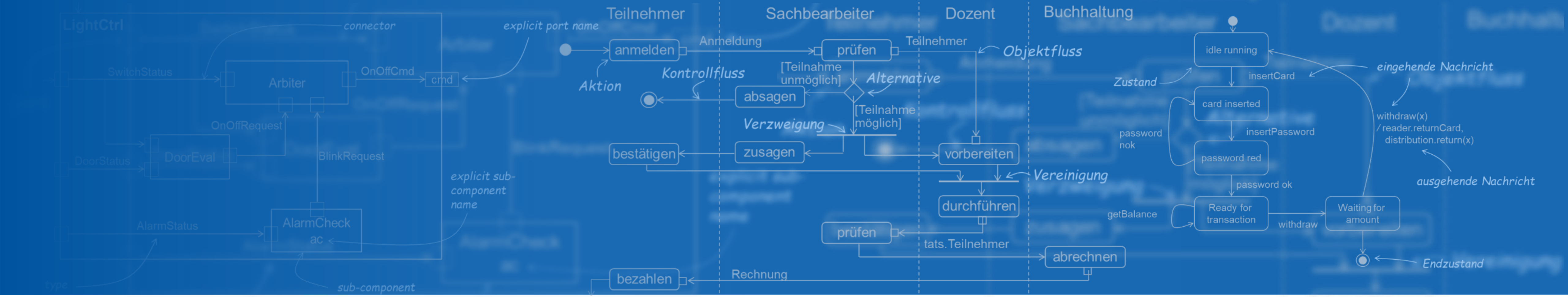


Seamless Generator Composition for Heterogeneous Modeling Languages

Nico Jansen
Software Engineering
RWTH Aachen

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

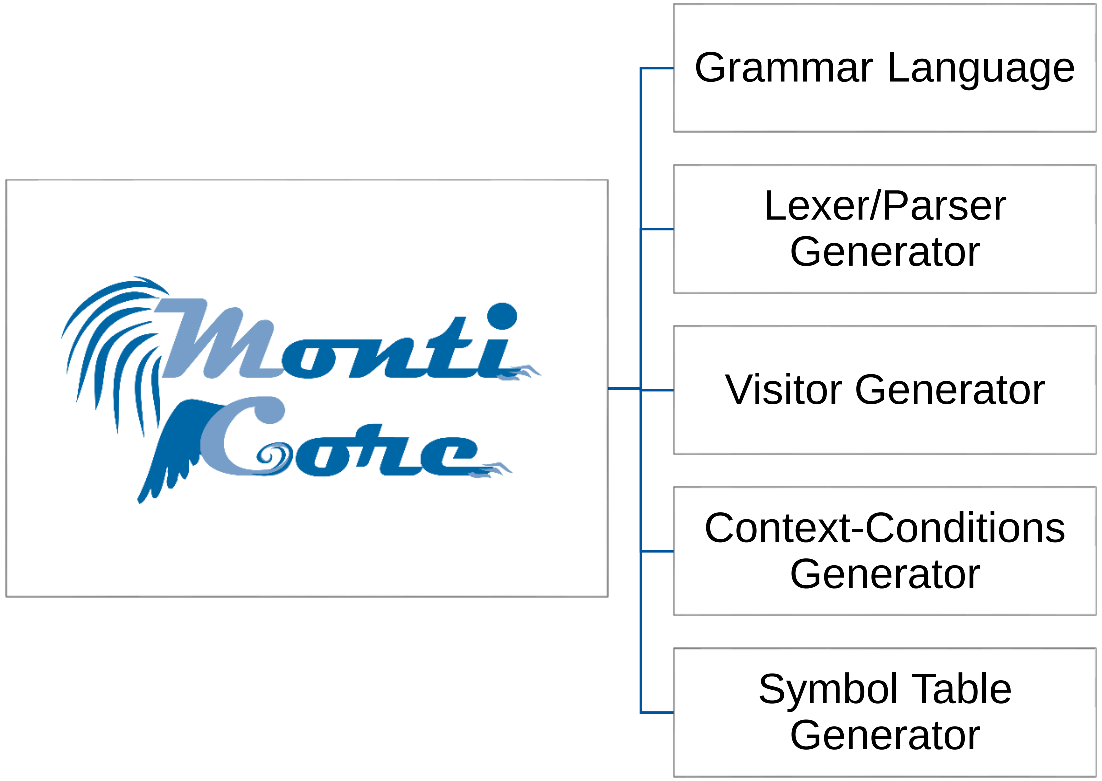
MontiCore – An Overview

Language Workbench MontiCore

- MontiCore is a language workbench (LWB) allowing to design DSL-tools.

- Common uses of DSL-tools:
  - **generating code**
  - generating tests
  - error detection, model and code analysis, metrics
  - synthesis, transformation

- History
  - Developed since 2004
  - Why? In 2004, the available tools were very poor in their functionalities and not extensible
  - Now: Flexible LWB for compositional language development



Grammar Language

Lexer/Parser Generator

Visitor Generator

Context-Conditions Generator

Symbol Table Generator

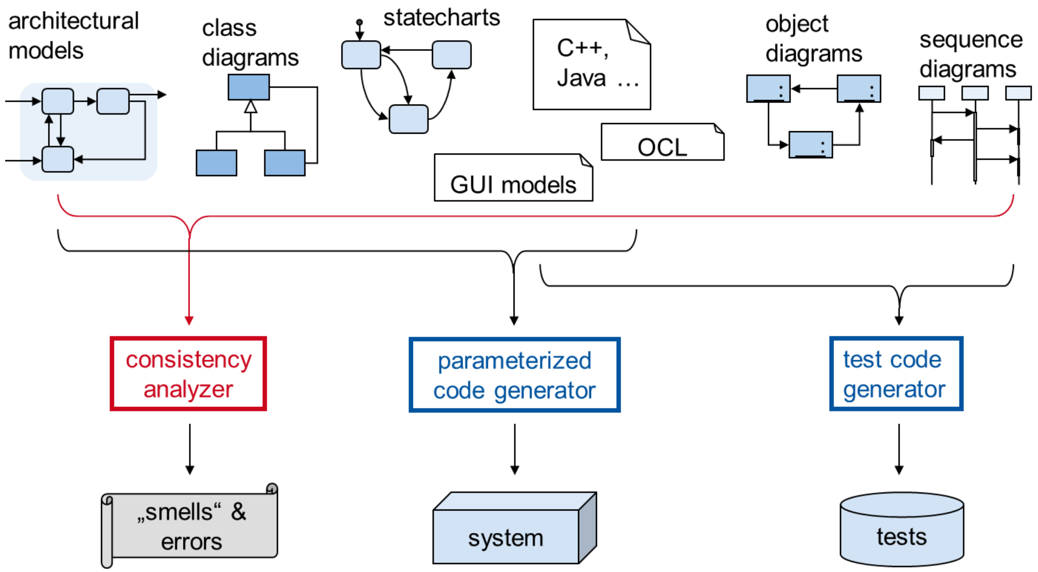
MontiCore Goals

Language & tooling workbench
MontiCore

- Definition of modular language components
- Interfaces between models/language components
  - Name spaces, typing (~ Java, UML)
  - Symbol „kinds“ + signatures
- Assistance for analysis and synthesis
- Assistance for transformations
- Pretty printing, editors (graphical + textual)

- Composition of languages:
  - independent language development
  - composition of languages and tools
  - language extension, aggregation
  - language inheritance (allows replacement)

- Quick definition of domain specific languages (DSLs)
  - by reusing existing languages
  - variability in syntax, context conditions, generation, semantics



architectural models

class diagrams

statecharts

C++, Java …

OCL

GUI models

object diagrams

sequence diagrams

consistency analyzer

parameterized code generator

test code generator

„smells“ & errors

system

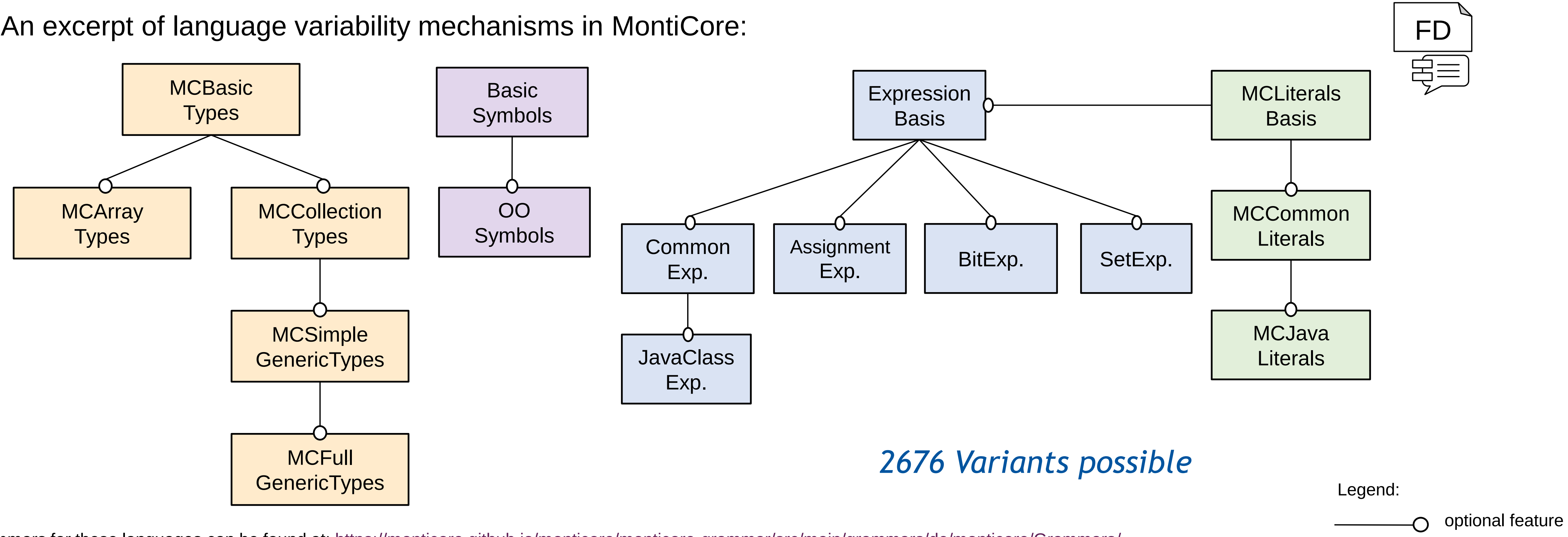
tests

Use of Models for Coding and Testing

Feature Diagram for MontiCore Language Components

- MontiCore provides a set of language components that can be used as features
  - Some dependencies exist, e.g. certain expressions rely on appropriate literals

- An excerpt of language variability mechanisms in MontiCore:

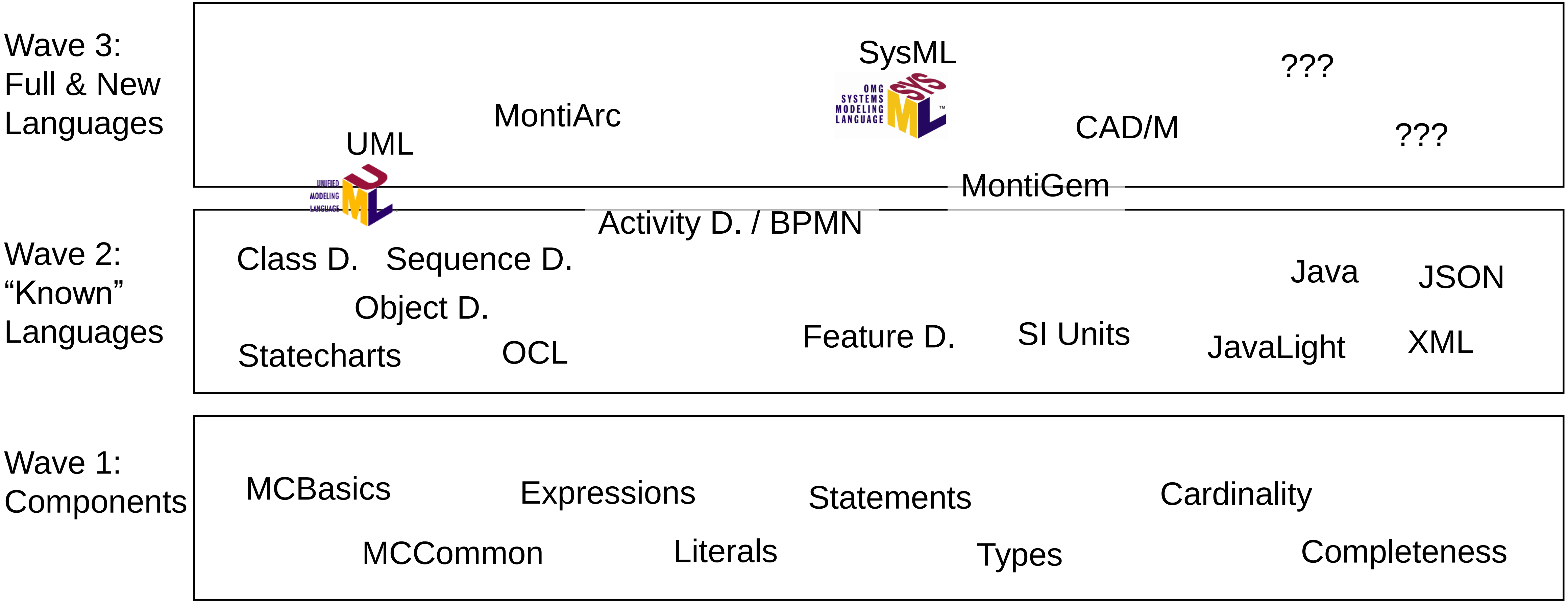


2676 Variants possible

Grammars for these languages can be found at: https://monticore.github.io/monticore/monticore-grammar/src/main/grammars/de/monticore/Grammars/

MontiCore Language Zoo: Development in three Waves

- Language library built in three phases



Wave 3:
Full & New
Languages

SysML

???

MontiArc

CAD/M

UML

???

MontiGem

Activity D. / BPMN

Wave 2:
“Known”
Languages

Class D. Sequence D.

Java

JSON

Object D.

Feature D.

SI Units

JavaLight

XML

Statecharts

OCL

Wave 1:
Components

MCBasics

Expressions

Statements

Cardinality

MCCommon

Literals

Types

Completeness

Legend: Many of these languages are defined using several grammars, CoCo-sets, etc.

MontiCore – Compositional Language Design

Language Extension

- Lets start with one language L1

L1

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

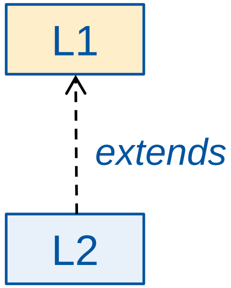
- Automaton language L1:

SC

```
automaton PingPong {

    state Ping, Pong;

    Ping -> Pong

    Pong -> Ping
}
```

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

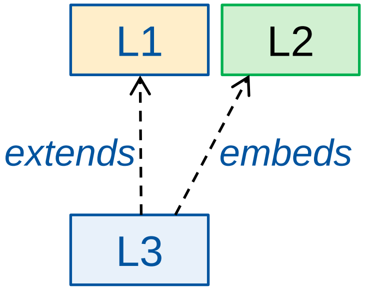
- Automaton language L1 is extended by actions in L2:
  - Actions are embedded at multiple places:

SC

```
automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

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

Language Embedding

- A new language L3 embeds model concepts from L2 in the language L1



- 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:

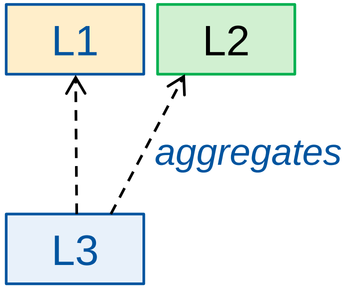
SC

```
automaton PingPong {

    state Ping, Pong;

    Ping -> Pong [ strokes++ ]

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

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

Language Aggregation

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



- Models are independent artifacts
  - 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

```
automaton PingPong {

    state Ping, Pong;

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

CD

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

Cross-Referencing & Symbol Resolution

- Symbol usages are often realized as names or expressions
  - Qualification often contained in import statements

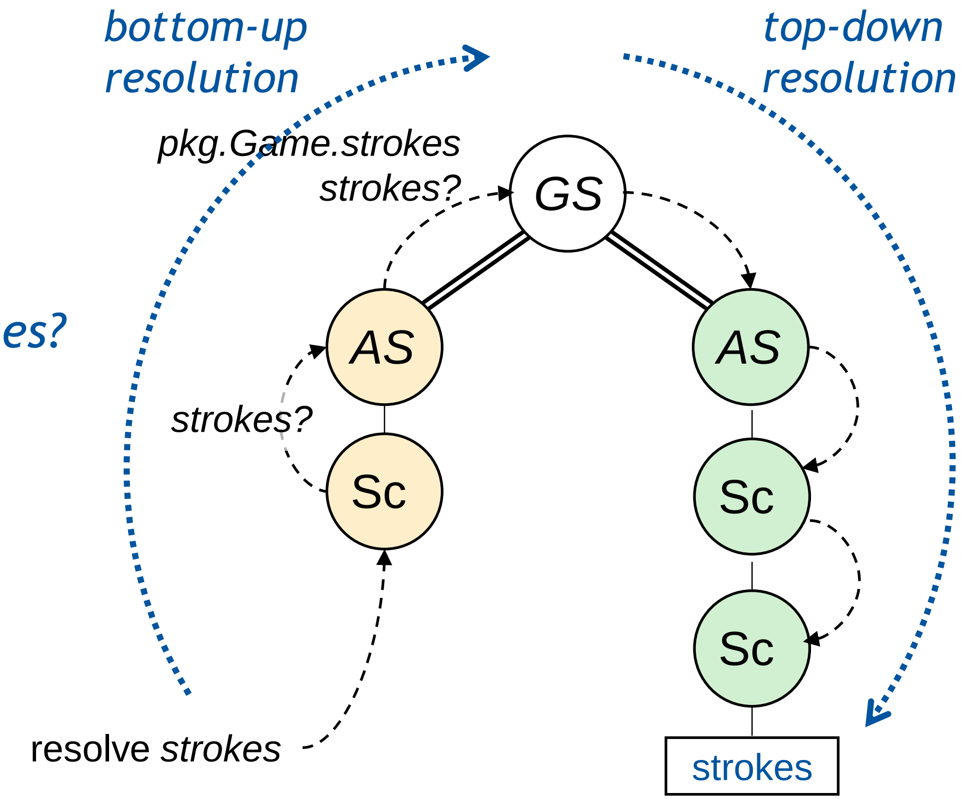
SC

```
import pkg.Game.*;

automaton PingPong {

    state Ping, Pong;

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

strokes?
play.Game.strokes?

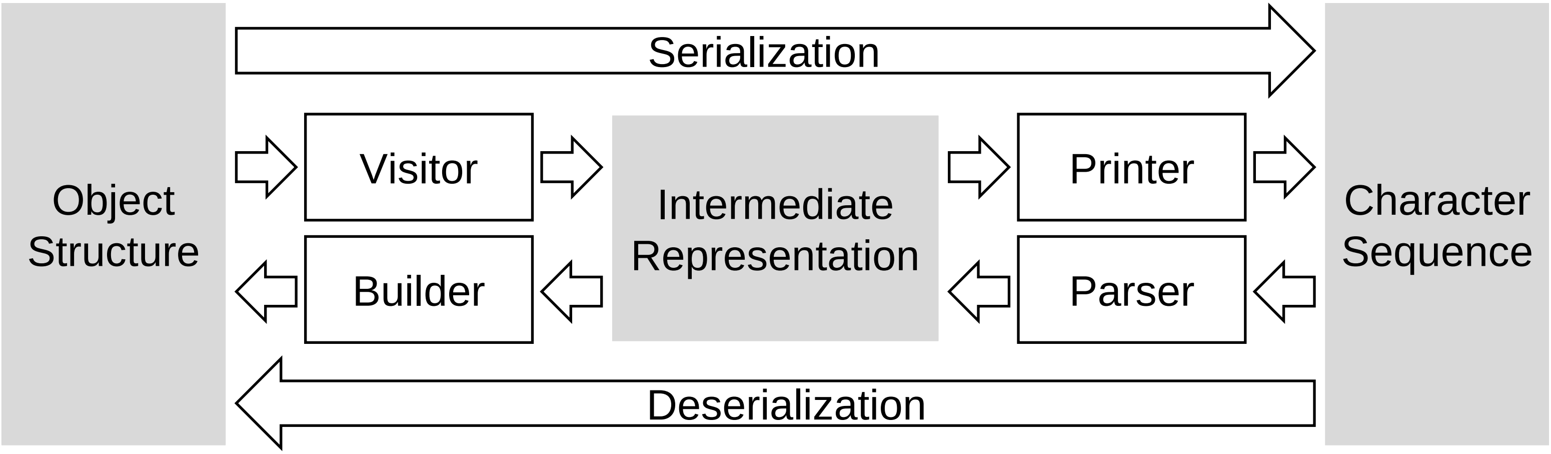
CD

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



Serialization with Intermediate Representation

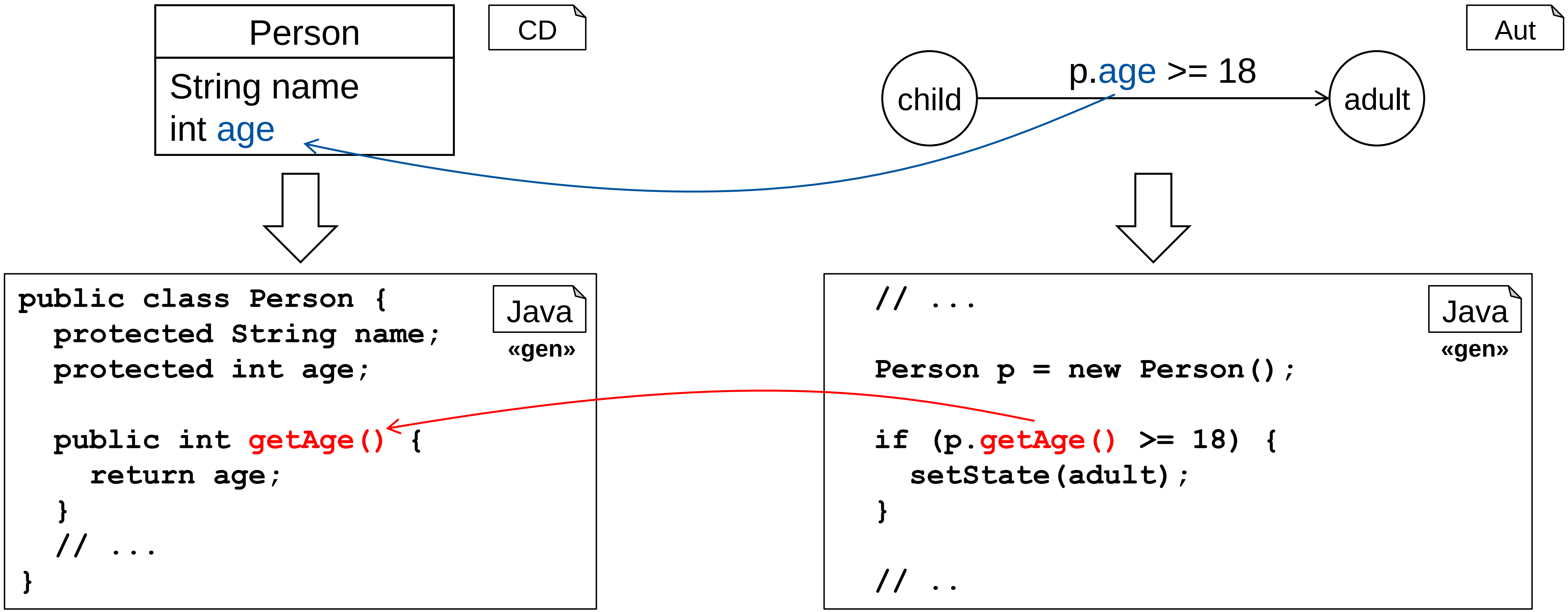
- **Serialization**: Translating object structure into character sequence
- **Deserialization**: Translating character sequence into object structure
- Character sequence is encoded in an encoding format (e.g., JSON, OD, but also binary formats)
- Efficient (de)serialization has to be aware of types that object structures conform to
- The **serialization strategy** describes how to translate between objects of a type and encoding format
- Transformation into an intermediate structure enables separating type-specific parts from type-agnostic parts
  - type-specific parts can be generated
  - relieves usage of reflection



MontiCore – Generator Composition for Aggregated Languages

Seamlessly Composing Generated Artifacts

- Challenge: Integrating Generated Artifacts of heterogeneous generators (mostly of different languages)

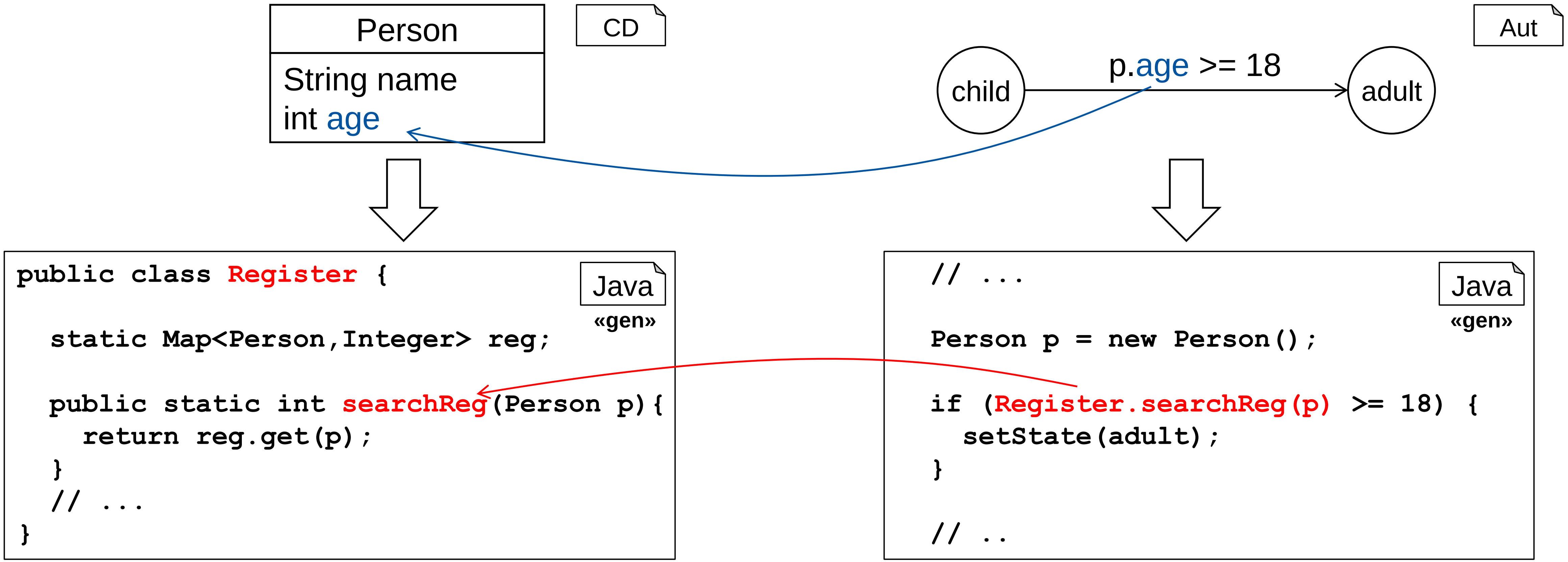


```
public class Person {
  protected String name;
  protected int age;

  public int getAge() {
    return age;
  }
  // ...
}
```

```
// ...

Person p = new Person();

if (p.getAge() >= 18) {
  setState(adult);
}

// ..
```

Seamlessly Composing Generated Artifacts

- Challenge: Integrating Generated Artifacts of heterogeneous generators (mostly of different languages)

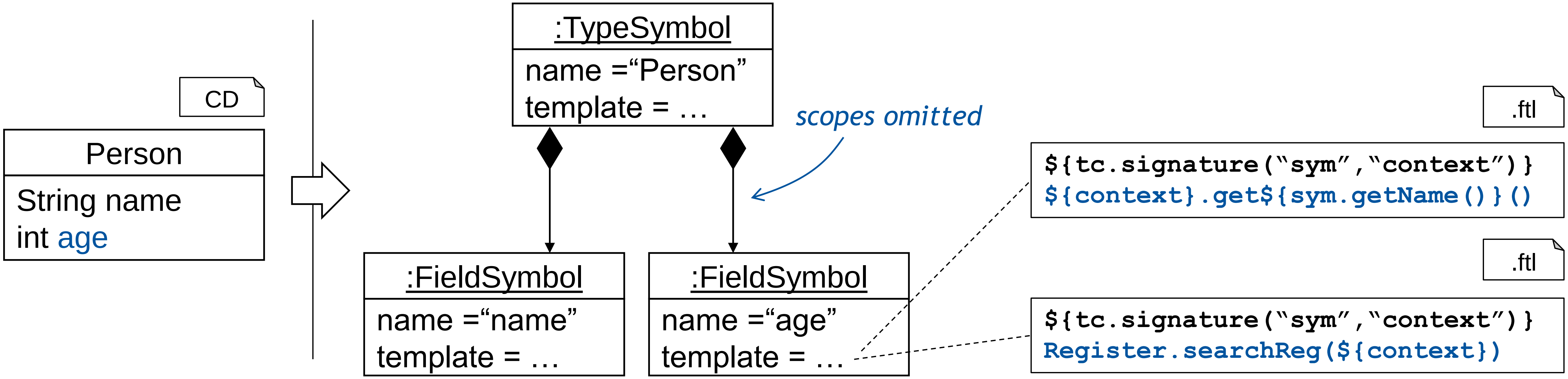


```
public class Register {

  static Map<Person,Integer> reg;

  public static int searchReg(Person p){
    return reg.get(p);
  }
  // ...
}
```

```
// ...

Person p = new Person();

if (Register.searchReg(p) >= 18) {
  setState(adult);
}

// ..
```

Composing Generators via Symbol Table

- Each symbol gets one (possibly more) templates that carries the corresponding accessor code
- When translating expressions in Freemarker templates, resolve for symbol and extract accessor

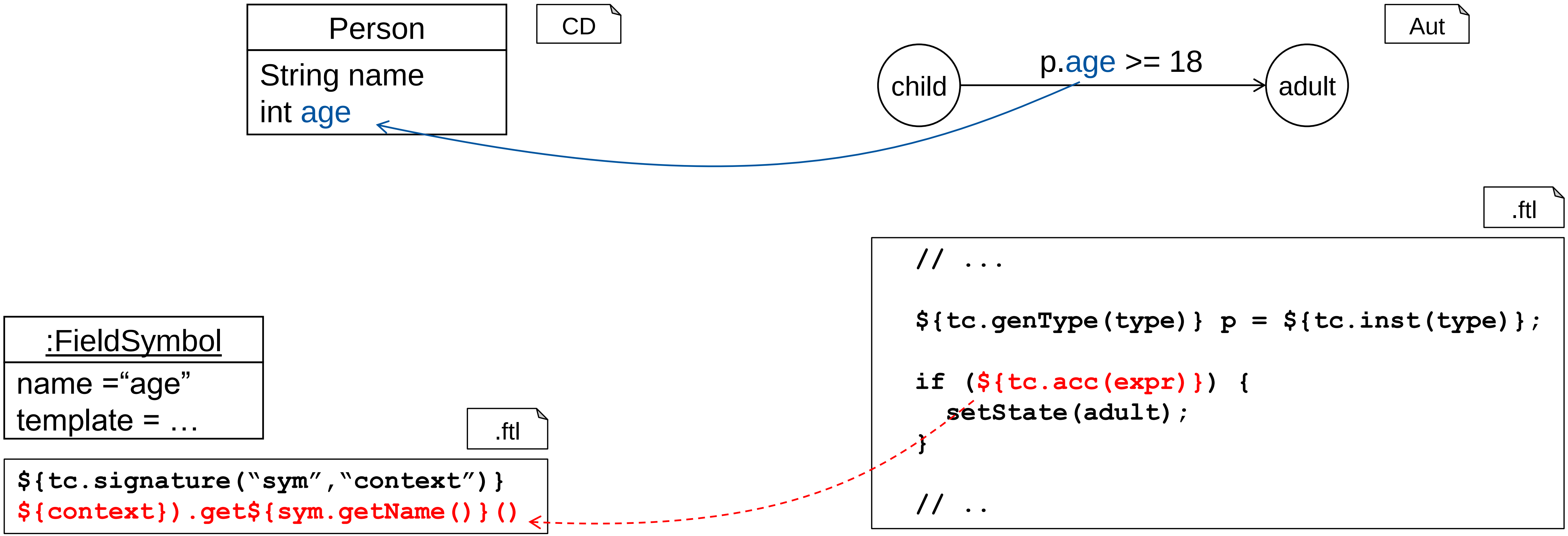


```
${tc.signature("sym","context")}
${context}.get${sym.getName()}()
```

```
${tc.signature("sym","context")}
Register.searchReg(${context})
```

Seamlessly Composing Generated Artifacts – Generator View

- Challenge: Integrating Generated Artifacts of heterogeneous generators (mostly of different languages)



```
${tc.signature(“sym”,“context”)}
${context}).get${sym.getName()}()
```

```
// ...

${tc.genType(type)} p = ${tc.inst(type)};

if (${tc.acc(expr)}) {
  setState(adult);
}

// ..
```

Thank you
for your attention