL

```
pre:
post:
```

write

Example: Method getAuction()

CD



- Example:
  The method Message.getAuction() returns the associated auction of a message:
  - **context Auction Message.getAuction()**

OCL

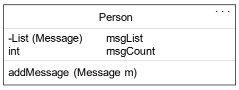
```
pre:  isAuctionSpecific()
post: this in result.message
```

queries can also be used here

"this" refers to the object that belongs to the method

@pre-Operator: Attribute Modifications

CD



- **addMessage** adds a new message, the timestamp of which must be newer than the last one:
  - **context Person.addMessage(Message m)**

OCL

```
pre:  (msgList.isEmpty || m.time > msgList.last.time)
         && !(m in msgList)
post: msgList == msgList@pre.add(m)
         && msgCount - msgCount@pre == 1
```

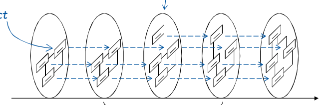
attribute@pre allows to access the state at method invocation time

Semantics of a Method Specification

- (An invariant is created based on one object structure (snapshot))
- A method specification uses two snapshots:
  - start snapshot for the precondition and
  - end and start snapshots for the postcondition:

a "snapshot" describes an object structure at a certain moment of the runtime of the system

one object



timeline

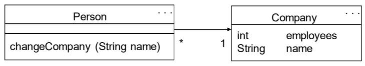
method call

start of the method call: this is the start snapshot

end snapshot for the method call: here the postcondition applies (in relation to the start snapshot)

Example: Complex, Composed Specifications

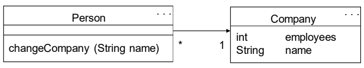
CD



- changeCompany() allows a person to change the company
  - if necessary, a new company is created
  - number of employees in the old and new companies will be changed
- This is a relatively complex situation, so we divide the specification into several cases
  - 1) new company already exists
  - 2) new company does not yet exist

Complex Specifications – Case 1

CD



- Case 1: new company object already exists
- Constraint: new company != old company
  - **context Person.changeCompany(String n)**

OCL

```
pre CC1pre:   company.name != n && exists Company co: co.name == n
post CC1post: company.name == n &&
              company.employees     == company.employees@pre +1 &&
              company@pre.employees == company@pre.employees@pre -1
```

pre-/postcondition with names

old company, old nr. of employees

Complex Specifications – Case 2

CD

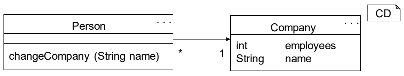
- Case 2: company object does not exist yet
  - **context Person.changeCompany(String n)**

OCL

```
pre  CC2pre:  !exists Company co: co.name == n
post CC2post:
```

write

Complex Specifications – Case 2



- Case 2: company object does not exist yet
  - `context Person.changeCompany(String n)`

```
pre  CC2pre:  !exists Company co: co.name == n
post CC2post: company.name == n &&
              company.employees == 1 &&
              company@pre.employees == company@pre.employees@pre -1 &&
              isnew(company)
```

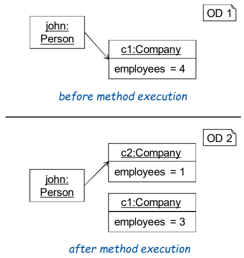
OCL

operator isnew(.) describes that an object was created

In Detail: @pre in Postconditions

- Example situation is illustrated by two object diagrams OD 1 and 2
- John moves from c1 to newly created c2.
- We evaluate the following expressions in the postcondition:

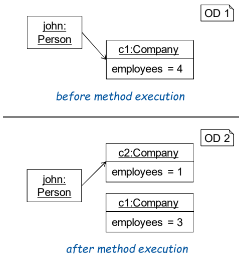
```
1 john.company.employees             ==
  –
2 john@pre.company.employees         ==
  –
3 john.company@pre.employees         ==
  –
4 john.company@pre.employees@pre     ==
  –
5 john.company.employees@pre         ==
```



In Detail: @pre in Postconditions

- Example situation is illustrated by two object diagrams OD 1 and 2
- John moves from c1 to newly created c2.
- We evaluate the following expressions in the postcondition:

```
1 john.company.employees             == 1
```
  – fully evaluated in the state after the method call
```
2 john@pre.company.employees         == 1
```
  – reference to the object john does not change: john==john@pre
```
3 john.company@pre.employees         == 3
```
  – company@pre is c1; c1.employees uses the current state of c1
```
4 john.company@pre.employees@pre     == 4
```
  – accesses original object c1 in the original condition
```
5 john.company.employees@pre         == undefined
```



Composition of Methods Specifications

- If the precondition is false, there are multiple interpretations:
  a) a program should detect errors and stop.
  b) a program should notice errors in the log and continue as robust as possible.
  c) the specification perspective:
     Nothing is stated.
     In particular, the postcondition need not be fullfilled.
- Case a) , b) can be used for defensive resp. robust programming
- Case c) is ideal for specifications: It allows composition of partial specifications
  - if one of the two conditions is true, the corresponding postcondition must hold
  - if both preconditions hold, also both postconditions.

OCL

```
context method()
 pre:    Apre
 post:   Apost
```

```
context method()
 pre:    Bpre
 post:   Bpost
```

composition
(in case c))

OCL

```
context method()
pre:     Apre || Bpre
post:    (Apre` implies Apost)   &&
         (Bpre` implies Bpost)
```

- `Apre`` is modified: Attributes, like var, become var@pre in the postcondition.

The Underspecification Principle with OCL

- Underspecification is the ability to describe the desired range of allowed behaviors (instead of a single, determined behavior)
- Advantages:
  - Easier to specify
  - Can be well combined with variant-building and methodical refinement
- OCL postconditions are a perfect way to underspecify
  - explicit choice of solutions
  - range of possibilities
    - e.g. because of (yet) unknown requirements
  - approximated results

OCL

```
context int random(int x)
pre:     x >= 1
post:    1 <= result && result <= x
```

OCL

```
context int choice(int x, int y)
pre:     true
post:    x == result || y == result
```

OCL

```
context float solver(Function f)
pre:     exists x: f(x)==0
post:    -0.0001 < f(result) < 0.0001
```

⇨ OCL assists controlled, explicit underspecification as specification principle

The Underspecification Principle in OCL: Incomplete Characterization

- In general, a method specification can be incomplete
- It focuses on the essential functionality and leaves open further details to the programmers
  - principle of angelic programmers/developer
- A demonic developer could obey the spec and still
  - change silently other objects or attributes
  - create/delete other objects
- For more precise restrictions, there are so-called “frame-rules”:
  - only the explicitly mentioned attributes and objects may be modified
  - implicit assumption: all others remain unchanged and are only adjusted when explicit invariants enforce this

OCL

```
context Person.addMessage(Message m)
pre:   ( msgList.isEmpty ||
         m.time > msgList.last.time) &&
       !(m in msgList)
post: msgList == (msgList@pre).add(m)  &&
       msgCount == msgCount@pre +1
```

Code Generation from OCL

- Many constructs of the OCL are implementable
  - use of the try-catch construct for errors
  - collections can be mapped to Java
  - quantifiers can be implemented with (slow) iterators
- Invariants / pre- / postconditions can be evaluated and thus used in tests
  - explicitly specify, where to evaluate: like Java asserts.
  - efficiency considerations: evaluate invariant on object changes only
    - infrastructure needed to observe object changes
- From many/some postconditions constructive code can be generated: But, not always and not always unambiguous.

OCL
```
context Person.incAge()
pre:  true
post: age == age@pre +1
```

constructive solution (when found)

Java
```
class Person {
  void incAge() {
    assert true; // precondition
    age = age+1; // postcondition
  }}
```

for testing manual implementation of incAge()

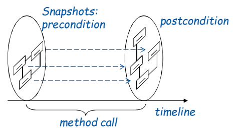
Java
```
class PersonSub extends Person {
  void incAge() {
    assert true; // precondition
    agePre=age;  // store old values
    super.incAge();  // call the real method
                    // postcondition
    assert age == agePre+1;
  }
}
```


Summary Method Specification

- OCL does not have method definitions but uses the underlying object system.
- OCL uses contracts, i.e.
  - preconditions and
  - postconditions

  to specify methods
- Methods belong to an underlying OO model
  - e.g. in the product code
  - in tests
  - in simulations of physical objects
- OCL specifications can also be used for behavior specs of physical objects (or humans)
  - Deficits:
    - no continuous behavior, but discrete
    - Builds on method calls, and not on message passing

OCL
```
context Person.incAge()
pre:  true
post: age == age@pre +1
```



MBSE

9. Specifying Behavior with the OCL
9.3. Specifications of Streams Processing Functions

OCL
```
context Class inv:
              invariant

context Method
    pre:  Precondition
    post: Postcondition
```

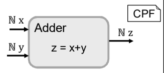
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Revisited Example: Simple Adder

- The Adder component has as signature:
  - $\mathbb{N}\,x \times \mathbb{N}\,y \rightarrow \mathbb{N}\,z$
- actually over time it processes streams of inputs:
  - stream<ℕ> x × stream<ℕ> y → stream<ℕ> z
- A choice could be to implement it via repeated call of method "doAdd" (written in OO style):
  - int doAdd (int x , int y)
- Spec omits:
  - Timing details
  - Absence of values (or waiting on values) on x or y inputs
- Care: "delivery" of incoming values to the appropriate method calls is often a schematic task based on explicit scheduling

CPF



OCL
```
context Adder.doAdd(int x, int y)
pre:  true
post: result = x + y
```

standard OCL: used a method concept

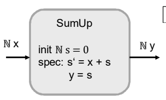
SpesML
```
spec Adder
  port in int x, int y;
  port out int z;
--------------------------------
pre:  true
post: z = x + y
```

SpesML is a currently developed function oriented specification language

Revisited Example: SumUp (with State)

- Building sum of arriving numbers:
  - We use an internal variable ℕ s
  - and as specification this invariant: $s' = x + s \;\wedge\; y = s$
- A choice could be to implement it via method "doIt" with the same signature (written in OO style):
  - ℕ doAdd (ℕ x) resp.
  - int doAdd (int x)
- State s is stored as attribute in the respective class "SumUp"
- Again: SpesML style is compact (no CD needed)
  - and SpesML can also handle several output channels

CPF



CD

SumUp
int s = 0;

OCL
```
context SumUp.doIt(int x)
pre:  true
post: s = x+s@pre && result = s@pre
```

SpesML
```
spec SumUp
  port in int x;
  port out int y;
  state int s = 0;
--------------------------------
pre:  true
post: s = x+s@pre  &&  y = s@pre
```


Revisited Example: R-S-Flip-Flop

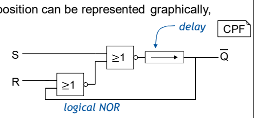
- R-S-Flip-Flop
  - is composed of two NOR and a Delay
  - includes a feedback loop:
  - this allows to store a state (a bit)

SpesML
```
spec NOR
  port in boolean x, y;
  port out boolean z;
------------------------
post: !x && !y  <=>  z
```

SpesML
```
spec Delay<T>
  port in T x;
  port out T z;
  state T buffer;
------------------------
post: buffer = x  &&  z = buffer@pre
```

- The composition can be represented graphically,

CPF

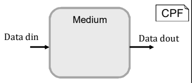


- but equivalently as textual (composition) formula:

SpesML
```
spec RSFlipFlop
  port in boolean R, S;
  port out boolean Q;
------------------------
  !Q = Delay<boolean> (
        NOR(S, NOR(R,Q)) )
```

Revisited Example: Underspecified Communication Medium

- Medium describes an unreliable communication device:
  - It may transport a signal (data) or may drop it
  - This behavior is nondeterministic in nature.
  - For simplicity: Medium does not replicate, alter or delay data, nor does it switch the order of data
  - Specification dout = din $\vee$ dout = ε
- Remarks:
  - The specification is based on a "current" snapshot behavior and cannot specify e.g. 99% success rate



SpesML

```
spec Medium<Data>
  port in Data din;
  port out Data dout;
-----------------------
  dout = din  ||  dout = epsilon
```


Summary Function Specification with SpesML

- SpesML is a logic derived from OCL
  - it specifies behavior of cyberphysical functions
  - with discrete behavior (i.e. message passing over channels)
- SpesML builds on the OCL logic and its contracts, i.e.
  - preconditions and
  - postconditions

  to constrain the inputs and relate them to outputs
- SpesML specifications are dedicated for behavior specs of software components, physical objects (or humans)
  - Deficits:
    - no continuous behavior, but discrete

SpesML

```
spec Medium<Data>
  port in Data din;
  port out Data dout;
-----------------------
  dout=din || dout=epsilon
```

SpesML

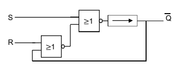
```
spec SumUp
  port in int x;
  port out int z;
  state int s = 0;
-------------------------------
pre:  true
post: s = x + s@pre  &&  y = s
```

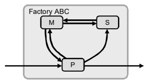

A Note On Recursive Definitions / Stateful Behavior

- Traditionally calculus is used to describe physical behavior (continuously)
  - Derivations condense history into infinitely small time frames
- Digital theory uses discrete changes of states, events, signaling
  - Transitions with instant changes based on state
- Recursion (i.e. a system continues its behavior based on its past) leads to significant differences in:
  - the used forms of "solving"
  - possible "executions"
  - numerical or similar simulations
  - Digital processes based on fixpoint theory
  - Continuous processes based on calculus

$$a = \frac{\delta v}{\delta t}$$

CPF



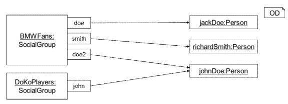


MBSE

10. Modeling Instance Structures with Object Diagrams
10.1. Introduction

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

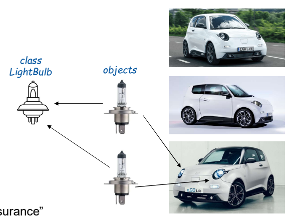
http://www.se-rwth.de/



Objects in the Physical World (Systems Engineering)

Rep.

- A system consists of a dynamically changing number of physical objects
- Objects represent entities of the domain and instances of exactly one class
- An object can be uniquely identified
- An object has a state as defined by its properties
  - result of an operation depends on the current state
- An object has a behavior modelled by the functions of its class

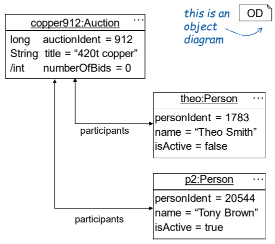


Kinds of objects:

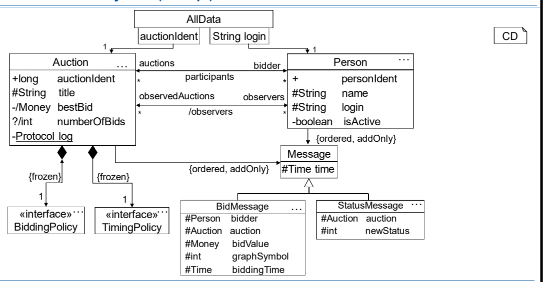
- Muhammad Ali, Albert Einstein of class "Person"
- The car with plate "AC-P-23" of class "Car"
- The software object at address 0x… of software class "Insurance"

Object Diagrams

- An object is an instance of a class.
- Object diagram shows a concrete situation (snapshot) in a system run
  - concrete, named objects
  - specific attribute values
  - link structure between objects
- Object diagram shows a single, possible situation
  - vs. class diagram characterizes all possible situations
- It is possible that the situation shown in an object diagram occurs never, several times, or even simultaneously
- Application patterns
  - Static structures without (much) dynamic changes
  - initial situations for system startup
  - undesirable situations, ...



Class Diagram of the Auction System (excerpt)

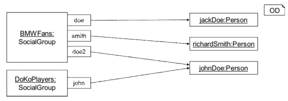


MBSE

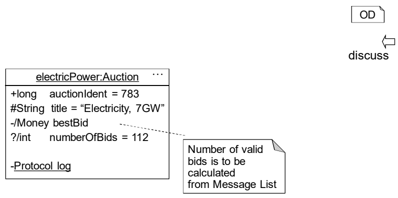
10. Modeling Instance Structures with Object Diagrams
10.2. Language

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

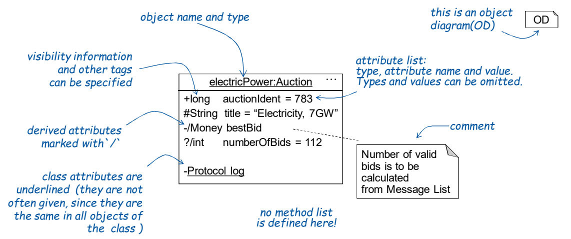
http://www.se-rwth.de/



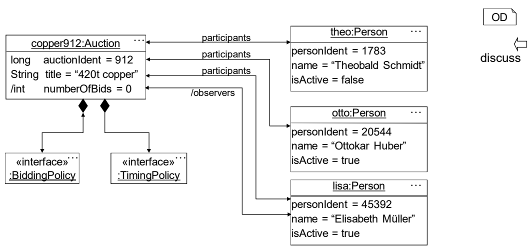
Example: Single Object



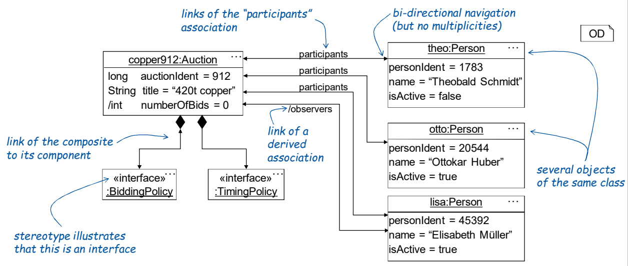
Example: Single Object



Example: Link Structure

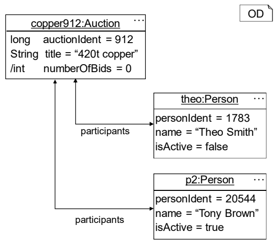


Example: Link Structure



Terminology

- Object
  - object is instance of a class
  - attributes have a value (but need not be shown)
  - object diagram uses prototypical objects
- Object name for identification of the prototypical object
- Attribute describes state of an object
  - always: attribute name
  - optional: type, value, and visibility
- Abstract object diagrams use variables and expressions instead of concrete values
- Link is an instance of an association between objects
  - optional: navigation direction, association and role names



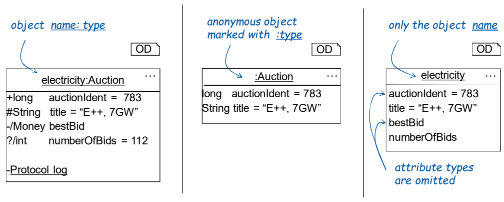
Exercise

home

- Goal: Dealing with OD
(not with perfect content, but syntactically correct):
- Design ODs that characterize the following situations (with the most interesting relationships between the involved elements):
  - your family with their residences
  - an aircraft and its technical equipment
  - a (multi) flight connection for the guest "Wolfgang" on 2.4.2024

Representation of an Object

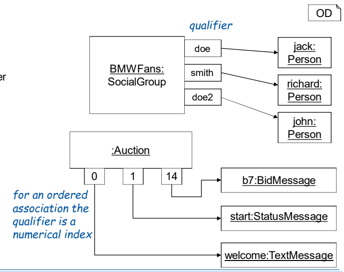
- ... is possible in many ways



- Representation indicators "…" and "©" indicate completeness of the attribute lists.

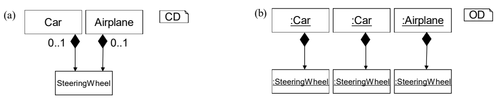
Links in Qualified and Ordered Associations

- Qualified links usually contain the actual value
  - In the BMWFans example it may be a nick name
  - Special form: an attribute of the target object can be used as qualifier
- Links of an ordered association use integers as qualifiers
  - In the example auction, the list of messages is shown
  - the list shown does not need to be complete

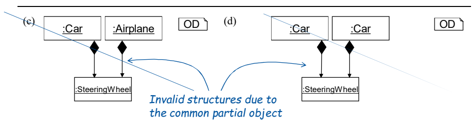


Composition in the Object Diagram

- Class diagram (a) is of course ok

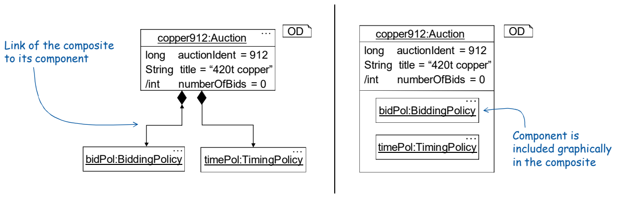


- Object diagram (b) is allowed
- (c) and (d) contain illegal composition structures, because one object belongs to several composites



Alternative Representation of the Composition

- ... by graphical containment
- Nesting is possible
- Both diagrams are equivalent (except for missing navigation information)

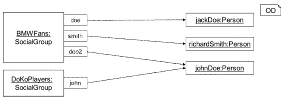


MBSE

10. Modeling Instance Structures with Object Diagrams
10.3. Meaning and Use

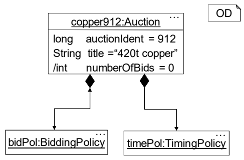
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Semantics of an Object Diagram?

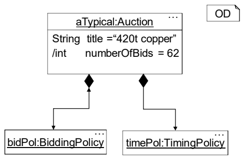
- An object diagram is exemplary
  - As opposed to a CD, which describes sets of possible object structures
- What is the meaning / semantics of such a diagram?
- For which purpose can object diagrams be used?
-
  -
  -
  -
-
  -
-



discuss

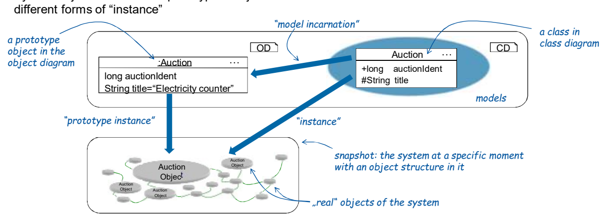
Semantics of an Object Diagram

- Exemplary means that
  - there can be more than one incarnation of the diagram
  - there need not be any incarnation
  - the number can vary over both time and the various system runs
- Do not confuse prototypical objects ("rectangles") in the diagram with objects of the system
  - there is no 1:1 relationship
- Expressiveness of object diagrams is very limited, e.g., properties like these, cannot be expressed:
  - "this OD is valid exactly once."
  - "this OD is valid at beginning"
  - "this OD never occurs"
  - "attribute x is within the range [-5,5]"



Model Incarnation vs. Instance in the System

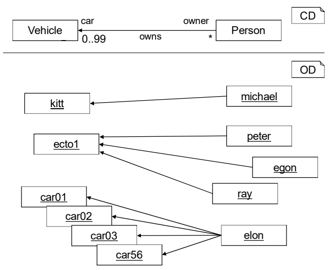
- OD and CD are models, but there is a kind of "model-instance" relationship between both of them: we call this "model-incarnation"
- System objects "instantiate" prototypical objects in the OD and also classes of the CD: but these are three different forms of "instance"



Object Diagrams and Class Diagrams

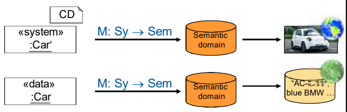
- There are many relations between CD and OD
  - classes must exist for objects
  - attributes of the OD must be defined in the CD
    - and have the same type
  - links must match the corresponding associations
  - multiplicities are obeyed
  - etc.
- These manifest in syntactic consistency conditions
- Representation Indicators "..." and "©" in both diagrams allow omission or indicate completeness
- Interesting questions
  - Which syntax checks applied when during development?
  - Can a CD be derived from exemplary OD's?
  - Which object structures can be derived as test examples from a CD?

OD ⟺ ? CD



Interpretation of OD's in the System

- Interpretation like with classes in a CD:
  - the real objects (on the street)
  - data objects
- Stereotypes apply:
  - «material»... elements, compounds, alloys, ...
  - «system»... machinery, ...
  - «component»... machine elements, ...
  - «energy»... types of energy
  - «being»... humans, animals, ...
  - «data»... for object structures, and other forms of data
  - no stereotype = no fixed interpretation
- More fine-grained stereotypes are possible, e.g:
  - «signal» ... data sent around
  - «subsystem»
  - «item»



Methodological Use of Object Diagrams

Various usage possibilities

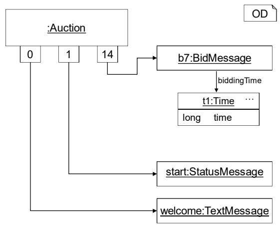
- exemplaric situations for discussion with users
- architecture description of static parts (i.e. a situation that always applies)

- precondition for a method call
- postcondition for a method call

- undesirable situation

- initial situation for a test
- expected result of a test

- Desirable:
  - 1) Combining ODs with OCL to describe logical parts
  - 2) Systematic code derivation from an OD



Prototypical Objects: OD as Matching Pattern

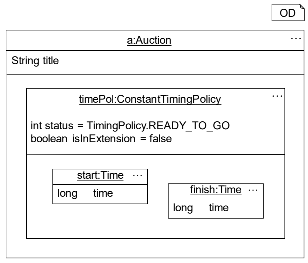
- An object diagrams can be understood as a pattern
  - Prototypes in the OD are patterns that can match real objects
  - Underspecification through incomplete descriptions of objects give freedom

- In addition OCL allows to restrict underspecification further, example:

```
extensionTime = 180s  &&
start.time + 2h <= finish.time;
```
OCL

- Possible Uses:
  - for testing and for programming

→ *Model situations with ODs instead of handcrafted pattern algorithms to recognize certain good/bad situations*



Prototypical Objects: OD as Matching Pattern

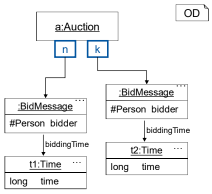
- An object diagrams can be understood as a pattern
  - Prototypes in the OD are patterns that can match real objects

- Abstract values are modeled by terms with variables

- Example: messages are ordered in the list in order of their timing

```
context Auction a,
        int n, k,
        Time t1, Time t2  inv:
    n<=k implies
         t1.timeSec <= t2.timeSec
```
OCL

- Use:
  - → *Model parts of logic formulae with ODs instead of OCL to recognize certain properties and situations*



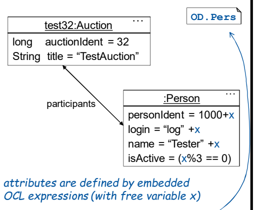
Prototypical Objects: OD as Instantiatable Template

- An object diagrams can be understood as a template
  - Objects and their links in the OD are used to inatantiate real objects accordingly

- In addition OCL (or PL code) allows to combine ODs
  - By using OD by their name (e.g. OD.Pers)

- Possible Uses:
  - for testing and for programming

  - → *Model OD templates instead of handcrafted patterns algorithms to instantiate object structures*



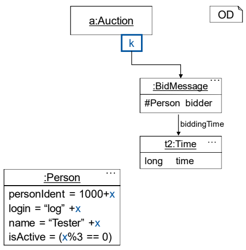

```
context Auction test32 inv MyNewTest32:
      forall int x in {1..100}: OD.Pers;
```
OCL

Methodical Use of ODs with OCL

- Each OD can be understood as logical constraint (with free variables and named accessible objects)
- OD describes an exemplaric situation
- Variables allow abstraction: “abstract values”

- OCL e.g. describes relationships among attributes
- OCL allows combination of object diagrams
  - composition   OD.A && OD.B
  - forbidden   !OD.A
  - alternatives   OD.A || OD.B
  - multiple instances   forall int x in {1..100}: OD.A

- Usage in
  - method specifications (pre-and postconditions)
  - description of the effect of constructors / modifiers



Code Derivation from Object Diagrams

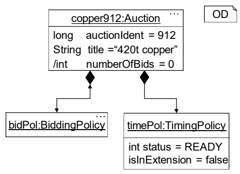
Code derivation / synthesis / generation from OD helps

1) OD as initial structure → code instantiates objects and sets attributes, e.g. like in a sophisticated factory

```java
Auction copper912() {
  Auction a = new Auction(912);
  a.setTitle = "420t copper";
  TimingPolicy timePol = new TimingPolicy();
  timePol.setStatus(READY);                // ...
}
```

2) OD as matching structure

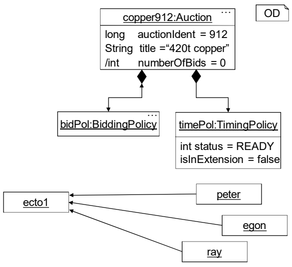
```java
boolean copper912Check(Auction a) {
  return a.auctionIdent == 912 &&
    a.getTitle == "420t copper" &&
    a.getTimePol.getStatus() == READY && ...;
}
```



- Many variants are possible, e.g.
  - Repair (i.e. objects exist and are partially adapted)
  - Marking (i.e. objects in illegal state are marked)
- Necessary: appropriate constructors, access to attributes, “pattern matching” algorithms, …

Summary Object Diagrams

- Object diagrams are exemplary
  - Close relation to class diagrams
  - Prototypical objects instantiate classes
  - Links instantiate associations
- OD can be applied to describe:
  - data in a system
  - physical objects of the system: «component»
  - events in a system: «event»
- OD can be understood as logic formula
- OD can be used in programming and testing
  - for discussion with users
  - architecture description of static parts
  - precondition / postcondition for a method call
  - initial situation / expected result of a test



(There is no chapter 11)

MBSE

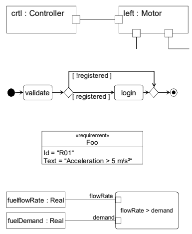
12. SysML v2 as Systems Modelling Language
12.1. SysML v2 Overview

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

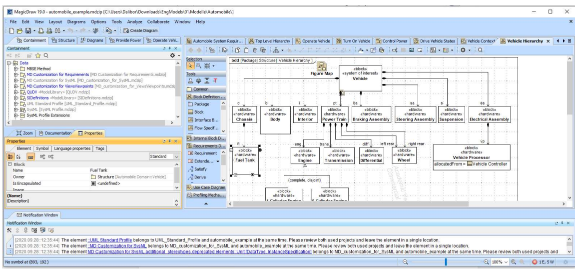
http://www.se-rwth.de/

Four Pillars of SysML

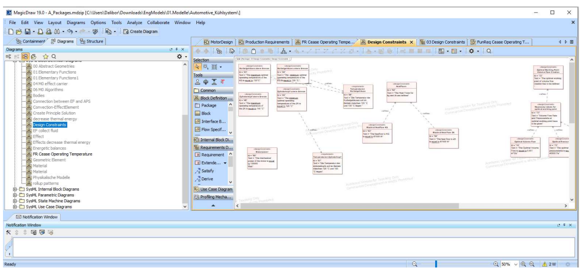
- Structure: Specification of hierarchies, ports, interconnection, model organization
- Behavior: Specification of sequences of actions, life cycle of a block, message-based behavior
- Requirements: Specification of requirements and relationships among model elements
- Parametrics: Constraints, enables integration of engineering analysis and design models



Examples of SysML Diagrams in Practice

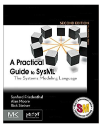


Examples of SysML Diagrams in Practice -2



Notes

- A Practical Guide to SysML, by Sanford Friedenthal, Alan Moore, and Rick Steiner, published by Morgan Kaufmann Publishers, Copyright 2009 Elsevier Inc.
- Lecture Introduction to Model Based Systems Engineering read by Joseph Wolfrom of the John Hopkins University. All rights reserved.
- Systems engineering is subject of ongoing research

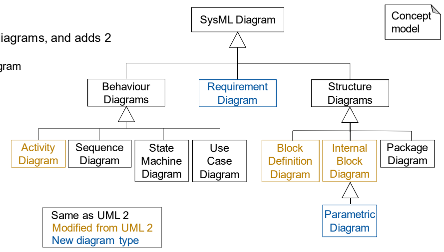


Clarification: SysML vs. UML vs. Systems Engineering

- Model-Based Systems Engineering ≠ SysML
  - May use SysML but not limited to it
- SysML is not a methodology or a tool
  - SysML is a modeling language
  - SysML is independent of methodologies and tools
- SysML is not intended to replace current modeling techniques of the other engineering disciplines (CAD, Simulink, …)
  - SysML intends to connect to others to allow for model interoperation
  - a "single-point-of-truth system model" might use SysML

Relationship Between SysML and UML as Concept Model

- SysML reuses 7 of UML's 14 diagrams, and adds 2 new diagrams
  - requirement and parametric diagram
- The most important two:
  - Block Definition Diagram (BDD)
  - Internal Block Diagram (IBD)
- which are similar to resp. derived from
  - Class Diagrams
  - Object Diagrams



MBSE

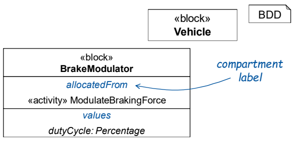
12. SysML v2 as Systems Modelling Language
12.2. Structure

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Blocks in SysML

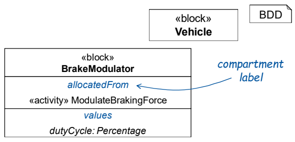
- A block is a modular unit in SysML that is used to define types of physical entities e.g.: system, system component, …
- The SysML block is based on the concept of UML class, but extended and interpreted in the physical world as
  - software, hardware, data, processes, material, energy, personnel, facilities, requirements, …
- Blocks feature various optional compartments that enable to describe block characteristics, e.g.
  - Blocks include Ports for physical flows



In contrast to a UML class
- Block is a stereotyped extension of a UML class
- SysML Blocks extend the syntax of UML classes by distinguishing among various kinds of properties
  - parts
  - values
- The semantic interpretation of blocks differs from software classes

Blocks as Basic Structural Elements

- Flow Specification Block: inputs and outputs are flows
  - A flow property signifies a single flow element to or from a Block
- Interface Block: to support nested ports
- Constraint Block: integration of engineering analyses
- Domain Block: … a component, location, or person
- External Block: Represents an external actor
- System Block: for structure organization
- Subsystem Block: for structure organization
- System Context Block: system embedded in its context



- Multiple compartments to describe blocks
  - properties (parts, references, values, ports) and operations (ca. CD)
  - constraints (used similar to OCL)
  - "allocations" from/to other model elements (i.e. where the information comes from)
  - requirements the block adresses
  - … and user defined compartments

Flow Specification Block

- Specifies inputs and outputs as a set of flow properties that list can be displayed in a flow properties compartment
- Flow specification is used to type Flow Ports, in order to specify items which can flow via the ports
- A Flow Port of type FuelFlow can (in this case) bidirectionally receive and emit Fuel
- May use both directions but does not need to

«flowSpecification»
FuelFlow

flow properties
out fuelSupply : Fuel{readOnly}
in fuelReturn : Fuel{readOnly}

Interface Block

- Special kind of block for typing proxy ports
  - no behavior or internal parts
- Contains a set of flow properties that can be shown in the flow properties compartment

«interfaceBlock»
ICE

flow properties
out engineData : ICEData
in mixture : Real
in throttlePosition : Real

Constraint Block

- Provide mechanism for integrating engineering analysis with other SysML models, such as
  - performance models
  - reliability models
- Constraint blocks can be used to specify a network of constraints
  - represent mathematical expressions that constrain the physical properties of a system
  - not typed or checked…

«constraint»
PositionEquation

constraints
{x(n+1) = x(n)+v*5280*3600*dt}

parameters
v : Vel
x : Dist

dt

Domain Blocks and External Blocks

- **Domain Block**
  - represents an entity, a concept, a location, or a person from the real-world domain
  - part of the system knowledge
- **External Block**
  - block that represents an actor
  - facilitates a more detailed modeling of actors like ports or internal structures

«domain»
AutomotiveDomain

parts
HSUV : HybridSUV

properties
: Driver
: Passenger
: Maintainer

general superset of the other property types of a block

«external»
Road

values
incline : Real

System Blocks and Subsystem Blocks

- **System Block**
  - artificial consisting of blocks that pursue a common goal which cannot be achieved by the system's individual elements
  - can be software, hardware, a person, or an arbitrary unit
- **Subsystem Block**
  - typically a large, encapsulated block within a larger system
  - any Block can be converted to a Subsystem if you decide that the appropriate Block is decomposed

«system»
HybridSUV

parts
c : ChassisSubsystem = c
p : PowerSubsystem = p
b : BodySubsystem = b

«subsystem»
PowerSubsystem

parts
fuelSupply : Fuel
i1 : ElectricCurrent

references
fuelReturn : Fuel

values
fuelFlow : FuelFlow

System Context Block

- Virtual container that includes the entire system and its actors
- Any Block can be converted to System Context if you decide that the appropriate Block is decomposed

«system context»
Car

constraints
fuelFlow : FuelFlow

parts
fuelSupply : Fuel
i1 : ElectricCurrent
i2 : ElectricCurrent

references
fuelReturn : Fuel

values
sn : ID

MBSE

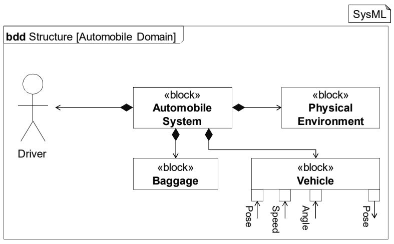
12. SysML v2 as Systems Modelling Language
12.3. Block Definition Diagrams

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

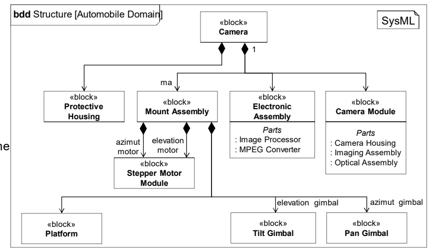
Block Definition Diagram (BDD)

- … represents structural elements called blocks, their composition and classification
- Describe relationships between blocks
  - composite association
  - generalization
  - (no associations)
- Define structural features of blocks
  - part properties
  - value properties
  - ports
- Define behavior of blocks
  - operations resp. at least their signatures



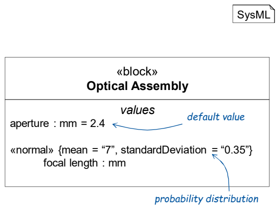
BDDs rely on Composite Association

- Composite associations depict parts that make up the whole
  - black diamond (like in UML)
  - role names can appear on the part end
- Semantics:
  Composite is composed of parts and parts don't change over lifetime
  → static structure



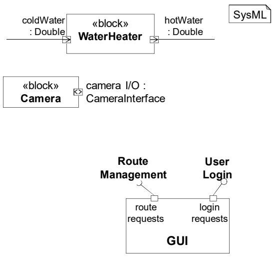
Value Properties

- Used to model quantifiable properties of the system component that is modelled (similar to attributes)
- Based on a value type, which describe the values for quantities
- Listed in compartments using the following syntax
  - value_property_name: value_type_name
- Value properties
  - can have default values
  - can also define a probability distribution for their values



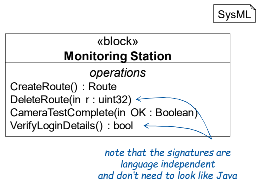
Ports

- Describes a structural interaction point of a SysML Block, which connects interacting parts of a block
  - equals interfaces in component and connector architectures
- Flow port (ca. MontiArc ports)
  - specifies what can flow in or out of a block
  - flow ports have a certain direction, which is indicated by the arrow direction
  - flow port type is defined by name:type
  - written over the port symbol
- Standard port: specifies a set of required or provided operations
  - Observe: this is bidirectional: OO only has provided operations
  - But otherwise operations are like method call signatures



Operations

- Operations describe something that a block can do
- Operations can have parameters
- Operations are typically synchronous (requestor waits for response)
- Operations are listed in the operations compartment of a block:
  - "operation name (parameter list): return type"
- Equivalent to methods of CDs because BDDs and CDs coincide



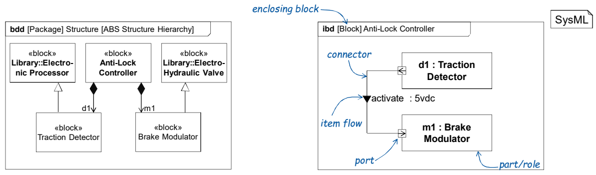
MBSE

12. SysML v2 as Systems Modelling Language
12.3. Internal Block Diagrams

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

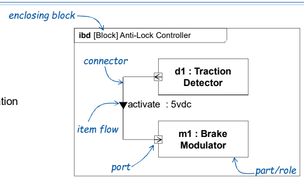
Block Usage (vs. Definition)



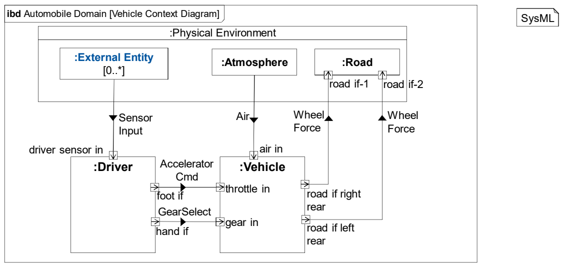
- **Definition**
  - Block definition diagram (BDD) defines blocks (types)
  - captures properties, etc.
  - reused in multiple contexts and varying connections
  - (similar to class diagram)
- **Usage**
  - Internal block diagram (IBD) defines the architecture of a block using other blocks
  - It provides one concrete connection structure
  - (similar to object diagram)

Syntactic Elements IBD

- Enclosing block
  - indicates that this ibd diagram defines the internal communication structure of a block
- Part/Role
  - define the role of the elements within the communication structure
- Port
  - indicates the inputs and outputs of a part
- Connector
  - connects two parts respectively their ports
- Item flow
  - defines the items (signals, messages) that are exchanged

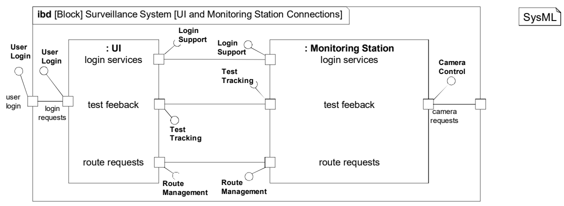


Context Block: A Vehicle System Context with External Interfaces

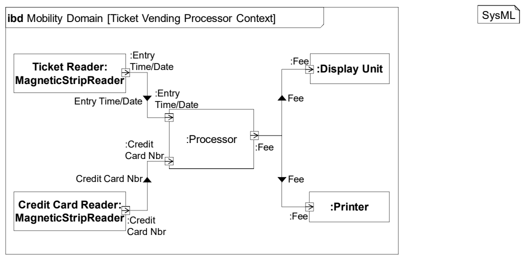


Modeling Standard Ports and their Connectors on an IBD

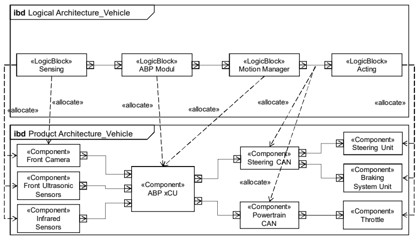
- Standard ports specify interactions as services
  - required interface specifies requests for services (socket symbol)
  - provided interface specifies provided services (ball symbol)



Context Block Showing External Interfaces



Mapping in SysML: Mapping Logic to Product Architecture

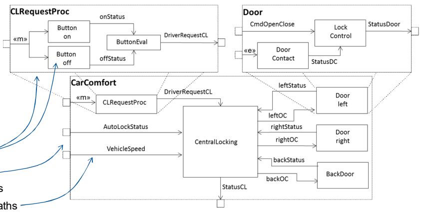


- Logical Architecture:
  - Functions and
  - Composition to an Architecture
- Product Architecture:
  - Decision on HW/SW-realization
  - Hardware components
- Next technical level would be:
  - Explicit inclusion of communication components

The MontiArc C&C ADL

Repetition

- MontiArc is an ADL
  - developed using MontiCore,
  - based on the stream approach
  - for modeling software and system architectures
  - extensible with component behavior languages
- Most important MontiArc elements
  - component: unit of computation
  - interface: has typed, directed ports
  - hierarchy: topology of subcomponents
  - connectors: realize communication paths

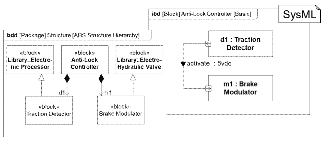


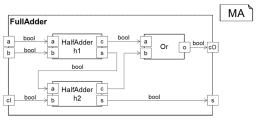
Comparison MontiArc and SysML (with respects to blocks)

- Both languages have much in common

MontiArc	SysML
component	block
port	port
channel	flow
composition	composition

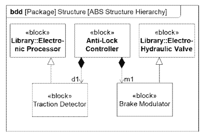
- Differences:
  - SysML needs separate diagrams (BDD & IBD) for definition and use of blocks → MontiArc only one
  - SysML includes behavior languages, MontiArc allows language composition to select your own languages
    - E.g. MontiArcAutomaton, Embedded MontiArc, MontiThings, MontiArc for KI

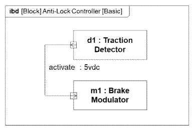




Summary

- SysML
  - Most import diagrams: IBD & BDD
  - Used for many purposes in Systems Engineering
  - Concepts:
    - block,
    - directed port,
    - connector,
    - composition





Model-Based Software Engineering

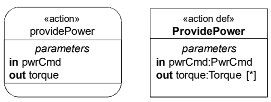
12. SysML v2 as Systems Modelling Language
12.3 Behavior

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

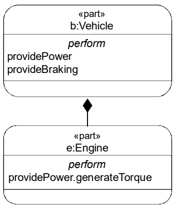
Actions

- Used to model the steps of the activity
- Accept inputs and create outputs
- Can be hierarchically decomposed
- Again, actions are defined by an action definitions

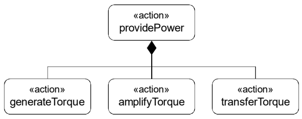


- They are used in industry (e.g., BMW, Daimler, Siemens) to model activity-oriented concepts

Parts Perform Actions

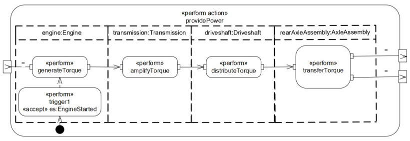


- A part can perform an action
- Part decomposition can follow action decomposition



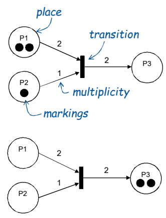
Action Flow in a Nutshell

- Actions
  - action parameters: inputs and outputs
  - special send and accept message actions
- Action flow
  - successions: control flow
  - inputs and/or outputs: object flow
  - control nodes: control flow
    - decision and merge
    - fork and join
- Filled circle: initial node
- Bulls eye: final node
- At the top are (optionally) the actors/parts
  - actions of each actor are in a swimlane



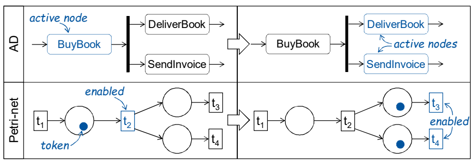
Excursion: Petri Nets

- A mathematical modelling language of bipartite graphs
- To describe discrete event dynamic systems (e.g., in Business Process Modeling, Simulation, Data Analysis)
- A petri net PN is defined as PN = (N, M, W) with
  - a net… N = (P,T,F) over
  - finite disjoint sets of places P and transitions T
  - set of arcs F ⊆ (P × T) cup (P × T)
  - a place multiset M : P → Z
  - an arc multiset W : F → Z



Petri Net Semantics of Activity Diagrams

- Active node in activity diagrams are recognized via implicit tokens
  - Semantics inherited from Petri-nets
- Firing a transition removes a token from the input nodes and adds a token to the output nodes
  - Petri net transitions can fire if all input nodes contain a token

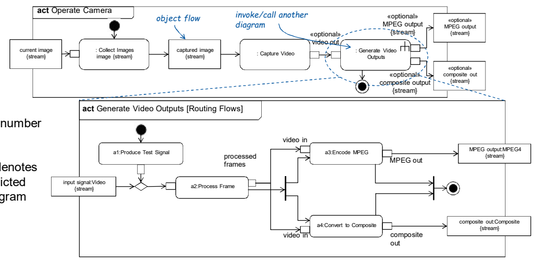


Control Flows vs. Object Flows

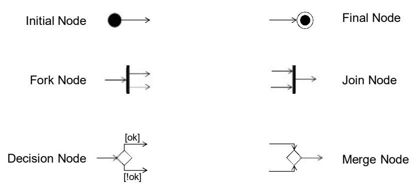
- Used to show sequences of actions
- Control relates to a control token
  - Actions cannot start until it receiving a control token on all input control flows
  - Upon completion, actions place control tokens on all outgoing control flows
- Can be depicted by a dashed arrow, to distinguish it from object flows
- Can be used with:
  - Forks and join nodes (parallel execution of control)
  - Decision and merge nodes (selective execution of control)
- Object flows only communicate data

Decomposing an Activity Diagram with Call Behavior Actions

- Pins match Parameters in number and type
- Rake symbol denotes details are depicted on another diagram



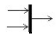
Most Important Types of Nodes



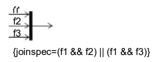
Fork and Join Nodes

- Fork nodes have one input flow, multiple output flows
  - Output flows are independent and concurrent

- Join node have multiple input flows, one output flow
  - Output occurs, only when all input flows are arrived at the node (default)

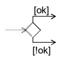


- Join specifications may override default join behavior of join nodes

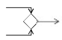


Decision and Merge Nodes

- Decision nodes have one input flow, multiple output paths
  - Guards must be mutually exclusive (non-overlapping)
  - Only one output path can be used, based on guard conditions

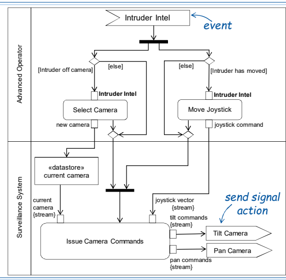


- Merge nodes have multiple input flows, one output flow
  - Output flow is triggered upon arrival of a token on any of the input flows

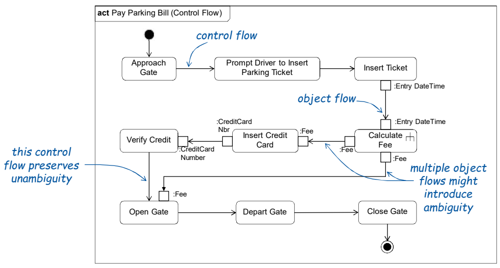


Partitions (aka Swimlanes)

- Allocates actions to an entity responsible for performing the action
- Can be used to specify functional requirements of an actor, component, or part
- Can be depicted horizontally or vertically

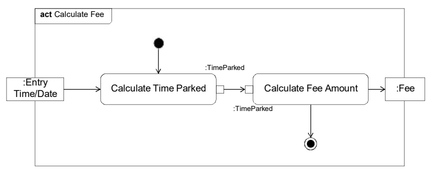


Activity Model (with Branching Object Flows)



Decomposition of Calculate Fee

- Example below shows use of Input and Output Parameters for the Calculate Fee Activity
- Hierarchical relationship of activities and actions



Summary

- Activity diagrams model behavior that specifies the transformation of inputs to outputs through a controlled sequence of actions
  - inputs/outputs can either be streaming or non-streaming
- Parameters: Multiple inputs or outputs of activities and contain multiple actions
  - Actions consume input tokens and produce output tokens via pins
- Object Flows are used to depict the flow of object tokens from one action to other actions
- Control Flows are used to depict the transfer of control from one action to other actions using control tokens
- Partitions are used to assign responsibility for actions to blocks or parts that the partition represent

Model-Based Software Engineering

12. SysML v2 as Systems Modelling Language
12.4 Constraints and Requirements

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Constraints

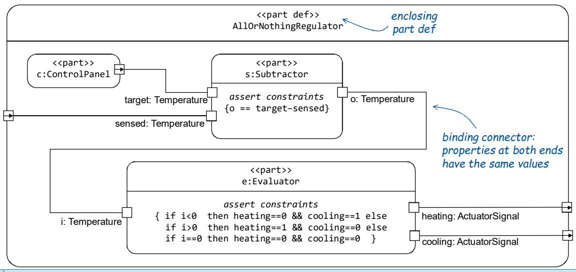
- Constraints are reusable, parametrized Boolean expressions
- Used to express specifications (equations)
  - Reference their inputs
  - They can model abstract (not computable) analytical constraints, value derivations, boundaries, etc.
- Constraints can be used in a context, typically a part
  - Their inputs can be bound
  - They can be asserted
- Asserted constraints cannot be violated
  - Otherwise the model is inconsistent

asserted constraint usage

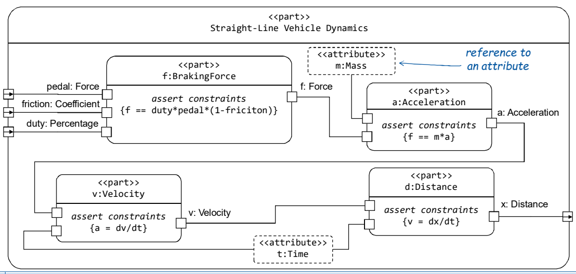
```
constraint def MassConstraint {
  in partMasses : MassValue[0..*];
  in massLimit : MassValue;

  sum(partMasses) <= massLimit
}

part def Vehicle {
  assert constraint m : MassConstraint {
    in partMasses = (
        chassisMass, engine.mass,
        transmission.mass
    );
    in massLimit = 2500[kg];
  }

  attribute chassisMass : MassValue;
  part engine : Engine {
    ...

}
```

binding of values

Examples from Industry



Examples from Industry



Requirements

- Constraint Definitions
  - define equations that can be re-used and inter-connected
  - define a set of parameters
  - define an expression that constrains the parameters
- Requirements are special Constraints
  - They group multiple constraints (or requirements, i.e., allow nesting)
  - They can have assumptions in the form of constraints
  - They have a subject and can reference its attributes
  - They can be satisfied by providing an actual subject (similar to constraints being asserted)
- non-satisfaction renders model inconsistent

```
requirement def VehicleMassLimit {
  subject vehicle: Vehicle;
  assume constraint {
    doc
    vehicle.fuelMass == vehicle.maxFuelMass
  }
  require massConstraint;
}


part commuter: Vehicle {
  attribute :>> chassisMass = 1500[kg];
}


satisfy r1: VehicleMassLimit by commuter;
```

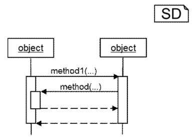
reference to constraint

13. Summary

- Parametric diagrams
  - capture the analysis as a network of equations
  - help ensure consistency between the system design model and multiple engineering analysis models
  - help to manage technical performance measures
- Constraint Blocks
  - define parameters and constraint expressions
  - represented on a Block Definition Diagram
- Constraint Property
  - usage of constraint blocks
  - represented on a Parametric Diagram

MBSE

13. Interactions with Sequence Diagrams
13.1. Concepts, Syntax



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

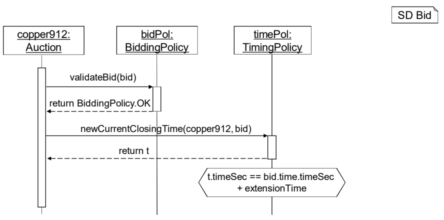
http://www.se-rwth.de/

Sequence Diagrams (SD)

- Uses of SDs
  - modeling of exemplary observations
  - representation of interaction patterns of objects
  - chronological order of calls
  - trigger for tests
- Core features of SD
  - exemplary nature
  - focus on interaction between objects
  - uses a timeline
- Comparison of SD and statechart: both are behavioral descriptions

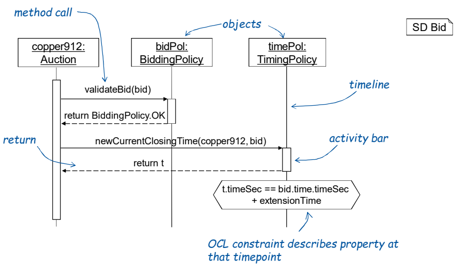
sequence diagrams	statecharts
interaction of several objects	behavior of one object
exemplary	complete
no internal state	state based

Example: Sequence Diagram



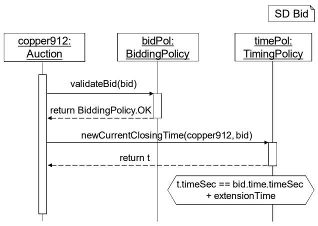
analyze

Example: Sequence Diagram



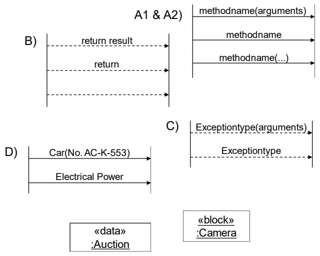
Terms for Sequence Diagrams

- Object: used as in the object diagram, only with the object name and type (no attribute values)
- Timeline describes the time elapsed of the object from top to down
  - no true scaling, but temporal order
- Interaction takes place between two objects
  - trigger is a stimulus (as discussed in statecharts)
  - parameters of interactions can be omitted
- Activity bar show the activity duration of a method call
  - activity bars can be represented nested in object recursion
- Logic condition describes a property at a certain point of time



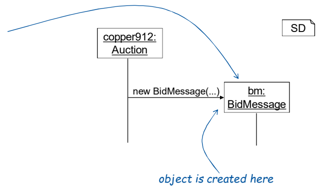
Forms of Interaction

- arrow types describe several forms of interaction
- for software
  A1) method call and
  A2) asynchronous message/signal transfer
  (not distinguished from method call)
  B) result of a previous method call (return)
  C) exception (as abnormal termination)
- for systems:
  D) flow of physical items
  flow of energy (i.e. power, heat, etc.)
  … and of course also data and messages
- The "objects" are the very same as in the object diagrams and SysML's IBDs, possibly tagged with their kinds



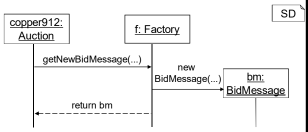
Object Creation via Constructor

- If an object doesn't exist yet at the at the beginning of the observation, but it will only be provided during creation then it is placed at the creation point.
- For C++ processes a similar construct exists for the destruction of an object.
  – Target language Java does not require this.

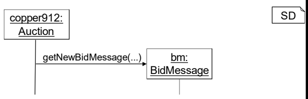


Object Creation with Factory

- Example, how an object is created via a factory:

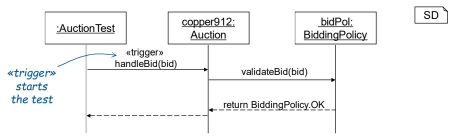


- However: abstract representation (i.e. omission of intermediate actions/objects) is useful
  – Here: omission of the factory in the model (although it will be needed in the code)
- The semantics of the SD has to permit this



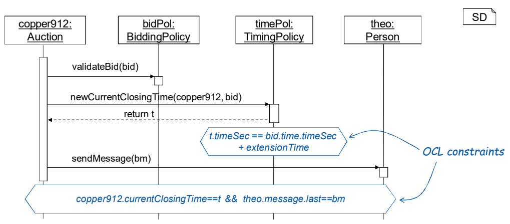
Stereotypes

- SD also has its own predefined stereotypes
- Example:
  – «trigger» marks the call, that triggers the interaction of the sequence diagram
  – used for e.g., modeling of tests:



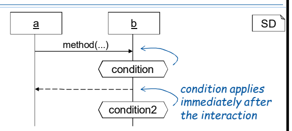
OCL Constraints in the Sequence Diagram

- OCL condition characterizes a property, that holds at a specific point (i.e. in the middle of the execution):

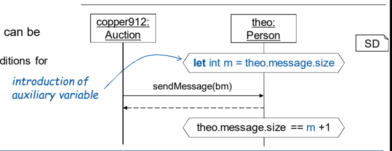


OCL Constraints in the Sequence Diagram

- OCL constraint must hold exactly during a specific point in the execution
- Variable names that can be used in OCL constraints:
  – all object names
  – attributes of the objects, whose timelines are touched
  – arguments of previous method calls



- Auxiliary Variables in Sequence Diagrams can be introduced with the let construct
  – in analogy to the let variables in pre-/postconditions for reuse in later OCL conditions



Examples: Sequence Diagrams for ...

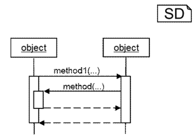
- Booking a ticket at the counter
- Money withdrawal
- Money transfer
- Direct phone call
- Phone call via operator

exercise

- *Was one SD sufficient?*

MBSE

13. Interactions with Sequence Diagrams
13.2. Semantics

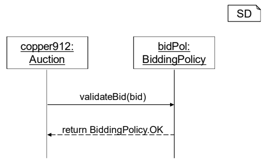


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

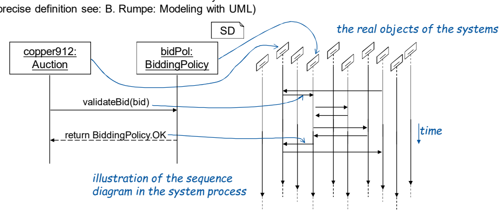
Exemplary Nature and Incompleteness of SDs

- A sequence diagram describes a period in a process of the system:
  - the object set is incomplete
  - arguments of method calls may be missing
  - further interactions take place before and after the shown ones
    - more interactions could even happen in between
  - the same sequence can occur multiple times
    - it may even occur temporally overlapping
    - it may also not at all occur
  - What is the semantics of a sequence diagram?
  - What does a sequence diagram tell us?



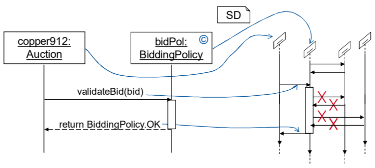
Precise Meaning of a SD

- is based on a mathematical mapping of
  - prototypical objects in the SD to real objects of the system
  - interactions of the SD to real interactions in the system
  - (for precise definition see: B. Rumpe: Modeling with UML)



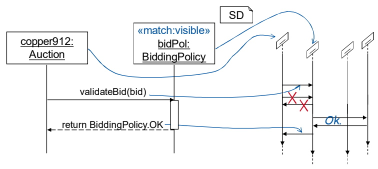
Complete Representation of Interaction

- To model that all object interactions that are taking place during the period of observation are shown tag «match:complete» (short: ©) is used
- «match:complete» prohibits all other interactions in between



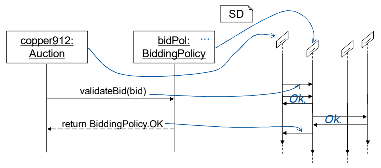
Nearly-Complete Interaction Shown

- All interactions with other visible objects that are taking place during the period of observation are shown when tag «match:visible» is used
- «match:visible» prohibits that other interactions in between with visible objects:



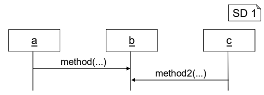
Interaction Shown Incompletely

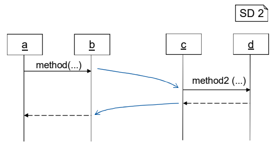
- Any other interactions are possible
- Tag «match:free» (short: ...) permits all other interactions in between



Non-causal SD

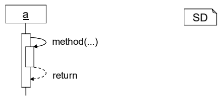
- Non-causal SD is a SD, in which the effect chain (causality) is not clear
  - Examples 1 and 2: why did method2(…) occur?
- These are non-causal but possible observations
  - In SD 1, e.g.. due to a not represented interaction from a to c (from b to c or an unknown object to a and c)
- In concurrent systems a temporal order may have causal reasons, but may also be based on pure coincidence.
- Causal SD can relatively well be used for constructive code generation
  - Step 1: Merge all SD into a kind of "regular expression" over interactions
  - Step 2: Transform this into state machines for each participating objects/component individually



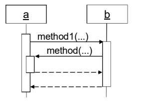


Object Recursion in SD

- In OOP Method calls have a corresponding return interaction
  - these interactions may be nested
- Method recursion: the same method is called again with other arguments
- Object recursion: the same object is called again
  - Which may be the same method (or another)
- Object recursion is very common in OOP, many design patterns use this



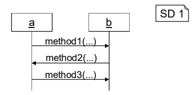
(a) direct object recursion

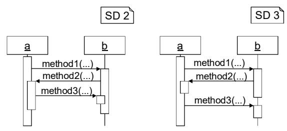


(b) indirect object recursion

Object Recursion introduces Underspecification

- SD 1 is a correct as a description of an observation
- However, the observation is incomplete (some details are not shown)
  - e.g. it is underspecified in what causes e.g. method3(…)
- Possible clarifications are SD 2 and SD3



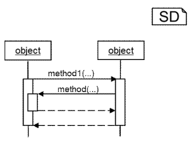


MBSE

13. Interactions with Sequence Diagrams
13.3. Methodical Use

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

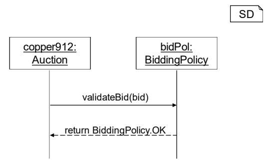
http://www.se-rwth.de/



Methodical Use of SD

- A: A SD can describe an observation:
  - A1: generated from an execution protocol for "debugging" (those are usually lengthy and detailed)
  - A2: manually developed during the analysis activity (used to specify desired interactions. e.g. in tests)
- B: A SD can be a constructive description of a necessary execution order:
  - e.g. if it is the only possible, unique execution order with no alternatives (unfortunately, this is rare)
- C: A SD can be a test driver:
  - «trigger» interactions are driving the test,
  - the other interactions are modelled as observations.

+ more



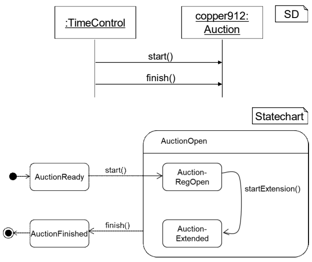
SD and Statecharts

Possible methodical uses of SD with Statecharts:
A: 1. SD as exemplary description from which
2. Statecharts are synthesized, or

B: given Statecharts are analyzed by review of specific system runs (SD)
– simulations of a Statechart produce SDs
(→ close to process analysis)

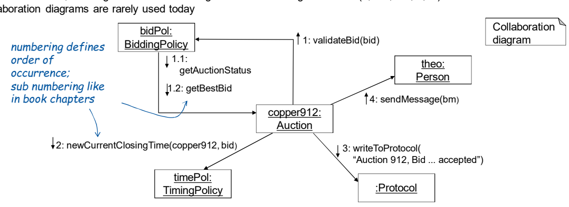
C: 1. SD and statecharts developed independently as two viewpoints of the system and
2. checked for consistency through
2a. appropriate matching techniques (call sequences, etc.) or
2b. through code generation from statecharts, test case generation from SD

- Further reading: Ingolf Krüger, LSC of D. Harel, et al.



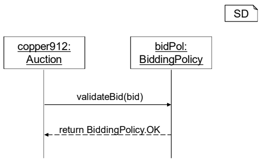
Collaboration Diagram vs. Sequence Diagram

- Collaboration diagrams describe the same information as SDs
  - but another form of presentation
  - instead of a timeline, message/interaction ordering is indicated through numbers (1, 1.1, 1.2, 2, ...)
    - collaboration diagrams are rarely used today



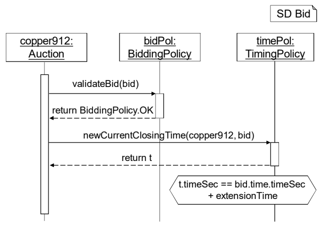
Variants / Extensions for SD

- History of SD:
  - Sequence diagrams were a variant from message sequence charts (MSC) that are used in the telecommunications field in combination with SDL
- Extensions for MSC's
  - concatenation, alternatives, repetition, parallel operations, recursion
  - with these enhancements SDs are more expressive, since e.g., the alternative processes and iteration can be described
  - comparable to formal languages:
    - single string = simple SD
    - regular expression = extended SD (without recursion)
    - context-free language = extended SD (with recursion)
  - further extensions in Harel's Life Sequence Charts



Summary for Sequence Diagrams

- A SD consists in its basic form of:
  - Objects (with name and type)
  - Timeline for objects
  - Interaction pattern
  - Activity bars
  - Conditions
- A SD describes an exemplaric behavior
  - extensions like «match:complete» allow to give SD more rigorous semantics
- SD can be checked with Statechart for consistency
- SD can be used for test case definition
- SD can be used for cutting out code



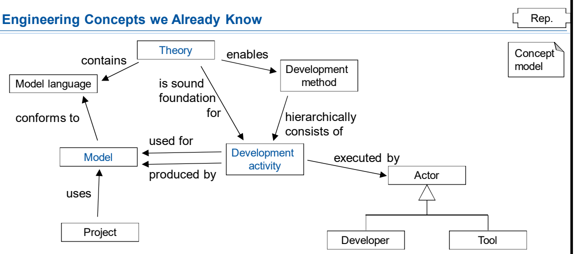
MBSE

14. Software and System Development Methods
14.1. Model-Based Development Methods

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Systems Engineering Concepts we Already Know



- The concept model illustrates some relevant concepts and their relationships.
- In this chapter we introduced: model, development method, and their underlying theory.

MBSE

14. Software and System Development Methods
14.1. Development Methods

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Systems Engineering is an Interdisciplinary Approach for the Realization of Systems

Rep.

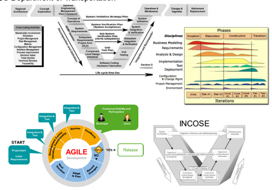
Definition (INCOSE 2016):

INCOSE Systems Engineering (SE) is an interdisciplinary approach and means to enable the realization of successful systems.

It focuses on
- holistically and concurrently understanding stakeholder needs;
- exploring opportunities;
- documenting requirements; and
- synthesizing,
- verifying,
- validating, and
- evolving solutions

while considering the complete problem, from system concept exploration through system disposal.

US Department of Transportation

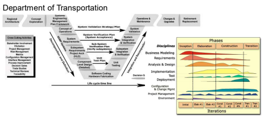


A method decomposes the big problem into smaller, manageable activities

Systems development life cycle

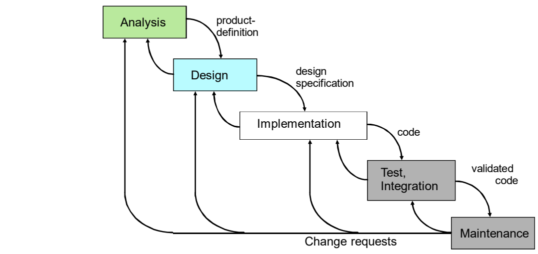
- A systems development process is the process of dividing system development work into smaller activities to improve design, product management, and project management.
- It is also known as a systems development life cycle (SDLC).
- Typical distinct activities
  - planning,
  - requirements gathering,
  - analysis,
  - design,
  - implementation,
  - testing,
  - deployment.
- and additional activities
  - architectural exploration,
  - verification,
  - validation
  - evolution,
  - maintenance,
  - bug fixing,

US Department of Transportation



- "Process" vs. "method" vs. "development model"
  - Some say: Process tells only what, method also how
  - Others say: Process is a concrete instance of the method (= development model) for a concrete project
  - Sometimes: synonymous use.

The Waterfall Model

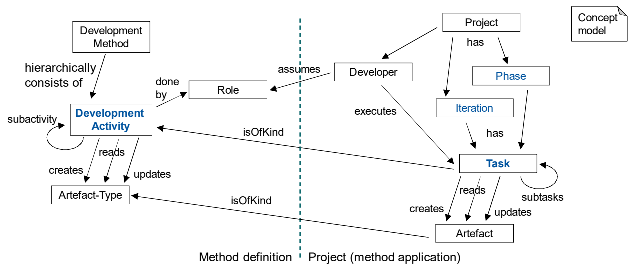


W. Royce (1970)

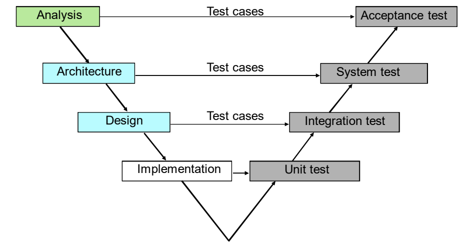
Most Important Development Activities

- Requirements analysis
  - Requirements, problems, objectives, and resources are identified. Stakeholders are involved.
  - Clarifies: "What to do?"
  - Subsumes: requirements elicitation, business analysis, system analysis
- Architecture
  - Defines the overall structure, i.e. the big picture.
  - Explores alternatives.
- Design
  - Defines a fine-grained specification of system elements.
  - Clarifies: "How to do it?"
- Implementation (also: Construction)
  - Coding, 3D-printing, any other form of construction and production.
- Testing
  - Any form of analyses, experiments, executions or reviewing of development artefacts, subsystems or the system that ensures desired quality.
  - Subsumes: Validation against stakeholder requirements and verification against design artefacts.
- Maintenance
  - Keeps a product in good condition by adapting it to changed needs, checking it, and repairing it when necessary.
  - Subsumes: Evolution (in the small), bug fixing

Concept Model for Development Methods and Projects



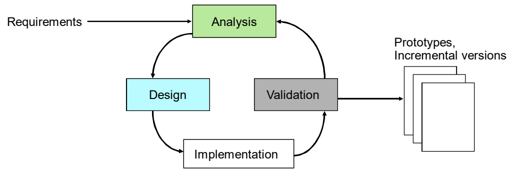
Quality Assurance in the V-Model



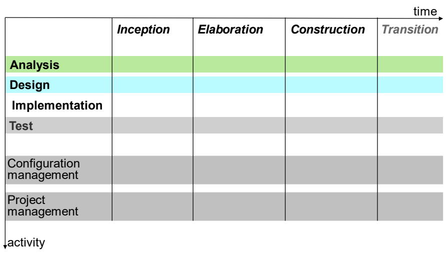
Boehm 1979 (initial, old „V-Model")

Evolutionary Development

- Rather useful for smaller projects or experimental systems
- But is increasingly used for larger projects
- Predecessor of methods Extreme Programming and Scrum

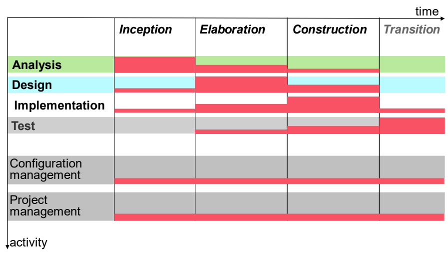


The Rational Unified Process (RUP) decouples Activities and Phases



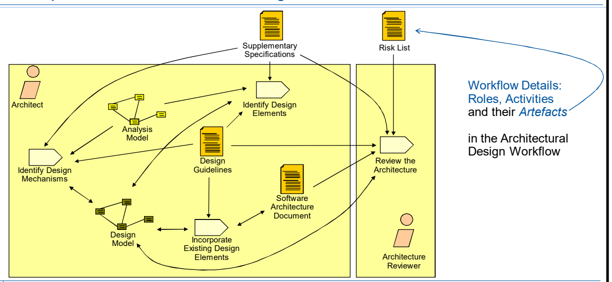
Rational Unified Process 1999 (Jacobson et al., Kruchten)

RUP supports labor distribution



Rational Unified Process 1999 (Jacobson et al., Kruchten)

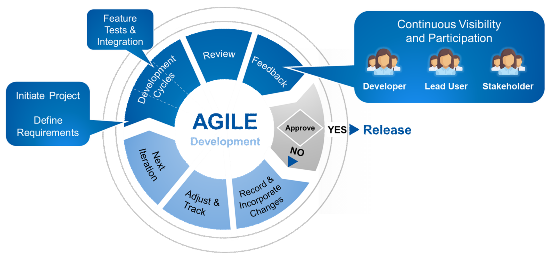
RUP: Example Workflow for Architectural Design



Workflow Details:
Roles, Activities
and their *Artefacts*

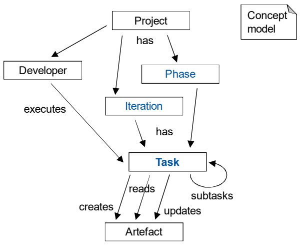
in the Architectural
Design Workflow

Scrum, XP: Highly Iterative Development Processes



Development Methods

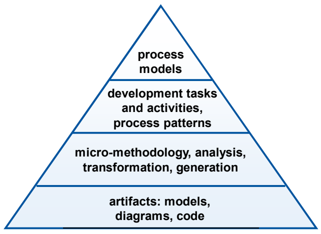
- A systems development process organizes its activities in phases and iterations.
- Phase
  - Structure for larger projects
- Iteration
  - A concrete project may have variably many iterations
- Waterfall: identifies phases = activities, no iterations
- RUP: has 4 phases, with many iterations in-between
  - Separates phase from activity (such as "design" act.)
- (Todays) V-Model: several iterations, with phases in-between
- XP: only iterations, no explicit phases



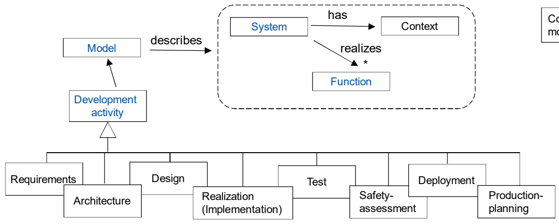
The Methodological Pyramid

Rep.

- Process models, such as RUP, V-Model, define the overall development process .
- They are composed of an appropriate set of development tasks and activities, such as "elicit requirements", "review the architecture"
- To accomplish these tasks a large set of "micro methods", e.g. using a best practice, a design pattern, tools for analysis, generation or synthesis, tools for evolution and transformations, etc.
- All these tasks are finally executed on the set of artifacts, that contains all relevant development information, such as requirements, all kinds of models, tests, code.



Summary as a Concept Model



- Systems and their functions are described by models, which are part of various development activities

MBSE

14. Software and System Development Methods
14.2. Variants of Methods

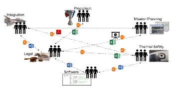
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Please recapitulate earlier chapter on Systems Engineering

Rep.

- Traditional Systems Engineering is Document-Based



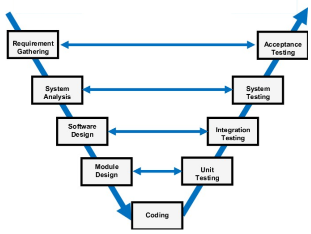
- Model-Based Systems Engineering is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.
  - INCOSE SE Vision 2020

- Opportunities of Model-Based Systems Engineering
  - consistent, related models ensure integrity and enable traceability
  - Enables top-down design decisions and drivers
  - Automated change propagation, ambiguity checking
  - Automated tracing of (changing) requirements to (changing) implementations
- Model-Driven:
  - Models even drive and guide the process
  - Models are primary development artifacts

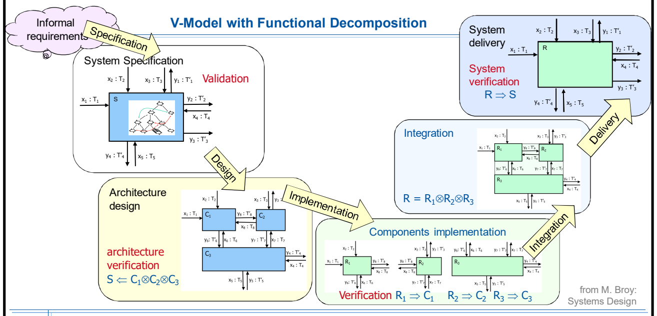
INCOSE

V-Model: A Standard Process to Develop Software

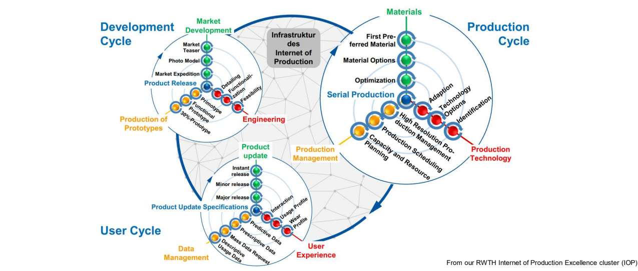
- The V-Model has
  - a constructive left wing:
    - from requirements to coding
  - and a quality assurance and testing right wing:
    - From unit tests to acceptance tests
  - Each activity on the left corresponds to tests on the right
  - The V-Model assumes manual work in all activities, it is agnostic to models and automation
  - In practice: more than 2/3 of the work happen on the right side



V-Model with Functional Decomposition

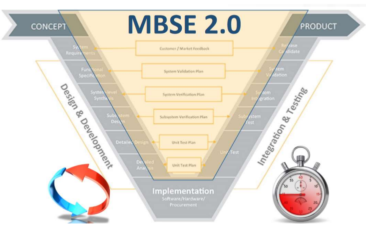


Agile Development in Engineering, Materials and Business



V-Model and MBSE

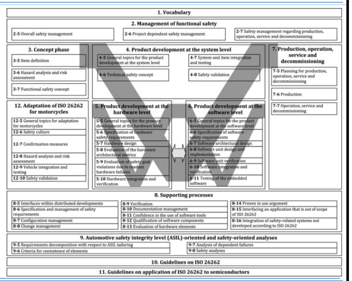
- Model based Systems Engineering 2.0



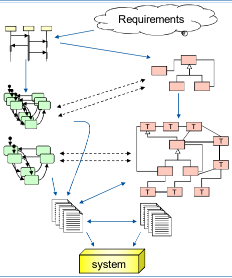
From David Long, Vitech, 2019

ISO 26262-1, Road vehicles — Functional safety

- ISO 26262-1 standard for vehicles
- Goal: quality of products
- Mechanism:
  - Enforces dedicated development and operation activities,
  - Organized in a development process
- Tries to be abstract
  - Allowing individual arrangements for each activity
  - Neither concrete tool, nor languages enforced
- V-Model compliant (if not enforcing)



Model-Driven Architecture (MDA)



use cases and scenarios:
sequence diagrams describe users' viewpoint

application classes define data structures

state machines describe states and behavior

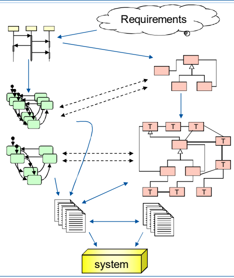
technical class diagram
adaptation, extension, technical design

+ behavior for technical classes

code generation
+ integration with manually written code

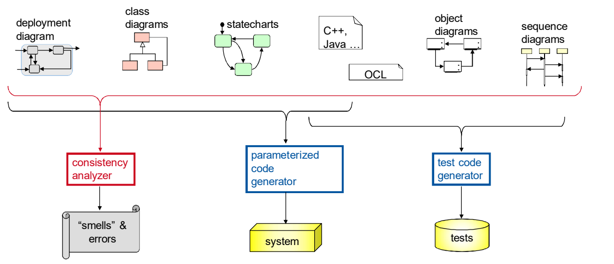
complete and running system

Problems of Model Driven Architecture

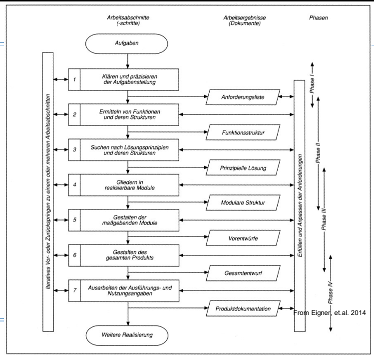


- Not much reuse (libraries …)
- Tool chain too deep
- No efficient tools
- Tracing problems
- Evolution is awkward
- Lot of information missing, e.g.,
  - design rationale
  - non-functional requirements
- “Agile” development is not possible
- SE-Models are not integrated with other Engineering Models (spatial, biological, …)

Agile UML-based Software Development: Constructive Use of Models for Coding and Testing



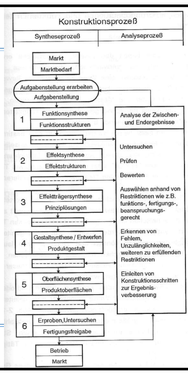
Produkt Entstehungs Prozess (PEP)



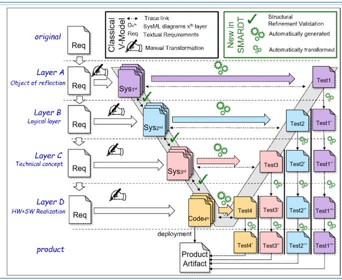
From Eigner, et.al. 2014

Koller/Kastrup: Construction Process

- Activities and intermediate results ("stations")
- Six constructive activities
  - Each accompanied with analysis techniques for quality assurance
    - Functions
    - Effects leading to the functions
    - Carriers of these effects
    - Physical layout
    - Surface design
    - Checks with prototypes
- It correctly decouples a project into manageable activities, but looks like waterfall

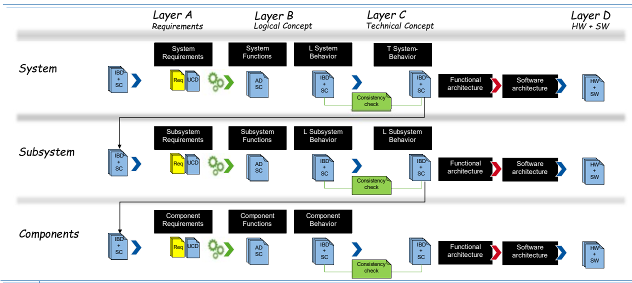


V-Model variant: the BMW SMArDT Process



From [DGH+19] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. von Wenckstern, A. Wortmann: SMArDT modeling for automotive software testing. In: Software: Practice and Experience, 49(2):301-328, Feb. 2019.

SMArDT Process Layers



Many Viewpoints

- Many Viewpoints for many stakeholders
- And many different modelling languages assisting these viewpoints
- This is why it is essential to ensure interoperability and consistency between models in a heterogeneous situation.
- Problem:
  - Tool manufacturers are not easily capable to achieve this
  - Manual transfer work is often needed
  - Consistency when evolving parts of the models?

MBSE

14. Software and System Development Methods
14.3. Models for Systems Engineering: An Overview of UML and SysML

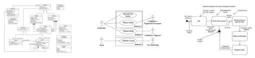
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

The Unified Modeling Language: Object-Oriented Modeling for Software-Intensive Systems

Features of the UML

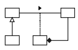
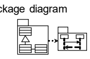
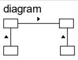
- Elements for specification, communication and documentation
  - among developers
  - developers with users
  - union of several previously existing methods
- Set of modeling concepts and concrete notations
- Standardized since September 1997 by OMG
- Developed by Booch, Rumbaugh, Jacobson, Selic, Kobryn, Cook and many others...



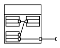
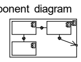
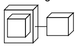
Goals of the UML

- Description of essential properties of a program (like a blueprint)
- Structuring of problem and solution
- Abstraction of implementation details
- Definition of various views:
  - task assignment and workflows
  - software and system architecture
  - interaction between components
  - behavior of components
  - implementation
  - physical distribution

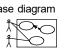
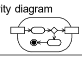
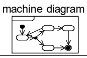
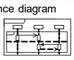
Overview of UML Diagram Types to Start

Diagram type	The central question answered by this kind of diagram	Strengths
class diagram 	Which classes form my system and how are they interrelated?	Describes the static structure of the system. Contains all relevant structural connections and data types. Bridge to dynamic diagrams.
package diagram 	How can I partition my program in order to retain an overview?	Logical group of model elements. Modeling dependencies/inclusion is possible.
object diagram 	What is the internal structure of my system at a specific moment at runtime. (snapshot)?	Displays objects and attribute values at a specific moment. Used as example for illustration. Level of details is the same as in the class diagram.

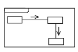
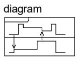
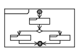
Overview of Diagram Types – 2

Diagram type	The central question answered by this kind of diagram	Strengths
composite structure diagram 	What is the inner structure of a class, a component, a part of the system?	Perfectly suited for top-down-modeling of the system (part-whole-relationship).
component diagram 	How are my classes aggregated in reusable, manageable components and in which ways are these components related to each other?	Shows the organization and dependencies of specific components of the system.
deployment diagram 	What is the operational environment (Hardware, Server, Databases, …) of the system? How are the components distributed at runtime?	Displays the runtime environment of the system with the 'tangible' system components. Presentation of 'Software Server' is possible. High level of abstraction, only few notational elements.

Overview of Diagram Types – 3

Diagram type	The central question answered by this kind of diagram	Strengths
use case diagram 	What does my system provide to its environment? (neighbor systems, stakeholders)?	External perspective of the system. Suitable for context identification. Strong abstraction, simple notation.
activity diagram 	How does a flow-oriented process or algorithm execute?	Very detailed visualization of processes with conditions, loops, branching. Parallelism and synchronization. Representation of data flow.
state machine diagram 	Which states can an object, an interface, a use case , etc accept and by which events are these states triggered?	Precise mapping of a state model with states, events, concurrency, conditions. Enter and exit actions. Nesting possible.
sequence diagram 	Who exchanges which information with whom and in which order?	Presentation of information interchange between communication partners. Accurate representation of the temporal order, including concurrency.

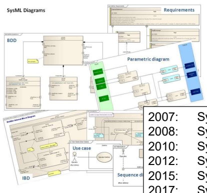
Overview of Diagram Types – 4

Diagram type	The central question answered by this kind of diagram	Strengths
communication diagram 	Who communicates with whom? Who is cooperating in the system?	Represents the exchange of information between communication partners. The focus is to give an overview. (Details and timing less important).
timing diagram 	When are interaction partners in which state?	Visualizes the exact timing behavior of classes, interfaces, protocols, ... Suitable for detailed observations, where it is important that an event occurs at the right time.
Interaction overview diagram 	How do interaction fit together?	Combines interaction diagrams (sequence, communication and timing diagrams) to a top-level. High level of abstraction.

Systems Modeling Language SysML

Rep.

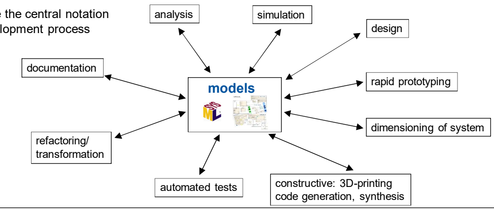
- SysML is dedicated to model the software part of (embedded) systems
- It started as variant of UML, but will probably become independent (with 2.0)
- SysML reuses 7 of UML's 14 diagrams, and adds 2 new diagrams
  - requirement and parametric diagrams



2007:	SysML 1.0
2008:	SysML 1.1
2010:	SysML 1.2
2012:	SysML 1.3
2015:	SysML 1.4
2017:	SysML 1.5
2019:	SysML 1.6
2023:	SysML 2.0

Model-Driven Development

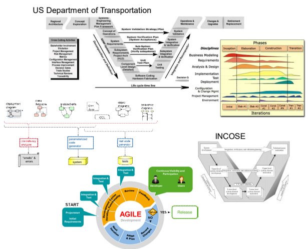
- Models are the central notation in the development process



⇨
- Models can serve as central notation for systems development
- A good modeling language can be used for analysis and synthesis

Summary

- Overview of methods
  - XP, Scrum, V-Model, RUP,
- Overview activities in a project
  - Analysis, design, implementation, testing + planning
- Agile, model-based systems development
  - Using SysML models
- Automation using tools
  - Generation
  - Consistency checking



Literature on UML

- **OMG UML 2 description** (www.omg.org):
  - Notation Guide, Semantics, Metamodel, OCL, Summary
- Martin Fowler, Kendall Scott: **UML Distilled**
- Desmond D'Souza, Allan Wills: Objects, Components, and Frameworks with UML, The **Catalysis Approach**
- Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, Stefan Queins: **UML 2.0 glasklar** (German)
- Martin Hitz, Gerti Kappel: **UML @ Work**

MBSE

14. Software and System Development Methods
14.4. Scrum-based Agile Methods

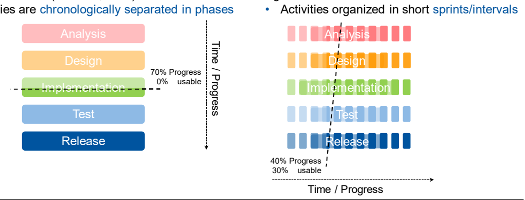
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Classical vs. Agile Processes

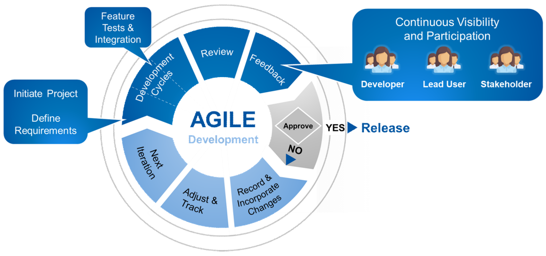
Waterfall Process (V-Model, RUP)
- Activities are chronologically separated in phases

Agile Process
- Activities organized in short sprints/intervals



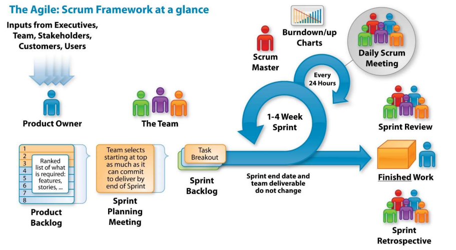
⇨ Agility and Evolution live from many small iterations

Development Processes … Scrum, XP

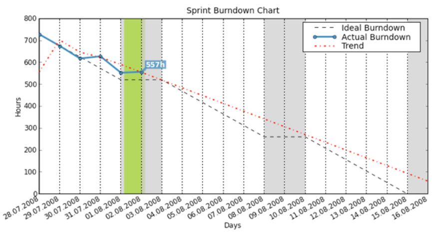


Scrum

- Artefacts:
  - Product Backlog, Sprint Backlog and Burndown Chart.
- Sprint
  - Time-boxed phase to develop the product (usually short)
- Sprint Backlog
  - List of User Stories (and bugs) to deliver



Scrum Burndown Chart



Burndown Chart shows actual state and desired plan of work progress within a sprint

Extreme Programming (XP)

- Light-weight, agile software development method
  - Comprises values, principles and practices
- Omits elements of software development:
  - Documentation, partitioning into (longer) phases
- Primary focus on
  - Source code, tests, communication
  - Short development iterations
- Needs: smaller teams, no life threats in product, available customer
- Consequence: XP cannot be used for every kind of project!

XP

Fundamental Principles of XP

- Fast feedback
  - Continuous project management
- Incremental Changes
  - No big-bang integration
  - Quantifiable progress
- High Qualitative Results
  - Ensured by different measures: testing, pair programming
- Simplicity
  - Clarity, elegance
- Support Changeability
  - To achieve flexibility and reduce costs of errors

XP

XP development best practices

Some of the most important:

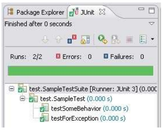
- Planning game
- 40 h. week
- Continuously available customer
- Small releases
- Continuous integration
- Simple design
- Testing
- Refactoring
- Pair programming
- Common code ownership
- Rigorous coding guidelines

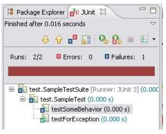
→ Observe: XP is a very rigorous process

XP

Testing in XP

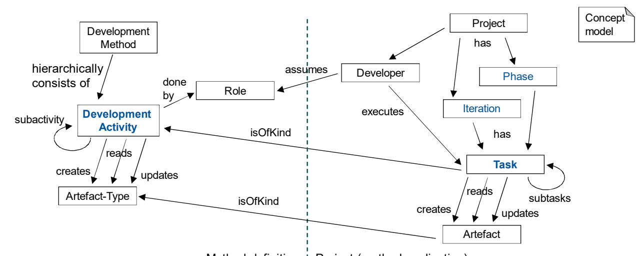
- … is most important in XP!
- Parallel development of code and tests
  - Best is "Test first"!
- Only fully automated tests
  - Setup of test
  - Execution
  - Evaluation of result to "green" or "red"
  - And: avoid manual test and debugging
- e.g. using junit, cppunit, etc.
- For all languages incl. Simulink available
- Run the tests every commit & every night
- In systems engineering: Test with simulations.

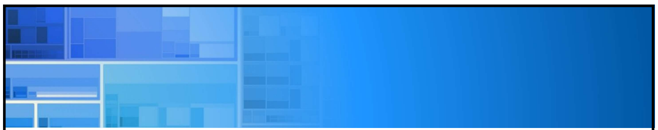




Summary: Various Development Methods fit Specific Project Needs

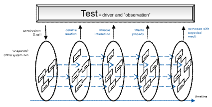
Rep.





MBSE

15. Testing and Simulation
15.1. Model-Based Testing



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

What is Testing? Definitions:

- *Testing is the process of executing a program with the intent of finding errors.*
  (Myers: The Art of Software Testing '79)

- *Software testing involves executing an implementation of the software with test data and examining the outputs of the software and its operational behavior to check that it is performing as required.*
  (Sommerville: Software Engineering '19)

- The term test means the process of planning, the preparation and the measurement, with the aim of determining the characteristics of a system and to demonstrate the difference between the current and the required condition.

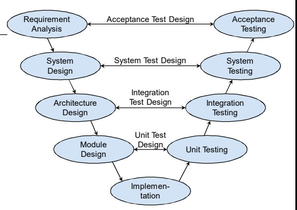
Testing Activities

- A test executes the system under test.
- A test is exemplary.
- A test is repeatable and determined.
- A test is goal-oriented.
- A test of a modified system can show behavioral equality with the original system exemplarily.
  - i.e. Regression testing
- A collection of tests form a software system on their own, which runs in conjunction with the system under examination.
- Tests should be automated.
- An automated test performs
  - (1) the setup of the test data,
  - (2) the test and
  - (3) examination of the test result.

  Success or failure of the test are detected and reported by the test run (green light).

Test Levels

Test type	Who created the test (or executes it)?	Test candidate
acceptance test	users, mostly interactive	the installed product
system test ("acceptance test")	test team with the help of users	the instrumented production system in the test environment
subsystem test ("integration test")	test team, developers	subsystem
component test ("class test", "module test", "unit test")	developers, test team	component, class
function test, method test	developers	function, method



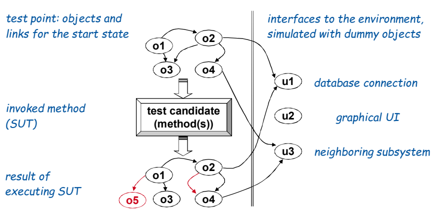
Terminology: Error

- Failure is the inability of a system or component to provide a required functionality within the specified limits.
- Failure manifests through wrong output, incorrect termination, or violation of time and storage conditions.
- Fault is a missing or incorrect code.
- Error is an action taken by the user or an environmental system that causes a failure.
- Omission is the lack of required functionality.
- Surprise is code that does not support any required functionality and is therefore useless.

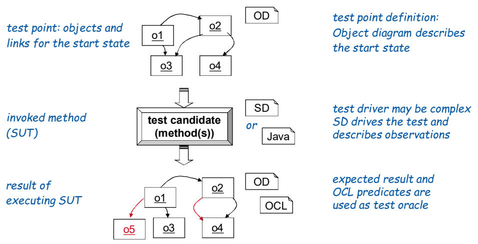
Terminology: Test

- Test object = system under test (SUT), system to be tested, test item, testee
- Test procedure: method how to create and carry out tests
- Test point / test data: concrete set of values for the input of a test, including object structure and objects to be tested
- Expected test result: the expected outcome of a test.
- Test case: description of the state of the test object to be tested and the environment before the test, the test point and the test result (includes test point + expected result).
- Test suite: set of test cases
- Test run: execution of a test with actual results
- Test driver organizes the test run from the setup of the test data until the examination of the success of the test.
- Test success iff actual result and expected result are compliant. Otherwise, the test has failed.
- Test verdict: statement on whether the test succeeded or failed.

Structure of a Software Test

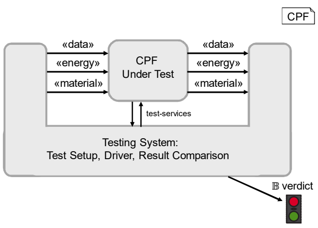


Structure of a Software Test



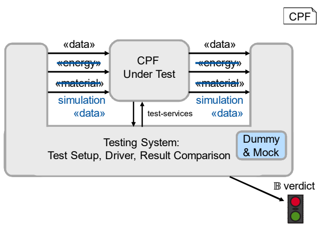
Structure of a CPF Test

- Test driver etc. is also CPF
  - test service channels help to setup the CPF Under Test
- A test provides a sequence of data, energy material to the interface
- Energy and material and their components need to be simulated
  - Data can remain as is, but may be transport and some processing components are simulated as well
- Output sequence is checked accordingly
  - test service channels are used to check internal state
- Test result: typically also contains detailed results
- CPF itself can be composed or an atomic function
  - Various forms of subsystems can be tested



Dummies, Mocks

- Critical
  - interfaces to the environment of the test candidate
- These are simulated by dummies
  a) to prevent side effects
  b) to check correctness of access
  c) to audition prepared results
- Dummies substitute real environment
  - database
  - neighboring systems: physical components
  - GUI
- Mocks are dummies with some intelligence (e.g., answering requests intelligently)
- Mocks can be part of testing system, but also replace part of the CPF Under Test.

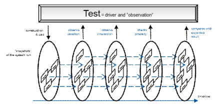


Effects of Automated Tests and Simulations

- Test with failure of the SUT documents an error.
- If all tests are successful:
- Developers' confidence in their own development and the results of colleagues is significantly higher.
- Increased self-confidence of a developer to adapt someone else's models.
- Knowledge about the system functionality is stored in repeatable tests.
- Comprehensive set of test cases and system specification are two models of the system.
- Testing effort is reduced. Manual regression tests and simulations would be too costly in the long run.
- Detailed test collection documents the quality of the system for the customer.

MBSE

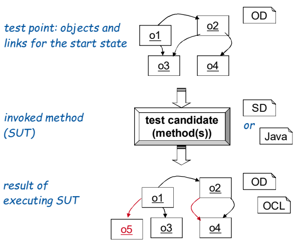
15. Testing and Simulation
15.2. Object Diagrams to Model Test Data



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

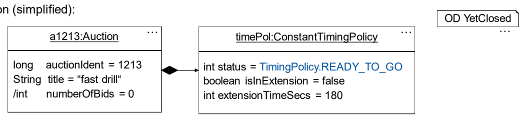
Structure of a Software Test



- Test data definition:
  - OD describes the start state
  - test data is sometimes complex, but only small number of them is needed. Highly reusable (if tuned to specific needs)
- Test driver may be
  - complex SD that drives the test and describes observations (usually compact, short)
  - or just program code to call the desired methods
- Test oracle
  - OD describes the expected (changed) result possibly refined by OCL predicates
  - Result OD needs to models differences only (i.e. usually also compact)
- Reusability of diagrams allows effectivness

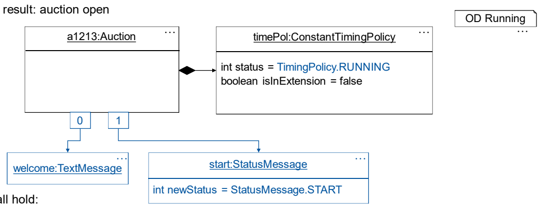
Example: Opening an auction - 1

- Initial situation (simplified):



Example: Opening an auction - 2

- Expected result: auction open



- And it shall hold:

OCL

```
context Auction a inv NoBidYet:
  { m in a1213.message | m instanceof BidMessage }.isEmpty
```


Example: The Test

- Description of a test

Test

```
test object:   Auction.start();
test data:     OD YetClosed;
driver:        a1213.start();
assert:        OD Running;
               inv NoBidYet; inv Bidders1;
```

- The generated code

Java/P

```
testStart() {
  Auction a1213 = setupYetClosed();        // generate test data
  a1213.start();                           // run the test
  assert isStructuredAsRunning(a1213);     // expected results met?
  checkNoBidYet(a1213);                    // property NoBidYet
  checkBidders1(a1213);                    // invariant Bidders1
}
```

- The diagrams and OCL are implemented as discussed.

General Test Case Structure

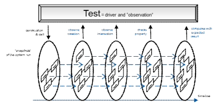
A test with all its elements can look like this. Often tabular representations are used as well:

test NameOfTest {
- **name**: *AuctionTest.testBid*
- **test data**: *object diagrams prepare the test data*
- **tune**: *Java code allows fine tuning of the test data*
- **driver**: *Java method call(s) or sequence diagram*
- **methodspec**: *OCL method specification checked for the method invocation*
- **interaction**: *sequence diagram used for monitoring the execution*
- **oracle**: *Java method call or Statechart produces expected results; these are compared to real results*
- **comparator**: *Java-Code | OCL-Code compares actual and expected results*
- **statechart**: *Statechart + an expected path are checked*
- **assert**: *object diagrams | OCL conditions | Java test code check actual result*
- **cleanup**: *Java code cleans up used resources (e.g. data base)*

}

MBSE

15. Testing and Simulation
15.3. OCL Invariants and Method Specifications

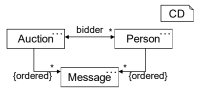


Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

OCL Invariants are Used as Code Instrumentation

- Example: Java method, extended by invariants:



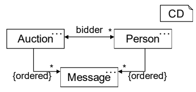
Java/P

```
class Auction {
  addMessage(Message m) {


    message.add(m);


    for (Iterator(Person)
            ip = bidder.iterator();
            ip.hasNext();){
      Person p = ip.next();
      p.receiveMessage(m);
    }

  }
}
```

discuss

OCL Invariants are Used as Code Instrumentation

- Example: Java method, extended by invariants:



Java/P

```
class Auction {
  addMessage(Message m) {
    ocl !this.message.contains(m);

    let int oldMessageSize = message.size;
    message.add(m);
    ocl message.size == oldMessageSize +1;

    for (Iterator(Person)
            ip = bidder.iterator();
            ip.hasNext();){
      Person p = ip.next();
      p.receiveMessage(m);
    }
    ocl forall p in bidder: m in p.message;
  }
}
```


Code Instrumentation by Invariants

Java/P

```
class Auction {
  addMessage(Message m) {
    ocl !this.message.contains(m);

    let int oldMessageSize = message.size;
    message.add(m);
    ocl message.size == oldMessageSize +1;

    for (Iterator(Person)
            ip = bidder.iterator();
            ip.hasNext();){
      Person p = ip.next();
      p.receiveMessage(m);
    }
    ocl forall p in bidder: m in p.message;
  }
}
```

- ocl-keyword is similar to the assert keyword in Java but followed by OCL conditions
- Implementation of OCL conditions as
  - assert (in normal code),
  - JUnit statement (tests), or
  - simply omission in the production code
- Code instrumentation is especially effective in combination with lots of covering tests
  - these extensively test the OCL invariants
- Invariants also give hints on where (more) tests should be defined, e.g., for boundary values

Methods Specifications (Pre-/Postcondition)

- Method specification can be used for testing.
- Code instrumentation:

OCL

```
context MyClass.method()
let   type a = value;
pre:  condition1;
post: condition2
```

Java

```
class MyClass {
  method() {
    // method body
}}
```

Java/P

```
class MyClass {
  method() {
    let type a = value;
    ocl condition1;
    // method body
    ocl condition2;
  }
}
```

- Problems with this approach
  - code instrumentation may not be possible because source is not available
  - returns in the method body must be treated separately

Methods Specifications (Pre-/Postcondition)

- Using subclassing: then instrumentation is not necessary

OCL
```
context MyClass.method()
let   type a = value;
pre:  condition1;
post: condition2
```

Java
```
class MyClass {
  method() {
    // method body (without return)
}}
```

Java/P
```
SubClass extends MyClass {
  method() {
    type a = value;
    ocl condition1;
    super.method();
    ocl condition2;
  }
}
```

- Necessary:
  - subclass objects to be instantiated,
    - use of a replaceable factory/builder (design pattern)
  - do not use static methods.

Test Cases Derived from Method Specification -1

- Basic idea:
  - analyze method specification to discover cases that should be tested
- E.g. each clause of a disjunction of the precondition should be tested as a separate case
  - This identifies two cases
- Additional case (3):
  what happens when a precondition (i.e. the disjunction) is not fulfilled at all?

OCL

- Example: changeCompany
```
context Person.changeCompany(String name)
pre: company.name == name ||
     forall Company co: co.name != name
```
- Case 1:
```
company.name == name
```
- Case 2:
```
forall Company co: co.name != name
```
- Case 3:
```
company.name != name &&
exists Company co: co.name == name
```


Test Cases Derived from Method Specification -2

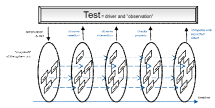
- In addition: use of postconditions
- Each case of the postcondition should be tested.
  - appropriate test data must be found!
- Often, the boundary values and their neighbors are of interest:
  - Classical boundary cases:
    empty string, null, 0, empty containers
- In this example, meaningful test data are:
  -n, -1, 0, 1, n (n large)
- More about this techniques in test lectures!

OCL

- Example:
```
context int abs(int val)
pre:  true
post: if (val>=0) then result ==  val
                  else result == -val
```
- Case 1:
```
val>=0
```
- Case 2:
```
val<0
```
- Data covering these cases:
  `-n, -1, 0, 1, n` (n large)

MBSE

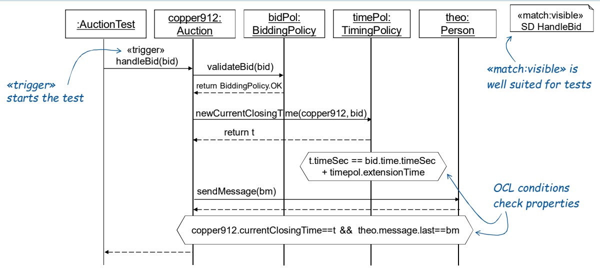
15. Testing and Simulation
15.4. Sequence Diagrams



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

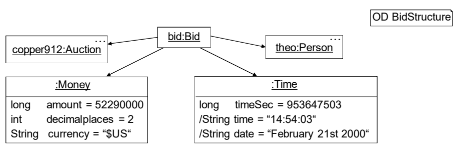
Sequence Diagram as a Test Case Description



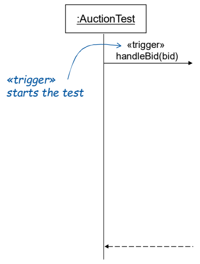
... And Associated Test Data

- Test case description:
  test object: Auction.handleBid(Bid bid)
  test data: OD copper912 && OD BidStructure;
  driver: SD HandleBid;
  assert: ...

Test



Sequence Diagram as Test Driver



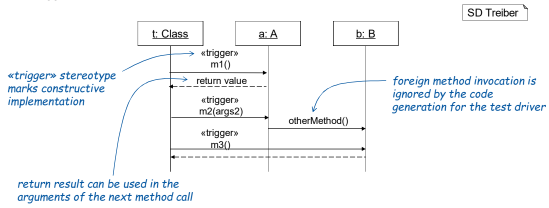
- JUnit is suitable as framework for SD tests
- setUp includes creation of the objects
- trigger maps to a simple method call

- More about JUnit under: www.junit.org

Java
```java
import junit.framework.*;

public class AuctionTest extends TestCase {
  Auction copper912;
  Bid bid;
  public void testHandleBid() {
    setUp();
    copper912.handleBid(bid)
    // check assertions
    tearDown();
  }
}
```

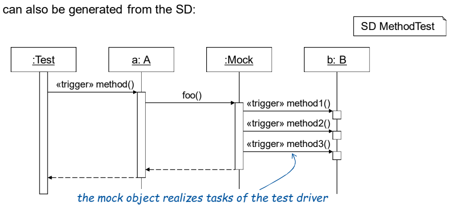

Complex Trigger - 1

- Multiple triggers require multiple method calls.



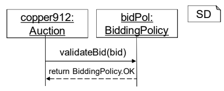
Complex Trigger - 2

- Triggers can be spread across several objects, this means a mock object is then be used in between.
- The mock object can also be generated from the SD:



Sequence Diagram as Observation

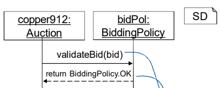
- When a part of the sequence diagram is to be observed as communication between the tested objects:



- Possible realization approaches
  - reflection / debugging API:
    not standard, not stable, thus better don't use it
  - instrumentation of object code
  - instrumentation of source code: only if available
  - instrumentation by subclasses

Monitoring of Calls

- Calls between the tested objects need to be observed. Here we use:
  - instrumentation by subclassing
  - including a monitor for tracking method calls, their orders and arguments:



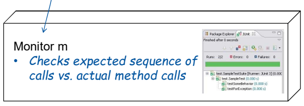
Java
```java
public class BiddingPolicyCheck extends BiddingPolicy {
  Monitor m;
  public BiddingPolicyCheck(Monitor m) { this.m = m; }

  public int validateBid(Bid bid) {
    m.callStarted(this, Monitor.ID_VALIDATE_BID, bid);
    int result = super.validateBid(bid);
    m.callEnded(this, Monitor.ID_VALIDATE_BID, result);
    return result;
} }
```


Monitor for Observation

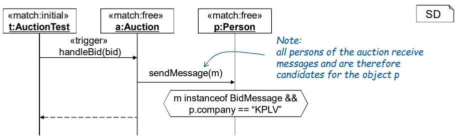
Main characteristics of the monitor:

- Calls and returns are registered at the monitor
  - arguments are: caller, method identifier, arguments
- Examined are
  - order of the sequence of calls and returns
  - correctness of the arguments
  - fulfillment of the invariants
- Stereotypes «match:*» have impact on allowed observations :
  - «match:free» for example allows an observed object to communicate with several others in a similar manner

Java
```java
...
  public int validateBid(Bid bid) {
    m.callStarted(this, Monitor.ID_VALIDATE_BID,bid);
    ...
    m.callEnded(this, Monitor.ID_VALIDATE_BID,result);
  ...
```

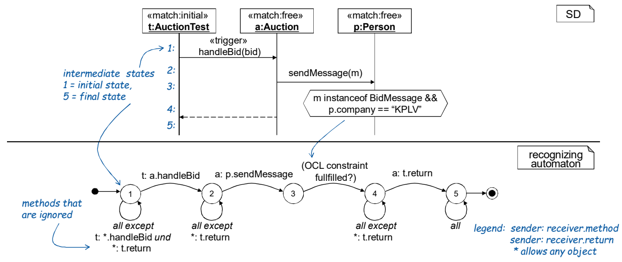


Monitoring a Sequence Diagram



- Because of «match:free» object p may be ambiguous, because several persons may receive m
- The following OCL condition on p may afterwards be invalid: this demands a sophisticated monitoring approach
- The efficient approach: Monitor uses a non-deterministic automaton to track the state, how far the sequence diagram has already progressed

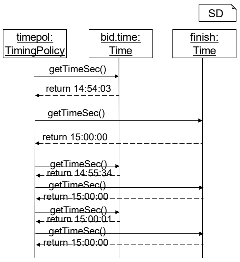
Deriving the Automaton for the Recognition Procedure



Sequence Diagram as Simulation Result

- Logs (protocols) can be represented as sequence diagrams
  - Both: derived from real systems and from simulations
  - Appropriate filters like «match:*»
    - on certain objects,
    - kinds of communication (resp. material flows), and
    - time frames

    reduce the length of the logs.
  - Appropriate visualizations and further aggregations are needed.
- SDs from logs are not necessarily causal, because they only describe observations

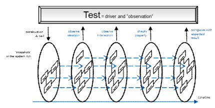


MBSE

15. Testing and Simulation
15.5. Statecharts

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

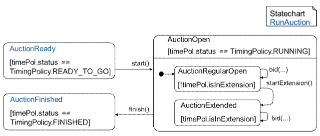


Applications for Statecharts

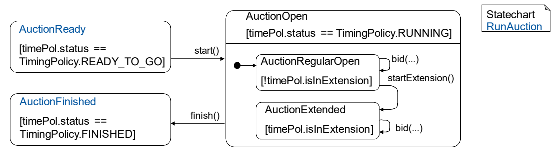
1: Constructive use of Statecharts for code generation
- typical: executable actions, rather detailed
- (already discussed in Statechart chapter)

2: Statecharts for tests
- typical: logic formulae as state invariants and transition postconditions
- → logics can be understood as state based method specifications

3: Statecharts as behavioral descriptions
- typical: few details, underspecified in various ways, abstracts from the real internal state
- → can be used
  - to check correct state transitions in test cases
  - or as a template for deriving test cases



Statechart to check a Control Flow



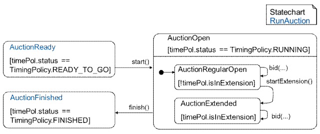
driver: auction.start(); auction.startExtension(); auction.finish();
statechart: auction RunAuction from AuctionReady to { AuctionFinished }

Test
...

observed object *name of the statechart* *initial state* *list of possible final states* *auction object is defined in an object diagram (not shown here)*

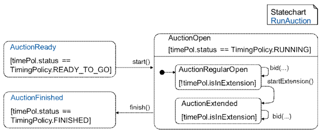
Test Coverage for a Statechart

- Coverage: How well does a set of tests cover possible behaviors?
- State coverage:
  - each state is traversed by a test
- Transition coverage:
  - each transition is traversed by a test
- Path coverage:
  - each path is traversed by a test
  - (but impossible when a loop is included)
- Minimal loop coverage:
  - acyclic paths + traverse through each loop once
- Further coverage criteria distinguish alternatives in conditions, invariants, ...
- Coverage can systematically be measured using a monitor.



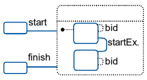
Deriving Tests from a Statechart

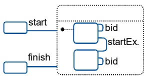
- Coverage can systematically be measured using a monitor.
- But it is also possible to systematically derive tests from a Statechart to reach the coverage:
- For each path:
  - calculate the test data that executes the path
  - i.e. an object structure including attribute values
- Strategies are:
  - Backward analysis: calculate attribute values from the desired result by stepping back along the transitions
  - Use symbolic execution (i.e. values remain abstract) along the symbolic computation
  - → related to verification (i.e. the symbolic manipulation in these tools)

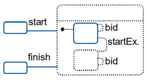


Example: Test Cases for the Auction Statechart

- State coverage requires only one test case
  Input: start; startExtension; finish
- Transition coverage and minimal path coverage coincide in this example:
- Two paths are sufficient, but are also necessary, because finish can be exited from both sub states.
  Input: start; bid; startExtension; bid; finish
  Input: start; finish
- Path coverage is not possible, because of two bid-loops, i.e. infinitely many paths of forms
  - start; bid;* startExtension; bid;* finish and
  - start; bid;* finish
  and their prefixes.



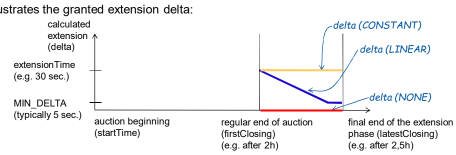




Appendix

Sample Task: Policies for Extending an Auction

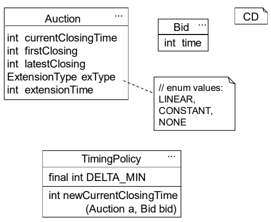
- Requirements: (1) If a bid is submitted just before the end of the auction, the auction will be extended by up to extensionTime seconds. (2) auctions always end up between firstClosing and latestClosing
- Three Policies:
  - NONE: there is no extension
  - CONSTANT: the extension is always the same delta.
  - LINEAR: linear decrease of extension, but at least MIN_DELTA.
- The graph illustrates the granted extension delta:



Appendix

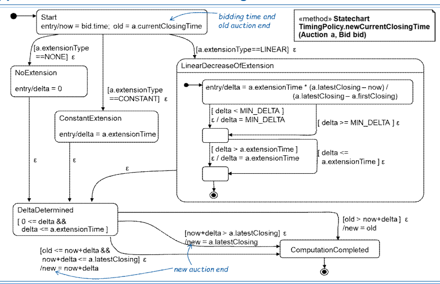
Exercise, Part 1:

1) Implement method newCurrentClosingTime that calculates the new closingTime in the structure given in the CD
2) Design a Statechart for the method representing the cases and variants over the control flow.
3) Identify a set of paths that achieves state, transition respectively path coverage.
4) Develop a set of test data for each path.
5) Test your implementation with each record.



Appendix

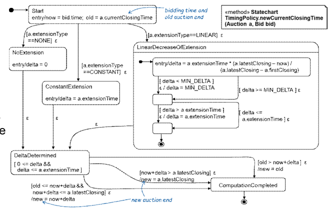
A Solution Approach/Oracle for newCurrentClosingTime



Appendix

Exercise, Part 2:

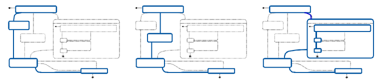
5) For this Statechart, identify a set of paths that achieves state, transition and path coverage.
6) Develop test data for each path
7) Derive/Generate an oracle from the Statechart
8) Test your implementation with each of the records and compare the actual result with the result of the oracle.



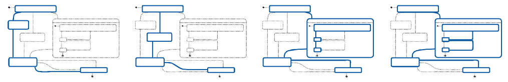
Appendix

Solution for Coverage - 1

- State coverage is possible with three paths:



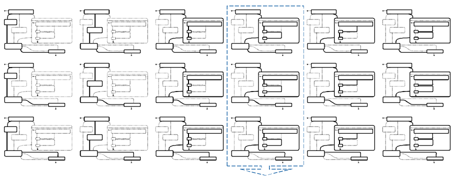
- Transition coverage has not yet been reached by this. However, four paths are sufficient:



Appendix

Solution for Overlapping - 2

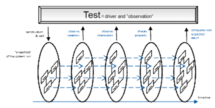
- Path coverage (identical to the minimal loop coverage, since no loop is present, 18 paths):



because of invariants in the algorithm, these paths cannot be taken by any chance and thus also cannot be tested

MBSE

15. Testing and Simulation
15.6. System Simulation



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

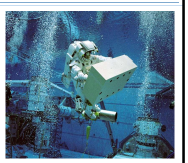
http://www.se-rwth.de/

Definition Simulation

A simulation is the imitation of the operation of a real-world process or system over time.

- Simulations require the use of models.
- Simulation is used in many contexts, such as
  - simulation of technology for performance tuning or optimizing, safety engineering,
  - testing,
  - training, education, and video games
  - scientific modelling of natural systems or human systems to gain insight
  - to show the eventual real effects of alternative courses of action

Human-in-the-loop simulation of outer space



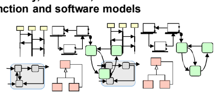
- Key issues in simulation:
  - relevant selection of key characteristics and behaviors used to build the model,
  - the use of simplifying approximations and assumptions within the model, and
  - fidelity and validity of the simulation outcomes.

(partly adapted from Wikipedia)

Simulation For Testing Model and System Quality

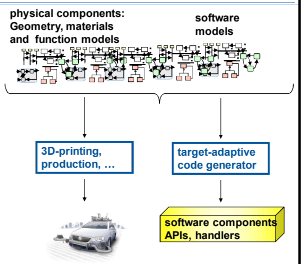
- Simulations require the use of models.
- Key questions that can be answered when simulating in engineering:
  - 1) Will the final system fulfill its requirements?
    - Given: requirements and models
  - 2) How will the system behave in specific situations?
    - Given: models and situation description of interest

Geometry, materials, function and software models

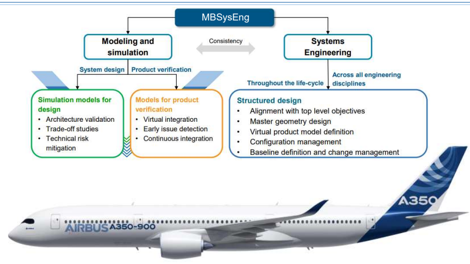


Simulation For Testing Model and System Quality

- 3) What is the quality of the designed models vs. fidelity and validity of the simulation outcomes?
- This question 3) has many facets, but this is especially relevant when automatic derivation of the system from the model exists, e.g.
  - in software with code generation
  - for systems with 3D printing
- i.e. when simulation and product partly coincide?
- 4) What is the quality of the designed models vs. correctness, reliability, security, safety, etc. of the generated outcomes?



Model-Based Systems Engineering at Airbus



Development of Autonomous Mobile Systems

- … on the example of participating in the DARPA GRAND CHALLENGE
- Design, implementation and quality assurance for
  - situation detection
  - situation classification
  - behavior generation
- For autonomous driving in sub-urban scenarios
- Team:
  - and Software Engineering

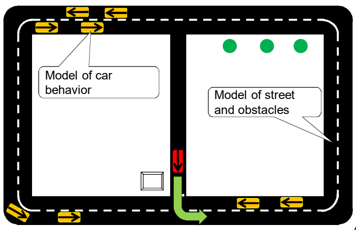
Conditions of the DARPA Urban Challenge

"Pass Californian driver's license test":
- Drive safely!
- 4 way stops, yield way, …
- Autonomous mission:
  - 60 miles in urban area

(Overview Testing Grounds)

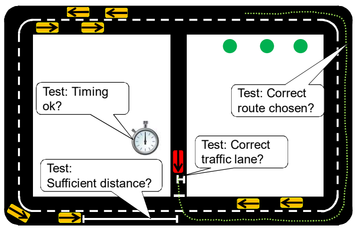
(4 Way Stop, Track C)

One of the Scenarios in the Semi Finals



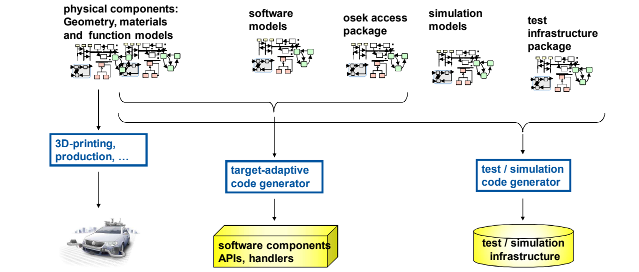
(Track A)

Assurance of Software Quality: Virtual Runs in the Simulator



Languages: DSLs for scenarios (CD+ layout+ Statecharts+ sequence diagram)
DSL for geographic & time-dependent traffic rules

Agile SysML-based Systems Development: Models for Code and Simulation

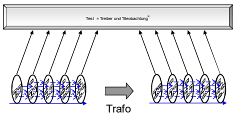


MBSE

16. Evolution through Model Refactoring
16.1. Evolution

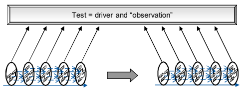
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Evolution

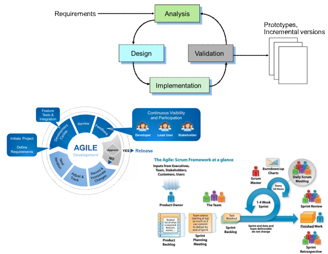
- Software must be adapted frequently:
  - new requirements
  - changed technology
  - new connections to neighbor systems
  - troubleshooting
- Techniques for the evolution of legacy systems, such as
  - reverse engineering: extraction of the original design models from the source code (object code)
  - wrapping: wrapping code of an older technology (Cobol, mainframe) into a modern access layer (Java, Web)
- Evolution traditionally consisted of one or a few large steps (transformations) with a high chance of failure



- Objective:
  - minimizing risk
  - increase developer effectiveness
- by:
  - small steps through systematic, manageable transformations
  - use of architecture and design given by models
- Prerequisite for quality-assured model-based evolution:
  - code generators
  - automated tests
  - library of model transformations

Evolution is an Intrinsic Concept in Agile Development Processes

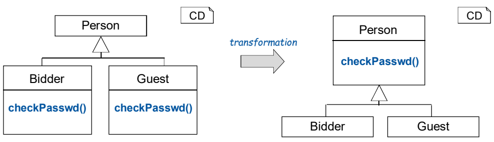
- Software must be enhanced frequently:
  - new requirements
  - changed technology
  - new connections to neighbor systems
  - troubleshooting
- And therefore modern development processes embrace evolution
  - minimizing risk
  - increase developer effectiveness
- through
  - small iterations
  - automated tests
  - refactoring techniques:
    transforming development artefacts (models, code)



Evolution of Models

- The goal of model evolution is the systematic transformation of a model to
  - improve the structure / architecture of a system while
  - maintaining the observed behavior

Example for a typical “refactoring”: subclass methods are generalized and moved into the superclass

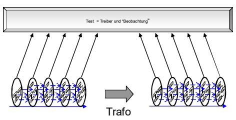


MBSE

16. Evolution through Model Refactoring
16.2. Principles of Refactoring

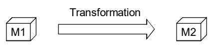
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Model Transformation

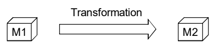
- A model transformation is a purposeful, executable mapping of a given model into different one.
  - Mapping is executable by a development tool



- Examples of transformations (that are not evolutionary, but helpful):
  - transformation of class diagram to Java
  - transformation of class diagram to SQL statements
  - extraction of analysis data
- Evolutionary transformations:
  - adding get/set methods
  - moving an attribute to another class
  - merging of two classes
  - minimization of statecharts
  - deletion of unused components in an architecture

Model Transformation

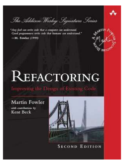
- A model transformation is a purposeful, executable mapping of a given model into different one.



- Properties of model transformations:
  - bijective (injective, surjective?)
  - bi-directionality?
  - abstracting (forgetting)?
  - adding details?
  - semantics preserving / refining?
  - within or between languages?
- Transformations within a language can be used for:
  - refinement / abstraction
  - normalization
  - evolution
  - information extraction
  - …

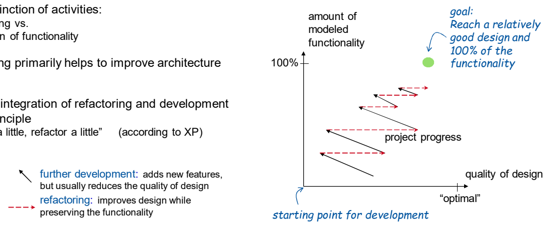
Refactoring

- Refactoring is a special case of transformations:
  - Fowler'99 uses refactoring on the code-level (Java)
  - refactoring was originally introduced in '92/93 by Opdyke /Johnson for C++
- Definition of refactoring [Fowler '99]:
  - Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code while improving its internal structure.
- Conclusion:
  - refactoring of models can be used for the evolution of systems.



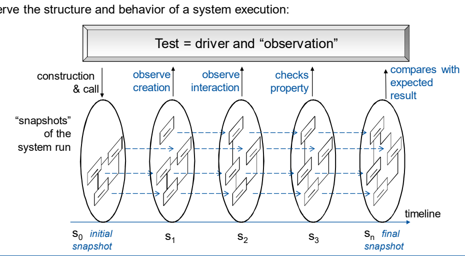
Methodology of Refactoring

- Strict distinction of activities:
  - refactoring vs.
  - extension of functionality
- Refactoring primarily helps to improve architecture
- But: tight integration of refactoring and development by the principle
  - "model a little, refactor a little" (according to XP)



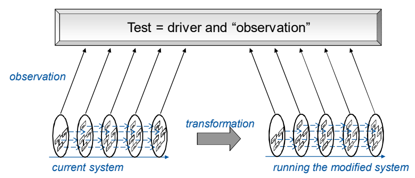
Tests are Observations for Evolutionary Transformations

- Tests observe the structure and behavior of a system execution:



Validation of Transformations

- The observation remains intact under the transformation.

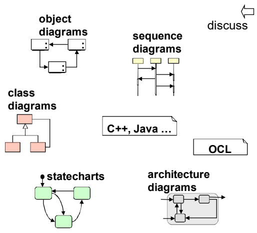


- But in practice: often structural parts are changed under the transformation
- Therefore: acceptance tests base on appropriate abstractions and fixed interfaces

Refactoring of UML / SysML Notations

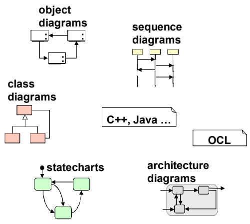
- Class diagrams
  –
- Architecture diagrams (IBDs)
  –
- Code
  –
- Object diagrams
  –
- OCL
  –
- Statecharts
  –
- Sequence diagrams
  –

discuss



Refactoring of UML / SysML Notations

- Class diagrams
  – architecture / design improvement: very worthwhile
- Architecture diagrams (IBDs)
  – architecture / design improvement: very worthwhile
- Code
  – cf. refactoring literature
- Object diagrams
  – necessary in the context of CD-transformations, but: rather unexplored
- OCL
  – logic has elaborated calculus, computation rules for containers, ...
- Statecharts
  – transformation rules for the simplification of statecharts
- Sequence diagrams
  – transformation techniques have not yet been developed



Example of a Transformation of Code

transformation source (here an expression with schema variable a)

Transformation

```
a + a
  ⇓
2 * a
```

|| expression *a* is free of side effects and deterministic

result

applicability conditions

- For both Java and OCL, the entire algebra is available, which is formulated as equations:
  – `a-a == 0,   a+b == b+a,    x && true == x`
- Data type-specific transformations, for example in OCL:
  – `List{a,b,c}.first == a`
- Procedural code often has applicability conditions of the form:
  term is defined | deterministic | side-effect free.

Applicability Conditions

- many transformations only apply under conditions, i.e. the applicability conditions:
- Example: replacement of "equal"

Transformation

```
a
⇓
b
```

|| 1. expressions *a* and *b* are free of side effects
2. *a* == *b*

- Example: replacing a method call

Transformation

```
... foo(x,y) ...
       ⇓
... bar(x,y,0) ...
```

|| when expression *x* has type *A*, *y* has type *B:*
1. **inv:**
```
    forall A a, B b:
      bar(a,b,0) == foo(a,b)
```


Applicability Conditions

- Expansion of a method body:
  – analogy to the compiler principle of method "inlining"

Transformation

```
... a.getX()
      ⇓
... 2 * a.getY()
```

|| when a is of type A
1. **class** `A { ...`
```
     getX() {
        return 2 * getY();
   }}
```
2. `getX()` is not redefined in any subclass

- When allowing redefinition of getX() in subclasses applicability conditions become more general, but also more complex
  – e.g., redefinition of getX() only within limits
- Q: What about checking the applicability conditions?

Correctness of Applicability Conditions

- Treatment of applicability conditions:
  – automatic check based on the syntax:
    - examples: strong typing system,
      correct initialization of variables, ...
  – semi-automatic check:
    - examples: model checking for system properties
  – interactive verification:
    - examples: correctness proofs in first-order logic e.g. based on a verification tool like MontiBelle
  – test:
    - example: verification of invariants at runtime
  – manual reviews:
    - reviewer gives his/her "OK" on being "confident"

Transformation

```
... foo(x,y) ...
       ⇓
... bar(x,y,0) ...
```

|| when expression *x* has type *A*, *y* has type *B:*
1. **inv:**
```
    forall A a, B b:
      bar(a,b,0) == foo(a,b)
```

Transformation

```
a + a
  ⇓
2 * a
```

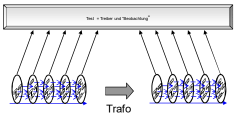
|| expression *a* is free of side effects and deterministic

MBSE

16. Evolution through Model Refactoring
16.3. Refactoring of Class Diagrams

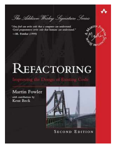
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Sources of Refactoring Rules

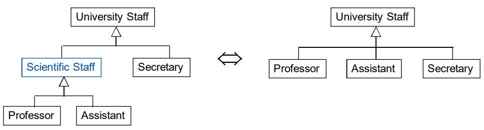
- Opdyke'93: 26 basic rules for C++:
  - often deleting and creating new program elements
- Fowler'99: 72 rules for Java:
  - many of them explained using class diagrams
  - 68 small refactoring, manipulating some of Java elements
  - 4 "big refactorings"



- Excerpt from the List of Refactorings (Fowler'99)
  - Add Parameter
  - Collapse Hierarchy
  - Encapsulate Collection
  - Extract Interface
  - Extract Method
  - Move Field (=Attribute)
  - Move Method
  - Pull Up Field
  - Remove Middle Man
  - Remove Parameter
  - Rename Method
  - Replace Array with Object
  - Replace Conditional with Polymorphism
  - Replace Delegation with Inheritance
  - Replace Inheritance with Delegation
  - Replace Error Code with Exception

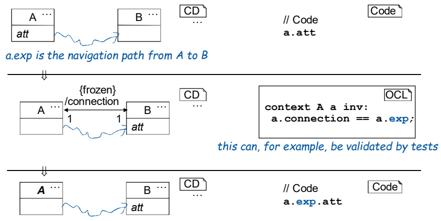
Refactoring "Collapse Hierarchy"

- Removing a class in the class hierarchy
- The rule can be applied in both directions
- When removing: inherited code is moved into subclasses
  - special case to handle: subclasses override method and call "super()"
  - special case to handle: constructors



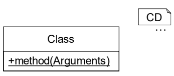
Example: Moving an Attribute

- Attribute "att" shall be moved from class A to B



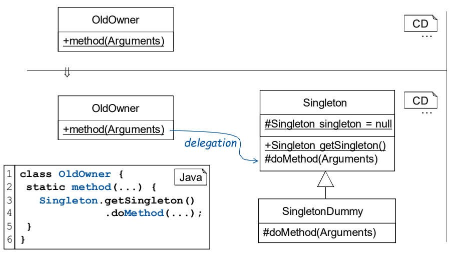
Refactoring: Introduction of a Test Pattern for Static Methods

- Problem to be addressed:
  - class has a static method
  - method has side effects
  - structure is there not suitable for testing, because method should be mockable in tests
- Solution: by transformation of the structure in three refactoring steps
  - 1) Replace Static Method with Singleton
  - 2) Mock the Singleton through Subclass
  - 3) Migrate Static Method for Encapsulation of the Singleton
- Applicability:
  - Static methods in general, e.g. protocols, logs, DB access
  - Constructors (see also factory / builder pattern)



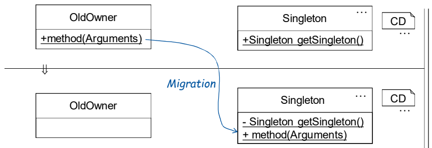
Replace Static Method with Singleton

- Method delegates its task to a singleton object
- `SingletonDummy` overrides the method in question thus allowing to mock the behavior in tests



... and Migrate Static Method for Encapsulation of the Singleton

- Encapsulation of the calling mechanism in the singleton
- … to prevent the users to have to cope with the object at all:
  - Users still have static method available

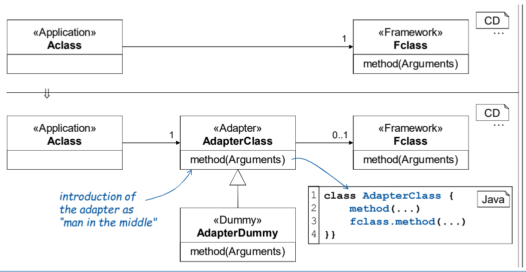



```java
class Singleton {                 Java
 static method(...) {
   getSingleton().doMethod(...);
} }
```


Refactoring: Decoupling Application – Framework

- Problem to be addressed:
  - the application uses a framework
  - framework has side effects / DB, GUI, Web, ...
  - use of the framework objects is inappropriate for testing, because of the side effects
- Solution:
  - decoupling by using an adapter as mediator
- Applicability:
  - Any kind of normal framework (non reflective)

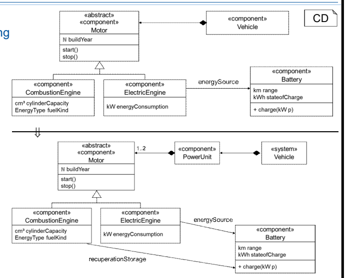
Decoupling with Adapter




```java
class AdapterClass {              Java
  method(...)
  fclass.method(...)
}}
```


Large Refactorings

- … are complex transformations that require planning
  - ideally, cut into small systematic steps
- Examples
  - separate domain from representation (Fowler)
  - convert procedural design to OO (Fowler)
  - decoupling of a complete application from a framework (GUI, Middleware, ...)
  - complex changes of structure (below)



Example: Changing a Data Structure

Action steps:

1. identify old data structure
   here: 'long' replaced by 'Money'

SellItem	…
long valueInDM	

2. Add new data structure (DS) + queries
   + compile & test

SellItem	…
long valueInDM	
Money value	

3. define invariants to relate both DS:

   ```
   context SellItem inv IV:
     valueInDM ==
           value.asDM()
   ```

based on

4. Add code for **changing** new DS wherever the old DS is **changed** + compile & test

   ```
   valueInDM = ...
   value.set(...)
   assert IV
   ```

5. Adjust code **using** the old DS to use new DS now
   + compile & test

   ```
   = ... valueInDM ...
          ⇓
   = ... value.asDM() ...
   ```

6. Simplify + compile & test
7. Remove old data structure
   + compile & test

SellItem	…
Money value	

State of the Art in Refactoring

- Extreme Programming provides the methodological foundation
- Eclipse, JUnit etc. provide techniques that assist refactoring
  - test case definition, execution, management
  - measuring "code smells" (metrics)
  - support for simple and increasingly many big refactorings
  - generation of dummies and mock objects for testing
  - status: still much to do!
- MDA provides methodology for model-based software development
  - code generators
  - transformation languages
  - status: MBSE tools are still improvable.

XP

eclipse

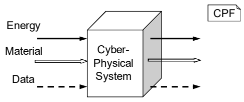
JUnit 5

MBSE

17. Functions Modelling Mechanics
17.1 Basics

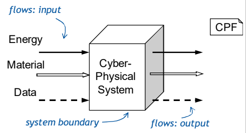
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

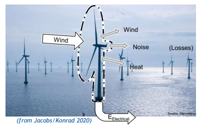
http://www.se-rwth.de/



System Specification through Functions

- A system defines a function
  - This includes mechanics and digital parts
  - A function describes the task of a product in a solution-neutral manner
- Advantages:
  - A) Mathematically very precise foundation
  - B) Function composition
  - C) Powerful modelling concepts
- In Mechanical Engineering there exist several design catalogues that list elementary functions and map them to physical effects
- Catalogue of mechanical functions and physical effects considered in this lecture:
  [KK98] Koller, R., Kastrup, N. *Prinziplösungen zur Konstruktion technischer Produkte*. Springer, 1998





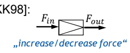
Elementary Function

- Function is a mathematical construct relating input and output (including history)
  - consequence: use of streams to model channels
  - functions can be composed

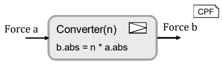
An elementary function is atomic in that sense that it is not further decomposed.

- [KK98] have shown that in Mechanics a catalogue of elementary functions can be defined
- The informal descriptions there usually contain
  - a picture or a verb + noun describing the I/O-relation
  - channel types identified through the arrow-kind and the variable name
  - often corresponds to a mathematical equation

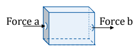
- Example from [KK98]:



- Modelled as Cyber Physical Function:

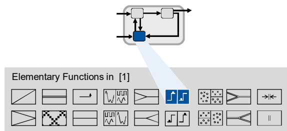


- Modelled as geometric object with two pointwise interfaces:

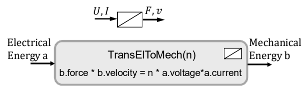


Catalogue of Elementary Functions

- [KK98] have also shown that a catalogue of elementary functions is useful in Mechanics
- Roughly 350 elementary functions, such as:
  - transform electrical to mechanical energy
  - apply mechanical energy on fluid
  - connect two solid materials
  - conduct force / light / current / …
  - … (many examples will be discussed in the following)
- An elementary function describes the functional interface, but not the physical (geometrical) realization:
  - for their solution we use *physical effects* discussed later

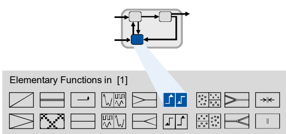


Example: „transform electrical to mechanical energy“



Catalogue of Elementary Functions in Software?

- [KK98] has ~350 elementary functions, which are successfully in use
- Software Engineering does not have a catalog of elementary functions, but also “building blocks”:
  - Math functions (+, *, pow)
  - Container operations (list, set, map operators)
  - OO methods? (but we define fresh ones all the time …)
  - Classes in a library?
  - Design patterns, such as factory, adapter, state pattern, … roughly 150, number increasing
- Software developers constantly create new elementary functions in OO methods and classes, which they compose to large systems.

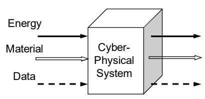


For software?

Signature of Input and Output of a Function

Rep.

- The signature of a function describes the forms of interactions of a system component with its environment.
- Interactions are broken down to streams of elements, which describe the flow and can be of the kinds
  - data,
  - energy or
  - material
- Interactions are organized through input and output channels.
- The Interface of a Cyber-Physical System is defined through its function signature



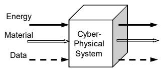
Stereotypes for CPF and their Interaction Channels

Rep.

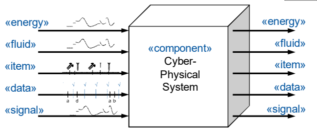
- We in this course define the following:
- for functions
  - «component» machinery, …
    - «system» machinery that is "complete"
  - «software» software (only), …
  - «being» humans, …
- for communication/flow channels:
  - «material» elements, compounds, alloys, …
    - «fluid» continuously flowing material, typically not countable (water, gas, sand)
    - «item» discrete physical items, e.g. cars
  - «energy» types of energy
  - «data» for data objects, basic data (e.g. int)
    - «event» for discrete data that triggers behavior
  - «signal» for continuously flowing data

… and omit them when unambiguous.

- Principle picture:

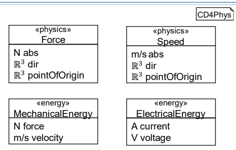


- is refined to:

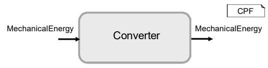


Modelling Types of Channels

- The input/output channels of a function are described with special datatypes that we model as special forms of classes
- We apply the interpretation of CDs for systems engineering
  - E.g., energy types model continuous flows → traditional interpretation is not suited
  - there is no classic "instance" and "identity" concept, only "values", but continuously many …
- Dependent on relevant properties e.g. Force may be modelled with or without direction, …
- A catalogue of common channel types, e.g. dependent on SI-Units, however, helps.



Classes to model flows of mechanical energy and electrical energy



Categories for the Catalog of Elementary Functions

- [KK98] categorizes elementary functions by the stereotypes of the function's channels and the kind of transformation performed by the function

Processing Energy (Inputs & Outputs are «energy»)
- Transform Energy
- Increase/Decrease
- Change Direction
- Conduct
- Isolate
- Collect
- Split
- Blend
- Separate

Processing Material (Inputs & Outputs are «fluid» xor «item»)
- Affix/Remove Materialistic Properties
- Increase/Decrease Values of Materialistic Properties
- Conduct/Isolate Material
- Mate/Unclamp Materials
- Blend/Split Materials
- Compound/Separate

Combining Material and Energy (Mixed Input/Output forms: «fluid», «item» and «energy»)
- Apply Energy to Materials
- Separate Energy from Material

Translational glossary for [KK98] (English - German)

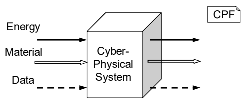
Elementary Function	Elementarfunktion
Design Catalogue	Konstruktionskatalog
Elementary Function	Elementarfunktion
Category of Elementary Functions	Elementaroperation
Transform	Wandeln
Collect	Sammeln
Split	Teilen
Blend	Mischen
Separate	Trennen
Affix	Hinzufügen
Mate	Fügen
Unclamp	Lösen
Principle Solution	Prinziplösung
(Physical) Effect	(Physikalischer) Effekt
Engineering Material	Werkstoff
Active Surface	Wirkfläche

MBSE

17. Functions Modelling Mechanics
17.2. Elementary Functions: Energy

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Energy Processing Functions

- Elementary functions in this category
  - have an interface with purely «energy»-channels
  - Perform one of the operations shown on the right
- These functions always obey energy conservation
  - Use this for specifying the behavior of these functions!
- Examples are:
  - Transforming electrical energy to mechanical energy using electromagnetic induction as effect (e.g., electric motor)
  - Increasing/decreasing moments of force using adhesion as effect (e.g. lever, gear box, wheel)
  - Split force/torque using the leverage effect (e.g., differentials)

electric motor (Electromagnetic Induction) · *differential (Leverage effect)* · *gear box (Adhesion Effect)*

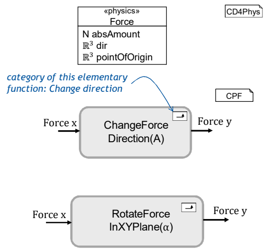
Processing Energy (Inputs & Outputs are «energy»)
- Transform Energy
- Increase/Decrease
- Change Direction
- Conduct
- Isolate
- Collect
- Split
- Blend
- Separate

Special Types in Energy Processing Functions

- Since it is unusual to work with energies directly in mechanical equations, [KK98] also allows the following types as inputs for Energy
  - Accelerations
  - Geometry descriptions in an energy equation (e.g. area, diameter, distance)
  - magnetic flux density; field strength
  - electric flux density; field strength
  - Force, moment
  - frequency
  - Current
  - Alternating current
  - Light intensity
  - Sound pressure
  - Charge
  - Warmth Q; Temperature T
  - Voltage
  - AC voltage
  - Velocities
  - Wave length
  - Density

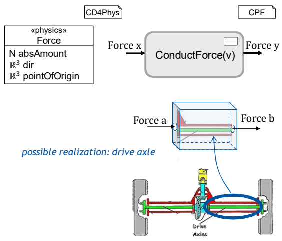
Elementary Function: Changing Force Direction

- We model Force as direction vector and point of origin
- A CPF may change this direction
  - Parameters: $A \in \mathbb{R}^{3\times3}$
  - Interface:
    - Input and ouput energy described by: Force x and Force y
- In general changing direction is modelled as linear transformation using matrix A
  - $\text{y.dir} = \text{A} * \text{x.dir}$
- Example: rotation around an angle α in the x-y plane realizes ChangeDirection with specialization:
  - $A = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$



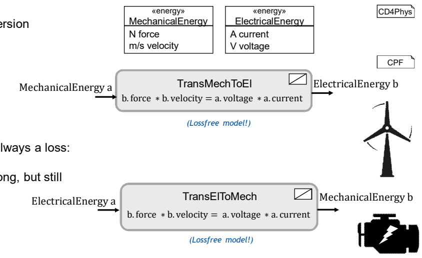
Elementary Function: Conduct Force

- Conducting force changes the point of origin of an incoming force
- Mathematically this corresponds to a translation:
  - Point of origin of the force vector is changed
  - Underspecify whether direction is changed
- We specify Conduct Force as follows:
  - Parameters: offset $v \in \mathbb{R}^3$
  - Interface: In/out as before
  - Behavior:
    - $\text{x.pointOfOrigin} = \text{t} + \text{y.pointOfOrigin}$
    - for given $A \in \mathbb{R}^{3\times3}$: $\text{x.dir} = \text{A} * \text{y.dir}$
  - We model it underspecified, whether the function also changes the direction of the force
- E.g. in a car, drive axles perform this function:



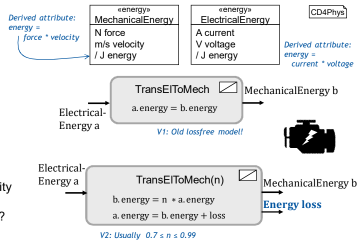
Elementary Function: Energy Transformation

- Energy transformation is a conversion
- Function interface:
  - Input Energy and type
  - Output Energy and type
  - Behavior:
    - law of energy conservation
- Tackling loss, because there is always a loss:
- A) Ignore the loss, deriving a wrong, but still useful model



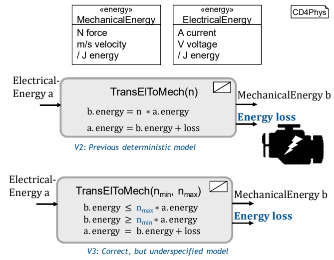
Energy Transformation with Loss (Deterministic Version)

- Tackling loss, because there is always a loss:
- B) Add loss as parameter (here: efficiency factor n):
  - Describing efficiency of the transformation
  - Observe: n is a parameter and thus fixed during operation
  - Thus, the equation is deterministic
  - and loss is fix over time.
- The model still is not completely correct (which n, n fixed?), but potentially closer to reality
- Open question: Where to connect channel loss?
  - E.g. as heat or out of the system ?



Energy Transformation with Loss (Underspecified, Correct Version)

- Tackling loss, because there is always a loss:
- C) Define loss as range (defined by parameters)
  - A range of efficiency values n_{min} and n_{max}
- The model is now underspecified:
  - A range of behaviors may occur
  - and thus many implementations are possible
  - deviations due to (uncaptured) system context influences can be taken into account
  - degeneration (over time) can be taken into account



Energy Transformation with Loss (Correct, Incomplete Version)

- Still tackling loss:
- Underspecification also allows to omit channels:
  - e.g. channel loss is omitted (1)
- This is a valid model
- But:
  - law of energy conservation cannot be stated anymore
  - (ok, let's live with it) (2)
- This allows various interpretations:
  - more channels are possible, but hidden (loss via wind, sound, heat)
  - or the lost energy is actually stored internally (e.g. as heat)

«physics» Power	«energy» MechanicalEnergy	«energy» ElectricalEnergy
W value	N force m/s velocity	A current V voltage

CD4Phys

Electrical-Energy a → **TransElToMech(n_{min}, n_{max})** → MechanicalEnergy b, Energy loss

$b.energy \leq n_{max} * a.energy$
$b.energy \geq n_{min} * a.energy$
(2) $a.energy = b.energy + loss$

V3: Correct, but underspecified model

Electrical-Energy a → **TransElToMech(n_{min}, n_{max})** → MechanicalEnergy b

$b.energy \leq n_{max} * a.energy$
$b.energy \geq n_{min} * a.energy$
~~a.energy = b.energy + loss~~ (1)

V4: Correct, but omitted output channel

Energy Transformation: Examples

- Other examples for energy transformation:
- (We use the simplified deterministic variants)
- An electrical heater transforms electrical energy to heat
  - Parameter: n (tells the efficiency)
  - Interface:
    - Incoming ElectricalEnergy a
    - Outgoing Heat b
- A light bulb transforms electrical energy to light:
  - Parameter: n (tells the efficiency)
  - Interface:
    - Incoming ElectricalEnergy a
    - Outgoing Light b
- Upcoming: There is an interesting dualism of light as energy carrier and light as a carrier of information!

«energy» HeatEnergy	«energy» ElectricalEnergy	«energy» Light
W heatflow	A current V voltage	W flux

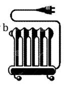
CD4Phys

CPF

ElectricalEnergy a → **TransElToHeat(n)** → Heat-Energy b

$b.heatflow = n * a.voltage * a.current$

Usually $n \approx 1$ [Wei09]



ElectricalEnergy a → **TransElToLight(n)** → Light b

$a.voltage * a.current = n * b.flux$

Usually $0.01 \leq n \leq 0.25$ [Wei09]

[Wei09] Weißberger, W. (2009). *Elektrotechnik für Ingenieure*. Vieweg + Teubner.

Electromagnetic Waves as a Carrier of Energy: Solar Generator

- Electromagnetic waves, such as light, carry energy
- In case of the sunlight this is due to nuclear fusion within the sun that releases energy
- Radiant flux (W) is the amount of energy emitted per unit of time
  - The radiant flux that is to be transformed depends on the geometry of the light-sender (e.g., the sun) and the surface it hits
  - Interface: Two contacts, one Data port
    - in: Light a
    - out: ElectricalEnergy b
  - Behavior:
    - $b.voltage * b.current = n * a.flux$

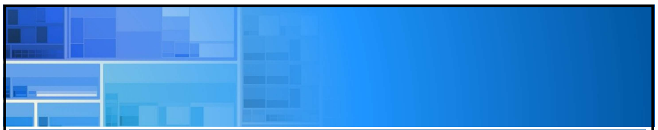
«energy» ElectricalEnergy	«energy» Light
A current V voltage	W flux

CD4Phys

CPF

Light a → **TransLightToEl(n, A)** → Electrical-Energy b

$b.voltage * b.current = n * a.flux$



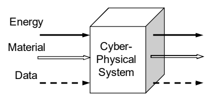
MBSE

17. Functions Modelling Mechanics
17.3. Elementary Functions: Material

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

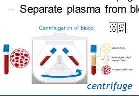
http://www.se-rwth.de/

CPF

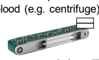


Material Processing Functions

- Elementary functions in this category
  - have an interface with purely «material»-channels (i.e. «fluid» or «item»)
  - Perform one of the operations shown on the right
- If the interface signature contains only «item»-channels the function exhibits discrete behavior
  - Very similar to the behavior of software functions!
- Examples are:
  - Mating physical parts made of metal (e.g. screw connection)
  - Conducting water (e.g. pipe), conducting screws (e.g., conveyor belt)
  - Blend water and oil (e.g. emulsion)
  - Separate plasma from blood (e.g. centrifuge)



centrifuge


conveyor belt


screw connection


pipes

Processing Material
(Inputs & Outputs are «fluid» xor «item»)

- Affix/Remove Materialistic Properties
- Increase/Decrease Values of Materialistic Properties
- Conduct/Isolate Material
- Mate/Unclamp Materials
- Blend/Split Materials
- Compound/Separate

Combination of the Car Body and the Engine

- Interface:
  - Two incoming material items are combined to form one outgoing material item
- Behavior:
  - The powertrain needs to arrive first
  - Once also the CarBody is there
  - The function builds the car from the two input items
- This is perfect for an object-oriented description
  - operating on material items
  - creating a new "object" of class Car
  - and attaching powertrain, and car body to it
  - Encoded as "method", e.g. c = buildCar(pt, cb)
- Please note: objects represent physical items, not data about them

«item» Powertrain	«item» CarBody	«item» Car
kg mass	kg mass	kg mass

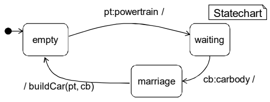
CD4Phys

CPF

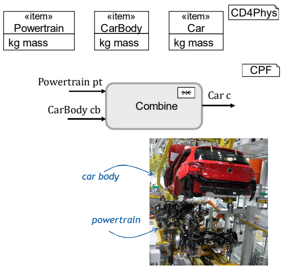
Powertrain pt, CarBody cb → **Combine** → Car c

Combination of the Car Body and the Engine: Behavior

- Behavior is specified by an automaton

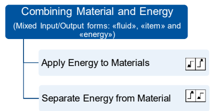


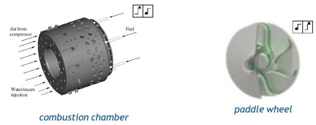
- describing:
  - powertrain and carbody are combined
- And also:
  - order of arrival: powertrain comes first
  - Statechart describes behavior of a component, but also obligations for the input



Functions that Combine Material and Energy

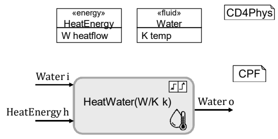
- Elementary functions in this category
  - have an interface with channels of mixed sorts i.e. «fluid» or «item» and «energy»
  - Perform one of the operations shown on the right
- Examples are:
  - Apply Water with Mechanical Energy (e.g., paddle wheel)
  - Separate Chemical Energy from Fuel (e.g. combustion chamber)





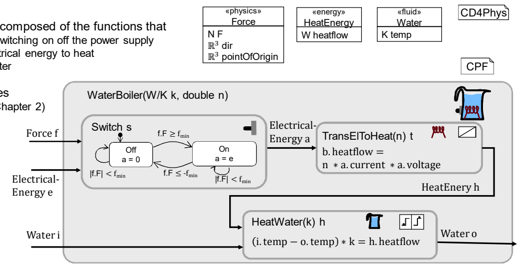
Apply Energy to Material: Heating up Water

- Heating up water:
  - apply thermal energy to the water
- Parameter:
  - Thermal conductance: W/K n
- Interface:
  - Incoming material: Water i
  - Incoming energy: HeatEnergy h
  - Outgoing material: Water o
- Behavior:
  - Generalization of the Law of convection that abstracts from geometry and materialistic properties:
    - $(o.temp - i.temp) * k = h.heatflow$



Functional Composition: Water Boiler

- A water boiler is composed of the functions that
  - Allows manual switching on off the power supply
  - Transforms electrical energy to heat
  - Heats up the water
- This model reuses
  - Switch s (from Chapter 2)
  - HeatWater(k)

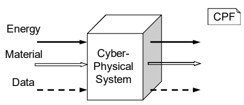


MBSE

17. Functions Modelling Mechanics
17.4. Elementary Functions: Sensors and Actuators

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

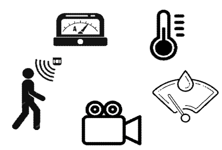


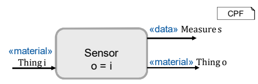
Sensors

- Broad definition of a sensor:

> [...] a sensor is a device, module, machine, or subsystem whose purpose is to detect events or changes in its environment and send the information to other electronics, frequently a computer processor. [Wikipedia]

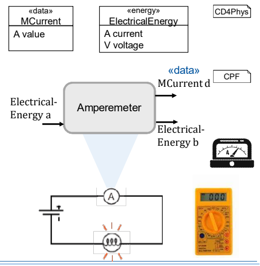
- Sensors
  - Read information from material or measure energy
  - Encode the information as data values or data objects
  - Sensors do not modify the measured things: o = I
    - (at least in this abstraction)





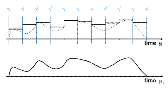
Ampere Meter

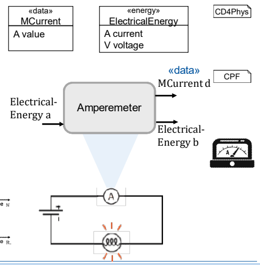
- An ampere meter measures a current
- Interface: Two contacts, one signal port
  - Input energy: ElEn a
  - Output energy: ElEn b
  - Output data: MCurrent d
- Behavior:
  - Defined by means of energy conservation:
    - $b = a$ (disregarding any losses)
  - Measurement result (ideally): $d.value = a.current$
- Note 1: Some deviation can be specified, e.g.
  - $|d.value - a.current| < 2\% * a.current +$ additiveMinDeviation



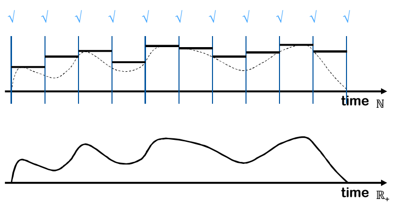
Ampere Meter and Digitalization

- Note 2: The data object uses ideal value i (real number) and is not digitized here.
  - a more implementation oriented spec could use another type like
    - A<float> value ampere measured as float
- Note 3: Channel d is discrete in time, while a, b are continuous
  - a precise definition encompasses
    - time
    - sampling frequency
    - measurement delay
    - + deviation



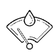


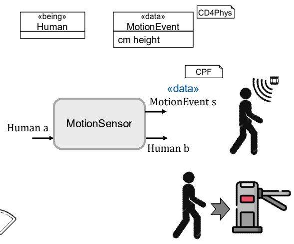
Vorlagen



Motion Sensor

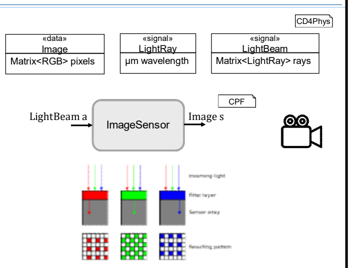
- Motion sensor detects movement within its area of reach
- Interface:
  - In/out: Human a
  - Out: Event that can be used for triggering some software, event can carry additional information e.g. height of person
- Behavior:
  - Humans remain unchanged: $b = a$
  - $a[t] \neq absent \Rightarrow s[t] = MotionEvent(\ldots)$
- Items or fluids, e.g., in production context, can be measured similarly





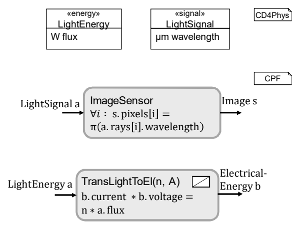
Light as A Carrier of Information: Image Sensor

- Image Sensors read information transmitted through light
- In this case, light acts as a carrier of information encoded in its wavelength (and we ignore energetic aspects)
- Function of the Image Sensor:
  - Interface: two contacts, one data port
    - In: Light a
    - Out: Image s
  - Behavior:
    - For π: $\mu m \rightarrow RGB$ from e.g. [Bru96]
    - $\forall i,j$: $s.pixels[i,j] = \pi(a.rays[i,j].wavelength)$
- Similar kinds of sensors: Microphones



Dualism of Light as Carrier of Information and of Energy

- Many items, forms of energy, can be model in various ways, e.g. light or also current
  - we model the relevant part and abstract the irrelevant
- A: Light carries information through its wavelength:
  - Encodes color
  - Information whether sb./sth. passes by
  - …
  - Image sensor reads the information encoded in a light signal and transforms it into data objects
- B: Light also transports energy through radiation, intensity
  - Transformers transform this energy e.g., into electricity



Actuators

- Definition of Actuator:

> An actuator is a component of a machine that is responsible for moving and controlling a mechanism or system, e.g., by opening a valve. […]
> An actuator requires a control signal and a source of energy. […] When it receives a control signal, an actuator responds by converting the source's energy into mechanical motion.
> [Wikipedia]

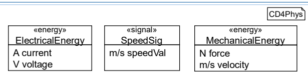
combustion and electric motor

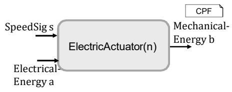
- Actuators
  - Transform an input energy
  - According to an input control signal or data

printer driver

Electric Motor Actuator

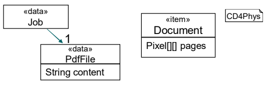
- The electric motor uses the input electrical energy to produce the power requested by the input signal
- Interface:
  - Incoming signal SpeedSig s
  - Incoming energy ElectricalEnergy a
  - Outgoing energy ElectricalEnergy b
- Behavior
  - The actuator produces the speed encoded in the signal:
    - $\mathrm{b.speed} = \mathrm{s.speedVal}$
  - By transforming the incoming electrical energy:
    - $\mathrm{b.force} * \mathrm{b.velocity} = \mathrm{n} * \mathrm{a.current} * \mathrm{a.voltage}$
- Parameter:
  - Efficiency of the energy conversion n
- Again with some simplifications / abstractions …





Printer

- The printer driver converts a file to be printed into a format that a printer can understand
- Interface:
  - Incoming data: Job printjob containing the file to be printed
  - Outgoing: Physically printed document
- Behavior:
  - d.text = printjob.f.content
- Abstracts from energy and paper sheets

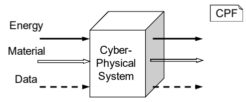


MBSE

17. Functions Modelling Mechanics
17.5. Elementary Functions: Transport

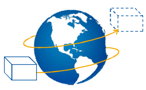
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

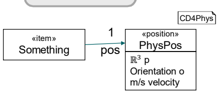
http://www.se-rwth.de/



Transport of Items

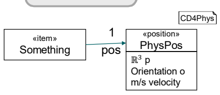
- Transport brings an item from one position to another
- Position is tricky …
- Position is usually modelled as state, but de facto it is not internal.
  - Component's context can experience the position of an item (e.g. see it)
  - Position is relative to a reference point, e.g. earth coordinates or relative to a point zero in a room
- The approach:
  - The incoming item has a position that is changed by the function to the desired destination
- PhysPos describes physical position, orientation, …





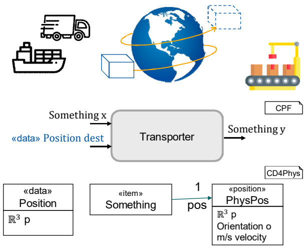
Transport Specification

- The approach:
  - The incoming item has a position that is changed by the function to the desired destination
- Transport interface:
  - Item in/out: Something x
- Only the destination changes:
  - $\mathrm{y.pos} \neq \mathrm{x.pos} \;\wedge$ rest remains unchanged
- We abstract from
  - The energy needed to transport x
  - The path taken by the function to transport x
  - The time needed
  - The load (multiple items in parallel?)



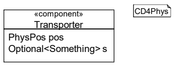
Transport With Destination Specification

- The approach:
  - Add as input the destination position (a data value)
- Transport interface:
  - Item in/out: Something x
  - In: «data» Position dest
- Item arrives at destination :
  $y.pos.p = dest.p$
- The approach can be used when item is passive
- The movement of the transporter component is not modelled yet

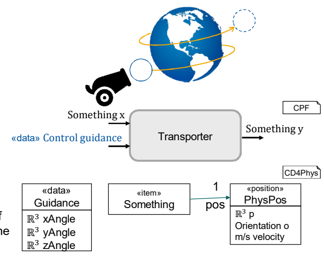


Transport With Destination Guidance

- The approach:
  - Add the element to be transport into the transporter together with guidance information leading to the destination
- Transport interface:
  - Item in/out: Something x
  - In: «data» Control guidance

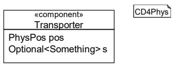


- A transport with destination specification is realized if the controller that calculates the guidance to reach the target is taken into the system boundaries

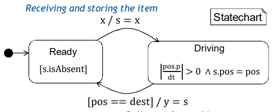


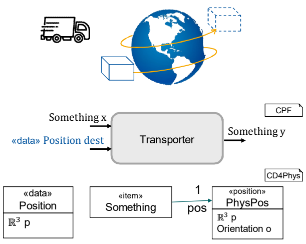
Mobile Transporter: Van

- The transporter has position and the transported item in its state:



- The behavior can be modeled by a hybrid automaton:





Example from Alur: Principles of Cyber-Physical Systems

Example: Motion of a Car

The component `NetHeat` is stateless. As an example of a stateful continuous-time component, let us build a model of how the speed of a car changes as a result of the force applied to it by the engine. For the purpose of designing a cruise controller, it typically suffices to make a number of simplifying assumptions. In particular, let us assume that the rotational inertia of the wheels is negligible and that the friction resisting the motion is proportional to the car's speed. The forces acting on the car are shown in figure 6.3. If x denotes the position of the car (measured with respect to an inertial reference) and F denotes the force applied to the car, then using the classical Newton's laws for motion, we can capture the dynamics of the car by the *differential equation*:

$$F - k\,\dot{x} = m\,\ddot{x}.$$

Here k is the coefficient of the frictional force, and m denotes the mass of the car. The quantity $\dot{x}$ denotes the first-order time derivative of the signal assigning values to the position variable x and thus captures the velocity of the car. Similarly, $\ddot{x}$ denotes the second-order derivative of this signal, that is, the acceleration of the car.

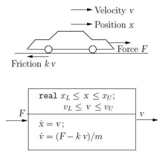
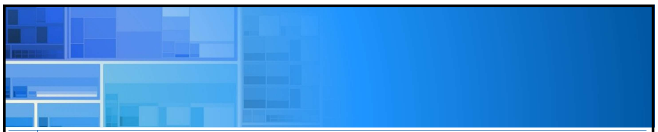
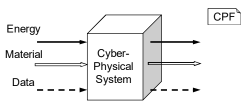


Figure 6.3: Continuous-time Component Car Modeling the Car Motion



MBSE

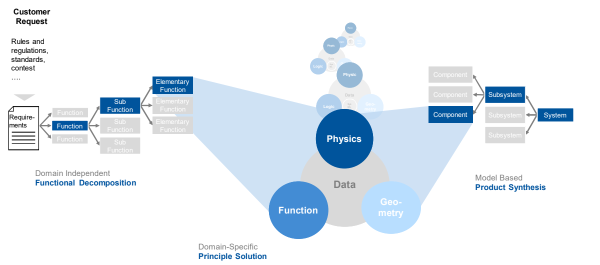
17. Functions Modelling Mechanics
17.6. Principle Solutions Through Physical Effects



Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Bridging the Methodical Gap: From Function to Component

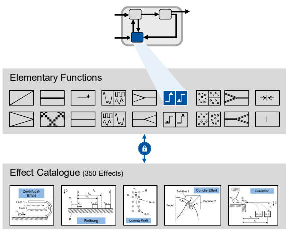


Physical Effects

- Remember: Elementary functions describe a function through their interface.

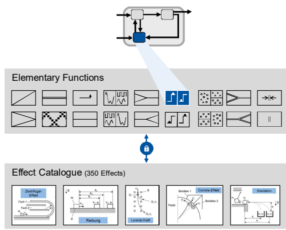
A physical effect implements the behavior of an elementary function.

- One elementary function can be realized by several physical effects (and vice versa)
- The physical effect also provides aspects of the system's geometry
  - Again there are many possible geometries for one effect



Catalogues of Mechanical Designs

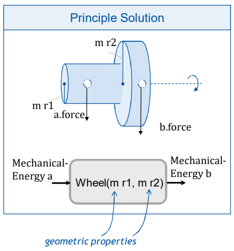
- Mechanical design catalogue provides
  - a set of elementary functions
  - a mapping from elementary functions to physical effects that are suited to realize the function
  - discussions on limitations, context constraints, etc.
  - geometry needed for the physical effect
- Catalogues can be used to find a technical principle for realizing an elementary function within a system
- Composition allows to construct complex machinery from the atomic solutions of a catalogue



Functional Modeling of Principle Solutions

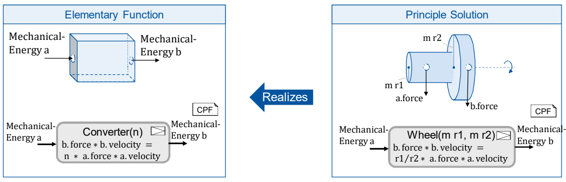
A principle solution realizes an elementary function such that
- its behavior is defined by a physical effect
- parametrized by the active surface and material

- Active surface: Geometric interface (parameterized)
- Engineering Material: Materialistic parameters
- Example: Wheel realizing a converter
  - interface has of two contact points:
    - In / outgoing for mechanical energy
  - Parameters: describe geometric properties:
    - Distances $m\,r_1, m\,r_2$ of contacts from rotation axis
  - Behavior described by the lever principle (physical effect)
    - $a.force * r_1 = b.force * r_2$ and
    - $a.velocity = \frac{r1}{r2} * b.velocity$



Physical Effects Realize the Behavior of Elementary Functions

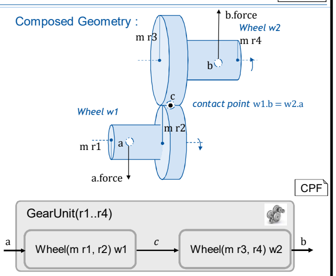
- A lever with distances r_1, r_2 realizes the converter with $n = \frac{r_1}{r_2}$
- Contact points are the active surfaces on which the modeled effect acts



Example: Gear Unit as Composition of Wheels

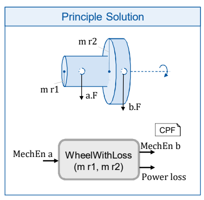
Rep.

- A gear unit is composed of multiple Wheels (here: two)
  - The transmission defines a function that is the composition of the functions of the two wheels:
- Functional model GearUnit(r1..r4)
  - In MechanicalEnergy a
  - Out MechanicalEnergy b
  - Internal components:
    - Wheel w1 = Wheel(r1,r2),
      w2 = Wheel(r3,r4);
  - Composition:
    - GearUnit(r1..r4)(a) = w2(w1(a))
- Result:
  - GearUnit(r1..r4) also realizes Converter(n)
    for n = (r1*r3) / (r2*r4)



Considering Losses

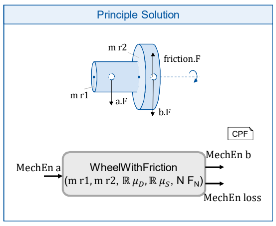
- A wheel that considers losses has an additional output channel:
  - Parameters: Distances $m\,r_1, m\,r_2$
  - Interface with three contact points:
    - In: Mechanical energy Force a
    - Out: Mechanical energy Force b
    - Out: energy loss
- The behavior is extended by an equation telling what is produced on the loss-output
  - $b.f = a.f * \frac{r1}{r2}$
  - $b.f * b.v = \frac{r1}{r2} * a.f * a.v$
  - $loss = b.f * b.v - a.f * a.v = a.f * a.v * \left(1 - \frac{r1}{r2}\right)$



Considering Losses: Friction

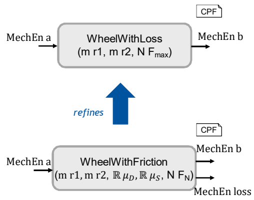
- Include losses, e.g., by considering friction:
- Additional Parameters:
  - Friction coefficients: $\mathbb{R}\,\mu_D, \mathbb{R}\,\mu_S$
  - Normal Force: N F_N
- Behavior :
  - Force implies only dry friction: $\mathrm{a.F} * \frac{r1}{r2} \leq \mu_D * F_N \Rightarrow$
    - $\mathrm{b.f} = \mathrm{a.f} * \frac{r1}{r2} \wedge$
    - $\mathrm{loss} = \mathrm{a.f} * \mathrm{a.v} * \left(1 - \frac{r1}{r2}\right)$

    Refines WheelWithLoss only in this case!
  - Force implies sliding friction: $\mathrm{a.f} * \frac{r1}{r2} > \mu_D * F_N \Rightarrow$
    - $\mathrm{b.f} = \mu_S * F_N \wedge$
    - $\mathrm{loss} = \mathrm{a.f} * \mathrm{a.v} - \mu_S * F_N * \mathrm{b.v}$



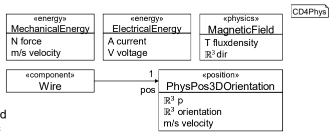
Refinement in Context

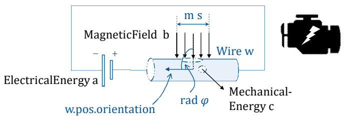
- Wheel with friction is more precise than the wheel but is formally not a refinement
- Add invariant & parameter F_{max} to the Wheel-Function:
  - $\mathrm{a.f} \leq F_{max} \Rightarrow$
    - $\mathrm{a.f} * r_1 = \mathrm{b.f} * r_2 \wedge$
    - $\mathrm{a.f} * \mathrm{a.v} = \frac{r1}{r2} * \mathrm{b.f} * \mathrm{b.v} \wedge$
    - $\mathrm{loss} = \mathrm{b.f} * \mathrm{b.v} - \mathrm{a.f} * \mathrm{a.v} = \mathrm{a.f} * \mathrm{a.v} * \left(1 - \frac{r1}{r2}\right)$
- Now, WheelWithFriction refines WheelWithLoss:
  - Refines the case $\mathrm{a.f} \leq F_{max} \rightarrow \mathrm{a.f} \leq \mu_D * F_N$
  - Completes the specification by adding a specification of the behavior in case $\mathrm{a.f} > \mu_D * F_N$



Modeling Physical Effects: Electromagnetic Induction

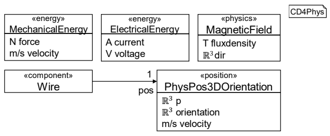
- Electromagnetic Induction is a physical effect
- Current-carrying wire within a Magnetic Field experiences a Force (Lorentz-Force):
  - Length is a geometric property → parameter of the function
- Wire has a physical position within the magnetic field
  - Orientation of the wire yields an angle φ to the magnetic field
- Defines e.g. the physical effect used in electric engines
- Abstracts from
  - energy losses, e.g. heat
  - tilting of the wire loop in the magnetic field

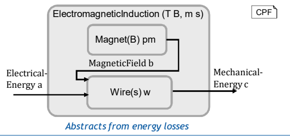




Modeling Physical Effects: Electromagnetic Induction II

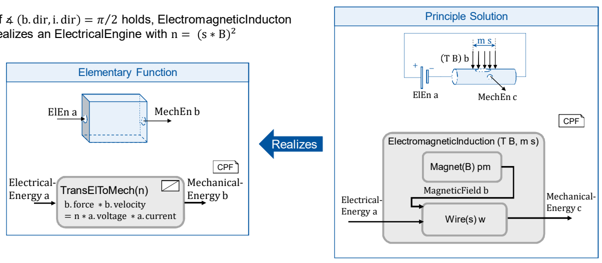
- By electromagnetic induction, a flowing current within a magnetic field induces a force
- Model this physical effect as composition of two functions:
  - Wire:
    - Parameter: Length s of the wire, flux density of the magnetic field
    - Interface with three contact points
      - Incoming electrical energy: ElectricalEnergy a
      - Incoming magnetic flux: MagneticField b
      - Outgoing mechanical energy: MechanicalEnergy c
    - Behavior, with $\measuredangle\,(\mathrm{b.dir}, \mathrm{w.orientation}) = \varphi$
      - $\mathrm{c.force} = s * \mathrm{b.fluxdensity} * \mathrm{a.current} * \cos(\varphi) * \sin(\varphi)$
      - $\mathrm{c.velocity} = s * \mathrm{b.fluxdensity} * \mathrm{a.voltage} * \cos(\varphi) * \sin(\varphi)$
  - Permanent magnet is a magnetic flux-source that supplies a constant magnetic flux
    - Parameter: Magnetic flux supplied (depends, among others on material)
    - Behavior: $\mathrm{b.fluxdensity} = B$





Electromagnetic Induction Realizes a Transformer

- Iff $\measuredangle\,(\mathrm{b.dir}, \mathrm{i.dir}) = \pi/2$ holds, ElectromagneticInducton realizes an ElectricalEngine with $n = (s * B)^2$



Literature

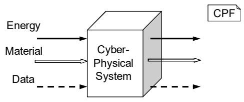
- [KK98] Koller, R., Kastrup, N. *Prinziplösungen zur Konstruktion technischer Produkte*. Springer, 1998
- [BCF+14] Legat, P., Mund, J., Campetelli, A., Hackenberg, G., Folmer, J., Schütz, D., Broy, M., Vogel-Heuser, B. (2014) *Interface Behavior Modeling for Automatic Verification of Industrial Automation Systems' Functional Conformance*. at - Automatisierungstechnik, *62*(11), 815-825.
- [JT01] Jany, P., Thieleke, G. (2001). *Thermodynamik für Ingenieure.* Vieweg + Teubner.
- [Wei09] Weißberger, W. (2009). *Elektrotechnik für Ingenieure.* Vieweg + Teubner.
- [DKV10] Dobrinski, P., Krakau, G., Vogel, A. (2010) *Physik für Ingenieure*. Vieweg + Teubner.
- [FG13] Feldhusen, J.; Grote, K.-H. (Hrsg:): *Pahl/Beitz Konstruktionslehre: Methoden und Anwendung erfolgreicher Produktentwicklung.* (2013) 8. Aufl., Springer-Vieweg-Verlag, Wiesbaden
- [Bru96] Bruton, Dan. *Approximate RGB values for Visible Wavelengths (*1996). Internet: http://www.physics.sfasu.edu/astro/color/spectra.html.
- [HMS+07] Hutcheson, R., Mcadams, D., Stone, R., Tumer, I. (2007). Function-based systems engineering (FuSE).

MBSE

18. Digital Twins
18.1. Foundations

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



Further Reading:
www.se-rwth.de/essay/Digital-Twin-Definition/

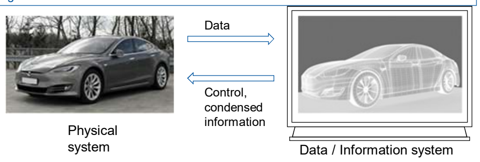
History of Digital Twins

- Goal: Increase system availability and performance of systems by
  - Analyzing physical processes and judging, predicting and optimizing virtually
  - Providing data from physical system to complete simulations, validate settings and dynamically adjust
  - Analyzing results and feeding back to respond to the changes
- Term "twin" originates from NASA: Build a physical copy of aircrafts to simulate and test control scenarios
- Today: Digital Twins normally are virtual representations of physical things:
  - digital models about the physical thing
  - data about/of the physical twin
- Realizing new technologies requires close collaboration of experts and connecting various models

Digital Twin Definition, V2.1

A Digital Twin of a system consists of
- a set of models of the system and
- a set of digital shadows, both of which are purposefully updated on a regular basis, and
- provides a set of services to use both purposefully with respect to the original system.

The digital twin interacts with the original system by
- providing useful information about the system's context and
- sending it control commands.

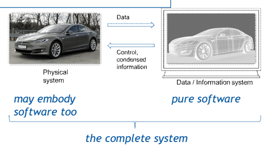


Digital Shadows as Part of the Digital Twin

A Digital Twin of a system consists of
- a set of models of the system and
- a set of digital shadows, both of which are purposefully updated on a regular basis, and
- provides a set of services to use both purposefully with respect to the original system.

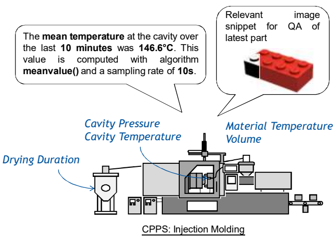
The digital twin interacts with the original system by
- providing useful information about the system's context and
- sending it control commands.

- Physical world contains observable elements that can be monitored, sensed, and may be actuated and controlled
- Data Collection & Device Control interacts with the physical world to observe and influence its behavior
- Creates digital shadows based on data about the physical world and queries/specifications from the digital twin applications
- Further Reading: www.se-rwth.de/essay/Digital-Twin-Definition/



Modern Systems are Monitored by Many Sensors

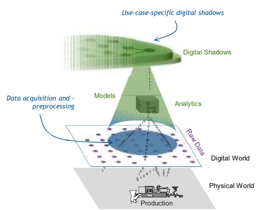
- Large amount of data
  - Real-time processing
  - History based
  - Storage, e.g., in cloud
- Reduced data set may be sufficient to gain insight about the system's state
- Data quality depends on sensor, sampling rate
- Metadata missing (units, date of measurement)
- Sensors over time create data sets, that may be:
  - Very detailed, or
  - Time reduced
  - Quantitatively reduced
  - Preprocessed
  - Qualitatively reduced (black and white instead of colored picture)
  - Enriched with metadata



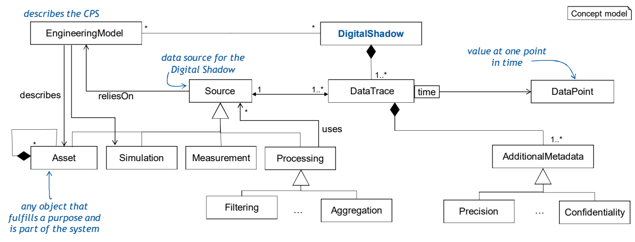
Digital Shadows

- A Digital Shadow is a set of contextual data traces and their aggregation and abstraction collected for a specific purpose with respect to an original system.

- A digital shadow is
  - a passive set of data
  - information source about a system's state and history
  - is collected, filtered and reduced for its dedicated purpose in varying forms of abstractions
  - a purely digital artifact
  - produced by a (physical) system.
- A system can have many different digital shadows describing a variety of different aspects of the system in different detail and at different times.
- Shadow may contain information about production systems, production processes, products, and human operations



A Digital Shadow Reference Model

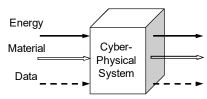


MBSE

18. Digital Twins
18.2. Typical Use Cases

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

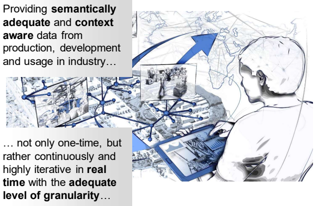


Further Reading:
www.se-rwth.de/essay/Digital-Twin-Definition/

The Internet of Production develops techniques for digital shadows and digital twins

The theme of the Internet of Production:

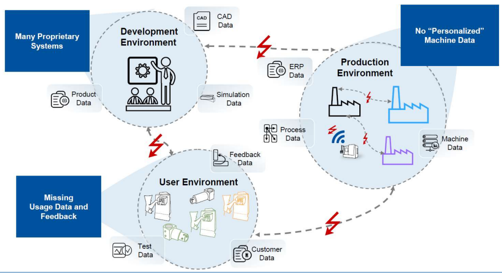
Providing **semantically adequate** and **context aware** data from production, development and usage in industry…



… not only one-time, but rather continuously and highly iterative in **real time** with the **adequate level of granularity**…

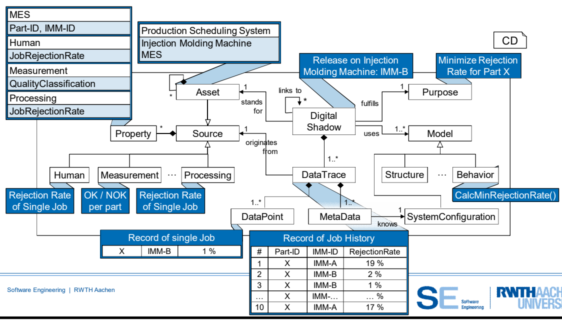
- The central scientific approach of the IoP are Digital Shadows as mediators between the vast amounts of heterogeneous data and detailed production engineering models, meaning:
  - Sufficiently aggregated, multi-perspective and persistent datasets
  - Generated by deliberate selection, cleaning, semantic integration and pre-analysis
  - Used for reporting, diagnosis, prediction and recommendation in domain-specific real-time
- The Internet of Production is a huge project:
  - 87,5 researchers (up to 2x7 years)
  - 13 research managers
  - 4 support positions
  - Overall ca. 200 employees

Data and Models must become available for cross domain use

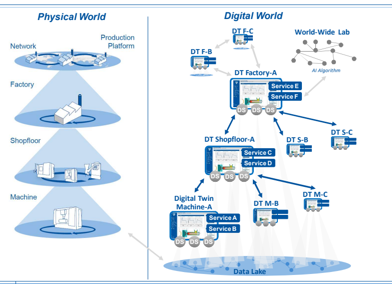


(from IOP)

Conceptual Model for Digital Shadows & Example in the IOP [BBD+21]



The SE-Vision within the IoP



Our aim:
Efficient development of digital twin services based on digital shadows

Provide Engineering Tools & Methods

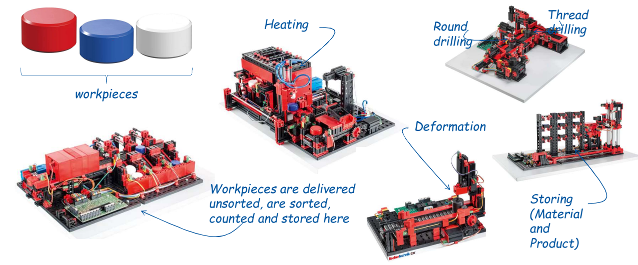
Tool 1: Digital Shadow Type Creator
- generate DS-Types which can be used during runtime to create DSs
- define relevant models, data and meta-data
- select data sources from the data lake
- based on MontiGem (GUI)

Tool 2: Low Code DT Platform
- create configurable DTs
- services for data extraction from engineering models
- definition of meta-data and connection to ontologies
- API's to other services, e.g. AI algorithms
- integrated process mining services
- based on MontiGem (GUI)

Use Case (Szenario): Factory for the production of metal blanks

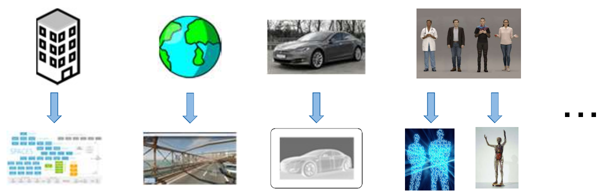
- Production of metal blanks
  - Producing factory
  - Blanks in various shapes
    - with threaded hole
    - with round bores
  - Deformed
  - Come in different colors
  - Can be heated for further processing steps
- Different views and roles
  - Users can view goods, place orders, view the status of orders
  - Monitoring for purchasing: material stock, throughput, demand prediction
  - Monitoring of the factory: capacity utilization, status, errors
- And then AR

Fischertechnik workpieces and functions



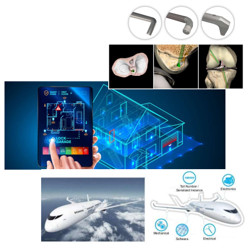
Kinds of Digital Shadows

- Different physical entities, very different, purpose specific kinds of data models
  - e.g., BIM, Google Earth, CAD, Conceptual Models



Various Purposes of Digital Twins in the Application Domains

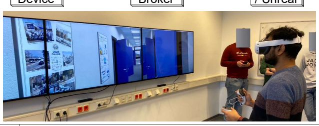
- Health: monitoring, diagnostics, and prognostics
  - Simulators for medical training and education
- Automotive:
  - Predicting driving behavior
  - Monitoring for predictive maintenance
- Aerospace: virtual product development and flight test scenarios
- Construction and Energy Efficiency:
  - Monitoring structural health of sensor modules
  - Process automation with intelligent sensors and methods for calibration
- Games, Media, and Entertainment:
  - Visual and physical motion sensing for three-dimensional motion capture
- Manufacturing: Automating production and reacting if necessary



Digital Twin of SE

- Chair's offices can be visited virtually
- Lights, a metal ring and fans are synchronized between the real world and the digital twin
- Technologies: Meta Quest 2&3, Unreal Engine, MQTT, Arduino/Raspberry Pi

IoT Device ⟷ MQTT Broker ⟶ VR / Unreal

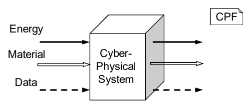


MBSE

18. Digital Twins
18.3. Services, Cockpits

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/



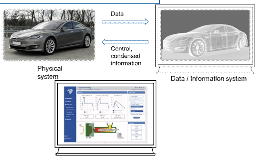
Further Reading:
www.se-rwth.de/essay/Digital-Twin-Definition/

Purposeful Services are Part of a Digital Twin

A Digital Twin of a system consists of
- a set of models of the system and
- a set of digital shadows, both of which are purposefully updated on a regular basis, and
- provides a set of services to use both purposefully with respect to the original system.

The digital twin interacts with the original system by
- providing useful information about the system's context and
- sending it control commands.

- Digital Twin Applications & Services take actions based on one or more situations that they sense in the environment
- User can access Digital Twin Applications & Services through appropriate interfaces
- Digital Twin has (some) strategic control over his physical twin
  - usually high-level, the real-time control strategies are embedded in the CPS



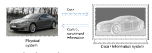
Digital Twin vs. Digital Shadow vs. Model

A Digital Twin of a system consists of
- a set of models of the system and
- a set of digital shadows, both of which are purposefully updated on a regular basis, and
- provides a set of services to use both purposefully with respect to the original system.

The digital twin interacts with the original system by
- providing useful information about the system's context and
- sending it control commands.

A Digital Shadow is a set of contextual data traces and their aggregation and abstraction collected concerning a system for a specific purpose with respect to the original system.

A model is essentially a reduced or abstracted representation of the original system in terms of measure, precision and functionality. (Stachowiak 1973)

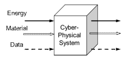


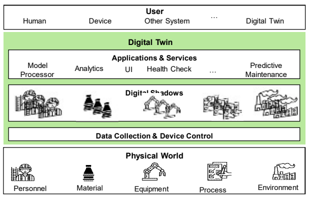
Terms share some characteristics, but:
- Twin is an active software system (through services)
- Model prescribes the system under development and at operation
- Shadow is passive data produced during operation

Cyber-Physical Systems and Digital Twins

Cyber-physical systems (CPS) are engineered systems where functionalities are emerging from the networked interaction of physical and computational processes. [BDS19]

- Literature currently is unclear, whether a digital twin is part of the CPS or beneath to it
- A useful viewpoint:
- The overall system contains the physical part, sensors, actuators the embedded controls, data collection, services and user interface
- The overall system is engineered in an integrated project considering physical and IT part in parallel
- Digital Twin builds a logical entity, but its software components may be distributed

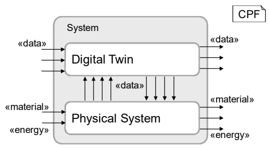




Cyber-Physical Systems and Digital Twins

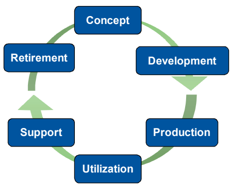
Cyber-physical systems (CPS) are engineered systems where functionalities are emerging from the networked interaction of physical and computational processes. [BDS19]

- Literature currently is unclear, whether a digital twin is part of the CPS or beneath to it
- A useful viewpoint:
- The overall system contains the physical part, sensors, actuators the embedded controls, data collection, services and user interface
- The overall system is engineered in an integrated project considering physical and IT part in parallel
- Digital Twin builds a logical entity, but its software components may be distributed



Digital Twins Support all Lifecycle Phases of a System

- Depending on the lifecycle phase a Digital Twin offers different services to control/adapt/represent the physical system
- During Concept and Development, a Digital Twin acts as integrated collection of development artefacts and/or simulates the behavior of a CPS
  - Supports communication engineers and designers while working together across departmental boundaries
  - Evaluates product variants to support design decisions
- During Production, a Digital Twin
  - Supervisee the production process, e.g., individual deviations from norm that require special treatment
  - Tracee the applied materials, components, processing steps
- During Utilization and Support a Digital Twin
  - Provides information on system state, history and usage
  - Enables optimization of a machine during operation
  - Facilitates the improvement of future products
  - Enables predictive maintenance



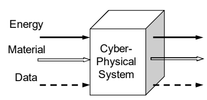
System Lifecycle from ISO/IEC 15288 (Systems Engineering standard)

MBSE

18. Digital Twins
18.4. Developing a Digital Twin

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

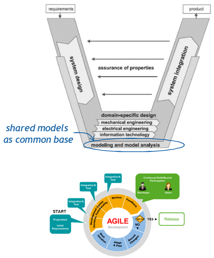
http://www.se-rwth.de/



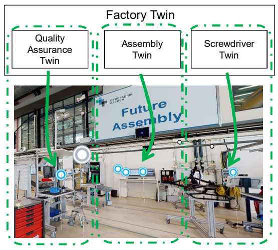
Further Reading:
www.se-rwth.de/essay/Digital-Twin-Definition/

Development Process of CPS and Digital Twins

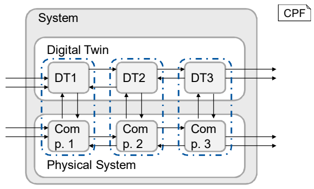
- Services of a Digital Twin are intensively connected to the CPS
  - Integrated parallel development
  - NOT only a spin-off product of the development of the physical system
  - may be configurable, can be parameterized or calibrated
  - new services may be coming over time
- Adaptivity through explicit models at runtime:
  1. Autonomous self-adaptation, e.g. induced by changes of the context, optimizations identified for example through continuous measurements or by a slow degradation of the system itself
  2. The user wishes to adapt the system behavior
  3. The manufacturer adapts the system behavior according to identified optimizations, fixing of bugs and failures, or upgrade of functionality
- Challenge:
  Development cycles/methods for CPS and IT differ radically



Composition of Digital Twins

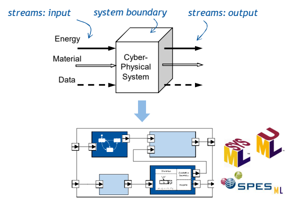


- Factory decomposed along two dimensions
  - DT's are composed
  - Physical components are composed to a system
  - But ideally: Physical component + its twin is developed / purchased / shipped together



The function paradigm provides the methodological and organizational foundation for smart digital twins services.

- A system defines a cyber-physical function
  - physical and computational structure
  - performs data, energetic and physical transformations
  - and is connected to its environment through its interface
- Functional modelling paradigm allows for
  - Clear definition of component interfaces / behavior
  - Data, energy, and material streams model interactions of components
  - Decomposition
  - Composition from atomic Principle Solution Models
  - Abstraction in modelling
  - Basis for simulation

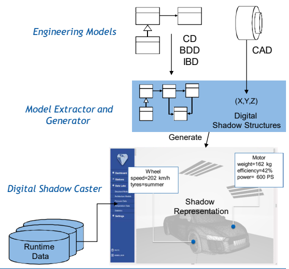


Functional system architecture

The functional paradigm specifies the energetic, materialistic and computational behavior of a CPS within a system architecture with a mathematically sound semantics.

Model-Driven Digital Twin Creation

- Cyber-physical systems are complex
  - Consist of multiple components
  - Offer different functionalities
- Reuse engineering models that are created during system design for systematic efficient definition of larger parts of a Digital Twin
- Generate a Digital Shadow Caster that accesses the CPS and displays potentially interesting Digital Shadows from Engineering Models
- Extract structural information about the CPS
  - How is the CPS composed
- Spatial Information
  - Where are the CPS and its internal components located
- Expected behavior
  - How should the system react to a specific situation
  - Derive, when the system does not behave as intended



Data Lake

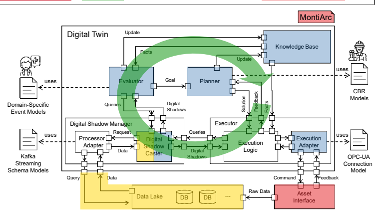
- Repository of data stored in its natural/raw format, usually object blobs or files
  - Include structured data from relational databases (rows and columns)
  - Semi-structured data (CSV, logs, XML, JSON)
  - Unstructured data (emails, documents, PDFs)
  - Binary data (images, audio, video)
- Data Lake stores all data - regardless of relevance, structure and purpose
- Data stored independently of source and structure
  - Remain in original form and only prepared when needed
  - "Schema on Read"-Principle: data is only structured when it is read
- Data Lake e.g., based on Hadoop
  - Distributes the storage and computation of the data over many nodes of a cluster
  - Data in large quantities can be processed quickly

Model-Driven Digital Twin Architecture [BBD+21a, DMR+20, BDH+20]

That realizes a MAPE-K self-adaptive loop over the CPPS

Digital twin: models + contextual data + services used purposefully w.r.t the physical system.

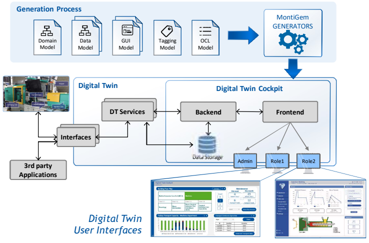
- **MontiArc reference architecture**
  - Uses data lake and asset
  - Creates digital shadows
  - Evaluates state and acts
- **Domain-specific models**
  - trigger shadow creation
  - connect to data lake
  - case-based reasoning
  - connect to CPS
- **Services:** Representation, monitoring, optimization



Creating Digital Twin Cockpits with MontiGem [DMR+20]

Generating digital twin cockpits from models with the generator framework MontiGem

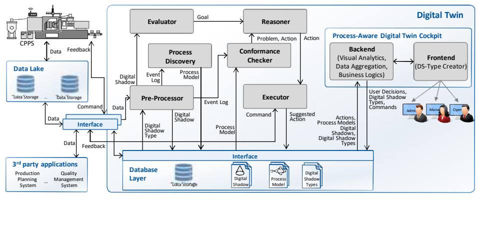
- Successfully applied to research and real-life projects
  - MaCoCo, Engineering Wind Turbines, InviDas see: https://www.se-rwth.de/projects
  - Digital twin cockpit: Injection Molding [DMR+20]
- Components of the digital twin cockpit application
  - Database, backend and frontend of a web application, communication infrastructure
- Used models, e.g.,
  - Domain model: data structure
  - GUI models: user interfaces
  - Data models: representation of parts of data in GUIs
  - OCL models: validation of data input



[DMR+20] M. Dalibor, J. Michael, B. Rumpe, S. Varga, A. Wortmann: Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. ER'20

Process Prediction with Digital Twins [BHK21]

- Aim: improve the operation of digital twins
  - process discovery from event logs and
  - process prediction from process models at runtime
- Models at designtime
  - Application-specific models, e.g. domain model, GUI models, …
  - Application independent models, e.g., architecture, basic DS and process structure,…
- Models at runtime
  - process models, goals, actions,…

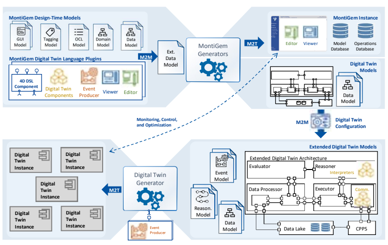


Presentation in Oct. 21 at MODELS@run.time Workshop

with Wil van der Aalst, István Koren, Merih Seran Uysal, Tobias Brockhoff (PADS RWTH Aachen University) and Andreas Wortmann (University of Stuttgart)

Low-Code Platforms for Model-Driven Digital Twins [MW21]

- Digital twins configured and operated by shop-floor experts (rarely professional software engineers)
- 2-step generation process
  - We generate the low-code platform
  - Shop-floor experts configure a digital twin via the low-code platform and
  - generate one or more digital twins
- Enablers
  - model-driven digital twin architecture and toolchain
  - model-driven toolchain for generating information systems
  - reuseable language components, services and models

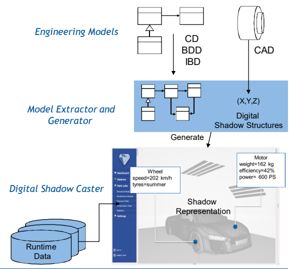


Presentation today at APMS conference

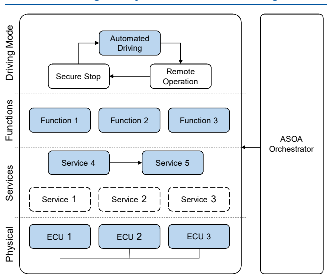
with Andreas Wortmann (University of Stuttgart)

Model-Driven Digital Shadow Creation

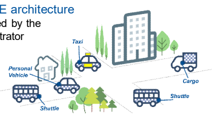
- Cyber-physical systems are complex
  - Consist of multiple components
  - Offer different functionalities
- Reuse engineering models that are created during system design for systematic efficient definition of larger parts of a Digital Twin
- Generate a Digital Shadow Caster that accesses the CPS and displays potentially interesting Digital Shadows from Engineering Models
- Extract structural information about the CPS
  - How is the CPS composed
- Spatial Information
  - Where are the CPS and its internal components located
- Expected behavior
  - How should the system react to a specific situation
  - Derive, when the system does not behave as intended



AUTOtech.agil Project: Model-Driven Digital Twin for Vehicle Diagnostics



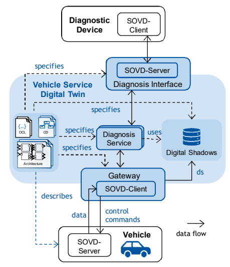
- Goal: shaping future software and E/E architecture
  - Layers controlled by the Service Orchestrator



- Our job: Model-driven generation of diagnostic DTs
- Four layers
  - The Driving Mode layer manages active modes
  - The Functions layer organizes vehicle functions
    - vehicle functions are functionalities dependent on multiple different services, e.g., door control
  - The Service layer: active services + connections
  - The Physical layer maps services to ECUs

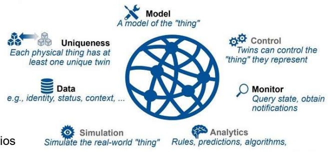
AUTOtech.agil Project: Model-Driven Digital Twin for Vehicle Diagnostics -2

- Our job: Model-driven generation of diagnostic DTs
  - services, SOVD-compliant interfaces, data containers for digital shadows
- Architecture models vehicle functions and their structure
  - to simulate the expected behavior
  - to compare of real vs expected behavior
- CDs + OCL as model basis
  - function classes model input/output data (storage logs of Digital Shadows)
  - kinds of software errors
  - predefined error-kinds specific diagnosis queries
- Pre-processing data in the vehicle services
  - mobile data-plan, because constraints on transmission
  - aggregation, compression



Outlook Digital Twins

- A digital twin can
  - Boost production efficiency
  - Optimize performance of products in the field
  - Enable more accurate predictions and what if scenarios from usage
  - Process and advise operators in complex scenarios
  - Give strategic control to the CPS
  - Identify correlations between scenarios, learn and improve during their application



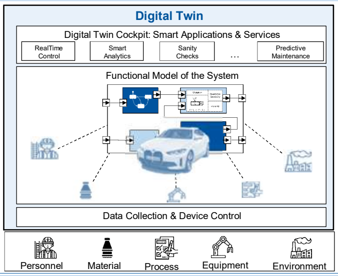
The digital twin enhances engineering models with data and synthesizes insights from system level analyses and optimizations.

System Level Analyses & Optimization
Machine learning and reasoning automate tasks e.g., verification and design optimization

Functional Systems Engineering
unifies system functions and physical parts in a model that serves as digital replica of the entire system and structures engineering models by the innovation driver – the function

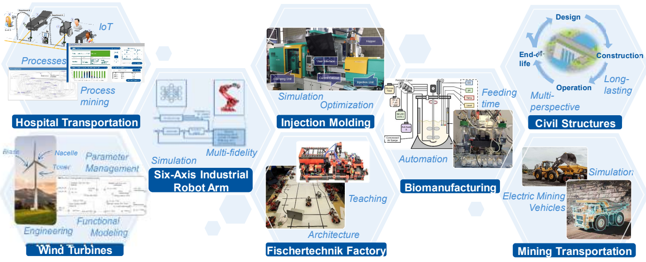
Knowledge Management
Artificial Intelligence extracts knowledge from data and propagates it as necessary

Data acquisition & propagation
Big Data systematizes empirical knowledge obtained from live-data



- Anhang

Some Digital Twin Use Cases



MBSE

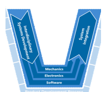
19. Advanced Methods (i.e. Methods, Part2)
19.1. Evolution in the Brownfield

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

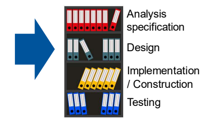
http://www.se-rwth.de/

Problem: Brownfield vs. Greenfield

- V-Model, RUP, etc. are greenfield development processes:
  - They assume a fresh project every time
  - They are good for understanding the overall organization, but ignore existing, reusable assets.
- Brownfield project relies on
  - Already existing product from the last version
  - Existing models and their relations of all sorts
  - Knowledge about problems, improvable functions, …
  - Changed requirements
- Existing project is legacy as well as a good starting point
- Brownfield also happens when:
  - Fixing bugs
  - Quickly adding a new feature (even after shipping …)



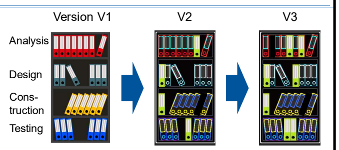
From greenfield process



to evolution of its artefacts:
a "cabinet" full of artifacts with relations of all kinds

Brownfield: Evolution of Existing Artefacts

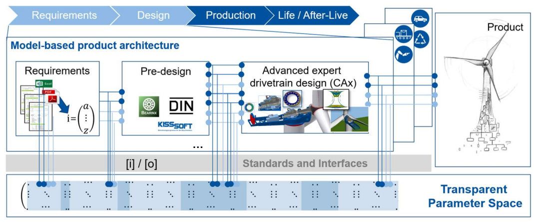
- Brownfield:
  - We evolve the cabinet of artefacts in parallel, but not top-down
    - Bug fixing → implementation artefacts
    - Redesign → Architecture / Design
    - Requirements change → Analysis
- Tracing the relations between artifacts is essential
  - Automatic consistency checking
  - Automatic change propagation (upward and downward)
- Even better:
  - Automated generation reduces the set of artifacts to evolve
  - Automate build with a build script
  - Automate testing (e.g., junit-like) and simulation (e.g., simunit)



"cabinet" full of artifacts with relations of all kinds is evolved, ideally while keeping all artefacts consistent to each other

Example: Wind Turbine

- MBSE – Parameter triggered processes by global transparent modelling of the entire design process



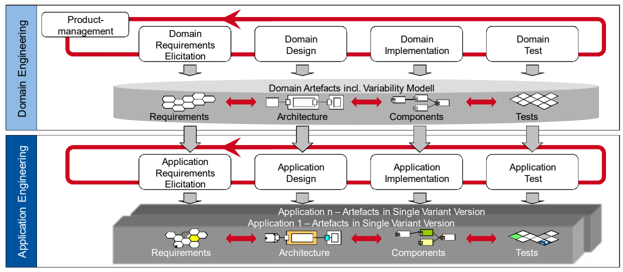
MBSE

19. Advanced Methods
19.2. Variability in Product Lines

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

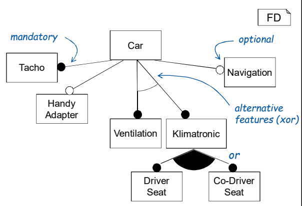
http://www.se-rwth.de/

Product Line Development: Separate Domain and Application Engineering



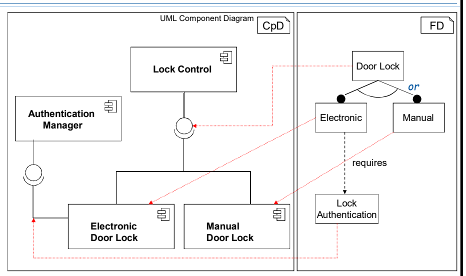
Description of Variability through Feature Diagrams

- A feature is a piece of functionality tangible by a customer
- A feature diagram describes the set of valid configurations
  - Box = feature
  - Dependencies of form and, or, xor
  - Additionally: B requires D , B excludes E
- A feature configuration is then a subset, e.g.
  - Tacho, Klimatronic, Driver Seat, Navigation
  - It describes possible configurations to ship a product
- FD's usually result from a domain analysis



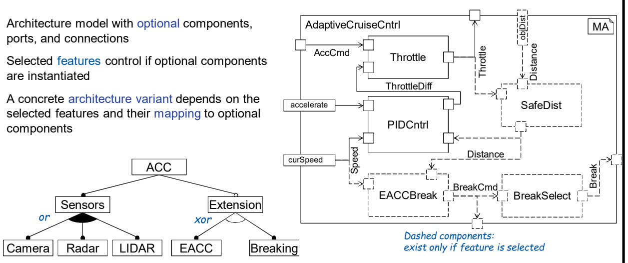
Features may Select Model Elements in Other Models

- Features in the FD correspond to model elements in other models
- 150% models may contain all features, and a subset is then selected
- Problems:
  - 150% models may not be valid models (e.g. alternatives cannot be expressed, etc.)
- Integrated 150% model belongs to a new language



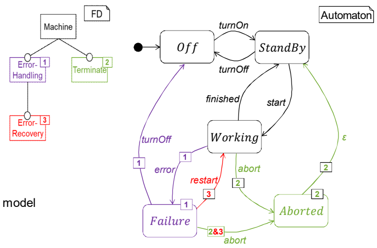
Variability in Component Definitions

- Architecture model with optional components, ports, and connections
- Selected features control if optional components are instantiated
- A concrete architecture variant depends on the selected features and their mapping to optional components



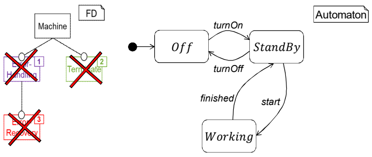
Automaton Example with 3 Features

- Base language: Automaton
- Color (and numbering) demonstrate the three features
  - (color is not part of automata)
- Variation points for automata can be the language constructs:
  - State (can be added)
  - Transition (can be added, redirected)
  - Alternate signals
  - Initial / final state marker
- Some states / transitions belong to the core model (black), others to specific features



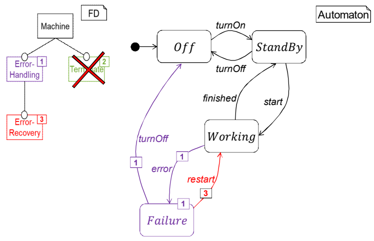
Automaton Example: No Feature Selected

- If no features are selected, a minimal automaton describes the core functionality
- Typical working engine with three states, but unable to cope with errors



Automaton Example: 2 Features Selected

- An extractor transforms the product line model into a base model
  - Based on a feature configuration
  - Calculating minimal complete set of features (ErrorRecovery requires ErrorHandling)
  - Deletes non-selected features
  - And keeps model elements of selected features
- The automaton model can now be further used for the defined variant of the product line.



MBSE

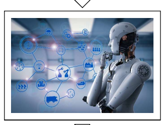
19. Advanced Methods
19.3. Artefacts and Automation

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

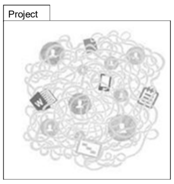
Automation Of Development Steps

- "Automation has proven to be the single most effective means of making dramatic improvements in both productivity and product quality"
  Bran Selic, 2019
- Automating :
  - Generation, synthesis
    - Constructive derivation of implementations or more detailed forms of models
    - Transformation
    - Correctness by construction
  - Analysis
    - Consistency checking, completeness, well-formedness rules, etc.
    - Applicability of transformations
    - Automatic verification
  - Testing
  - Simulation
    - As a technique for testing and dynamic analysis



Artefact View: Many Artefacts and Many Types of Artefacts

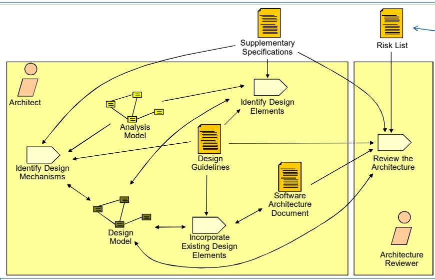
An artifact is an individually stored and referenceable unit containing relevant information in a software or systems development project. — Definition



- Examples:
  - Requirement document,
  - SysML model, UML, model, config file, build script,
  - Variant list, task,
  - CAx file, decision description,
  - Code, prototype,
  - Test, test result, …
- Roughly:
  - Each file is an artefact,
  - But data bases and archives also contain artefacts

Example 2: RUP Workflow with Activities and Artefacts

Rep.



Workflow Details: Roles, Activities and their *Artifacts*

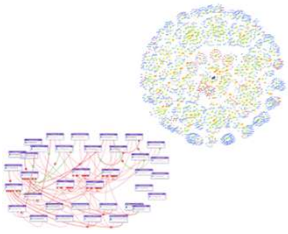
in the Architectural Design Workflow

Artefacts and Their Relations

- … are the basic constituents for the development process

Artifact 1 —relates to→ *Artifact 2*

- Various kinds of relations exist:
  - Derived from
  - Refinement of
  - Compiled to
  - Replacement of
  - Trace
- Observations:
  - Understanding artefacts and relations is key for assessing a development project
  - Artefact model similar to PLM for the development

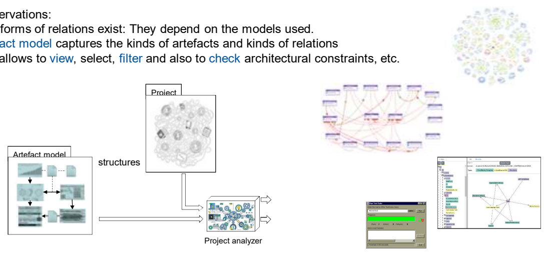


(Visualizations of retrieved artifact models)

Artefact Model

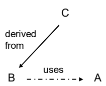
More Observations:

- Various forms of relations exist: They depend on the models used.
- An artefact model captures the kinds of artefacts and kinds of relations
- Tooling allows to view, select, filter and also to check architectural constraints, etc.



Two Classes of Artefact Relations

- Many kinds of relations, but: they can be cklassified
- in only two forms of relations
  - "B uses A",
    e.g. import in Java (also called "dependency"),
  - "B is derived from C",
    e.g. class files are derived from Java source



- If "B uses A" then both artefacts A, B are relevant for the subsequent activities.
- They contain different pieces of information, but B uses symbols that have been introduced in A.
- B is syntactically coupled to A.
  - use of B enforces use of A.
  - change of A may lead to a change in B.
- Usage may be cyclic

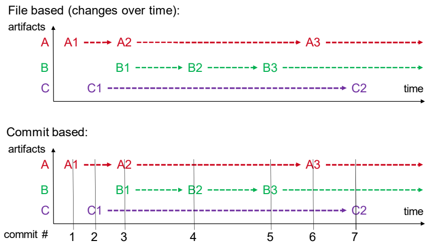
- If "B is derived from C", then B contains (partially) the same information from C. B may have more information or a different representation.
- C could be thrown away (or ignored), if all information was derived to B.
  - C was a result of an "earlier" activity than B.
- Derivation is acyclic (a directed acyclic graph)

Degrees of Automation of Artefact Relations

- For the derivation relations, we see several degrees of automation:
- B is manually derived from A — A - - - - → B
  - Costly labor has been spent by a developer
- B is generated fully automatically from A — A ——→ B
  - Cloud / computer power is cheap: No cost at all
- Mixtures with variable degrees of automation are possible:
  - (1) B is manually derived from A, but inconsistencies can be detected if A or B change.
  - (2) a trace can be established allowing to constructively propagate changes on A to B.
  - (3) … and also to propagate changes backward from B to A.
  - (4) an explicit transformation has been scripted that allows to redo the derivation of B from A
    - But: Trace and transformation are also artifacts!

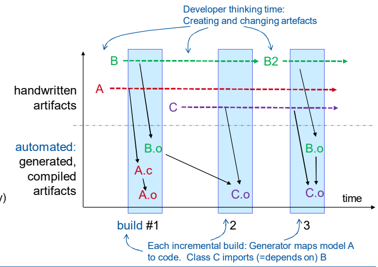
Artifacts Evolution Over Time

- Artifacts evolve
- This is also a derivation relation, usually a manual derivation
- Artifact A exists in versions A1, A2 , A3, …
- Only the current version is relevant
- Typical version controls:
  - git, svn



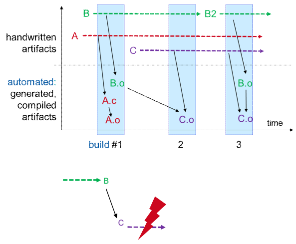
Artifacts Over Time: Development

- Artifacts are manually derived from others
  - A time consuming, human activity
- Some artifacts are automatically generated or compiled
  - a quick build process that can be repeated anytime
- A development process takes time:
  - makro time scale (e.g. a year)
- Manual derivation is slow and extensive:
  - development step: micro time scale (e.g. a day)
- Automatic generation, testing is quick:
  - nano time scale (e.g. 1-10min)



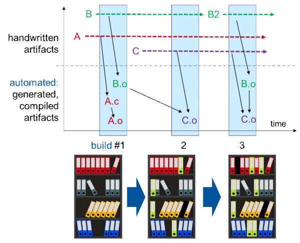
Automation in Development

- Manual derivation is extensive even on the micro steps / time scale
- Automatic generation, testing is quick: nano time scale
- Consequences:
  - automate build with a robust build script
  - automate testing (e.g. junit like) and simulation (e.g. simunit)
  - automate analyses (e.g. continuous integration tools)
  - keep project buildable / testable at all times
  - reduce redundancy (single source of information)
  - use generators from abstract to concrete
  - avoid lengthy manual tool chains
  - reuse generators to stay agile
    - DO NOT make one-shot generations
    - DO NOT modify generated artifacts



Build Scripts and Development Steps

- Macro time: development process
- Micro steps: manual activities
- Nano time scale: automatic activities
- Build scripts (gradle, make, mvn) include generation, compilation, testing, deployment, consistency checks, transformation, etc.
- They are a natural part of the methodical steps of the development process.
- Automation changes the development process!
- Agile evolution needs as much automation as possible.



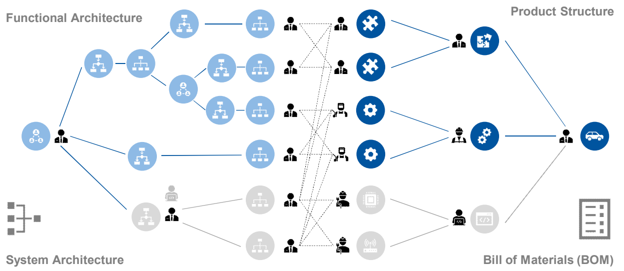
MBSE

19. Advanced Methods
19.4. Some Industrial Insights

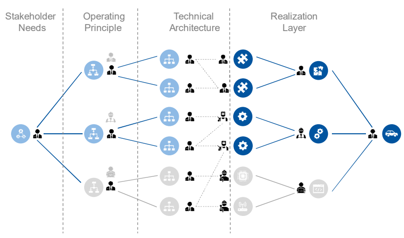
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Model-Based Systems Engineering for Automotive System Development



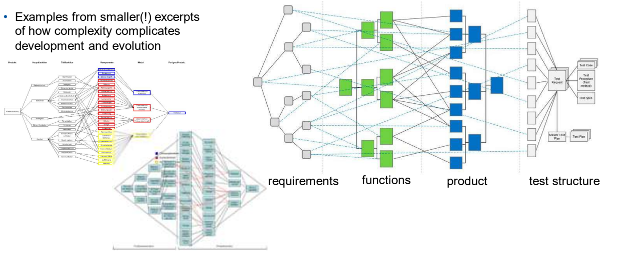
Challenges in Model-Based Systems Engineering of Automotive Systems



- Stakeholder Needs
  - From customer to stakeholder needs
  - Use Case Templates (Cockburn & Fowler)
- Operating Principle
  - Semantical Feature & System Foundation
  - Design pattern
- Technical Architecture
  - Connection between Logical & Technical Architecture (DSL)
  - Integrate Features
- Realization Layer
  - Tracing solutions to requirements
  - Generative connection

Challenge: Complexity is Overwhelming

- Examples from smaller(!) excerpts of how complexity complicates development and evolution



MBSE

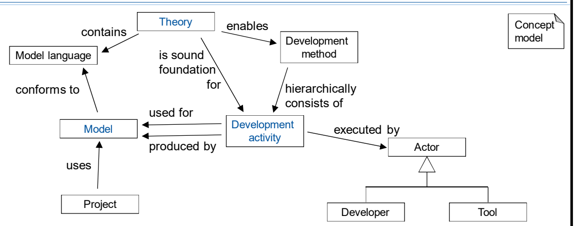
20. Wrap Up / Summary

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Systems Engineering Concepts we Already Know

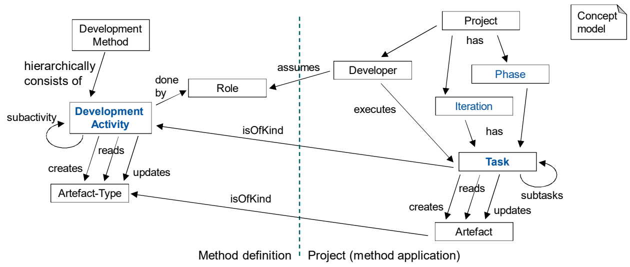
Rep.



- The concept model illustrates some relevant concepts and their relationships.
- In this chapter we introduced: model, development method, and their underlying theory.

Concept Model for Development Methods and Projects

Rep.

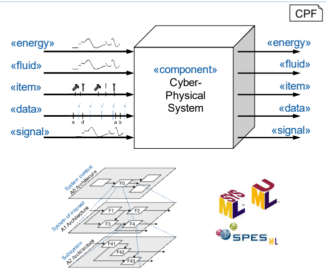


Function-based Universal Specification and Construction Principles

Rep.

CPF

1. The function paradigm is the foundation
   - Clear boundaries, clear input/ouput signatures
2. Abstraction with dedicated models to master complexity
3. Controlled, explicit underspecification
   - abstraction, variability, ability to describe the desired range of allowed behaviors
4. The concept of stream
   - as mathematically precise, time dependent model of input/output behavior
5. Composition into hierarchies of function nets
6. Static dimensioning of parameterized functions
7. Adequate modelling techniques center around the function paradigm, e.g. SysML



Content of the Lecture

- Modelling in Development
- Software and Systems Engineering
- Development Methods (Agility, Scrum, V)
- Modelling Paradigms:
  - Data, Function, Structure, Behavior
- Modelling languages, e.g.
  - Class Diagrams for data and physical entities
  - StateCharts for state-based behavior
  - Architecture for function, component and gadgets
- Modelling Cyberphysical Systems
- Modelling Software
- Software synthesis (code generation)
- Composition
- Refinement
- Evolution (Agility)
- Variability

Benefits of Model-Based Software Engineering (MBSE)

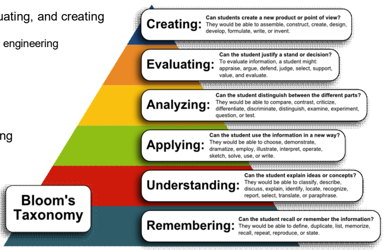
- *You would not design an airplane by putting together nuts and bolts – why should we do this with software?*
- MBSE reduces the conceptual gap [FR07]
  - Between problem domains (e.g., robotics, medicine, law) and solution domain (software engineering)
- Models increase abstraction
  - the model of a software application is specified on a higher abstraction level than traditional programming languages
- This eases communication, documentation, and integration of domain experts
- MBSE enables and facilitates automation
  - Model checking, artefact tracing, integration
  - Model transformation (model-to-text, model-to-model)
- MBSE facilitates producing high-quality software
  - Depends on the generator. Easy to improve (generated) code base

[FR07] R. France, B. Rumpe. Model-Driven Development of Complex Software: A Research Roadmap.. In: Future of Software Engineering 2007 at ICSE.

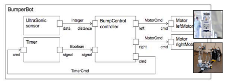
Learning Objectives

Rep.

- Understanding, applying, analyzing, evaluating, and creating
  - Models by applying modeling methods
  - Functional modeling and models in systems engineering
  - Requirements modeling
  - Data modeling
  - Geometric and physical modeling
  - Structure and behavior modeling
  - Systematic CPS engineering
- Syntax and semantics of selected modeling languages (including UML, SysML)
- Digital twins
- Quality assurance

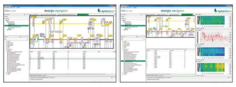


Model-Based Systems Engineering

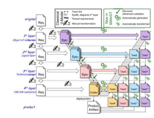


MontiArc Architecture Modeling

Management Cockpit for Controlling



Energie Navigator modeling infrastructure for building information management



SMArDT specification method for requirements, design, and testing

We hope you had fun and will actually be able to productively use the knowledge learned.

Further research and deepening the knowledge: Bachelor and Master theses on sub-topics are available

All the best,

Bernhard Rumpe and Team