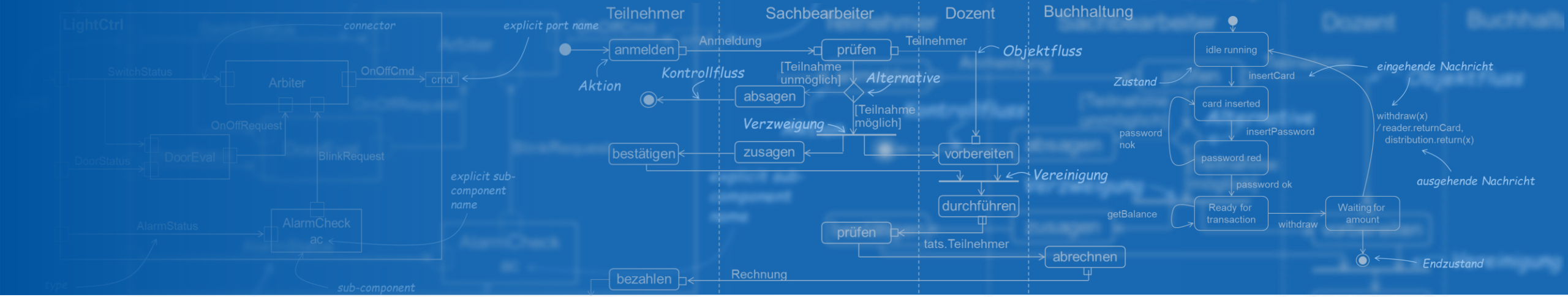


Seamless Interpreter Integration for Compositional Modeling Languages

Nico Jansen, Bernhard Rumpe

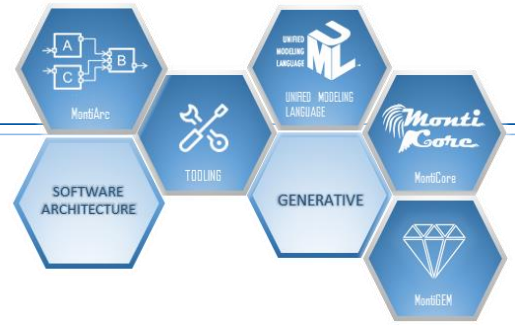
Software Engineering
RWTH Aachen

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

MontiCore – An Overview

MontiCore Goals

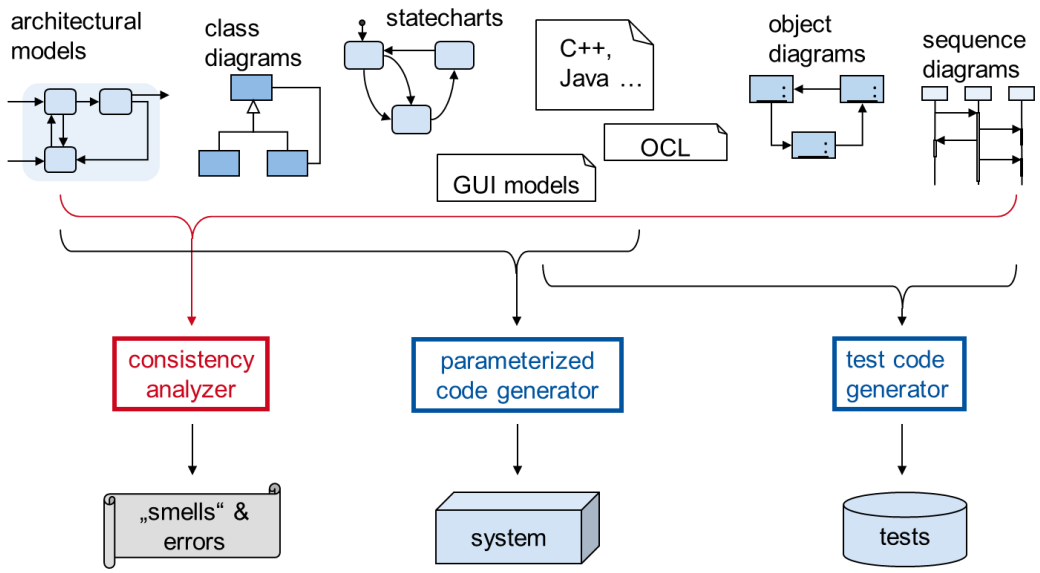
Language & tooling workbench
MontiCore



- 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)

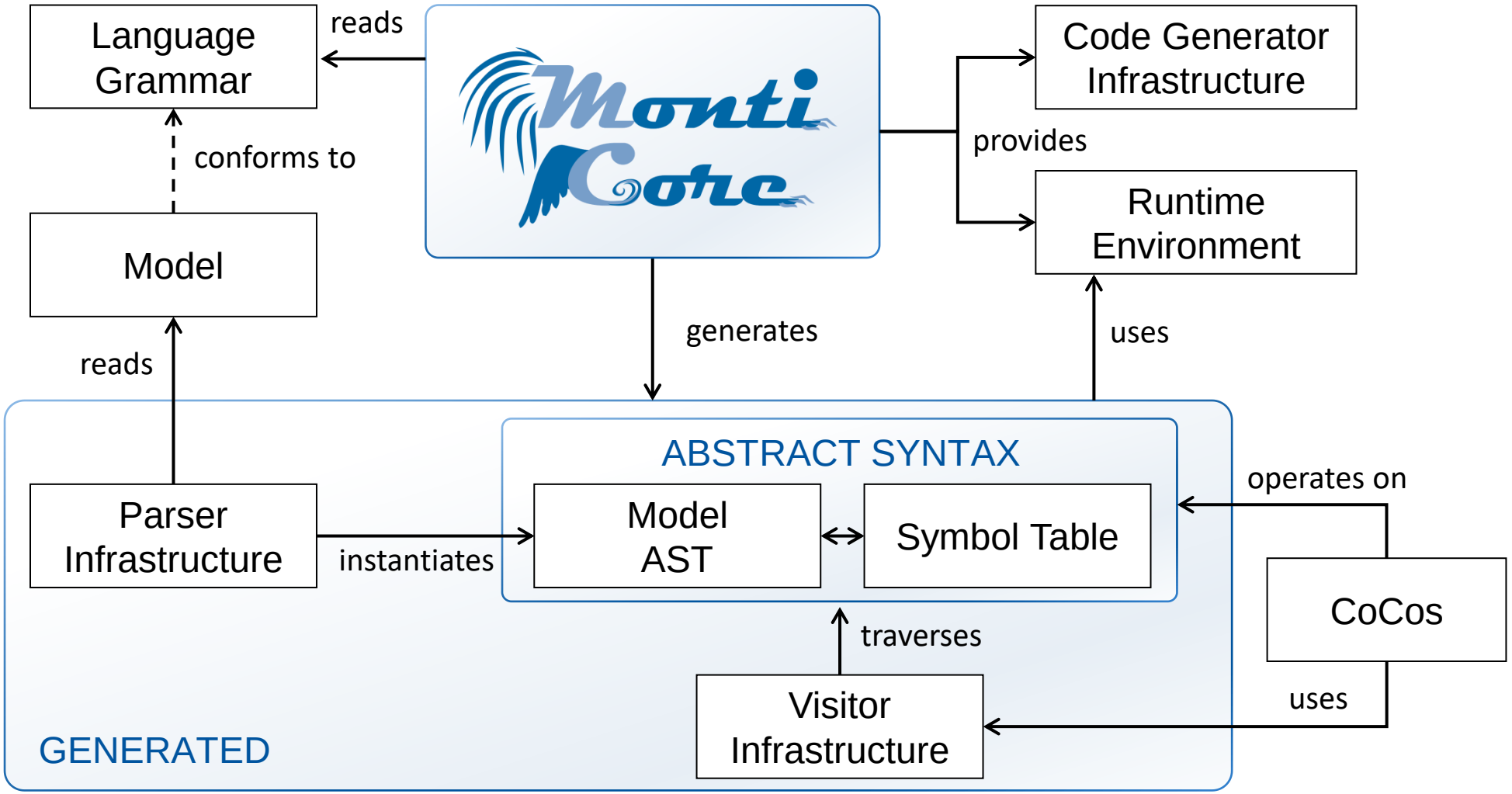
- Composition of languages:
  - independent language development
  - composition of languages and tools
  - language extension, aggregation
  - language inheritance (allows replacement)

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



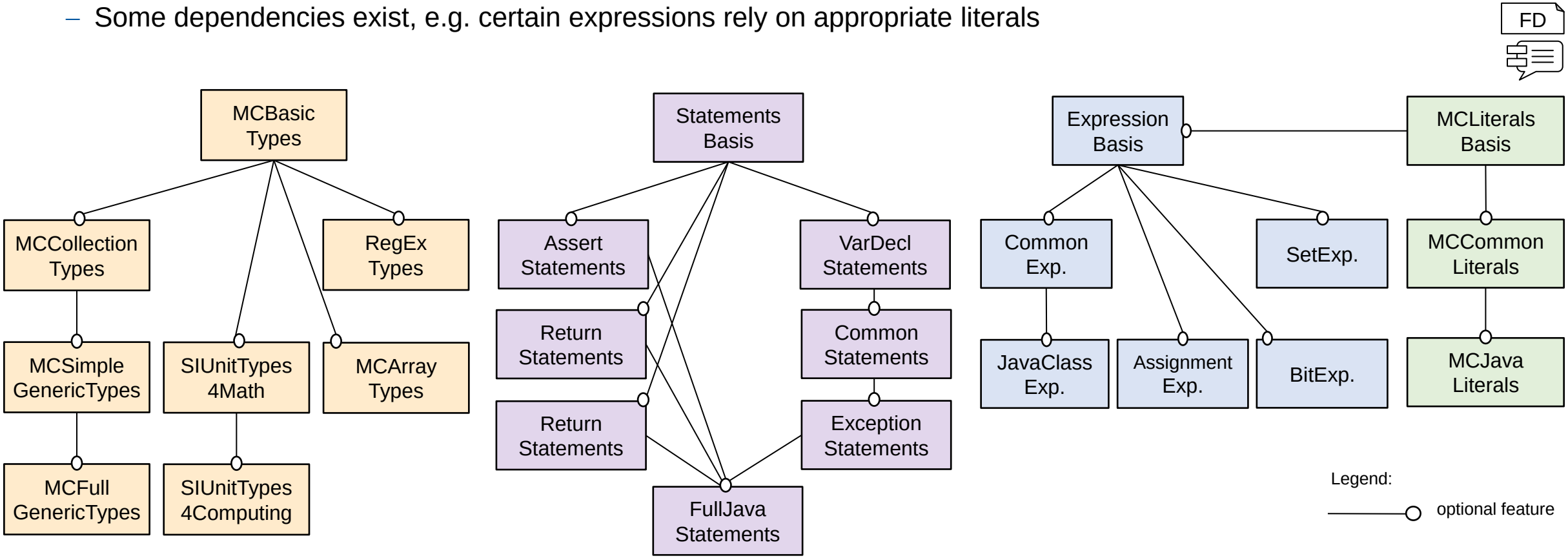
Use of Models for Coding and Testing

Language Workbench MontiCore and its Symbol Management Infrastructure



Reuse & Modularization of Languages – Modular Language Components

- Reusable language components that can be used as features
  - Some dependencies exist, e.g. certain expressions rely on appropriate literals

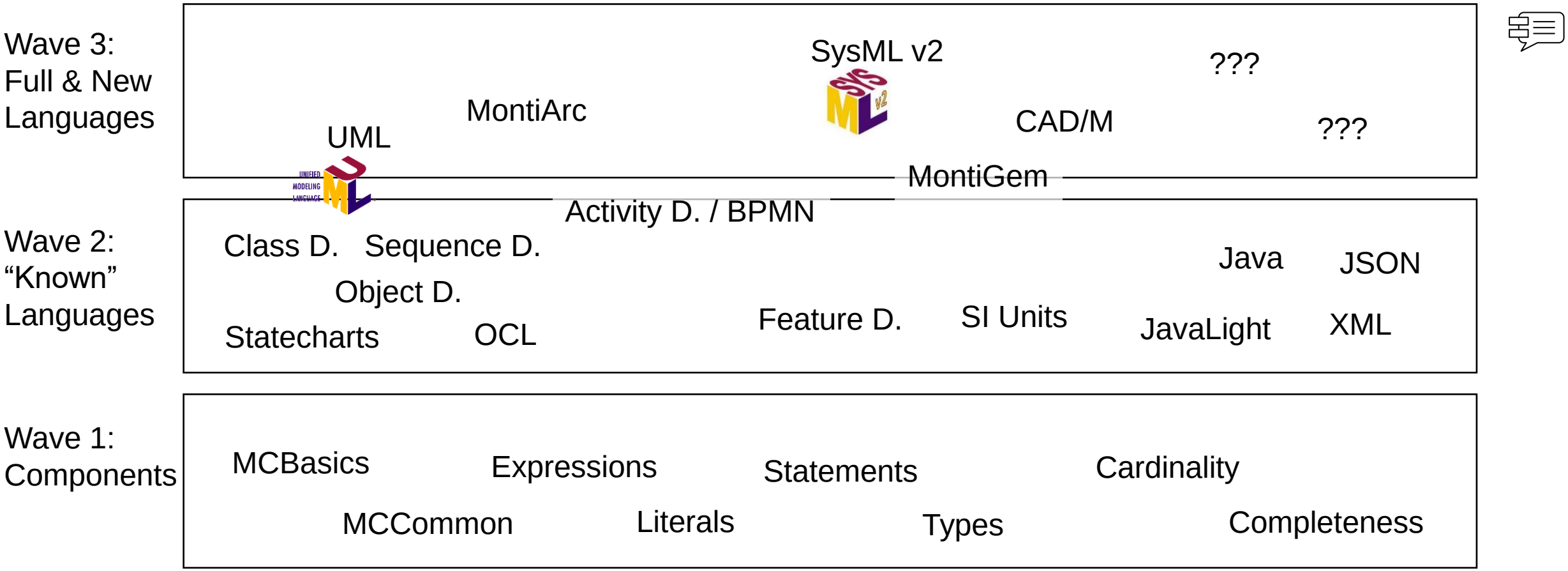


Grammars for these languages can be found at: https://monticore.github.io/monticore/monticore-grammar/src/main/grammars/de/monticore/Grammars/

[BEH+20] A. Butting, R. Eikermann, K. Hölldobler, N. Jansen, B. Rumpe, A. Wortmann: A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. JOT 19 (3), 2020.

MontiCore Language Zoo: Development in three Waves

- Languages are built in three phases



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

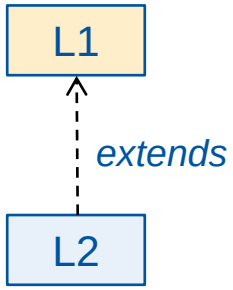
- Automaton language L1:

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

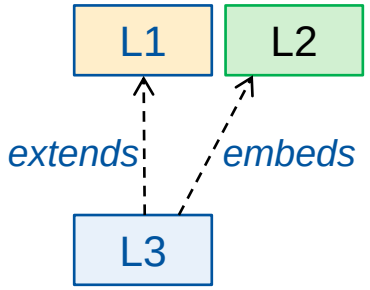
- Automaton language L1 is extended by actions in L2:
  - Actions are embedded at multiple places:

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.

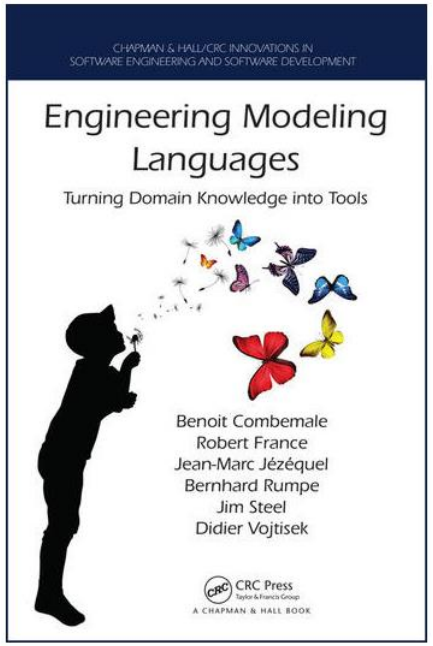
- Automaton language L1 and action language L2 are combined to a language embedding the actions into the automaton:

SC

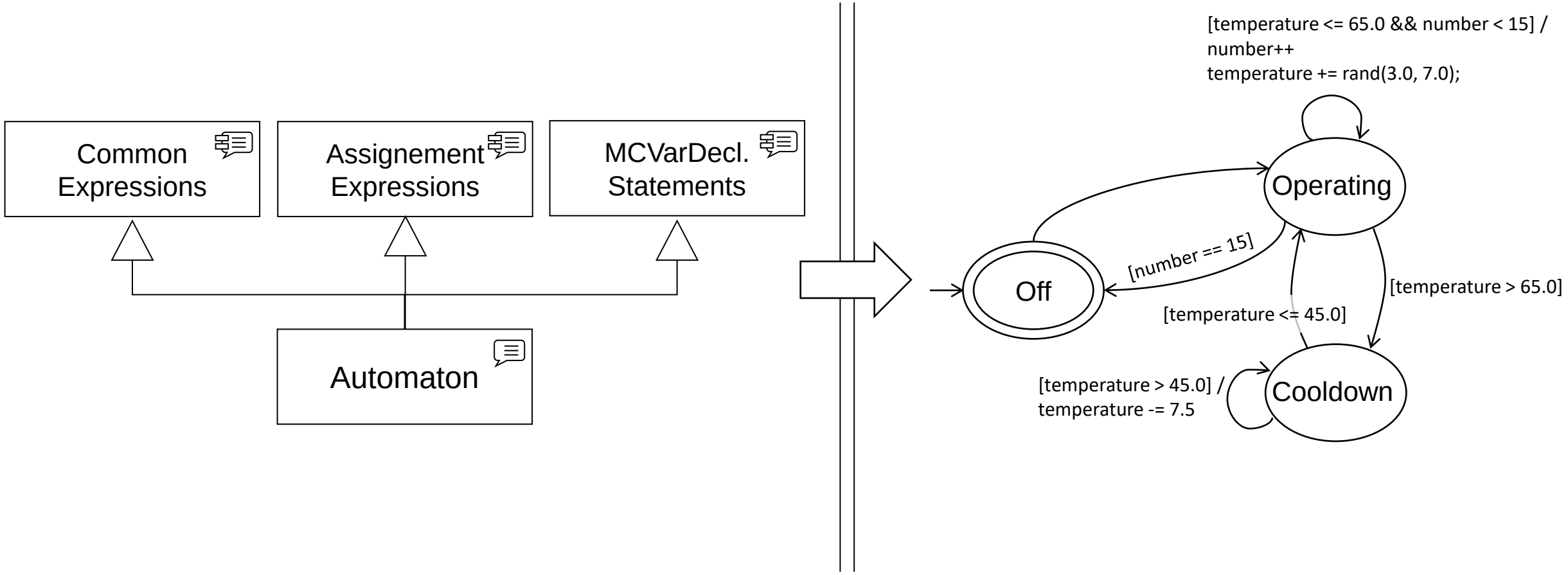
```
automaton PingPong {

    state Ping, Pong;

    Ping -> Pong / strokes++

    Pong -> Ping / strokes++
}
```

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

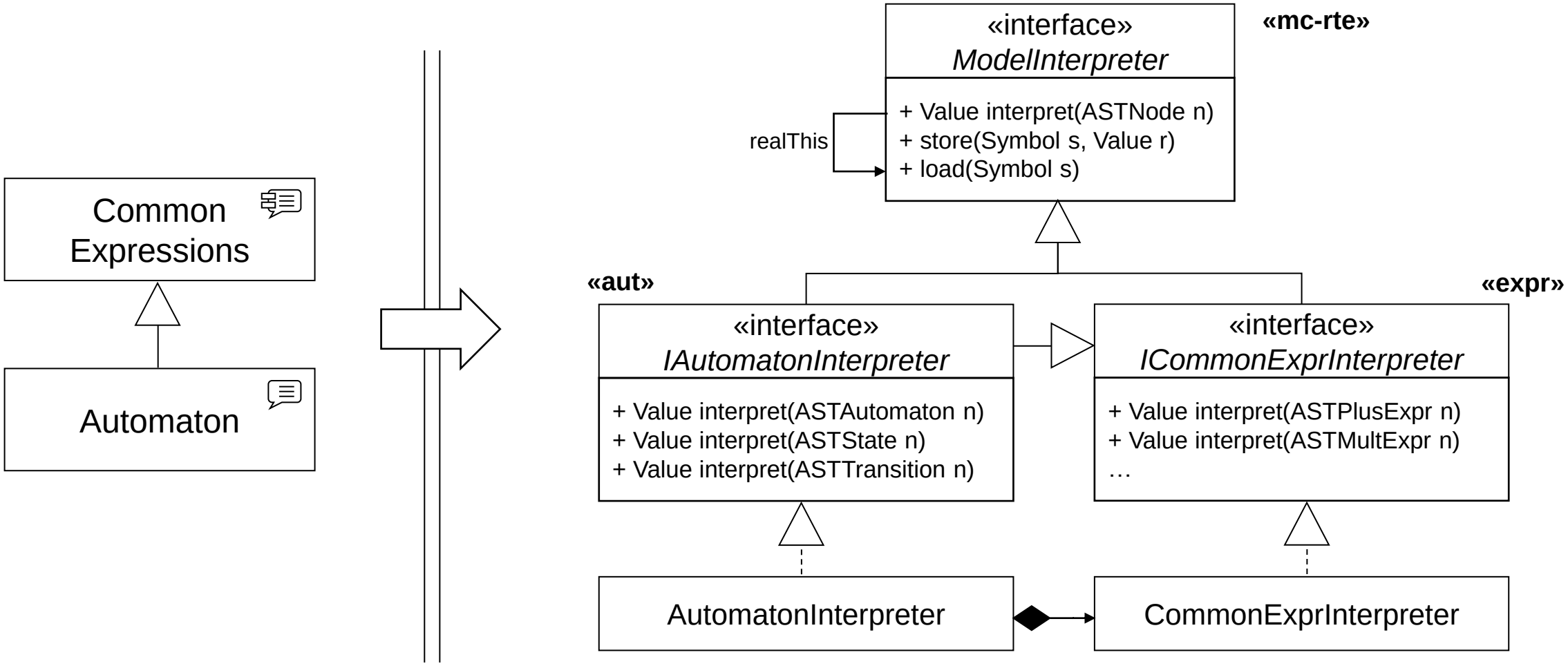
MontiCore – Interpreter Composition for Embedded Languages



Example


Common Expressions
Assignement Expressions
MCVarDecl. Statements
Automaton
[temperature <= 65.0 && number < 15] /
number++
temperature += rand(3.0, 7.0);
Operating
Off
[number == 15]
[temperature > 65.0]
[temperature <= 45.0]
[temperature > 45.0] /
temperature -= 7.5
Cooldown

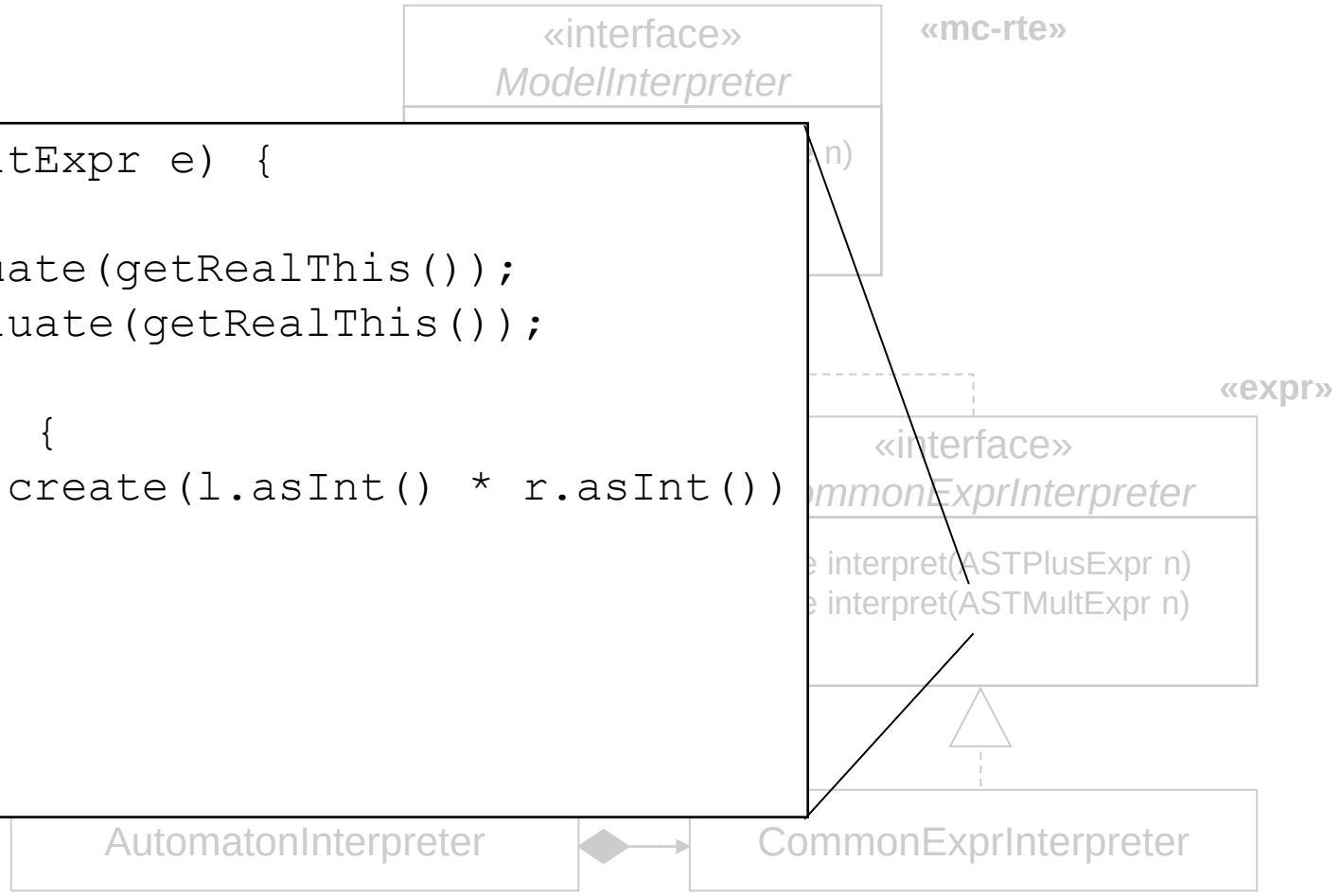
Compositional Model Interpreter in MontiCore



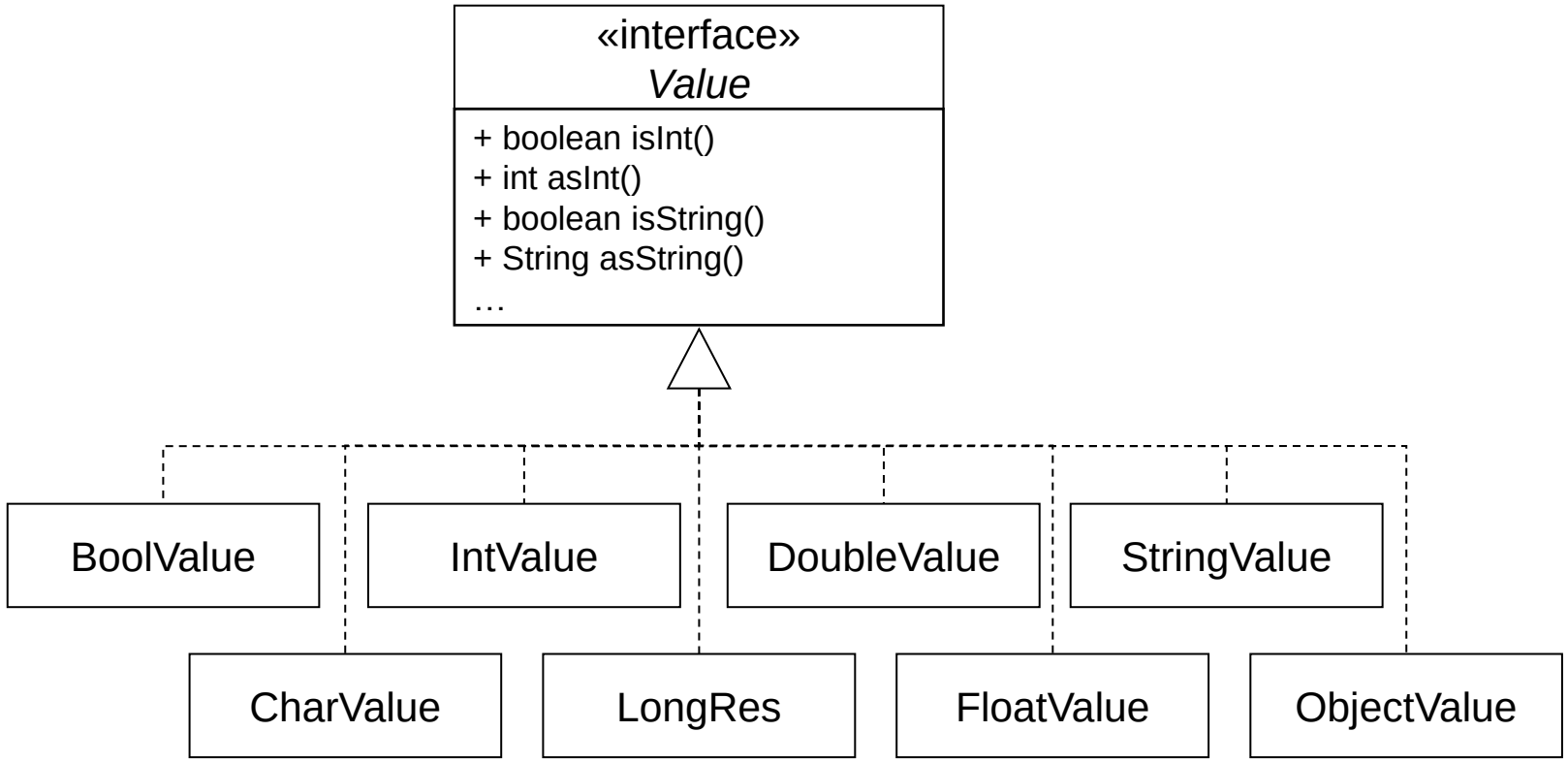
Compositional Model Interpreter in MontiCore

double dispatching step

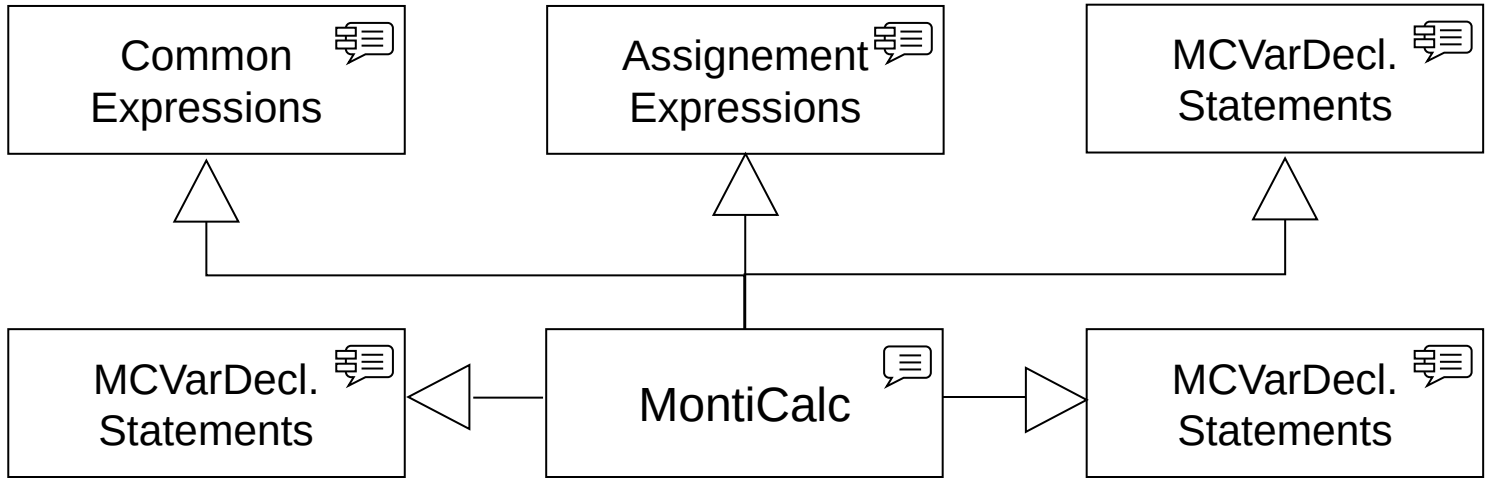
```
01 public Value interpret(ASTMultExpr e) {
02   // e.left * e.right
03   Value r = e.getLeft().evaluate(getRealThis());
04   Value l = e.getRight().evaluate(getRealThis());
05
06   if (r.isInt() && l.isInt()) {
07     Value res = ValueFactory.create(l.asInt() * r.asInt());
08     return res;
09   } else {
10     // ...
11   }
12 }
```



Value as Interpretation Result



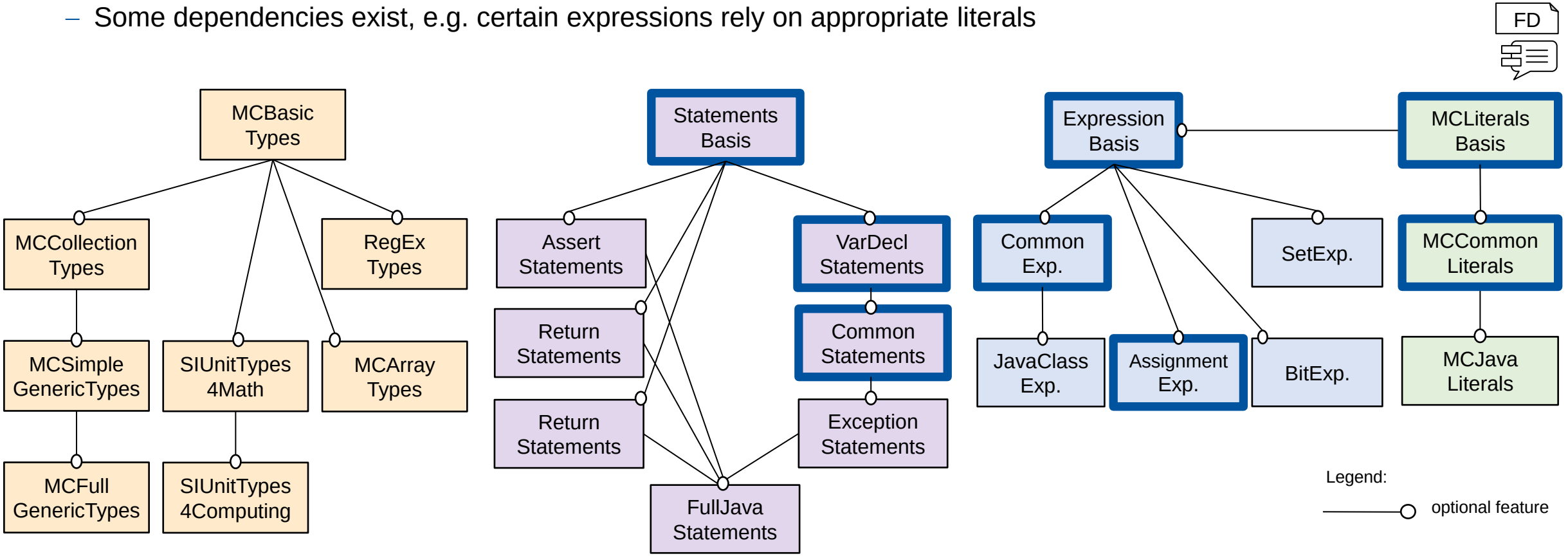
Demo – Building a Calculator




```
01 grammar MontiCalc extends BasicSymbols, MCCommonLiterals,
02                           CommonExpressions, AssignmentExpressions
03                           MCVarDeclarationStatements {
04
05   CalcCompilationUnit = (CalcCmd)*;
06
07   CalcCmd = LocalVariableDeclaration | Expression | MCStatement;
08
09   PrintCmd implements MCStatement = "print" Name@Variable;
10 }
```

Current Work: Interpreter Library for Modular Language Components

- Reusable language components that can be used as features
  - Some dependencies exist, e.g. certain expressions rely on appropriate literals



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

[BEH+20] A. Butting, R. Eikermann, K. Hölldobler, N. Jansen, B. Rumpe, A. Wortmann: A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. JOT 19 (3), 2020.

Thank you
for your attention