Addressing the "Engineering" in "Software Language Engineering"

Bernhard Rumpe
Software Engineering
RWTH Aachen
http://www.se-rwth.de/

SE Software Engineering | RWTH AACHEN UNIVERSITY

RWTH Informatics: Facts and Figures

	RWTH	Informatics
professors	569	40
staff (FTE)	9.300	330
first semester students	8700	750
students	45.000	4.600

CHE-Ranking: in the top group in all categories

WiWo: places 1-3 top universities, for years

Guide2Research: 61. of 1255 of the top universities worldwide

State: 06/2025

Software Language

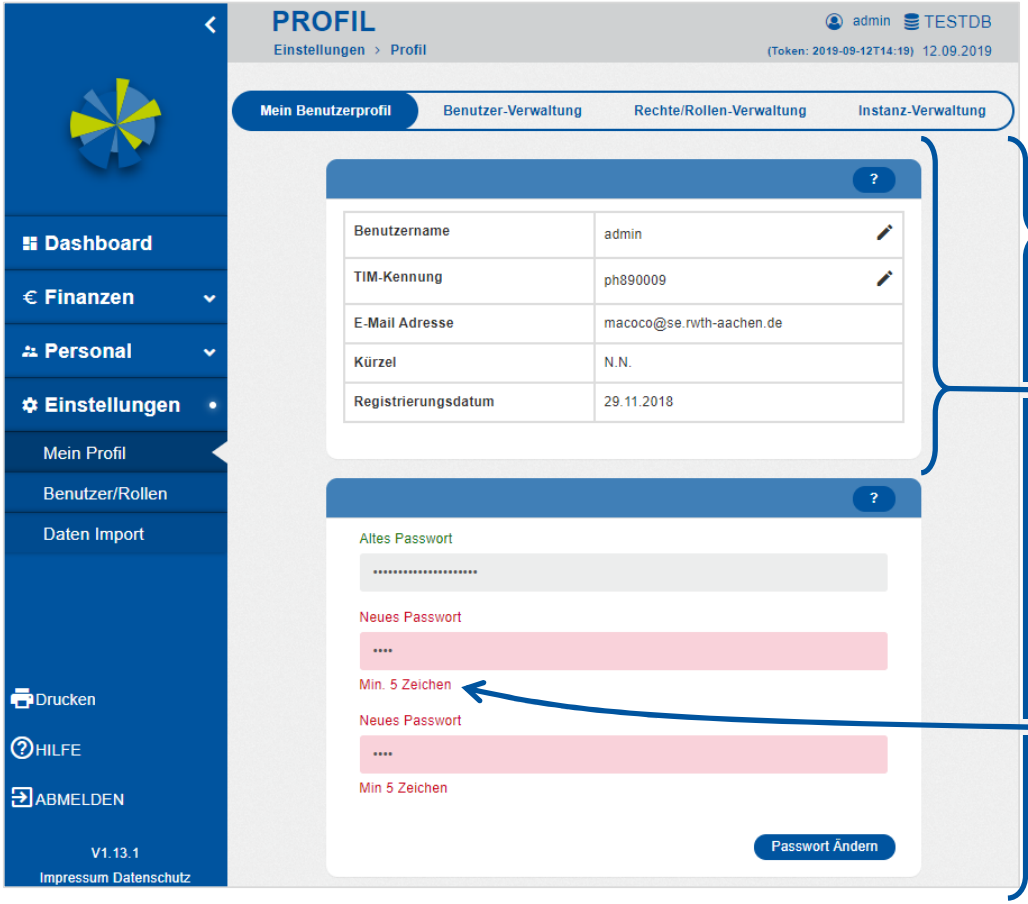
A software language is a human readable and computer processable language addressing a particular problem.

A modeling language is a software language used for modeling software or systems.

- Software languages facilitate
  - automated tool-based analysis, synthesis and code generation on models
  - re-use of models
  - engineering productivity
  - adaptivity, for example with Models@Runtime
- Any form of automation needs a precisely and explicitly defined language.

- Examples:
  - UML: a general-purpose modeling language
  - Java: a general-purpose programming language
  - XML: a format for structured data

GUI Modeling in MontiGem: For Information Systems, Digital Twins, IOT-Services



CD4A

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

GUI-DSL

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

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. ";
```

Data structure

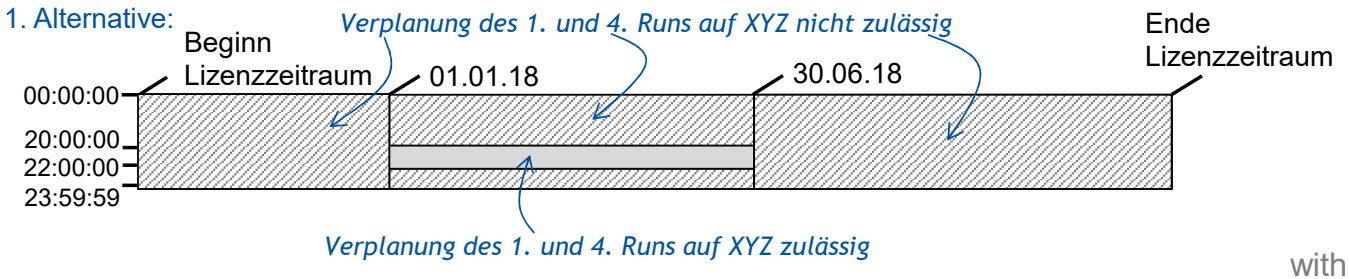
User interface

Constraints

Description Language for Planning TV Broadcasting

- Planning audio-visual offers such as TV program and video-on-demand
- Restrictions in licensing contracts
- Risks: Misinterpretation and resulting planning errors
- DSL for
  - Verification of plans
  - Calculation of allowed planning periods

```
1 Alternative 1:
2 Der 1. & 4. Run innerhalb von 01.01.2018 bis 30.06.2018 turnus immer von 20:00 bis
3 22:00 auf Nutzer XYZ
4 Alternative 2:
5 Alle Run innerhalb von 01.01.2018 bis 30.06.2018 turnus ohne auf Nutzer XYZ
```

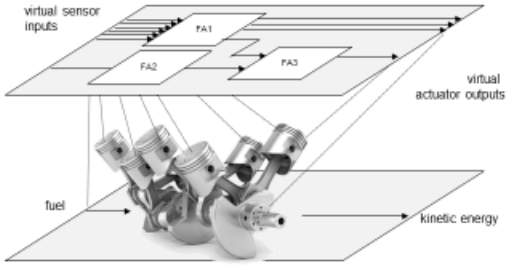


with I. Drave, K. Hölldobler, O. Kautz, J. Michael

Some Examples For Usage of DSLs

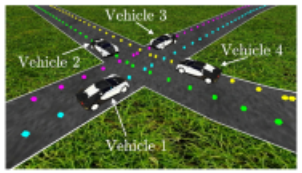
Modelling Mechanics and Software

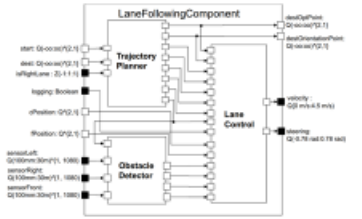
- when specifying software and mechanics use very different kinds of models
- Cyber Physical Systems need their sound semantic integration
- SysML Version 2.0 will help

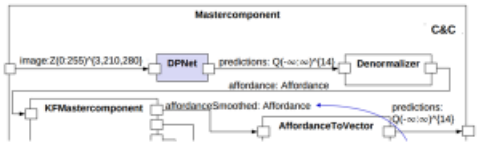


Cooperating Local Traffic Systems

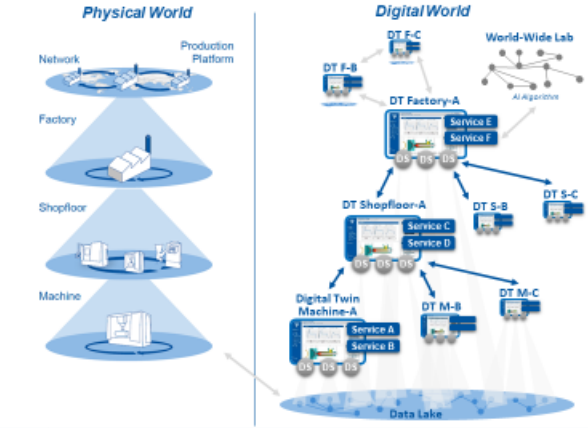
- Dynamic reconfiguration
- Deep Learning Architectures for Autonomous Vehicles







Using Models to create a Digital Twin of a Production Line



Efficient development of digital twin services based on digital shadows

Provide Engineering Tools & Methods

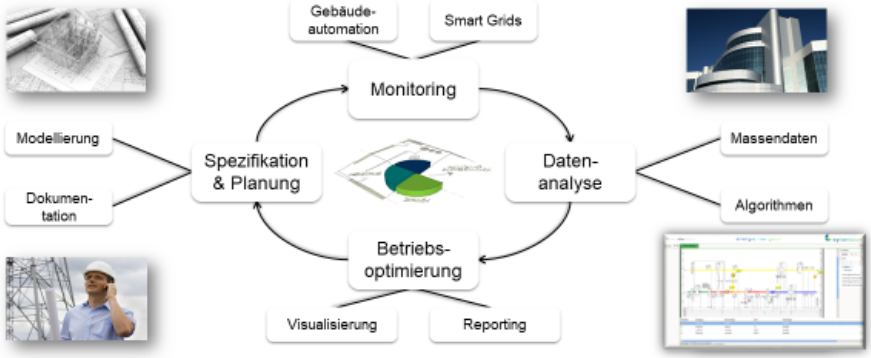
Tool Suite 1: Digital Shadow Type Creator
- generate DS-Types which can be used during runtime to create DSs
- define relevant models, data and meta-data
- select data sources from the data lake
- based on MontiGem (GUI)

Tool Suite 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)

Modeling of Energy Efficient Buildings

Ausgründung: synavision

Zentrales Thema: Energieoptimierung durch Softwaresysteme → CO_2 Reduktion



Some Examples For Usage of DSLs

Model-Driven Software Engineering of Information Systems

The **MontiGEM** framework provides support to generate large parts of a **web application** for a domain specific Enterprise Information System consisting of **data base, persistence layer, command infrastructure** and **graphical user interface** based on the **input** of a hand full of **models.**

1 Define models
- Required: Class Diagrams (Data structure, subsets for views), GUI Models (Views)
- Optional: OCL (Input data validation), BPMN (Workflows)

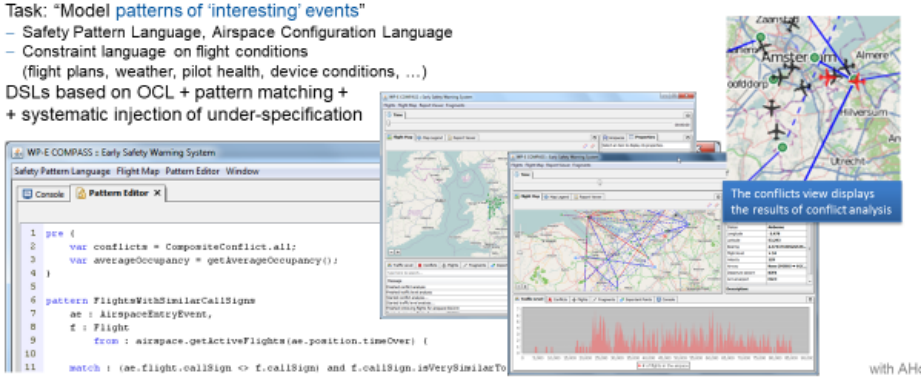
2 Generation of the „System Core“
- Database
- Communication Infrastructure between Frontend and Backend
- Graphical User Interfaces

3 Hand-Written Extensions
- Allow for re-generation
- Customize Appearance
- Add business logic

11

Sesar - Air Traffic Management (EU)

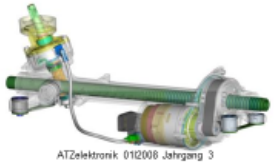
- Task: “Model patterns of ‘interesting’ events”
  - Safety Pattern Language, Airspace Configuration Language
  - Constraint language on flight conditions (flight plans, weather, pilot health, device conditions, …)
- DSLs based on OCL + pattern matching + + systematic injection of under-specification



with AHo

Software Product Line for Steering Control Software in Cars

- From product to product line
  - merging of similar developments
  - reduction of redundant activities/inefficiency
    - coding, testing, evolution
- Re-use strategy
- Schematic generation of similar functions
  - for communication, scheduling, safety
- Variation management
- Basis for automated testing / deployment



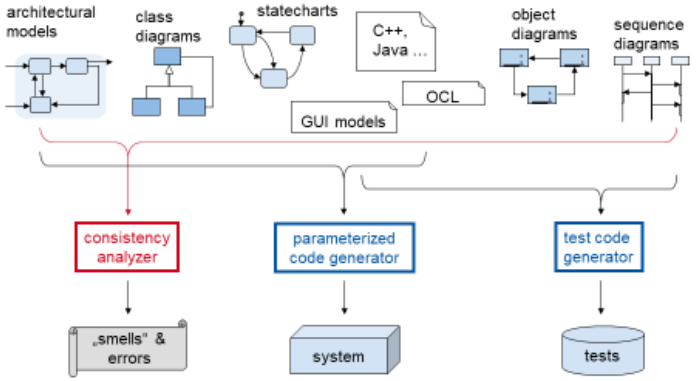
ATZelektronik 0112008 Jahrgang 3

Language:
DSL für Signal-/Schedule-Management (embedded in Doors)
Generated: "Envelope"-Functions for signal transport & safety.

with H. Rendel

43 Prof. Dr. Bernhard Rumpe | Software Engineering

Example UML: Constructive use of Models for Coding and Testing:



12 Prof. Dr. Bernhard Rumpe | Software Engineering

Some Examples For Usage of Our DSLs

Wie kann ein Gesetz als Modell definiert werden?
Beispiel natürlichsprachliche Modellierung: Fristen in der Abgabenordnung

```
§ 170 Beginn der Festsetzungsfrist
/* Absätze 1 bis 4 */
(5) FÜR die [Erbschaftsteuer (Schenkungsteuer): Wahrheitswert] BEGINNT die
    Festsetzungsfrist nach den Absätzen 1 oder 2

1. BEI einem [Erwerb von Todes wegen: Wahrheitswert] NICHT VOR [Ablauf des
   Kalenderjahrs, in dem der Erwerber Kenntnis von dem Erwerb erlangt hat: Datum],
2. BEI einer [Schenkung: Wahrheitswert] NICHT VOR [Ablauf des Kalenderjahrs, in dem
   der Schenker gestorben ist oder die Finanzbehörde von der vollzogenen Schenkung
   Kenntnis erlangt hat: Datum],
3. BEI einer [Zweckzuwendung unter Lebenden: Wahrheitswert] NICHT VOR
   [Ablauf des Kalenderjahrs, in dem die Verpflichtung erfüllt worden ist: Datum].
```

DSL-Variante für § 170 des Steuergesetzes:
- Vom Computer verarbeitbar
- für Menschen lesbar, verstehbar, veränderbar

(Teil einer NEGZ-Studie u.a. mit dem Bayerischen Landesamt für Steuern)

Milestone Definition Language

- Used for the PEP of a Car OEM
  - basis for project planning
- Menu layout and guidance
- Drag & drop of all graphical elements,
- Undo & redo
- Context-sensitive search, auto completion
- Wizard support for complex objects
- Development is based on a generic editor framework

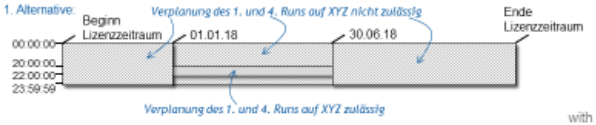
Languages:
DSL + Feature Diags for editor configuration;
Meta-DSL for modelling of diagram data

with T. Gülke, A. Horst

Description Language for Planning TV Broadcasting

- Planning audio-visual offers such as TV program and video-on-demand
- Restrictions in licensing contracts
- Risks: Misinterpretation and resulting planning errors
- DSL for
  - Verification of plans
  - Calculation of allowed planning periods

```
1 Alternative 1:
2 Der 1. & 4. Run innerhalb von 01.01.2018 bis 30.06.2018 turnus immer von 20:00 bis
3 22:00 auf Nutzer XYZ
4 Alternative 2:
5 Alle Run innerhalb von 01.01.2018 bis 30.06.2018 turnus ohne auf Nutzer XYZ
```



with I. Drave, K. Hölldobler, O. Kautz, J. Michael

MaCoCo: Full-size real world MDSE application

Problem:
Implement Enterprise Information System while new requirements emerge during development process

Solution:
Agile, evolutionary development of Information System infrastructure from common data structures, extend custom features with handwritten components

100+ instances (=databases) @RWTH Aachen
- Individual instances per chair/department
- One for all chairs of Faculty 1 (Mathematics, Computer Science and Natural Sciences)

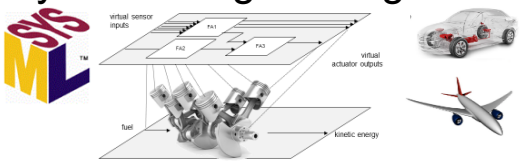
Handwritten code (Java, TS, HTML)
- 41.000 + 38.000 + 4.000 = 83.000 LOC
- app. 13% backend, 30% frontend

Generated code (Java, TS, HTML)
- 189.000 + 79.000 + 13.000 = 281.000 LOC

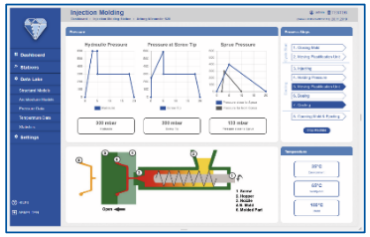
www.se-rwth.de/projects/MaCoCo.php
with A. Gerasimov, J. Michael, L. Netz, S. Varga

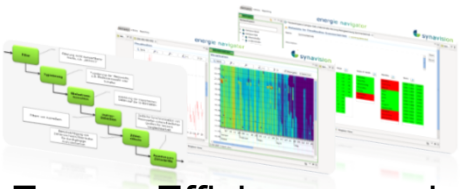
Software Engineering: Our Mission: Improving Software and Systems Development

Systems Engineering



Digital Twin Cockpits





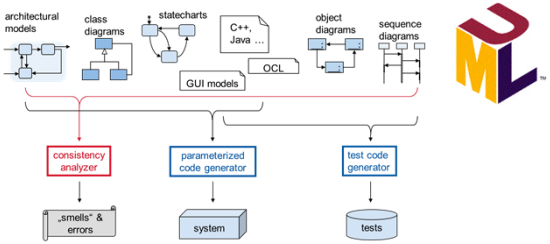
Energy Efficiency e.g. in Buildings

Contracts, Regulations, Laws, Requirements

Languages, methods, concepts, tools and infrastructures for

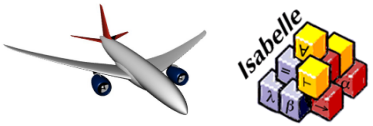
- better and faster agile development,
- resulting in high quality products.

Software, Tests, Deployment, Architecture, Design, Agility, Process management



Information Systems: Management Cockpits

Systems Verification



Software Languages, DSLs, LowCode

A basic question:

How to **engineer** a DSL?

Steps for Designing a Language

1. Understand the goal of the DSL:
   What to be described?
2. Write down examples of the DSL
3. Identify reusable language components from a language library
4. Build the language from components using extension and adaptation as glue for
   - abstract and concrete syntax,
   - symbols: names, kinds and visibility,
   - context conditions
5. What kinds of tools users need?
   - editor, wizard,
   - guidelines, metrics/smell analyzer,
   - code generator, interpreter,
   - diffing of models, etc.

Examples

Markdown

```
* list item 3
* list item 5
* list item 8
```

LaTeX

```
\alpha for α
```

Java

```
int a = 0, b = 1;
while (a <= 100) {
  System.out.println(a + " -> " + (a * a));
  int next = a + b; a = b; b = next;
}
```

Language Extension - Starter

- Lets start with one language L1

L1

- The automaton has
  - 2 states and
  - 2 transitions
  - describing a ping pong game

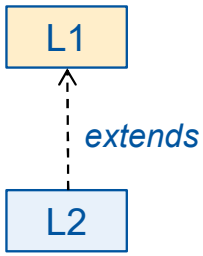
- Automaton language L1:

SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong

    Ping -> Pong
}
```

Language Extension

- L2 extends L1
  - by new language concepts



- One model contains language concepts of both languages
- Either L1 or L2 becomes the master language and the other the multiply embedded sub-language
- Semantics, code generation is often defined together, but ideally reuse L1-semantics, generators, etc. should be possible

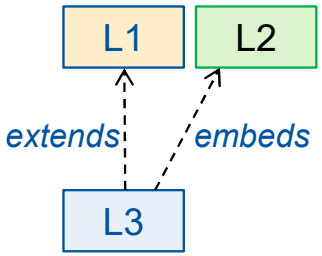
- Automaton language L1 is extended by actions in L2:
  - Actions are embedded at multiple places:

SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

    Ping -> Pong [ strokes++ ]
}
```

Language Embedding

- A new language L3 embeds model concepts from L2 in the language L2



- Models have parts conforming to sublanguages
- Languages L1 and L2 were independently developed
- Enables reuse and extension of languages
- Allows to define language components
  - E.g. expressions, literals, type definitions.

- Automaton language L1 and action language L2 are combined to a language embedding the actions into the automaton:

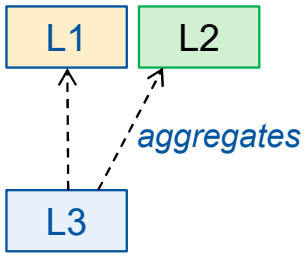
SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

    Ping -> Pong [ strokes++ ]
}
```

- “Glue” can be added, e.g. the square brackets

Language Aggregation

- An aggregated language
  L3 combines L1, L2, and more …



- Models are independent artefacts
  - they can be edited, reused, etc. individually
- Models are only semantically composed
  - there is no model belonging "only" to L3
- Models syntactically refer to each other
  - "Symbols" are imported / exported

- Two models:
  - An automaton and a java class sharing symbols (e.g. **`strokes`**)

SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ];
}
```

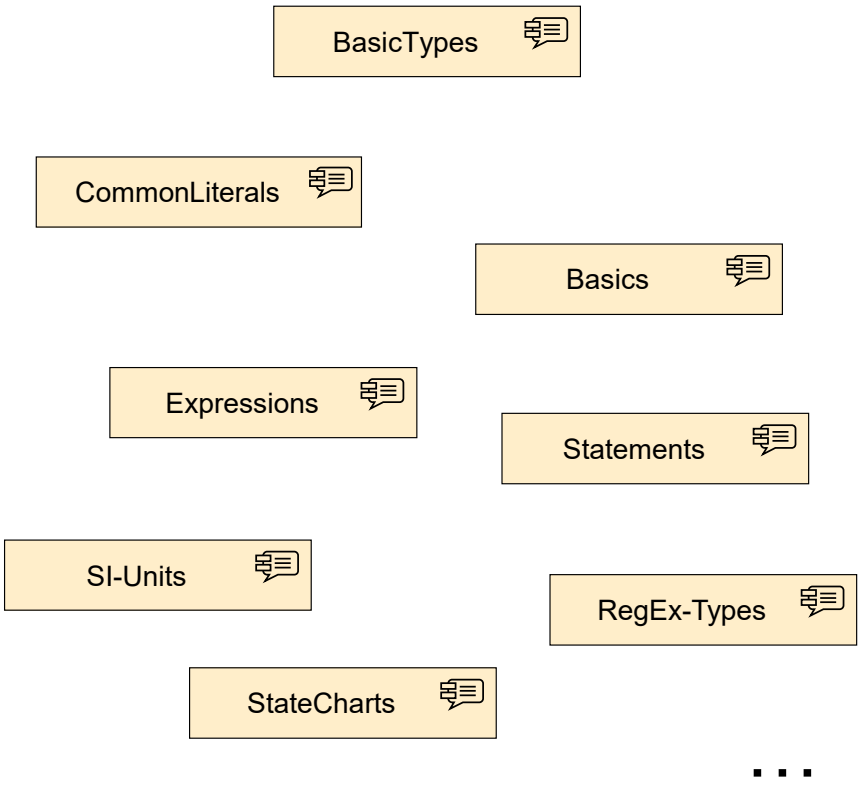
Java

```
class Game {
   Player a, b;
   int strokes = 0;
}
```

Language Component Library builds on Composition

A language component consists of
- one (or more) grammars
- handwritten extensions for integration
- additional handwritten Java classes (RTE)
- templates to generate code, reports,…

A language component library is a set of relatively independent language components.



- MontiCore provides a basic set of components:
  - www.monticore.de > Grammars
- and a set of complete languages:
  - www.monticore.de > Languages > List of Languages

In MontiCore: Grammars define a DSL

```
1  grammar StateChart {                                                    MG
2    Automaton  =  "automaton"  Name  "{"  Body  "}"
3    Body       =  ( State | Transition )*  ";"
4    State      =  "state"  Name  "<<initial>>"?
5    Transition =  from:Name  "-"  input:Name  ">"  to:Name    }
```

- MontiCore Grammar (MC) has
  - nonterminals,
  - keywords “…”,
  - grouping (…) ,
  - alternatives … | … ,
  - iteration …* , ...+ , (body | delimiter)*
  - optionals …?
  - token, etc.

- MontiCore generates from the grammar:
  - Abstract Syntax Tree - Classes
  - Parser + Tree Builder
  - Visitors
  - Pretty printing

- And furthermore:
  - Symbol Tables
  - Typecheck
  - Context Condition Infrastructure
  - Transformation engine, …

In MontiCore: Grammars extend other Grammars

```
1  grammar StateChart {                                              MG
2    Automaton    =  "automaton"  Name  "{"  Body  "}"
3    Body         =  ( State | Transition )*  ";"
4    State        =  "state"  Name  "<<initial>>"?
5    Transition   =  from:Name  "-"  input:Name  ">"  to:Name    }
```

redefines some NTs

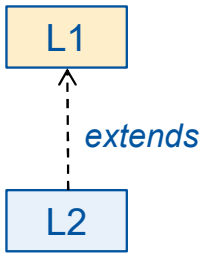
extends grammar

```
11 grammar MyStateChart extends StateChart, Expressions {            MG
12   Automaton    =  "statemachine"  Name  "{"  VariableDef*  Body  "}"
13   VariableDef  =  Name "=" Expression ";"
14   Transition   =  from:Name  "[" pre:Expression "]"  "-"  input:Name  ">"  to:Name    }
```

- MontiCore allows to extend grammars
  - and overwrite existing nonterminals (like in OOP)
- MontiCore generates only the delta and thus allows
  - black-box reuse of language components
  - (parser is an exception)

Conservative Language Extension

- L2 extends L1
  - by new language concepts



L2 is a conservative extension of L1, if Models(L2) are a superset of Models(L1)

- … enables reuse of models

L2 is a AST (metamodel) conservative extension of L1, if internal representation remains valid

- … enables reuse of L1-functionality also operating on L2-models

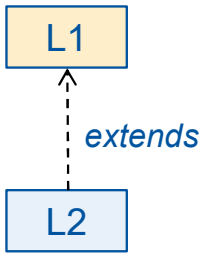
- Automaton language L1 is extended by actions in L2:
  - Actions are embedded at multiple places:

SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

    Ping -> Pong [ strokes++ ]
}
```

Language Restriction

- L2 extends L1
  - … by eliminating syntactic concepts or additional CoCos



L2 is a conservative restriction of L1, if Models(L2) are a subset of Models(L1)

- Enables:
  - Reuse of tooling and some of the models
  - Simpler L2-functions possible (e.g. generator may now be computable)

- Automaton language L2 does not allow the Java actions of L2 anymore:

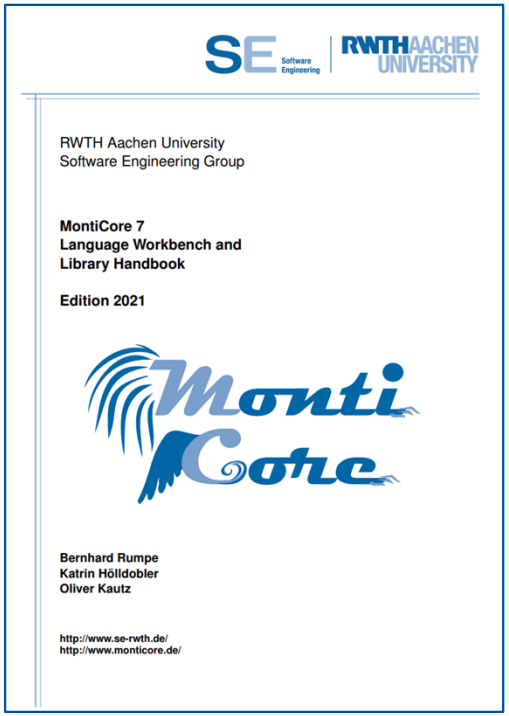
SC

```
behavior automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

    Ping -> Pong [ strokes++ ]
}
```

Examples:
- Forbid actions, forbid hierarchy of states
- Only basic types (integer …) used
- Disallow certain names,

MontiCore Language Workbench

- Definition of modular language components
- Quick definition of domain specific languages (DSLs)
- Library of existing languages
- Code generation
- Assistance for analysis
- Assistance for transformations
- Pretty printing, editors (graphical + textual)
- Namespaces/scopes, typing (fits GPL, UML)
- Variability in syntax, context conditions, generation, semantics



with NJ, Alu, MSh, FDr, AHe, FDr, DS, KH, AW, PN, AR, HK, SV, HG, MS, AHo, IW, AHa, AP, ML, GV, MB et.al.

MontiCore's Expressions as Language Components

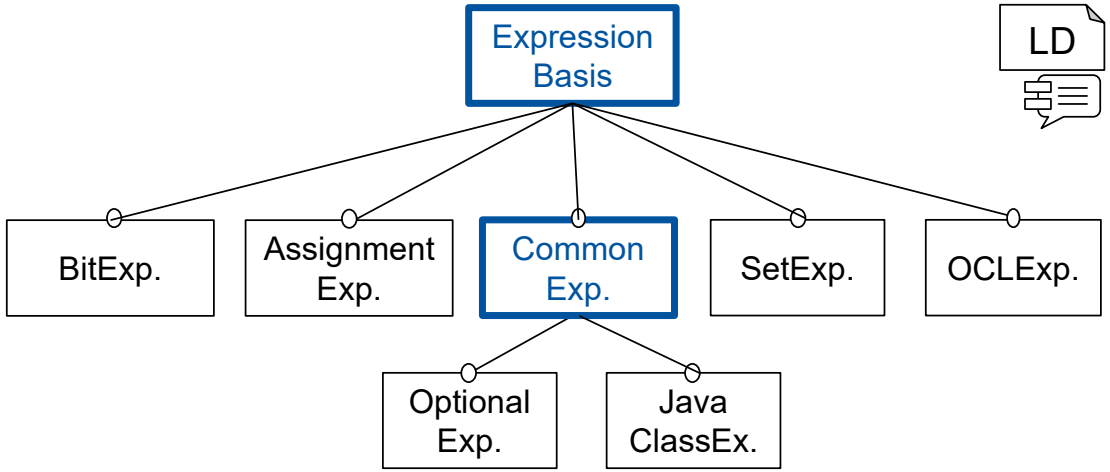
- 9 Grammars; $2^4 * (2^2 + 1) = 80$ variants
- Explanation to be found at:
  - MC/…/Grammars.md
  - Details: Expressions.md

- ~100 nonterminals, most relevant: Expression

- Type checks implemented as visitors
- After parsing a small predefined transformation is applied

- Example:

```
!(i+1 >= 3 * foo(1,0xFE,x))
```



Grammar	*Operators*
CommonExp:	/ % + - <= >= == > < != ~. !. .?.:. && || ~.
AssignExp:	++ -- = += -= *= /= &= |= ^= >>= >>>= <<= %=
BitExp:	& | ^ << >> <<< >>>
JavaClass:	this .[.] (.). super .instanceof.
SetExp:	.isin. .in. union intersect setand setor
OCLExp:	forall exists any let.in. .@pre {.|.}
Option.Exp:	?: ?<= ?>= ?< ?> ?== ?!= ?~~ ?!~

MontiCore's Literals as Language Components

- 4 Grammars; 4 variants
- Explanation to be found at:
  - MC/…/Grammars.md
  - Details: Literals.md

- Used for primitives in Expression grammars

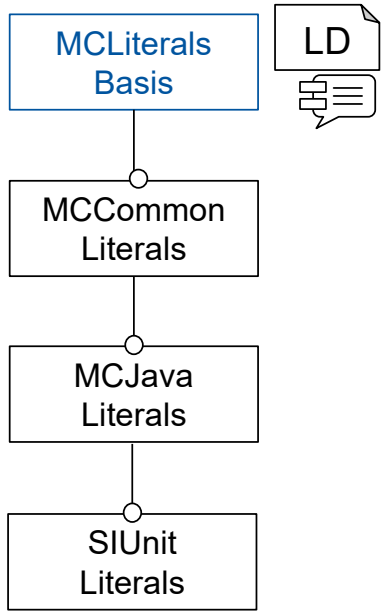
- ~95 nonterminals, most relevant:
  - Literal implements Expression

- Type checks implemented as visitors

- Example for an expression:

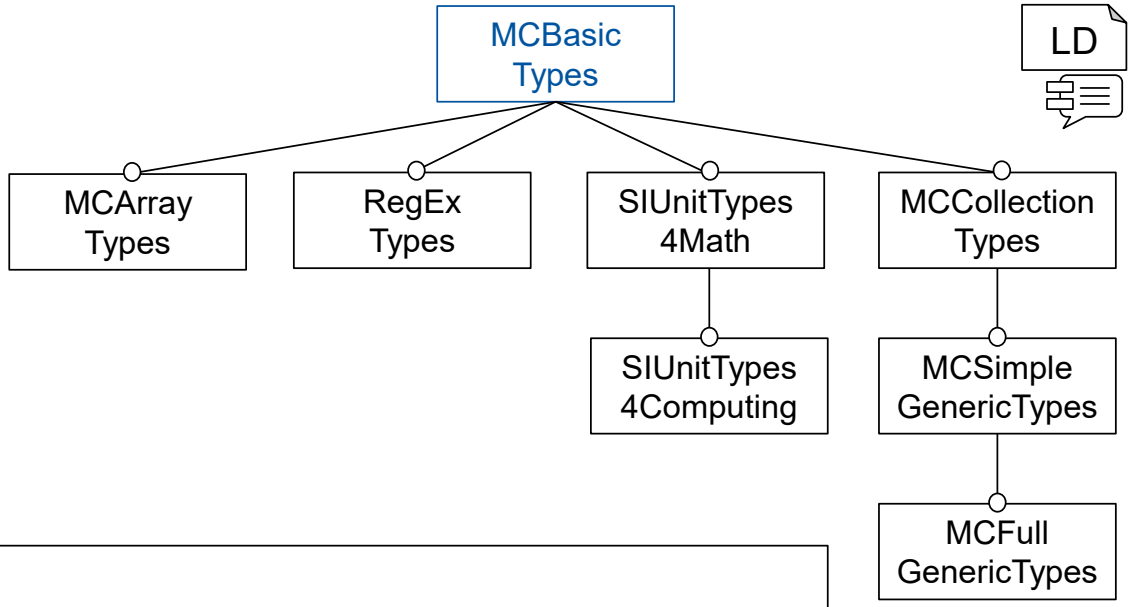
```
i + 1mm >= 3km/h * foo("!")
```

Literal language examples:	
MCCommonLit	3 -3 2.17 -4 true false 'c' '\03AE' 3L 2.17d 2.17f 0xAF "string" "str\b\n\\" "str\uAF01\u0001\377" null
MCJavaLiterals	999_999 0567 0x3F_2A 0b0001_0101 1.2e-7F
SIUnitLiterals	3km/h 2.7mg 1 km^2 3.54l m*deg/(h^2*mg)



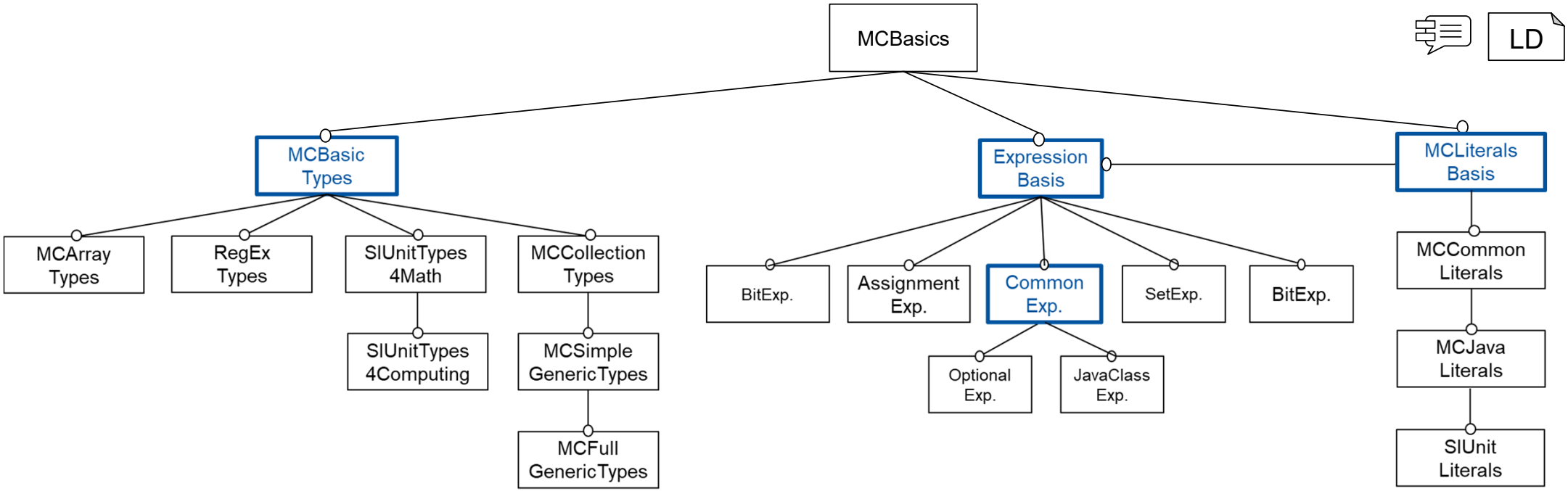
Types as Language Components

- 8 Grammars; 2*2*3*4 = 49 variants
- Explanation to be found at:
  - MC/…/Grammars.md
  - Details: Types.md
- ~30 nonterminals, most relevant:
  - `MCType` all allowed kinds of types
  - `MCQualifiedName` like in Java: `a.b.Foo`
  - `MCImportStatement` like in Java: `import a.b.*;`




```
Language examples:

MCBasicTypes           boolean byte short int long char float double void  Person a.b.Person
MCCollectionTypes      List<.>   Set<.>  Optional<.>   Map<.,.>
MCSimpleGenericTypes   Foo<.>  a.b.Bar<.,..,.>
MCFullGenericTypes     Foo<? extends .>    Foo<? super .>
MCArrayTypes           Person[][]
SIUnitTypes            kg  m/s  km^2
RegExTypes             R"goo(d(ies)?|gle)"   R"[^abx]+"       // define possible values
```

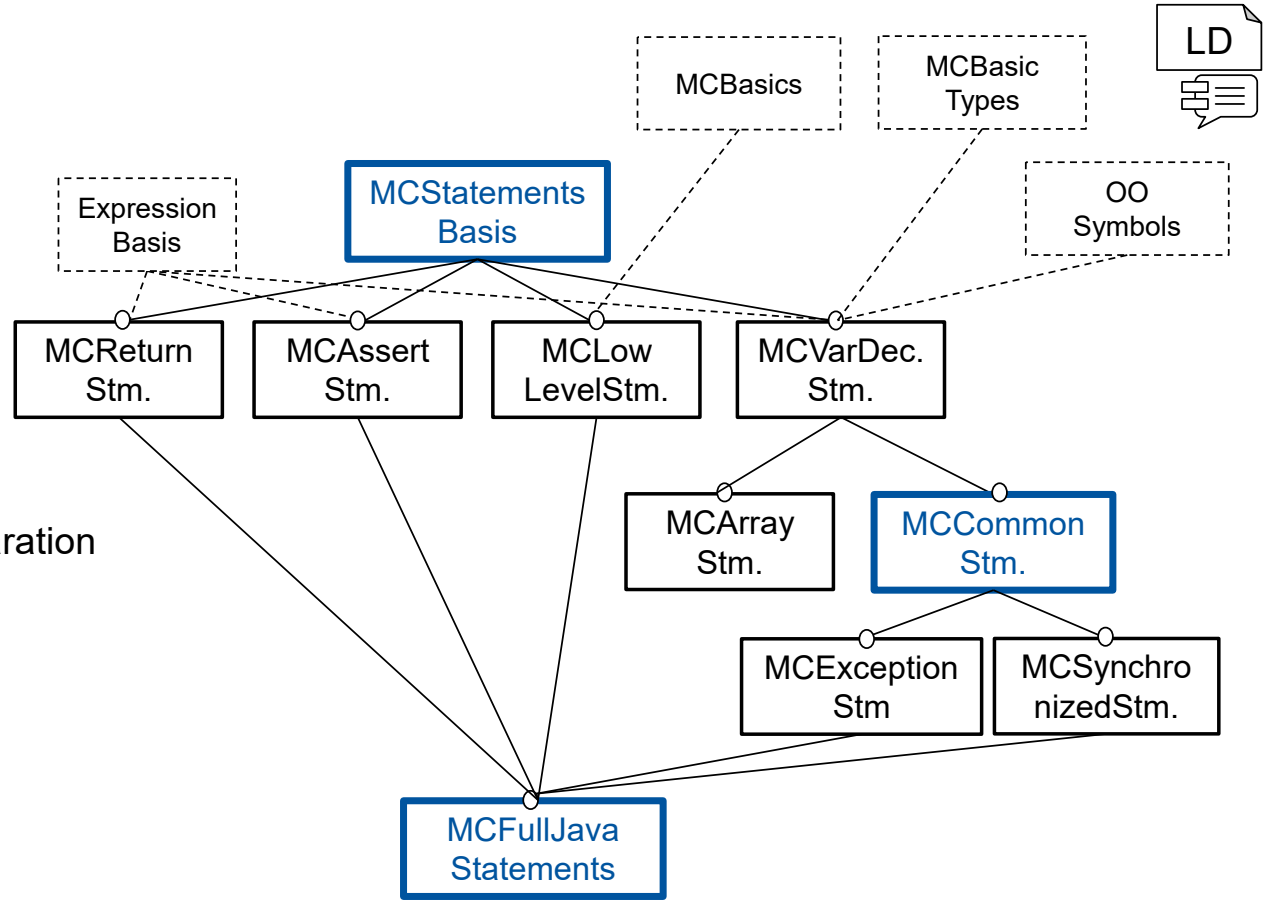
Summary: Component Hierarchy for Expression, Type and Literal



Overall Grammar Hierarchy for Expression, Type and Literals
15925 Variants possible

Statements as Language Components

- 10 Grammars; 90 variants
- Explanation to be found at:
  - MC/…/Grammars.md
  - Details: Statements.md
- ~45 nonterminals, most relevant:
  - `Statement` all allowed forms of statements
  - `MCBlockStatement` includes also var. declaration
- All statements / actions (~ Java) covered
- Statement also include Expressions, like i++ (when imported).
- Relies on Expressions, Types and others.



MontiCore Libraries of Reusable Language Components Build a Language Zoo:

Layer 3: "Multi-Viewpoint" Languages

UML

SysML / SpesML

MontiArc

BPMN

MontiGem

Layer 2: Focused Languages

Class D.

Sequence D.

Neuronal ANNA

Java

JSON

XML

Object D.

OCL

Feature D.

JavaLight

Statecharts

C, C++, Python

Base Layer: Components

MCBasics

Expressions

Statements

Cardinality

SI Units

MCCommon

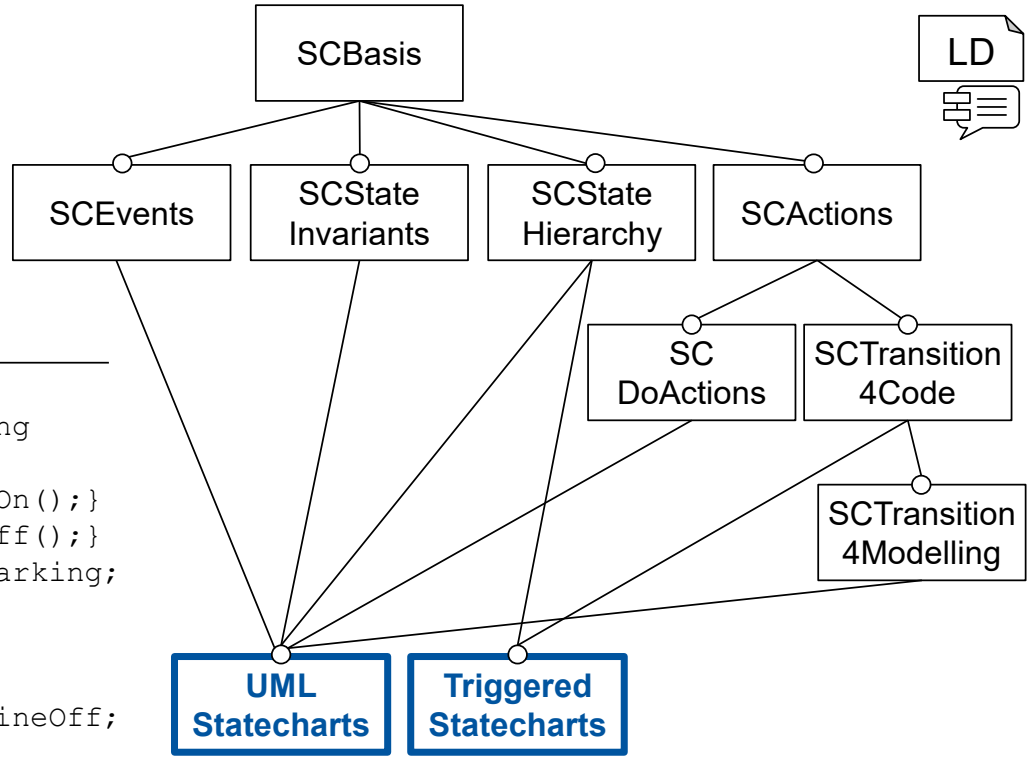
Literals

Types

Completeness

Statecharts

- Modeling of state-based behavior:
  - states, state hierarchies, and invariants
  - transitions with stimuli, conditions, and actions
  - entry-, exit-, and do-actions
- Statecharts are mapped to code through the state pattern
- Two key languages:
  - TriggeredStatecharts for describing automata with signal triggers
  - UMLStatecharts supports Statecharts with methods as triggers
- May import type, field, and function symbols from other languages
- #of Nonterminals: 29

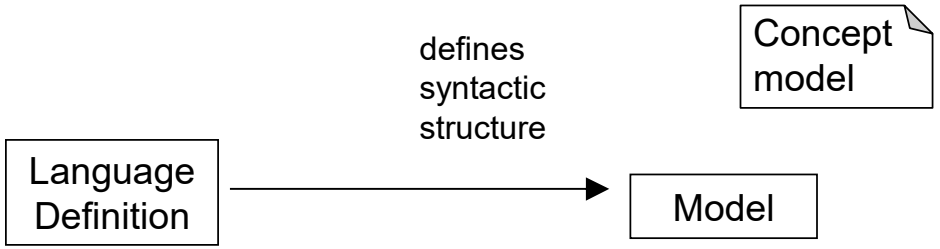



```
statechart Car {
  state EngineRunning
   [!fuelIsEmpty] {
    entry / {lightsOn();}
    exit / {lightsOff();}
    initial state Parking;
    state Driving;
  };

  initial state EngineOff;

  EngineOff -> EngineRunning on();
  EngineRunning -> EngineOff off();
}
```

Is the role of libraries in modelling underestimated?

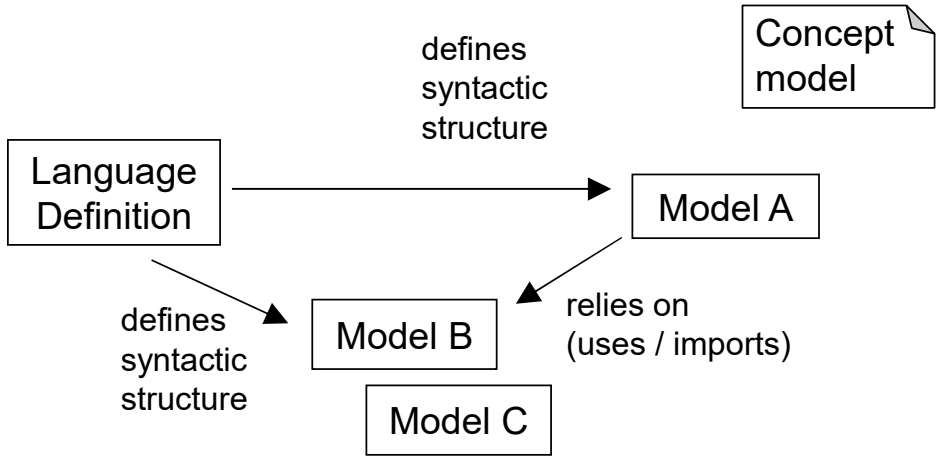
Role of Libraries in Language Definitions

- Language ~ set of wellformed models.
- Wellformedness is defined in two levels of constraints:
  1. Basic definition (context free syntax)
  2. Context conditions (aka constraints)



Role of Libraries in Language Definitions

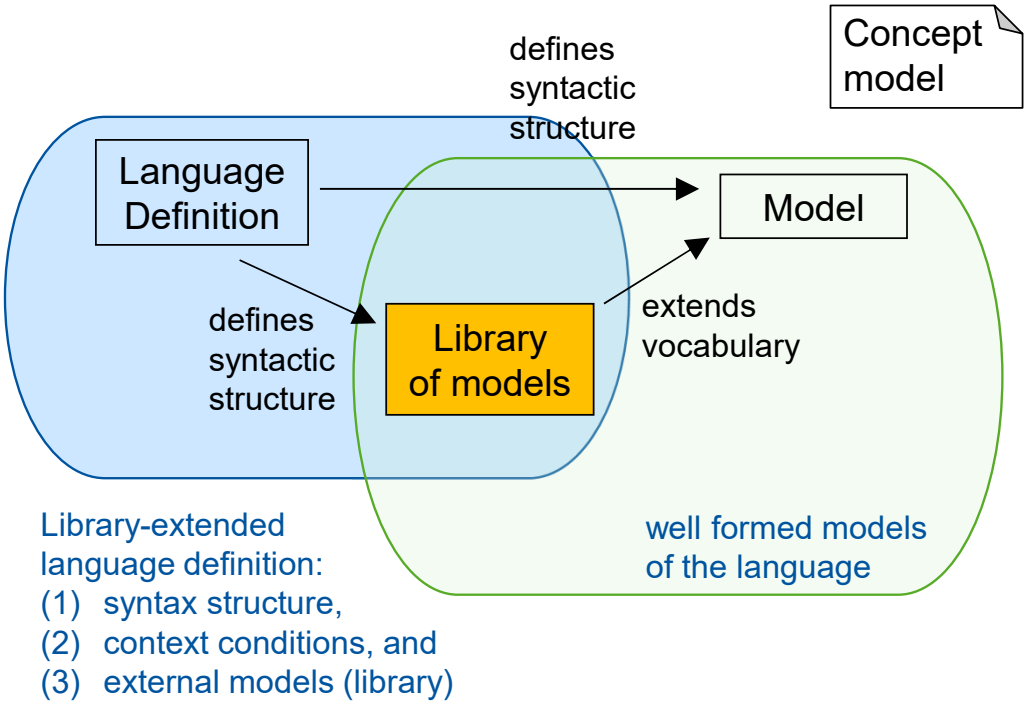
- A library ~ set of models that can be “imported”.
- A model library
  extends the language vocabulary
- We distinguish:
  - Language structure (such as given by a grammar)
  - Language vocabulary:
    - Usually lightweight extensions of a language that can be defined within the language itself



well formed models

Role of Libraries in Language Definitions

- Examples for vocabulary definitions:
  - Java allows to define classes, methods, variables
  - CD's allow to define classes,
  - Statecharts: states, …
  - Scala, C++ allow infix operations: .>>.
  - Natural language has glossaries
- Good languages allow
  - introduce new symbols,
  - define "meaning" using the language itself,
- Model libraries
  - 1) modularize/decompose models to allow reuse
  - 2) allow lightweight language extension

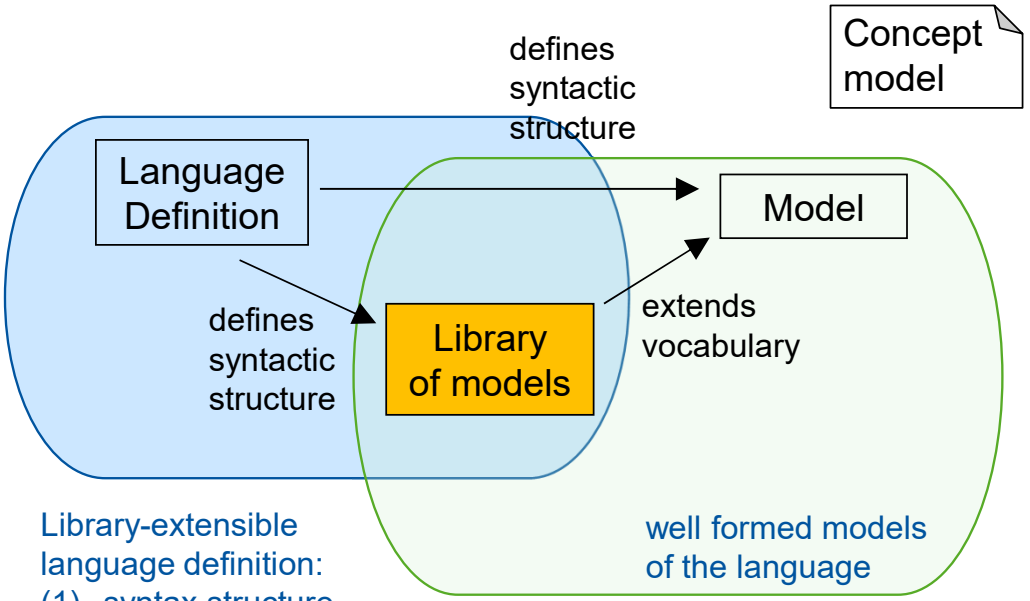


Extensible Language?

- A language is defined in three stages:

  1. syntactic structure
  2. context conditions
     → well-formed models:
  3. predefined, external models (libraries)
     → library-extensible language definition

- Consequence:
  - SLE needs to engineer extensible languages:
  - Definition of symbols, their meaning, and their use
  - Import / include / rely on other models



In MontiCore: Grammars define where symbols are defined and used

Old:

MG

```
1  grammar StateChart {
2                Automaton  =  "automaton"  Name  "{"  Body  "}"
3                Body       =  ( State | Transition )*  ";"
4                State      =  "state"  Name  "<<initial>>"?
5                Transition =  from:Name       "-"  input:Name  ">"  to:Name        }
```

Symbol enhanced grammar:

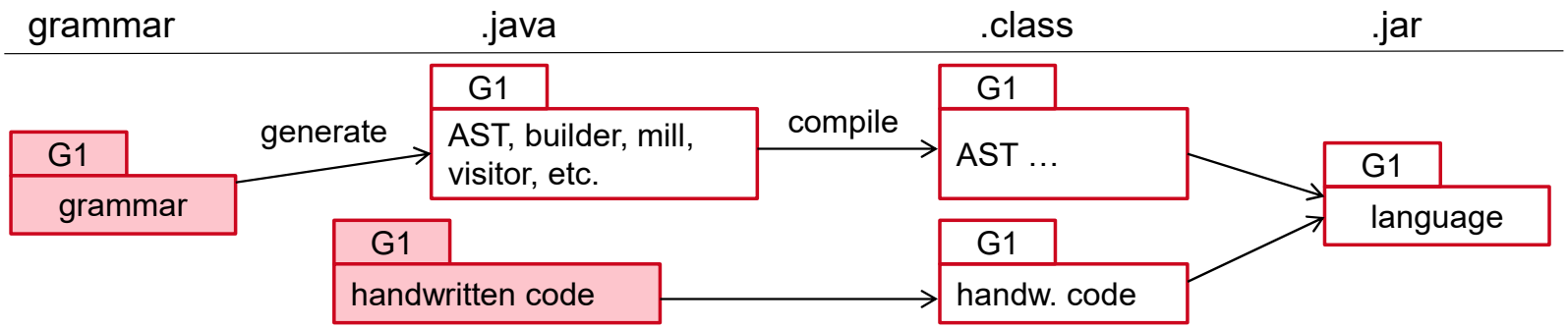
MG

```
11 grammar StateChart {
12   symbol scope Automaton  =  "automaton"  Name  "{"  Body  "}"
13                Body       =  ( State | Transition )*  ";"
14   symbol       State      =  "state"  Name  "<<initial>>"?
15                Transition =  from:Name@State  "-"  input:Name  ">"  to:Name@State   }
```

- „symbol“ : here symbols are defined
- „Name@State“ : here State-symbols are used

- MontiCore generates the full infrastructure for symbol management including typecheck, load/store symbol tables.
  (and its also language compositional!)

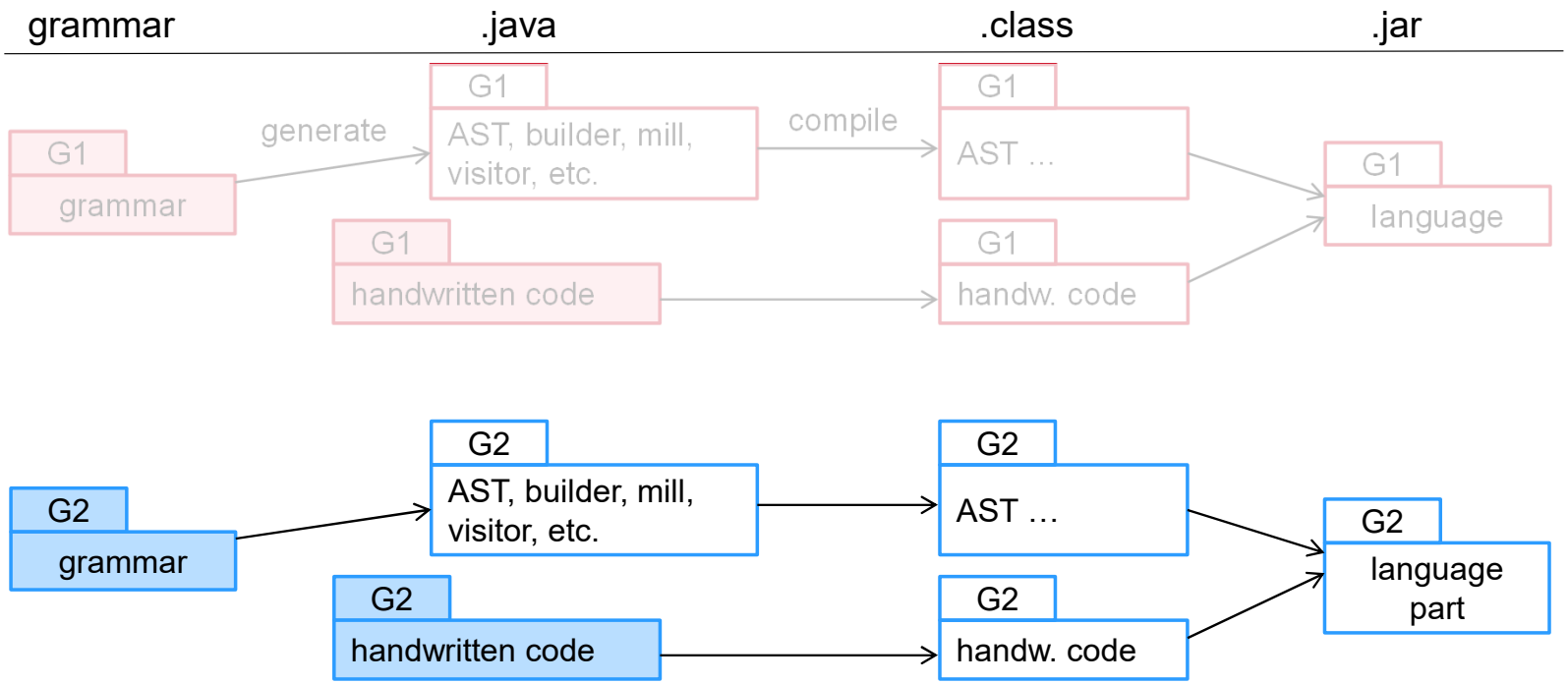
Generation Chain (in MontiCore)



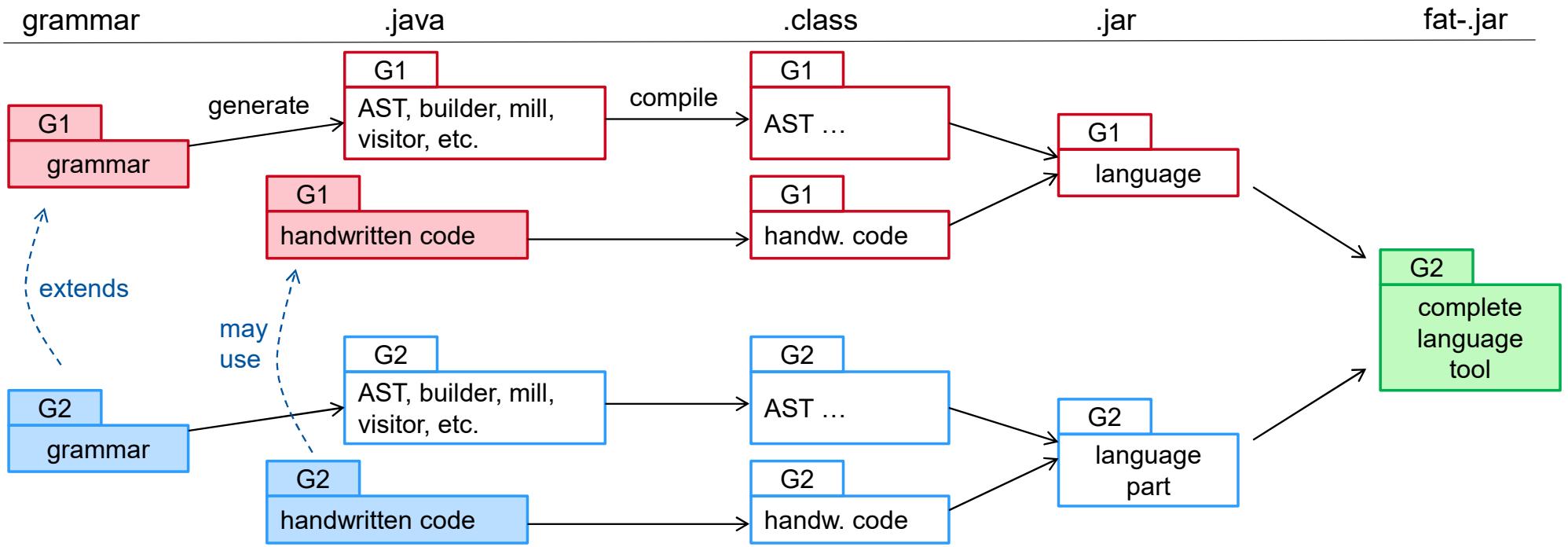
- MontiCore generates the full infrastructure for
  - syntax tree,
  - parsing,
  - traversing/visitors,
  - symbol management,
  - typecheck,
  - load/store symbol tables
  - syntax tree transformations
  - prettyprinting

- MontiCore automatically integrates handwritten code via Design Patterns
- (and NOT via copy/paste into generated code)

Modularity in the Generation Chain (in MontiCore)



Compositionality in the Generation Chain (in MontiCore)



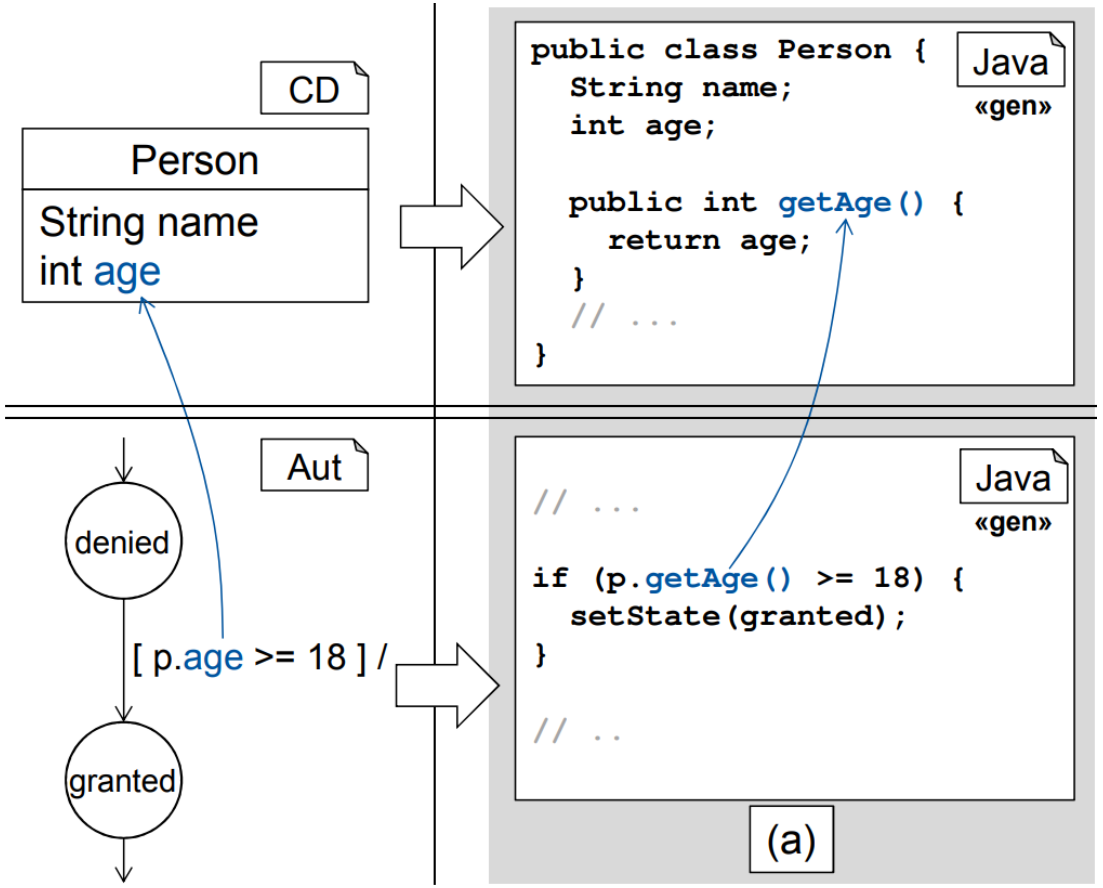
- Generated parts are compositional:
  - late binding of pre-generated/pre-compiled code
- and reuse of handwritten code is without adaptation

Question:

Composition of backends:
generators resp. their artifacts?

Generated Artifacts: They use, extend, configure each other: An Example

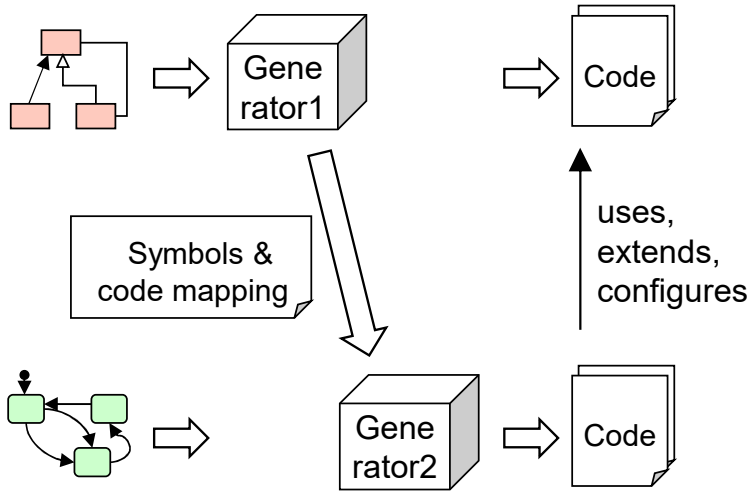
- Generator 1 creates class „Person“
  - and functions for attribute “age”
- Generator 2 needs to know,
  - how to assess „age“
    - Here using “getAge()”
  - and thus imports Person
- Gen.2 knows the result of Gen.1
- Gen.1 is independent of Gen.2



Modularity of Code Generation: Collaboration between Generators

- Main Solution: Modular mapping of models,
  - i.e. process only one model at a time: completely decoupled generators
  - code uses various design patterns
  - let generators "communicate" via stored symbol table
- Stored symbol table contains information about the mapping of a symbol to code (typically CRUD using Freemarker templates)
- Advantage:
  - Independent run; incremental re-run; efficient (if no circular dependency)
- MontiCore doesn't force to compose generator backends, but keeps them independent & communicating

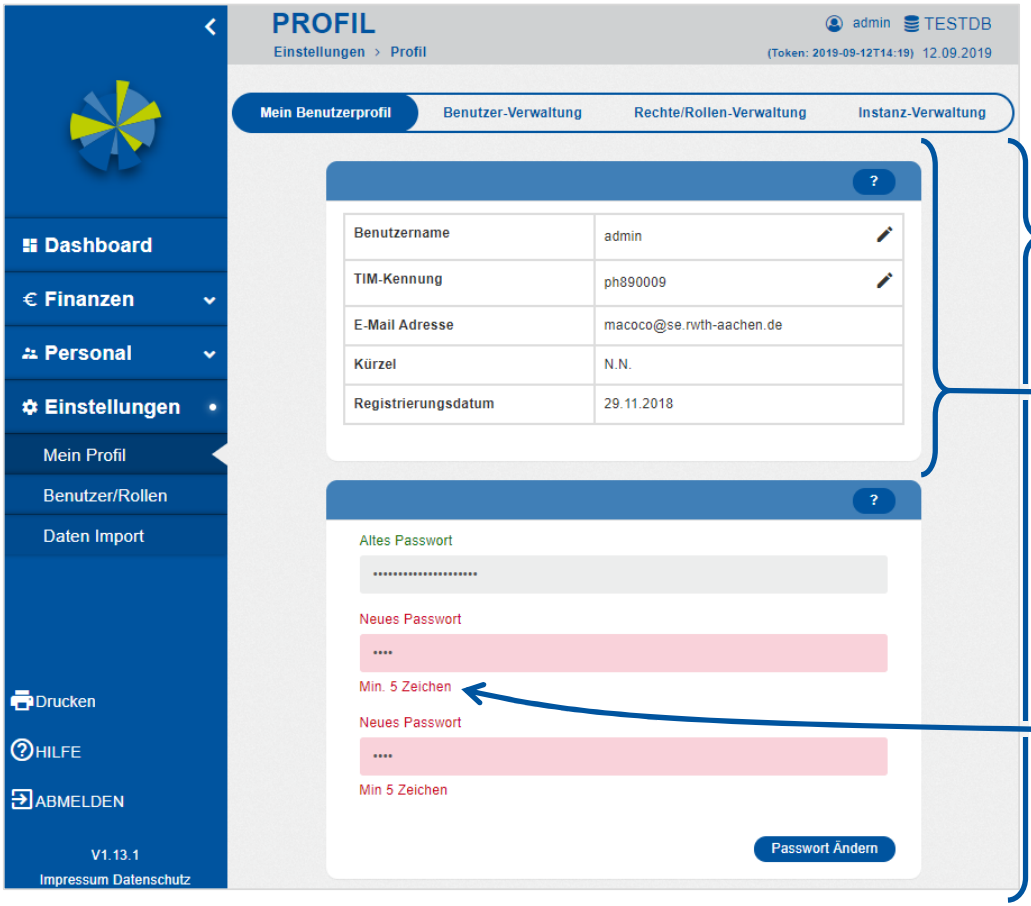
Solution approach:



Question:

What can be done with such an infrastructure?

GUI Modeling in MontiGem: For Information Systems, Digital Twins, IOT-Services



CD4A

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

GUI-DSL

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

OCL/P

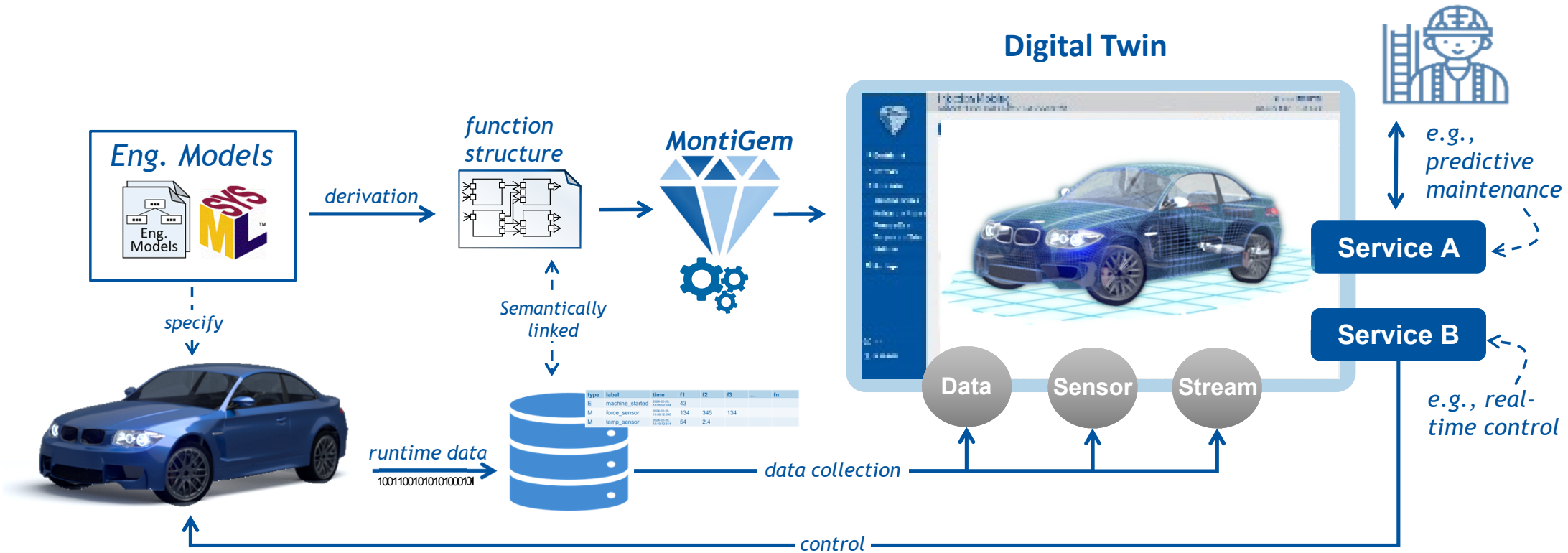
```
context User inv isPasswordValid:
  password.length() < 5;
  shortError: "Min. 5 Zeichen";
  error: "Das Passwort muss aus mindestens 5 Zeichen
    bestehen, hat aber nur " +
    passwort.length() + " Zeichen. ";
```

Data structure

User interface

Constraints

Transformation from the Engineering Models to the Digital Twin



Summary: Software Languages

- … are essential for the progress of digitalization.

- Engineering a language means:
  - language composition
  - … refinement
  - … embedding
  - … aggregation
  - … extension
  - … derivation
  - reusing language components
  - libraries of language components
  - tools

  - And language variants + tools are almost for free

- Ludwig Wittgenstein (philosopher):
  “The limits of my language are the limits of my world.”