TO WHAT DEGREE IS A MODELICA COMPILER ACTUALLY A COMPILER?

Dirk Zimmer
Institute of Robotics and Mechatronics, DLR

Open Modelica Workshop 2026, Feb 2, Linköping, Sweden

DLR

Why this question?

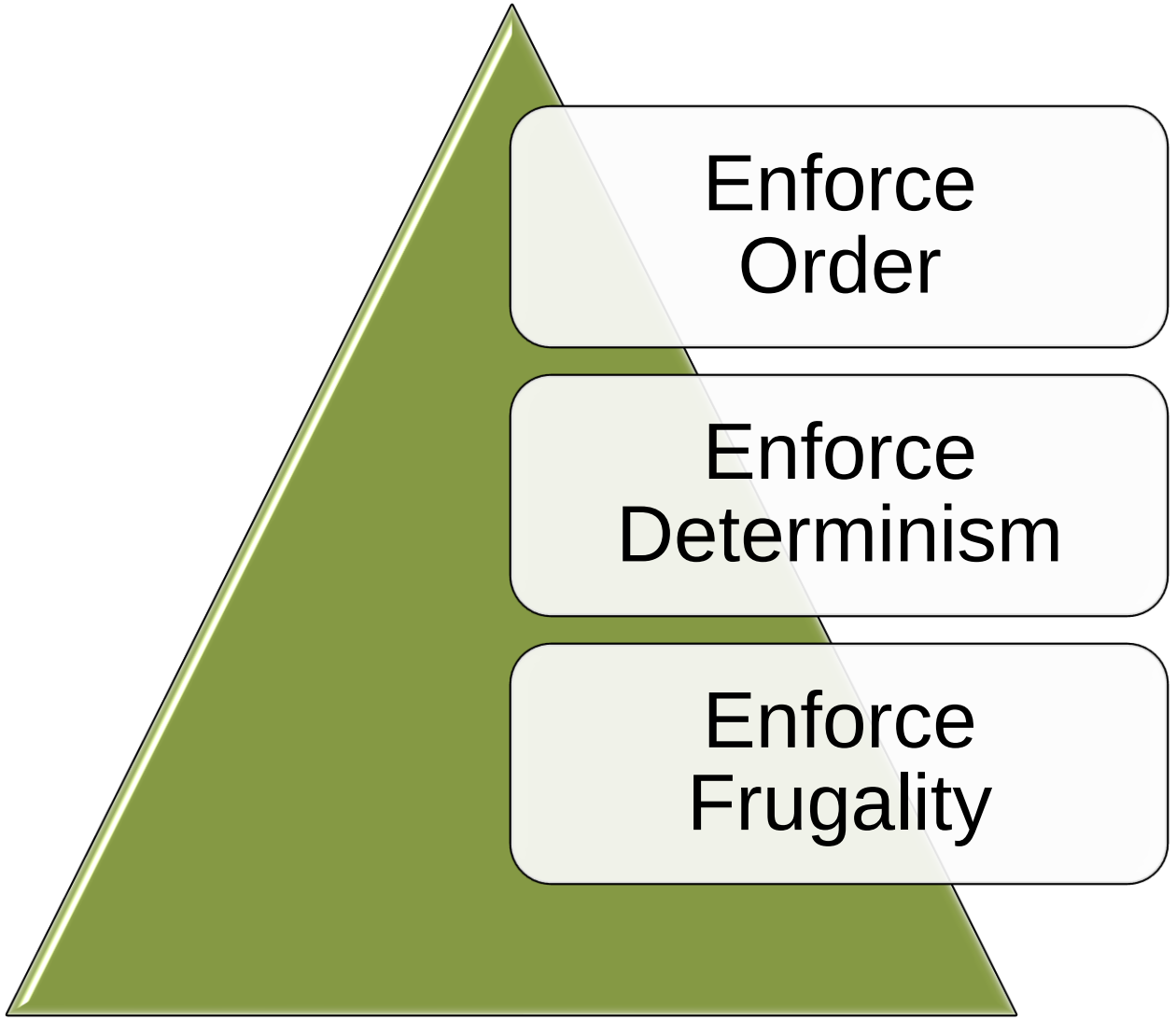
- Clarification of Computational Framework is needed to reduce complexity
  - Meaning of higher-level type system?
  - Way to formulate the semantic rules

- The Processing of the Language is also important on how to achieve Modularity.
  - At what level shall modularity be applied
  - What entities shall be mapped?

- Work conducted for the context of ModelicaLite

ModelicaLite: Easy to learn, Easy to process

>20x complexity reduction



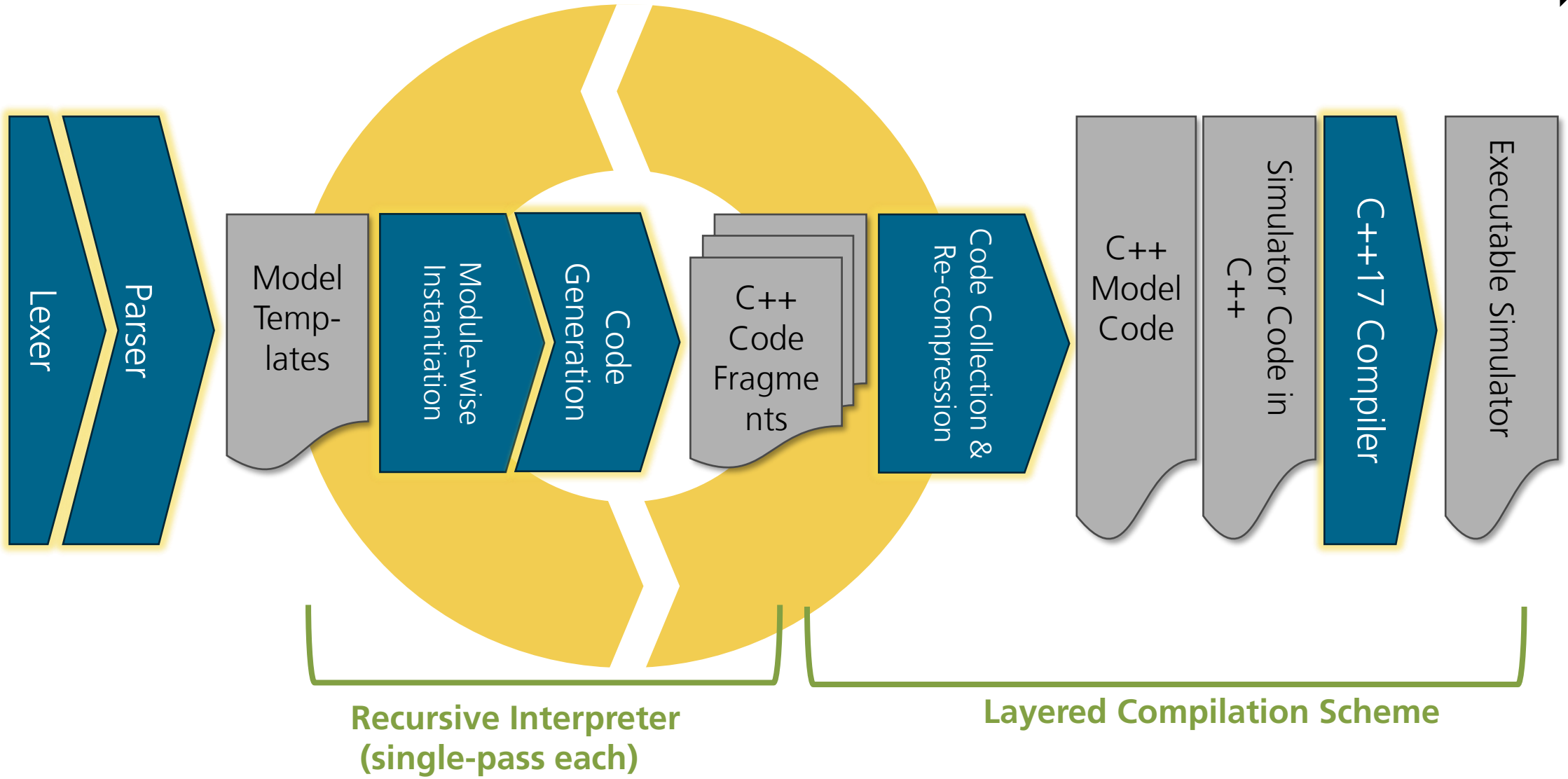
ModelicaLite: Easy to learn, Easy to process

Demo

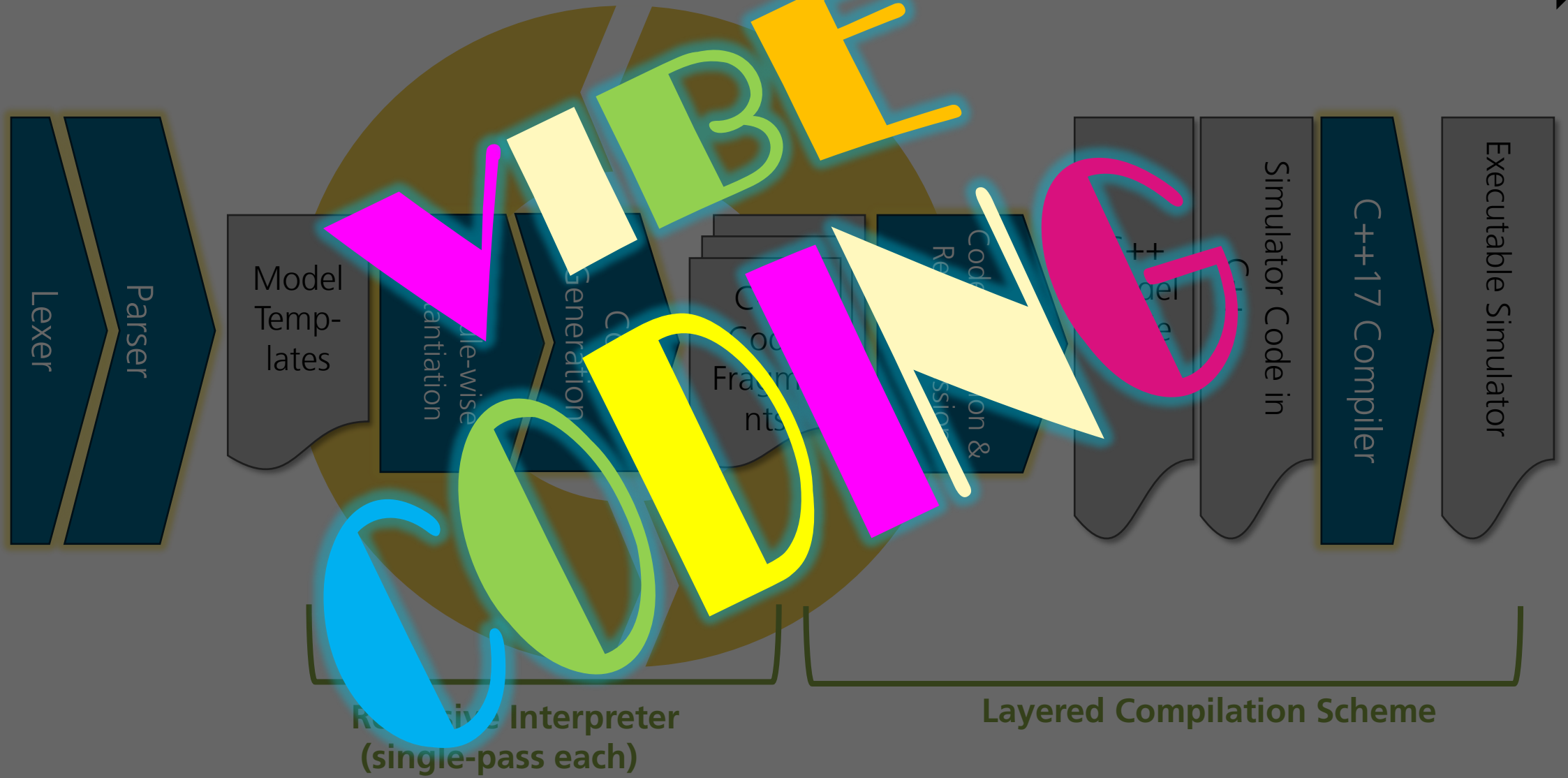
SOFTWARE ARCHITECTURE

Midjourney

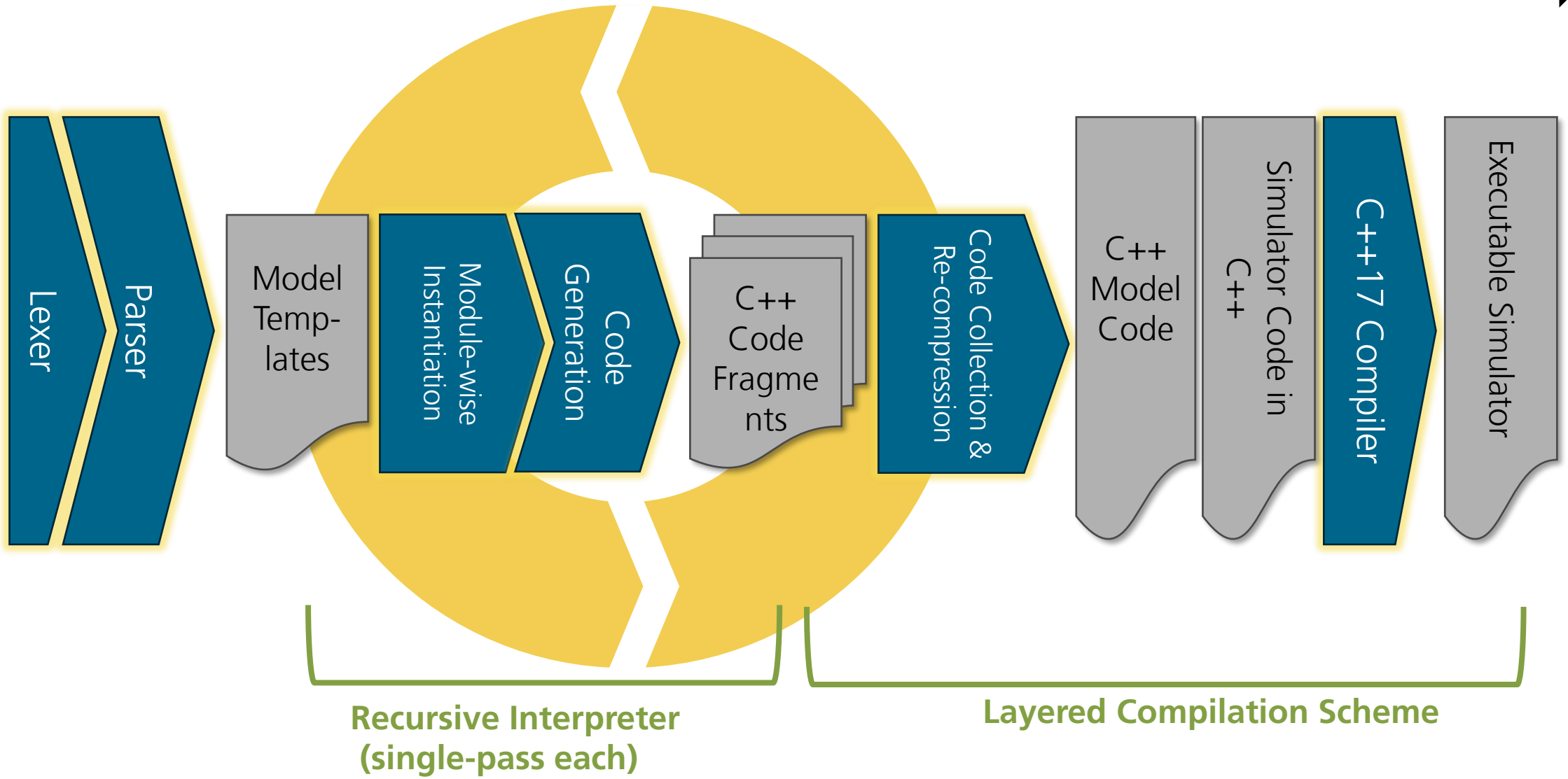
Modelica Lite Compiler Architecture (Current State)



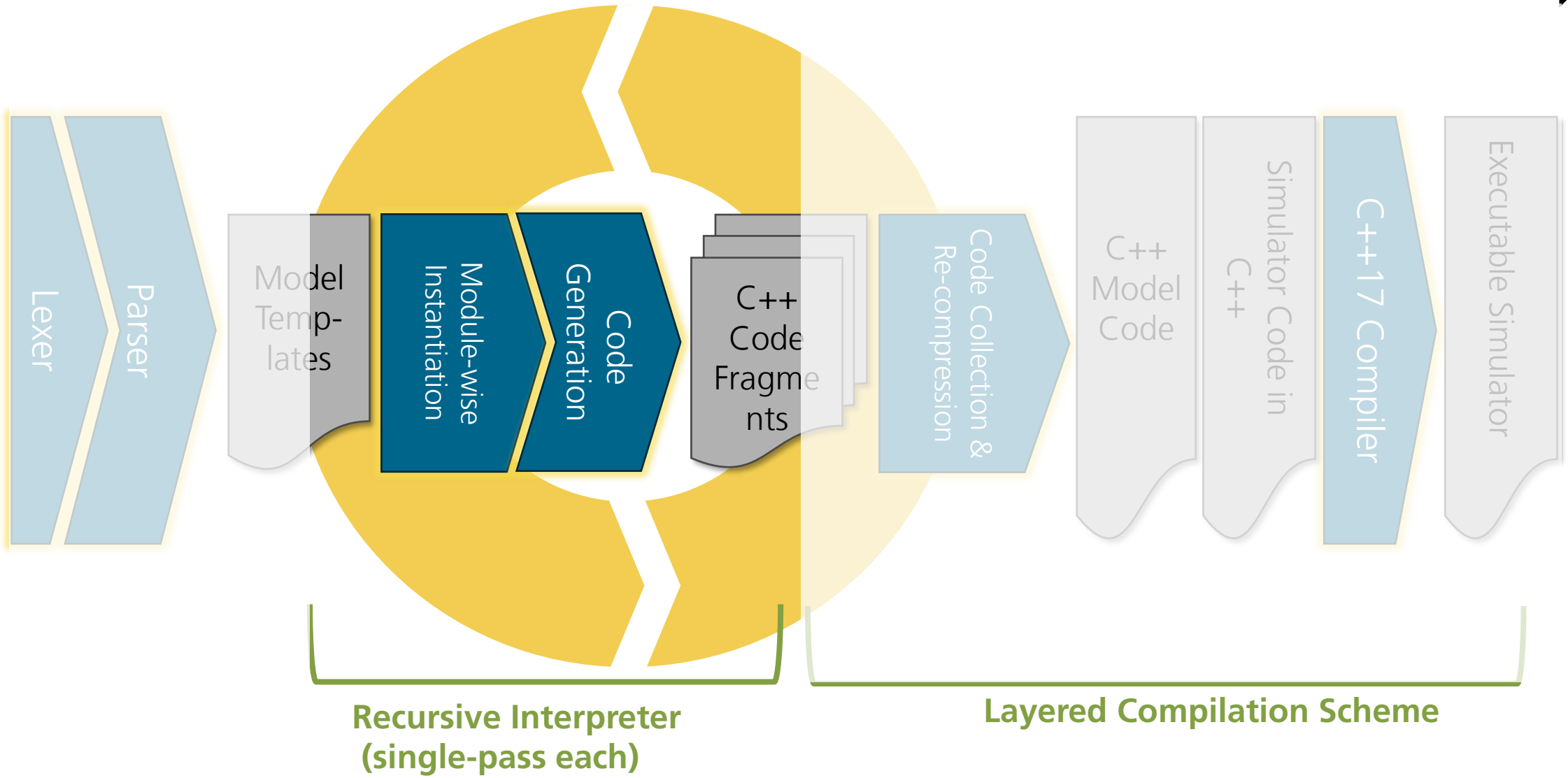
Modelica Lite Compiler Architecture



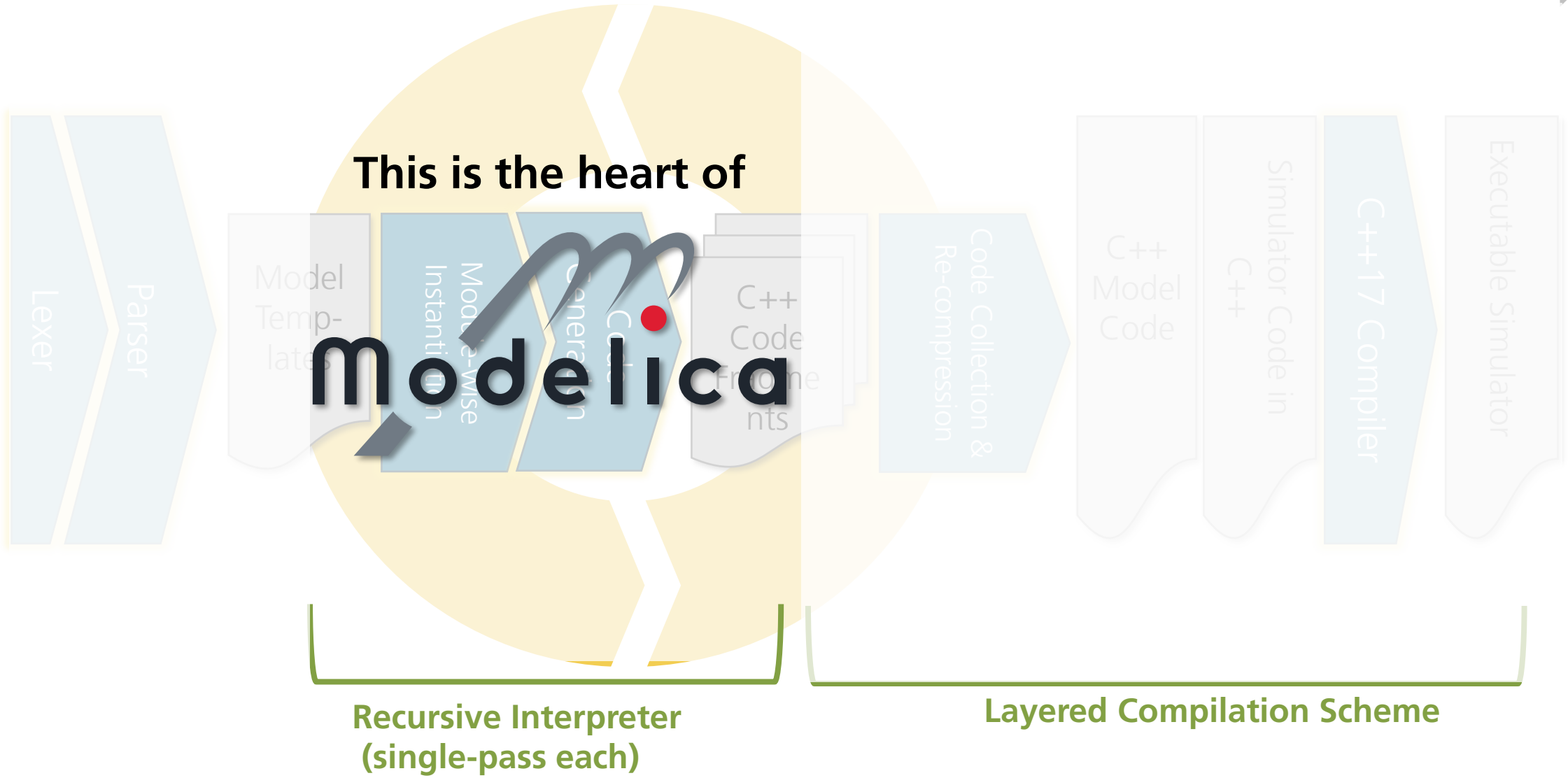
Modelica Lite Compiler Architecture (Current State)



Modelica Lite Compiler Architecture (Current State)

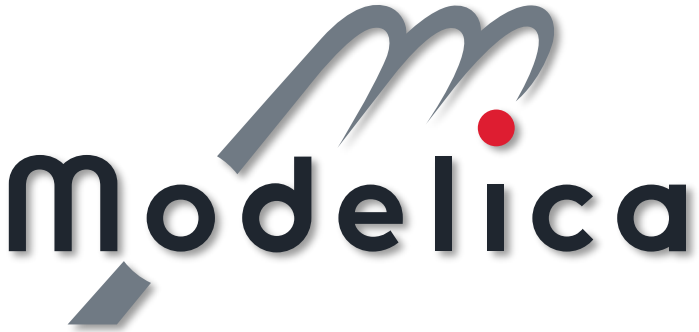


Predominantly Interpreter?



Predominantly Interpreter?

This is the heart of



- Extentions of classes
- Modification of classes
- Redefinition
- Redeclaration
- Conditional Components
- Connections
- Equation generation and collection…

… all become part of an interpreter

Recursive Interpreter (single-pass each)

Compiler vs. Interpreter

- Typical for an interpreter is a wholistic recursive processing.
  - Avoidance of intermediate form ➔ direct processing
  - Recursive scheme with intermediate results on Stack

- Typical for a compiler is a layered and piecewise processing.
  - Generation of intermediate forms
  - Piecewise/Modular generation of code

Recursive Interpreter (single-pass each)

Layered Compilation Scheme

Drivers of Complexity in Software

- The principal of minimal visible surface
  ➔Low ratio of code for interface/implementation
- Persistent data-structures on the heap often drive complexity
  - Need to be kept internally consistent
  - Are accessed by many parts of the program
  - Synchronization issues
- Temporary data-structures on the stack avoid complexity
  - Access limited to the top of the stack
  - No need to keep consistent since ultimately discarded.
- The stack is your friend (but you cannot only work with friends)

Single Recursive Interpreter

- We shall interprete ModelicaLite by a single-recursive interpreter.
  - All class definitions are just regarded as templates
  - No intermediate data structures that persist
  - Everything on the stack. (e.g. ad-hoc type-generation)
  - Once code for one module has been generated, stack is empty.
- Take advantage of being an interpreter
  - No type-system for high-level objects. No-one cares whether the error is prompted before or after instantiation
  - Only low-level type check before you go to compilation stage.
  - We could allow more flexibility than now.
- Side-result: Recursive rules for semantics

MODULAR CODE GENERATION

Midjourney

Modelling Formats/Methodologies

Quasi-ODE

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, t)$$

(resulting from Hamiltonian)

- Local analysis sufficient
- A priori statement on solvability
- Limited means to suppress high frequencies

idealization

LIED

$$\begin{bmatrix} \mathbf{w}_E \\ \dot{\boldsymbol{x}}_E \end{bmatrix} = \mathrm{g}(\boldsymbol{x}_I, \boldsymbol{x}_E, t)$$

$$\mathbf{A}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E) \begin{bmatrix} \mathbf{w}_I \\ \dot{\boldsymbol{x}}_I \end{bmatrix} = \mathrm{f}(\boldsymbol{x}_I, \boldsymbol{x}_E, \mathbf{w}_E, t)$$

- Local analysis sufficient
- A priori statement on solvability
- Adequate means to suppress high frequencies

over-idealization

DAE

$$\mathbf{0} = \boldsymbol{F}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \mathbf{w}, t)$$

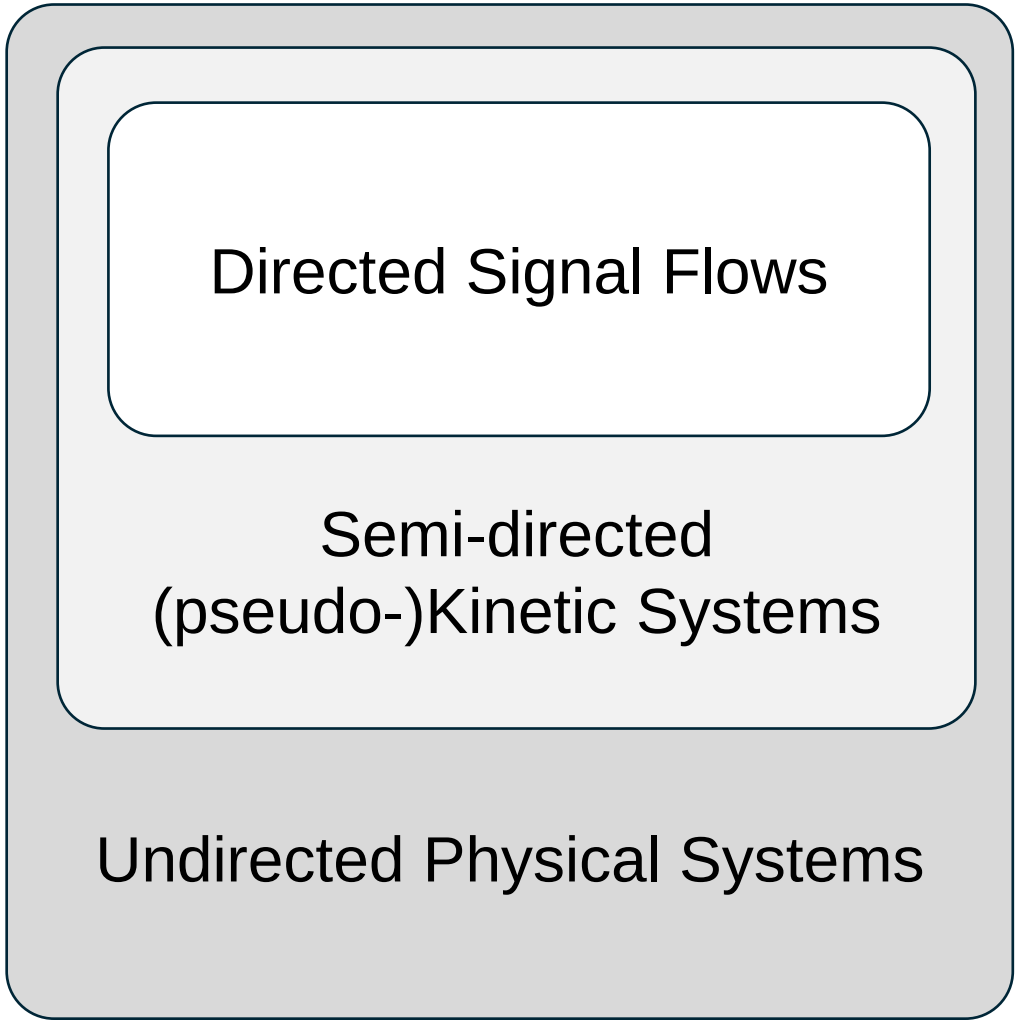
- Global analysis needed
- No a priori statement on solvability possible
- Excellent means to suppress high frequencies

An integral Part of Modeling Techniques

- To model this systems, we will learn 3 methodologies that build on top of each other:

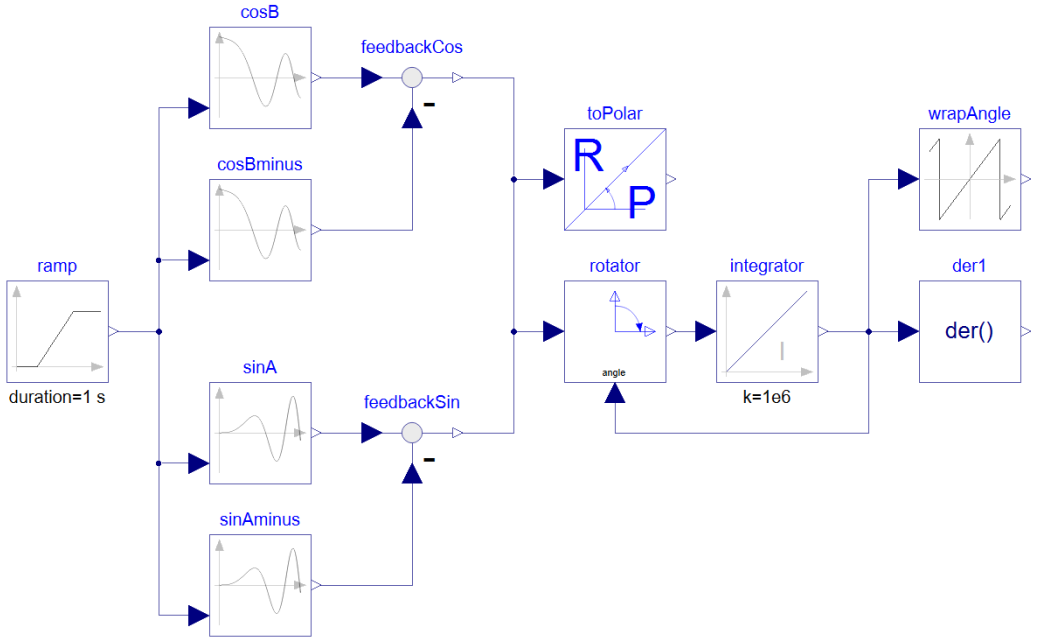
1. Directed Signal Flows
2. Semi-directed (pseudo-) Kinetic Systems
3. Undirected Physical Systems

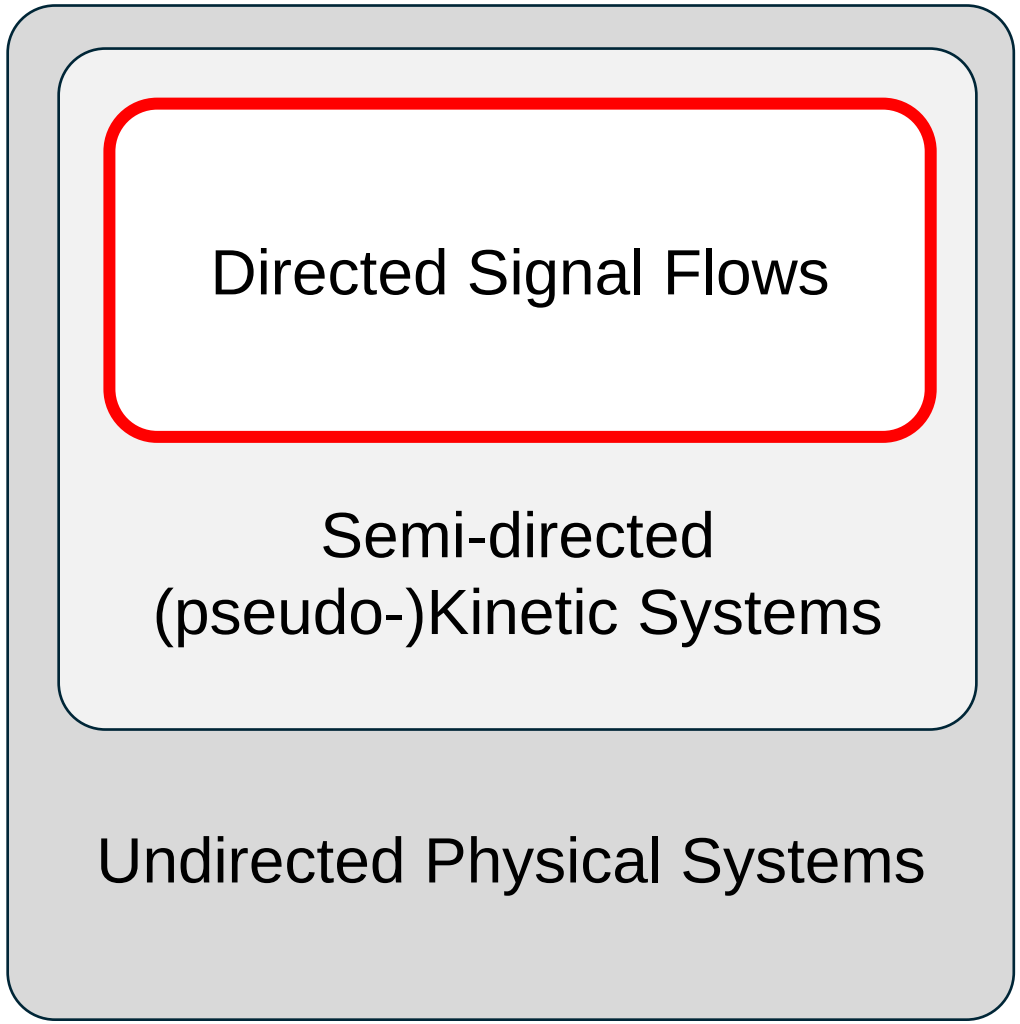
- Each class contains the prior.
- Each class demands more expertise and applies to more systems



Teaching an integral Part of Modeling Techniques

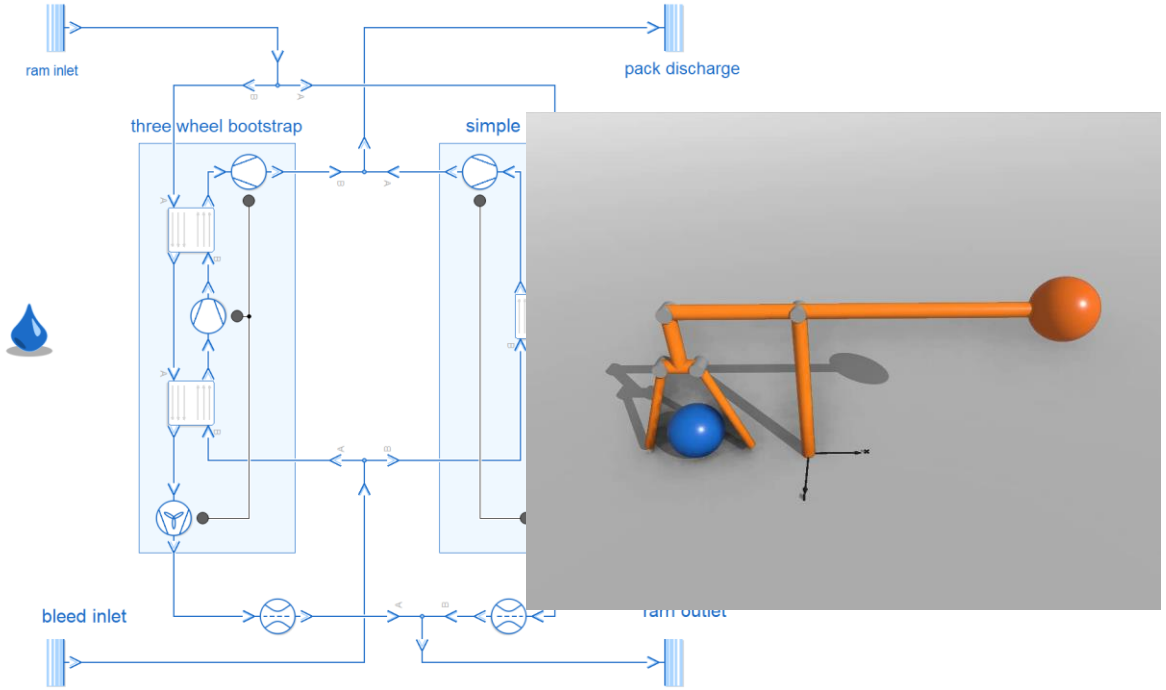
Signal Flow Systems are useful for control but can also represent systems that are modeled by “cause-and-effect”. A very useful class.





Teaching an integral Part of Modeling Techniques

Semi-directed Kinetic Systems model the dynamics of complex non-linear configuration that is described from a root point.



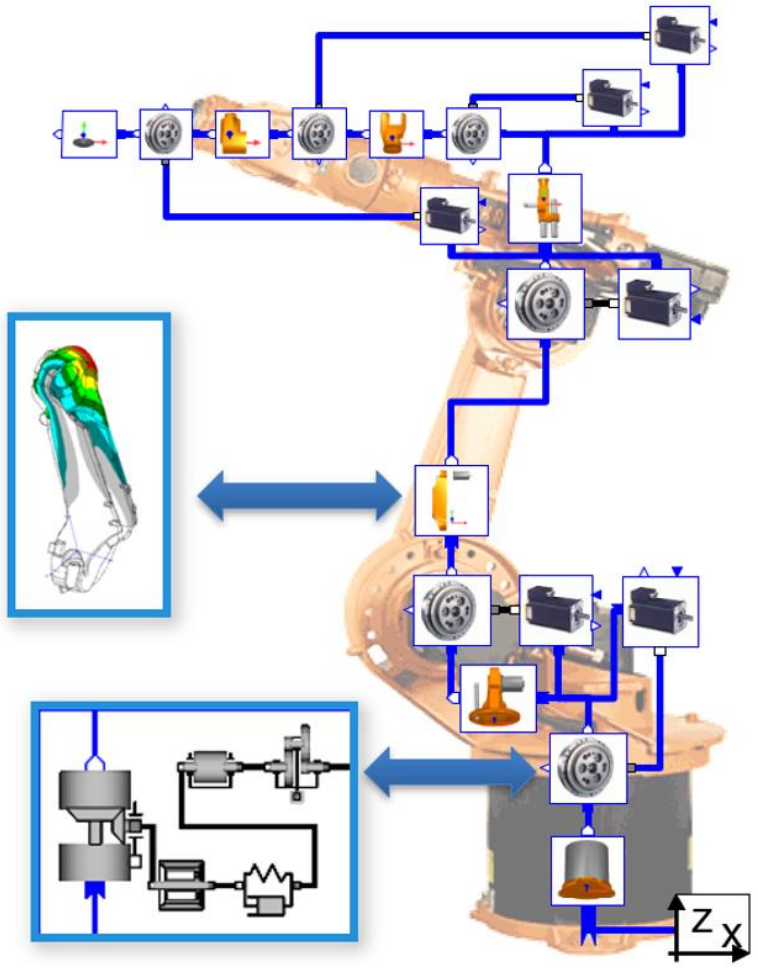
Directed Signal Flows

Semi-directed
(pseudo-)Kinetic Systems

Undirected Physical Systems

Teaching an integral Part of Modeling Techniques

Undirected physical systems work on arbitrar energy flows anc can enable the highest degree o idealization but also need expertise



Directed Signal Flows

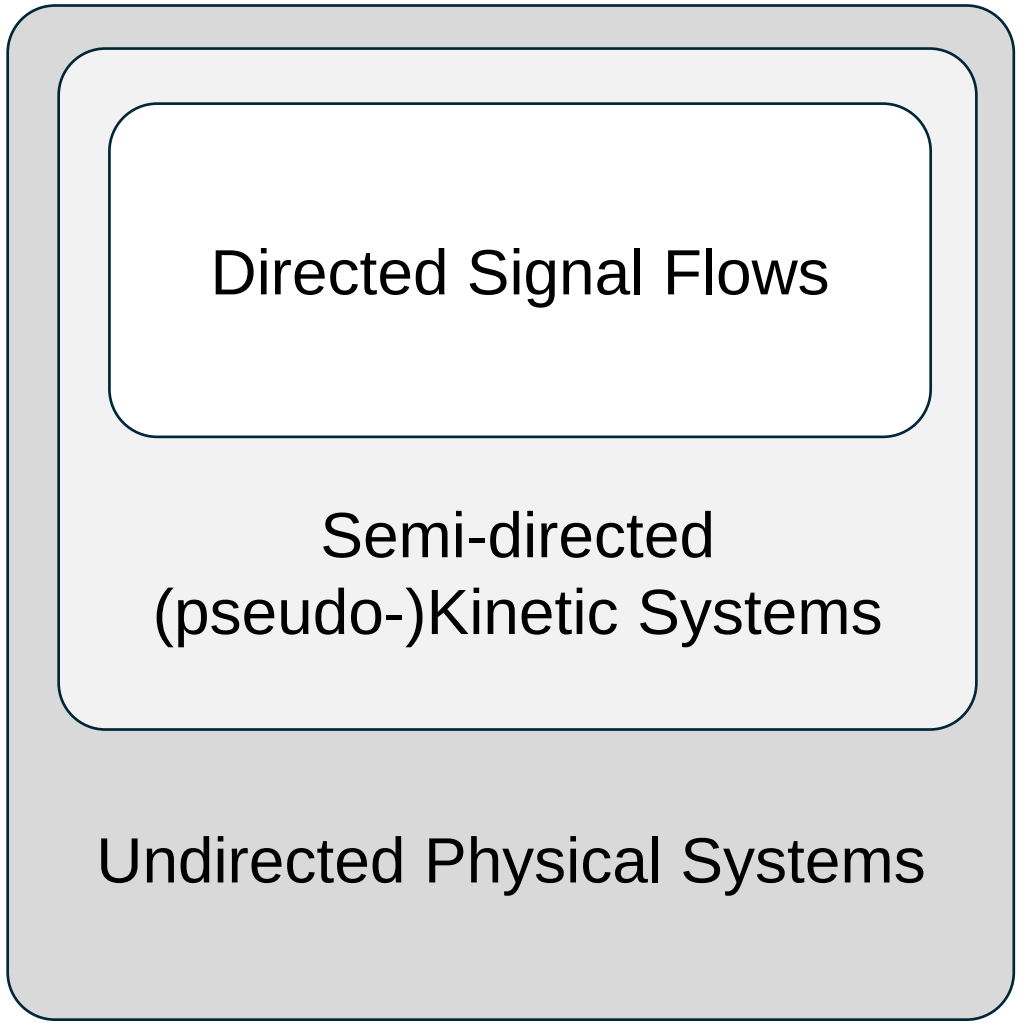
Semi-directed (pseudo-)Kinetic Systems

Undirected Physical Systems

Teaching an integral Part of Modeling Techniques

- We will use one language to learn all these 3 methodologies:

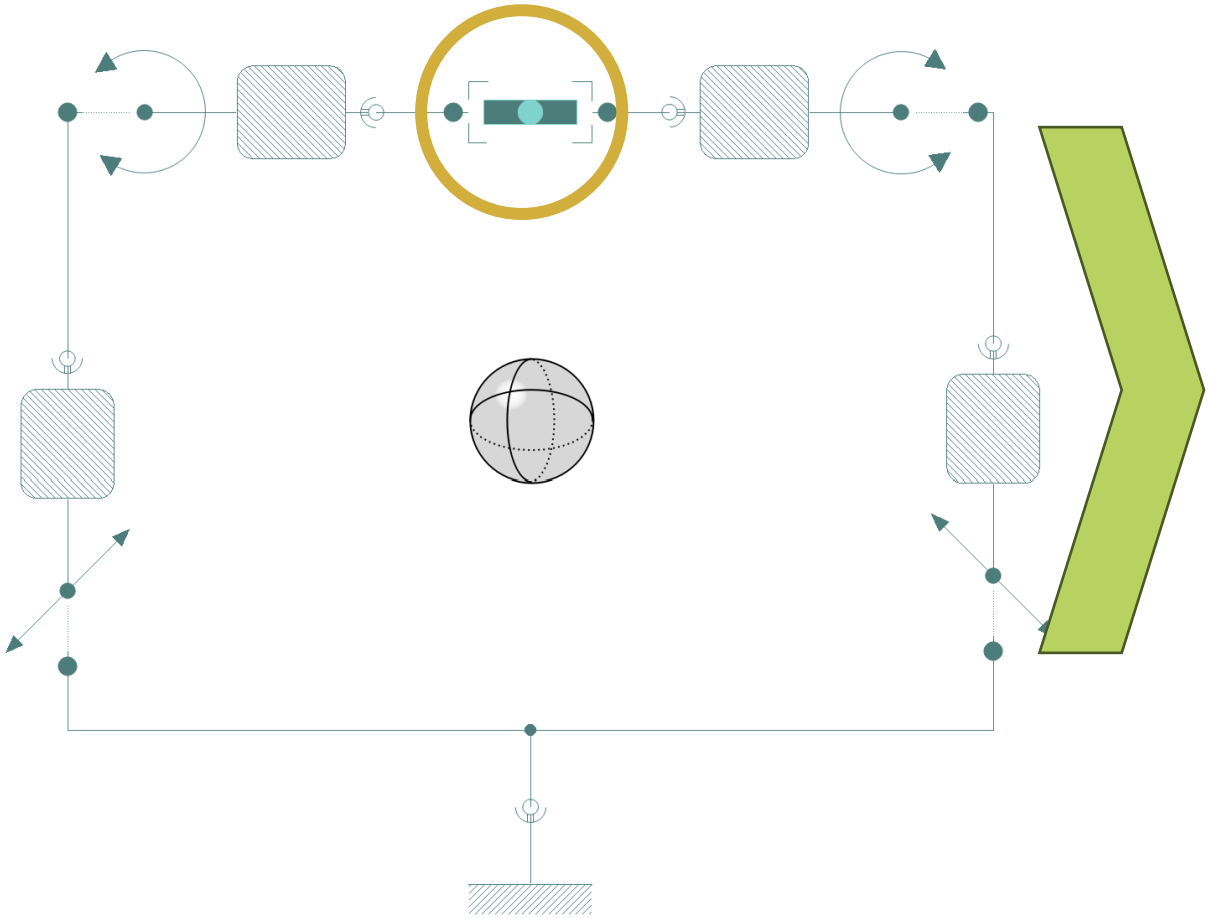
Modelica Language



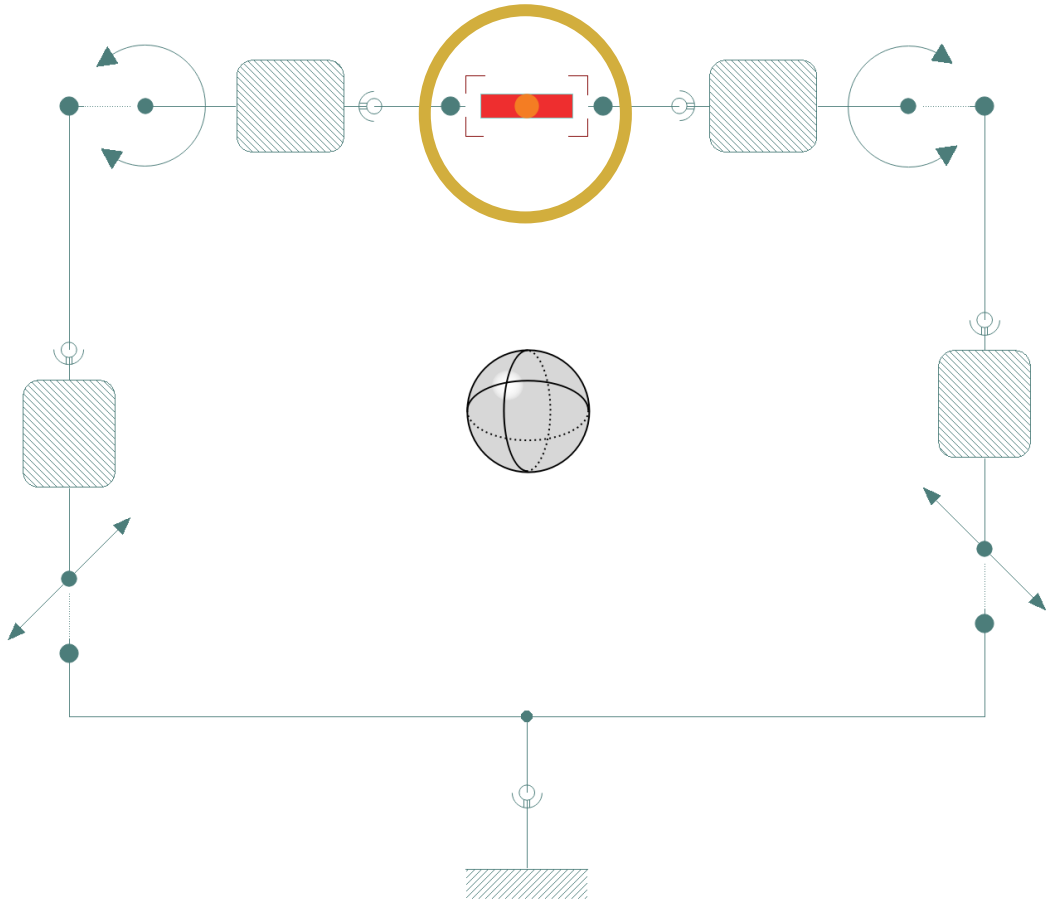
- There are many other modeling languages but Modelica is suited for all 3.
- Hence we can use one language and focus on the modeling.

Example of Combined Integrated Use

Elastic Constraint:
Compatible to ModelicaLite



Ideal Constraint:
Requires Full Modelica Support



Technical Prerequisites of Modularity

We need to know

- Which pairs of effort in flow correspond to which carrier signal
- What connector variables can be differentiated
- What are states
- What are pairs of tearing and residuals for the linear equation system

```
class BodyComp : public Component {
public:
    const Real J;
    FlangeToRot flangeToR;
    Real z;
    BodyComp(Real J) : J(J){ };
    void eval1();
    void metainfo(Meta& meta) {
      meta.regComp(&flangeToR, " " );
      meta.regVariable(&z, " " );
      meta.addBlock(this,
      [](Component* c) { return ((BodyComp*)c)->eval1(); },
      Signals{&flangeToR.signal2},Signals{&flangeToR.signal3}
      );  };
};

void BodyComp::eval1() {
    z=flangeToR.w__der;
    flangeToR.t=((1*J)*z);
};
```

Technical Prerequisites of Modularity

We need to know

- **Which pairs of effort in flow correspond to which carrier signal**
- What connector variables can be differentiated
- What are states
- What are pairs of tearing and residuals for the linear equation system

```
type AngularVelocity2 =
B.RealWithDerivative(unit="rad/s");


connector FlangeOnRot
  output SI.Angle phi;
  AngularVelocity2 w;
  flow SI.Torque t;
end FlangeOnRot;


connector FlangeToRot
  input SI.Angle phi;
  AngularVelocity2 w;
  flow SI.Torque t;
end FlangeToRot;
```

Fixed Order:
1. Signal
2. Effort
3. Flow

Technical Prerequisites of Modularity

We need to know

- Which pairs of effort in flow correspond to which carrier signal
- **What connector variables can be differentiated**
- **What are states**
- **What are pairs of tearing and residuals for the linear equation system**
- **These definitions are compatible and orthogonal to existing Modelica**

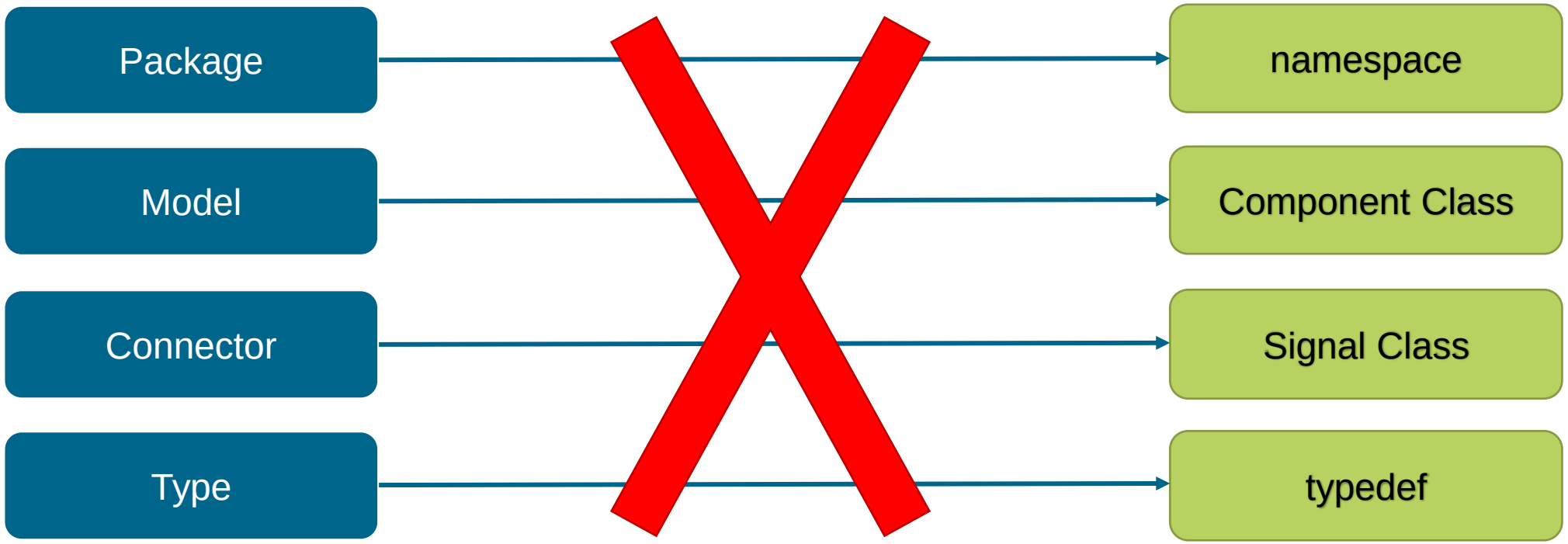
```
encapsulated package Base

  model LinearRespons
    input Real residual;
    output Real balance;
  equation
    residual = 0;
    annotation(
    defaultConnectionStructurallyInconsistent=true,
    …);
  end LinearResponse;

  type RealWithDerivative = Real;
  type ContinuousState = Real;
```

Implementation of Modularity

Modelica is not designed for component-wise compilation.

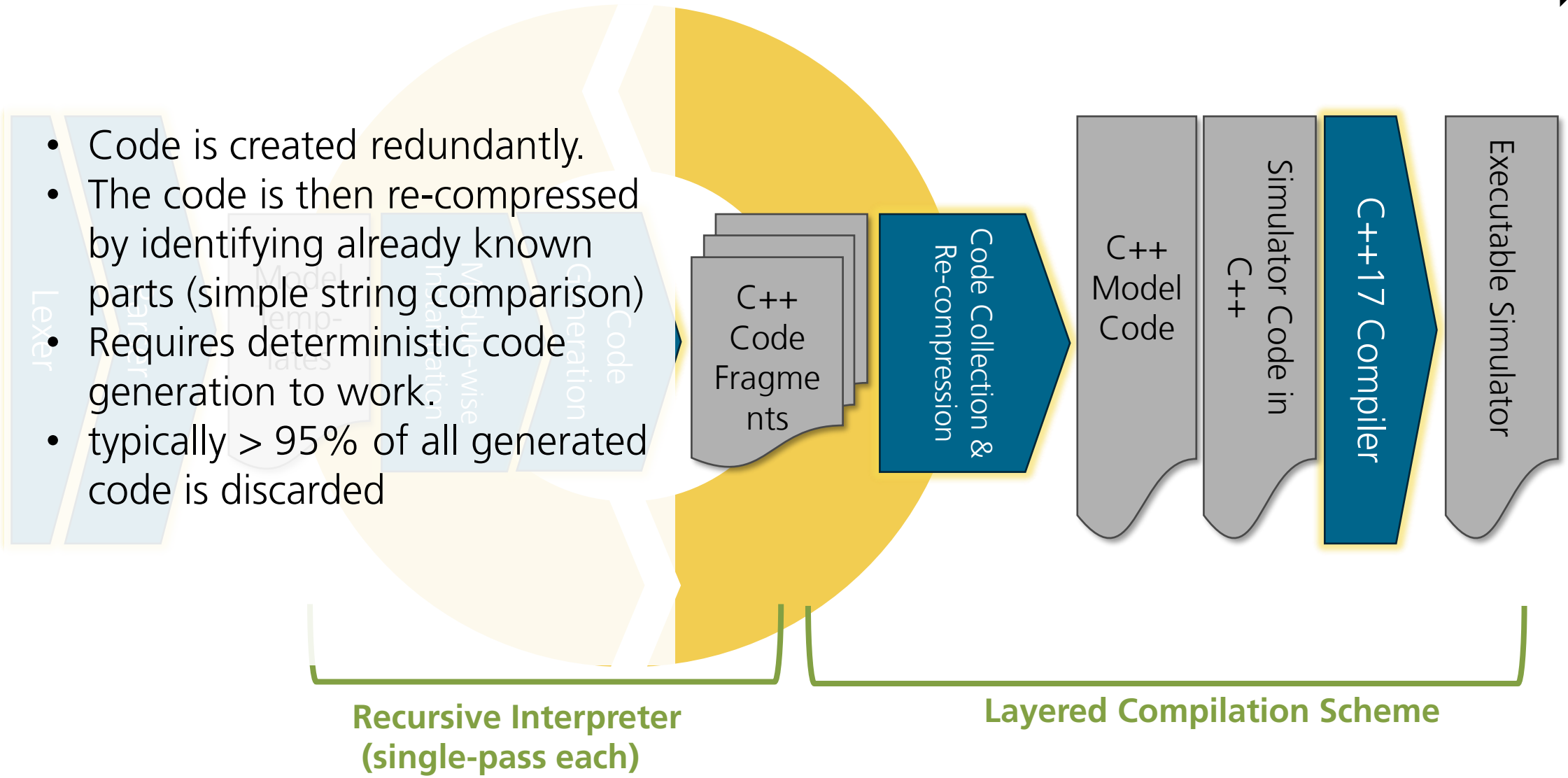
Classes are just templates. Only Instances can be treated as modules

Hence no attempt is made to relate Modelica classes to C++ classes directly.

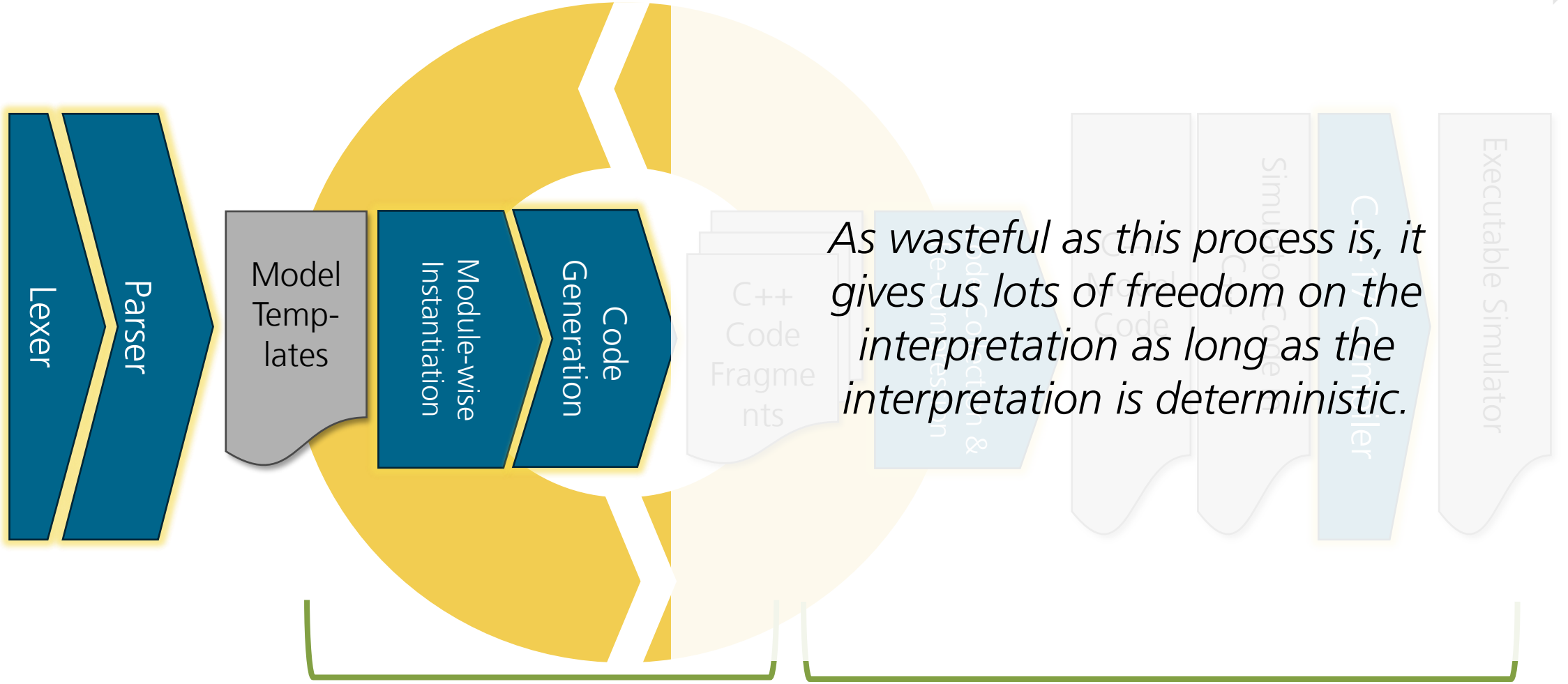


Redundant Code Generation

- Code is created redundantly.
- The code is then re-compressed by identifying already known parts (simple string comparison)
- Requires deterministic code generation to work.
- typically > 95% of all generated code is discarded



Redundant Code Generation



As wasteful as this process is, it gives us lots of freedom on the interpretation as long as the interpretation is deterministic.

```
89      return false;
90  }
91
92
93  bool Parser::expectModel(shared_ptr<HierachicalElem> Parent) {
94      bool partial = expect(Tokens::_partial, false);
95      if (expect(Tokens::_model, partial)) {
96          if (expect(Tokens::_name, true)) {
97              auto curDef = make_shared_tk<Model>(lastToken, lastToken.text, Parent, partial);
98              bool valid = true;
99
100             while (expectExtendsStmt(curDef));
101
102             ScopeSpecifier scope = ScopeSpecifier::publicScope;
103             while (expectScopeSpecifier(scope) || expectDeclaration(curDef, false, scope));
104
105             if (expect(Tokens::_equ_section, false)) {
106                 while (expectEquation(curDef) || expectConnection(curDef)) {
107                     expect(Tokens::_scolon, true);
108                 }
109             };
110
111             if (expect(Tokens::_annotation, false)) {
112                 valid &= expect(Tokens::_scolon, true);
113             }
114
115             valid &= expect(Tokens::_end, true);
116             if (expect(Tokens::_name, true)) {
117                 if (curDef->getName() != lastToken.text) {
118                     errorstream << ErrorMsgToken(curToken) << "Identifiers do not match. Found : " << lastToken.text
```

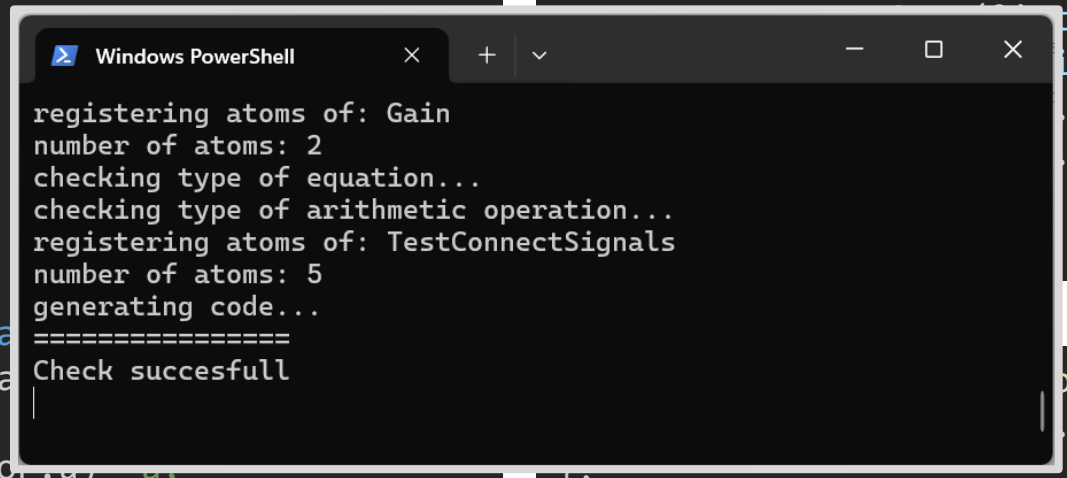
CURRENT STATE OF IMPLEMENTATION

Demonstration

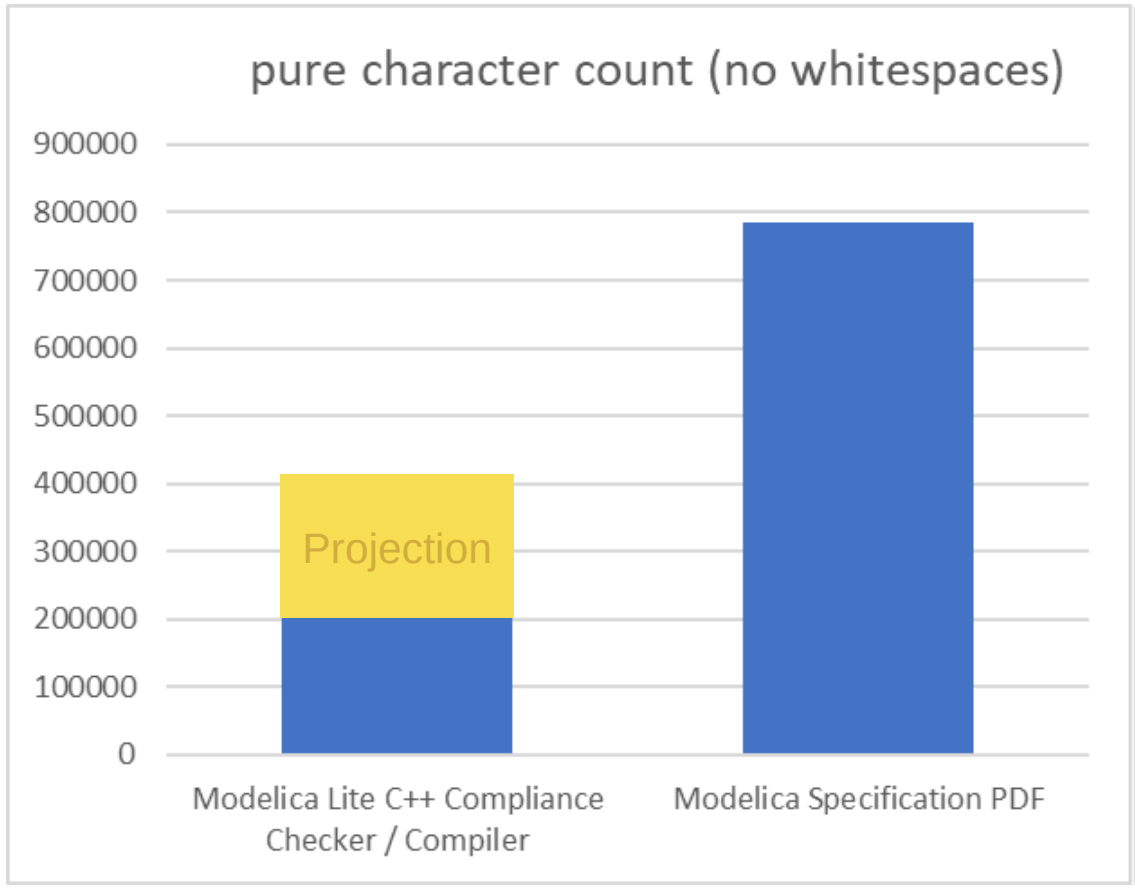
```
within ;
encapsulated package Base

  model ContinuousState "continuous state"
    input Real derivative;
    output Real state;
  equation
      der(state) = derivative;
  end ContinuousState;

[…]


package Test

  model TestConnectSignals
  protected
    Signals.Blocks.Constant
      Signals.Blocks.Integra
      Signals.Blocks.Gain ga
    equation
      connect(c.y, integrator.u)  a;
      connect(integrator.y, gain.u)  a;
      a;
    end TestConnectSignals;
  end Test;
```

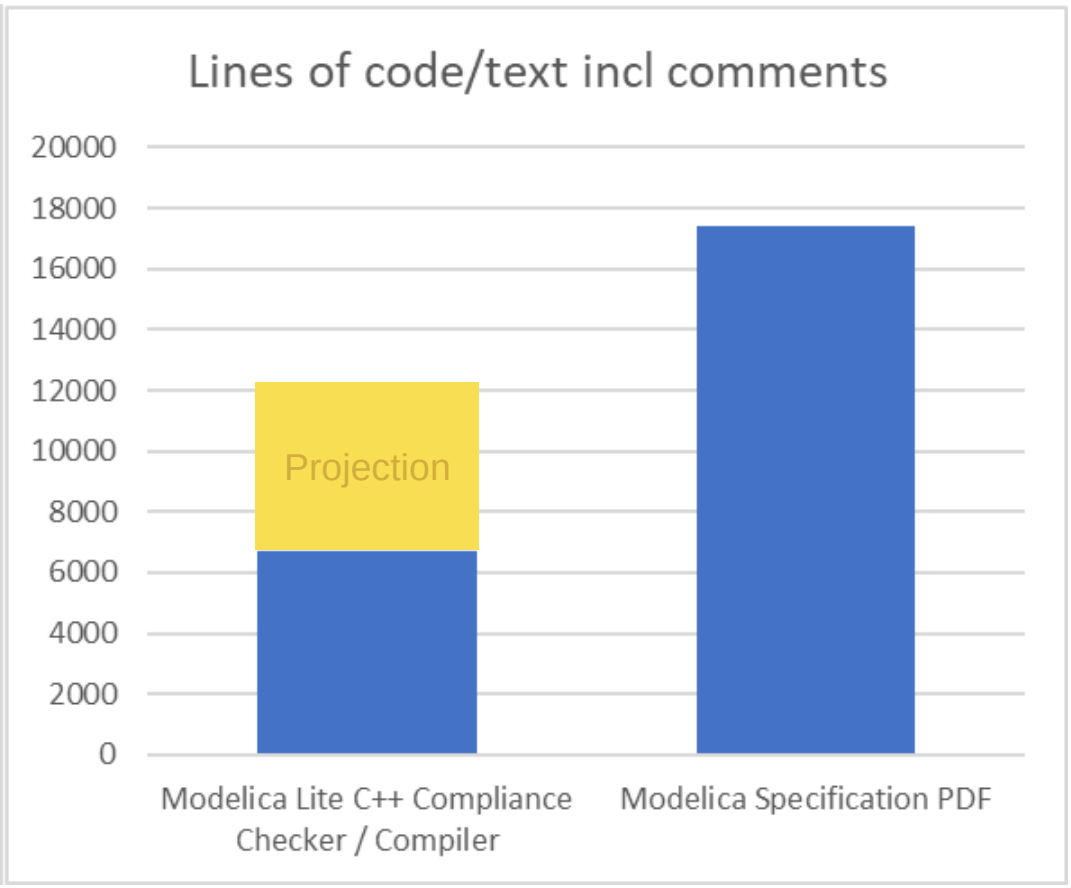
```
class TestConnectSignalsComp : public Component {
public:
    ConstantComp c{5};
    IntegratorComp integrator{1};
    GainComp gain{1.2};
    TestConnectSignalsComp(){
    };
    void eval1();
    void eval2();
    void metainfo(Meta& meta) {
        meta.regComp(&c, " " );
                                  tegrator, " " );
                                  in, " " );
                                  );
                                  );
```

```
                                  omp::eval1() {
                                  .val;
};
void TestConnectSignalsComp::eval2() {
    gain.u.val=integrator.y.val;
};
```



Compliance Checker / Compiler: Complexity Reduction





- Goal is to avoid the writing of an explicit specification but make the code so good and transparent that it serves as specification.
- Reusing existing IDEs and tools is also of big help

Conclusions

- ModelicaLite shall be designed such that it can be interpreted by a single-recursive interpreter, avoiding intermediate data-structures
- Modelica shall be a language that supports different modeling styles. We are pragmatic not dogmatic.
- ModelicaLite shall be an integral to Modelica not an extra path. Hence the modular prerequisites (Tearing, Connectors, StateSelection) are defined in a non-conflicting manner.
- Future tasks will be to deal with the Code Modules for CPUs and GPUs
- We have also not yet talked about discrete-event systems.