The Open-Source Modelica Compiler

Martin Sjölund

Department of Computer and Information Science
Linköping University

2021-02-12

LINKÖPING
UNIVERSITY

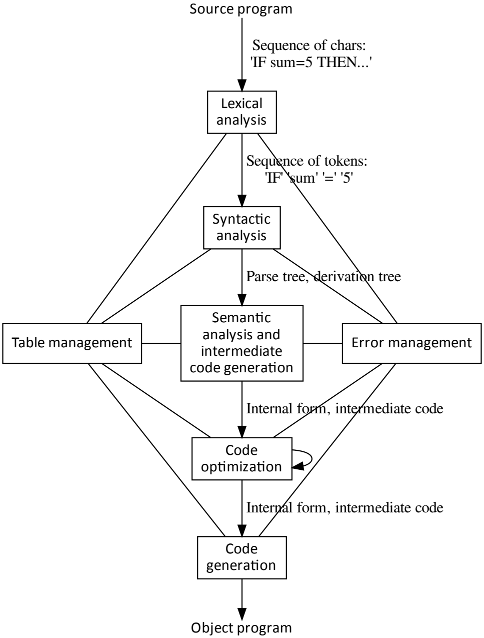
Overview

Part I	Background
Part II	Implementation of a Modelica Compiler

Part I

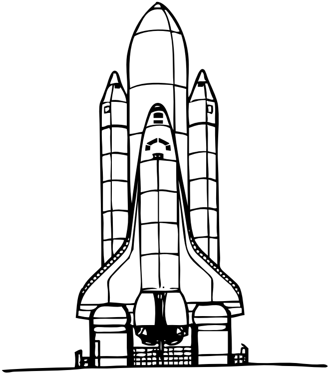
Background

The Phases of the Classic Compiler



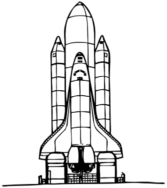
Systems Engineering

- Handling large, complex projects.
- Combining requirements, modeling, simulation, deployment, support, etc.
- Inter-disciplinary.
- You often end up with many different tools because different domains traditionally used different tools.



Systems Engineering – Example Tools

- Early stage – Administrators and managers email around Word (contracts) and Excel (requirements) sheets
- Requirements are formalized and stored in a database somewhere
- Requirements are mapped to UML models
- UML is mapped to a design (empty classes)
- The actual code is written (perhaps C)
- The code is adapted/generated to run on a certain platform (MISRA C perhaps?)
- The code tested/certified, etc



Domain-Specific Languages

- Many are similar to classic, general-purpose programming languages (e.g. PHP).
- Examples include unix shells, SQL, HTML, regular expressions, parser generators, some XML schemas, and many more.
- Compilers are usually implemented partially using domain-specific languages (grammars, special languages to describe architectures, etc).
- Why? It is easier to program and maintain such code.

DSLs: Markup Languages

```html
<!DOCTYPE html>
<html>
<body>

<h1>My first HTML page</h1>

<p>Hello, world!</p>

</body>
</html>
```

DSLs: Template Languages

```
<!DOCTYPE html>
<html>
<body>

<h1>My first PHP page</h1>
<?php
echo $_SERVER["REMOTE_ADDR"];
?>
</body>
</html>
```

DSLs: Embedded Scripting Languages

```html
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

DSLs: Regular expressions

```
grep "status: *correct" "$test"
grep -R "openmodelica[.]org" /etc/apache2
```

Modelica

- An equation-based object-oriented modeling language (a DSL).
- Modeling using a graphical user interface (or the equivalent textual representation).
- Used for simulation and/or control of multi-domain (physical) systems.
- Centered around making it easy for a (mechanical, electrical, etc) engineer to use Modelica.

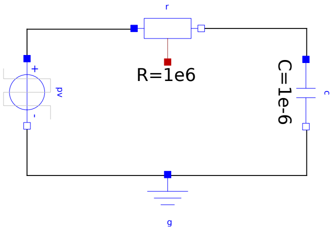


Figure: An RC-circuit implemented in Modelica.

Simulating the RC-circuit

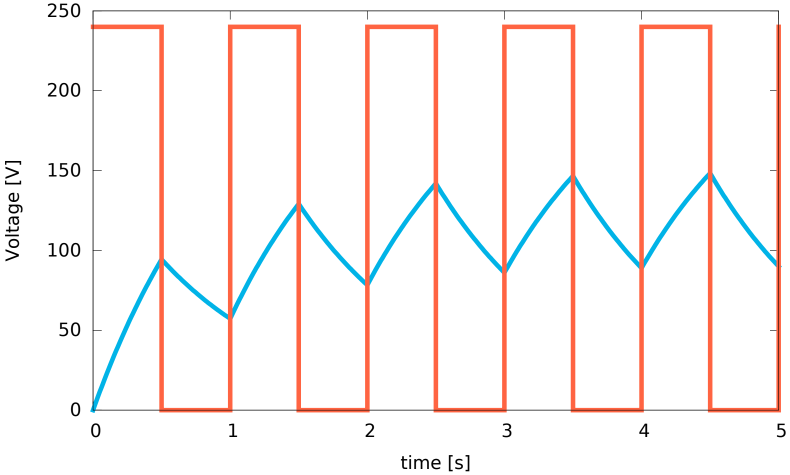


Figure: Result of simulating the RC-circuit.

Equations

Physics is described by *equations*, not statements. Thus, Modelica primarily uses equations instead of imperative programming (like C).

- Equations look like $\frac{V}{R} = I$

However, code needs to be translated to imperative programming (or similar) in order to run numerical solvers on a CPU. So it could be solved as either of:

- $V := R * I$
- $I := \frac{V}{R}$
- $R := \frac{V}{I}$

Ordinary Differential Equations (ODEs)

The numerical solvers we use require an ODE formulation. For example:

$$\frac{\partial x}{\partial t} = y \tag{1}$$

$$d = 2.0 * t \tag{2}$$

$$\frac{\partial y}{\partial t} = -d * x \tag{3}$$

Where these equations can be solved sequentially and the start value for each state variable is known or can be solved during initialization.

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00
x	3.00
y	2.00
$\frac{\partial x}{\partial t} = y$	
$d = 2.0 * t$	
$\frac{\partial y}{\partial t} = -d * x$	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00
x	3.00
y	2.00
$\frac{\partial x}{\partial t} = y$	2.00
$d = 2.0 * t$	
$\frac{\partial y}{\partial t} = -d * x$	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t=0)=3$ and $y(t=0)=2$:

t	0.00
x	3.00
y	2.00
$\frac{\partial x}{\partial t} = y$	2.00
$d = 2.0 * t$	0.00
$\frac{\partial y}{\partial t} = -d * x$	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t=0) = 3$ and $y(t=0) = 2$:

t	0.00
x	3.00
y	2.00
$\frac{\partial x}{\partial t} = y$	2.00
$d = 2.0 * t$	0.00
$\frac{\partial y}{\partial t} = -d * x$	0.00

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t=0) = 3$ and $y(t=0) = 2$:

t	0.00	0.25
x	3.00	3.50
y	2.00	
$\frac{\partial x}{\partial t} = y$	2.00	
$d = 2.0 * t$	0.00	
$\frac{\partial y}{\partial t} = -d * x$	0.00	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25
x	3.00	3.50
y	2.00	2.00
$\frac{\partial x}{\partial t} = y$	2.00	
$d = 2.0 * t$	0.00	
$\frac{\partial y}{\partial t} = -d * x$	0.00	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25
x	3.00	3.50
y	2.00	2.00
$\frac{\partial x}{\partial t} = y$	2.00	2.00
$d = 2.0 * t$	0.00	0.50
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25	0.50
x	3.00	3.50	4.00
y	2.00	2.00	1.5625
$\frac{\partial x}{\partial t} = y$	2.00	2.00	
$d = 2.0 * t$	0.00	0.50	
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25	0.50
x	3.00	3.50	4.00
y	2.00	2.00	1.5625
$\frac{\partial x}{\partial t} = y$	2.00	2.00	1.5625
$d = 2.0 * t$	0.00	0.50	1.00
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75	-4.00

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25	0.50	0.75
x	3.00	3.50	4.00	4.390625
y	2.00	2.00	1.5625	0.5625
$\frac{\partial x}{\partial t} = y$	2.00	2.00	1.5625	
$d = 2.0 * t$	0.00	0.50	1.00	
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75	-4.00	

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t=0)=3$ and $y(t=0)=2$:

t	0.00	0.25	0.50	0.75
x	3.00	3.50	4.00	4.390625
y	2.00	2.00	1.5625	0.5625
$\frac{\partial x}{\partial t} = y$	2.00	2.00	1.5625	0.5625
$d = 2.0 * t$	0.00	0.50	1.00	1.50
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75	-4.00	-6.5859375

Solving the Ordinary Differential Equation

Solving the ODE from t=0 to t=1 using a step size of 0.25 with explicit (forward) Euler, using $x(t = 0) = 3$ and $y(t = 0) = 2$:

t	0.00	0.25	0.50	0.75	1.00
x	3.00	3.50	4.00	4.390625	4.53125
y	2.00	2.00	1.5625	0.5625	-1.083984375
$\frac{\partial x}{\partial t} = y$	2.00	2.00	1.5625	0.5625	
$d = 2.0 * t$	0.00	0.50	1.00	1.50	
$\frac{\partial y}{\partial t} = -d * x$	0.00	-1.75	-4.00	-6.5859375	

Better Solutions for the Ordinary Differential Equation

Modelica tools need to know about ODEs, and can solve them better than the explicit Euler. The end values for different solvers:

- Euler stepSize=0.25, x=4.531, y=-1.084, 4 rhs calls
- Euler stepSize=0.10, x=4.087, y=-1.600, 10 rhs calls
- RK4 stepSize=1.00, x=3.667, y=-1.667, 4 rhs calls
- RK4 stepSize=0.25, x=3.747, y=-1.841, 16 rhs calls
- DASSL, tolerance=1e-3, x=3.745, y=-1.838, 15 rhs calls
- DASSL, x=3.747, y=-1.842, 67 rhs calls
- Euler stepSize=1e-5, x=3.747, y=-1.842, 10000 rhs calls

Ability to choose numerical solver based on needs (predictable execution time, fast execution time, or accurate).

```
model ODE
 Real d=2*time;
 Real x(start=3.0,
        fixed=true);
 Real y(start=2.0,
        fixed=true);
equation
 der(x) = y;
 der(y) = -d*x;
end ODE;
```

Better Solutions for the Ordinary Differential Equation

Modelica tools need to know about ODEs, and can solve them better than the explicit Euler. The end values for different solvers:

- Euler stepSize=0.25, x=4.531, y=-1.084, 4 rhs calls
- Euler stepSize=0.10, x=4.087, y=-1.600, 10 rhs calls
- RK4 stepSize=1.00, x=3.667, y=-1.667, 4 rhs calls
- RK4 stepSize=0.25, x=3.747, y=-1.841, 16 rhs calls
- DASSL, tolerance=1e-3, x=3.745, y=-1.838, 15 rhs calls
- DASSL, x=3.747, y=-1.842, 67 rhs calls
- Euler stepSize=1e-5, x=3.747, y=-1.842, 10000 rhs calls

Ability to choose numerical solver based on needs (predictable execution time, fast execution time, or accurate).

```
model ODE
 Real d=2*time;
 Real x(start=3.0,
        fixed=true);
 Real y(start=2.0,
        fixed=true);
equation
 der(x) = y;
 der(y) = -d*x;
end ODE;
```

Multi-Domain Approach: Systems Engineering

Modelica is a multi-domain approach based on math:

- It is suitable to simulate different physical domains independently or multiple domains in the same model.
- Can be used to simulate full systems or synthesize parts of a system (such as digital controllers).
- The language fulfills the requirements of Systems Engineering.

Part II

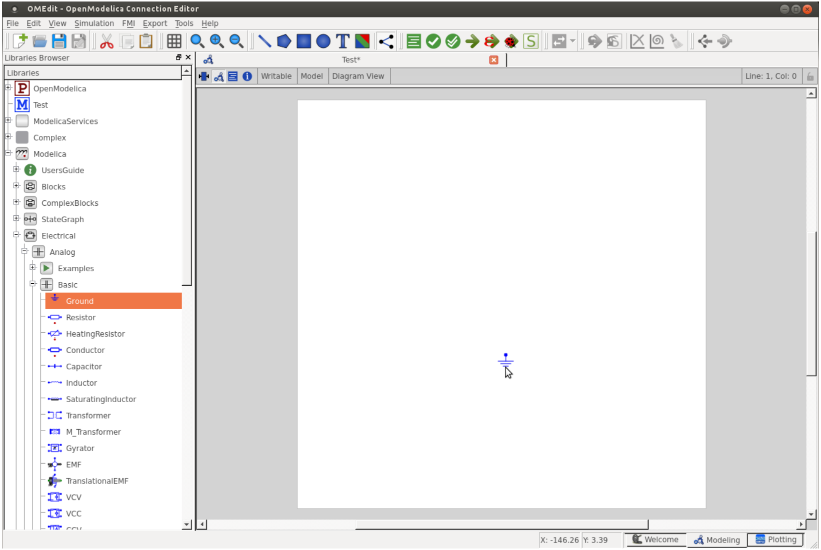
Implementation of a Modelica Compiler

User in the Focus

Modelica is designed around the user of the language being the focus:

- This makes implementation of the compiler harder.
- NP-hard problems need to be solved at *compile-time*.
- Compilation time is unbounded and includes interpretation of arbitrary code.
- Certain language features are a little weird when you consider the textual representation, but make sense when for the graphical user interface.

User in the Focus – OMEdit / OMWebBook Demo



The Compiler Design

A Modelica compiler needs to have lots of domain knowledge. It also depends on heuristics to translate equations into ODEs (the main job of a Modelica compiler), which are translated to executable code.

- The heuristics may fail to create an ODE from the given code even when a solution exists.
- Solutions do not necessarily exist.
- Numerical solvers may fail to solve the model (require infinite resolution).
- Solution: good error messages and debuggers.

In practice, this works very well.

OpenModelica Overview

- Written in MetaModelica (general-purpose programming extension to Modelica).
- The code generator uses our own DSL Susan for text generation (which translates Susan code to MetaModelica).
- ANTLR parser which translates an ANTLR grammar with a C-code target, which uses the C interface to MetaModelica to create data types.
- Our own flex-based lexer which generates a MetaModelica-based lexer.
- Kernel written in Modelica, MetaModelica, C, C++, Susan, ANTLR, flex. DSL's almost everywhere.

OpenModelica Parts

- Parser (using the ANTLR parser generator).
- Front-end (semantic analysis, like a traditional compiler). Old version was one huge monolithic step.
- Equation back-end (symbolic math, outputs imperative code from equations). (Very) old version was one huge monolithic step.
- Code generator (takes imperative code and generates of C-code, skipping middle-end and back-end of a traditional compiler).
- Utilities.
- Scripting environment.
- Front-end + code generator handles MetaModelica (functions).
- The compiler is also written in MetaModelica (bootstrapping).

Modelica code for RC-circuit (GUI annotations stripped)

```
model RC
  Modelica.Electrical.Analog.Basic.Ground g;
  Modelica.Electrical.Analog.Basic.Resistor r(R = 1e6);
  Modelica.Electrical.Analog.Basic.Capacitor c(C = 1e-6);
  Modelica.Electrical.Analog.Sources.SineVoltage sineVoltage;
equation
  connect(r.n, c.p);
  connect(c.n, g.p);
  connect(sineVoltage.p, r.p);
  connect(g.p, sineVoltage.n);
end RC;
```

FrontEnd phases

- FrontEnd - loaded program (parsing dependent libraries, etc)
- Absyn->SCode (`Real[3] x[2], y[4]; -> Real x[2,3]; Real y[4,3];`)
- NFInst.instantiate
- NFInst.instExpressions
- NFInst.updateImplicitVariability (`Real arr[x] // x is structural`)
- NFTyping.typeComponents (`Real r`)
- NFTyping.typeBindings (`Real r = 1.5`)
- NFTyping.typeClassSections (equations)
- NFFlatten.flatten
- NFFlatten.resolveConnections
- NFEvalConstants.evaluate
- NFSimplifyModel.simplify ("constant folding")
- NFPackage.collectConstants
- NFFlatten.collectFunctions (removes now unused functions)
- NFFlatModel.toFlatString

Flat Modelica IR; can be fed into a Modelica compiler

```
class 'RC'
  public Real 'g.p.v'(unit = "V", quantity = "ElectricPotential");
  public Real 'g.p.i'(unit = "A", quantity = "ElectricCurrent");
  public parameter Real 'r.R'(start = 1.0, unit = "Ohm", quantity = "Resistance") = 1000000.0;
  public parameter Real 'r.T_ref'(nominal = 300.0, start = 288.15, min = 0.0, displayUnit = "degC", unit =
  public parameter Real 'r.alpha'(unit = "1/K", quantity = "LinearTemperatureCoefficient") = 0.0;
  public Real 'r.v'(unit = "V", quantity = "ElectricPotential");
  // ...
equation
  'r.n.v' = 'c.p.v';
  'g.p.v' = 'sineVoltage.n.v';
  'g.p.v' = 'c.n.v';
  'sineVoltage.p.v' = 'r.p.v';
  'c.p.i' + 'r.n.i' = 0.0;
  'sineVoltage.n.i' + 'c.n.i' + 'g.p.i' = 0.0;
  'sineVoltage.p.i' + 'r.p.i' = 0.0;
  'g.p.v' = 0.0;
  assert(1.0 + 'r.alpha' * ('r.T_heatPort' - 'r.T_ref') >= 1e-15, "Temperature outside scope of model!",
  'r.R_actual' = 'r.R' * (1.0 + 'r.alpha' * ('r.T_heatPort' - 'r.T_ref'));
  'r.v' = 'r.R_actual' * 'r.i';
  'r.LossPower' = 'r.v' * 'r.i';
  'r.T_heatPort' = 'r.T';
  0.0 = 'r.p.i' + 'r.n.i';
  'r.i' = 'r.p.i';
  'r.v' = 'r.p.v' - 'r.n.v';
  'c.i' = 'c.C' * der('c.v');
  0.0 = 'c.p.i' + 'c.n.i';
  'c.i' = 'c.p.i';
  'c.v' = 'c.p.v' - 'c.n.v';
  'sineVoltage.signalSource.y' = 'sineVoltage.signalSource.offset' + (if time < 'sineVoltage.signalSource.
  'sineVoltage.v' = 'sineVoltage.signalSource.y';
  0.0 = 'sineVoltage.p.i' + 'sineVoltage.n.i';
  'sineVoltage.i' = 'sineVoltage.p.i';
  'sineVoltage.v' = 'sineVoltage.p.v' - 'sineVoltage.n.v';
end 'RC';
```

FrontEnd-Backend conversion phases

- NFScalarize.scalarize (`Real r[3] -> Real 'r[1]'; Real 'r[2]'; Real 'r[3]';`)
- NFVerifyModel.verify
- NFConvertDAE.convert (convert to the scalar-only backend)
- FrontEnd - DAE generated
- Transformations before Dump
- DAEDump done
- Misc Dump
- Transformations before backend

Backend conversion phases (except initialization)

- Generate backend data structure
- preOpt normalInlineFunction
- preOpt evaluateParameters
- preOpt simplifyIfEquations
- preOpt expandDerOperator (`der(2*x) -> 2*der(x)`)
- preOpt clockPartitioning
- preOpt findStateOrder
- preOpt replaceEdgeChange
- preOpt inlineArrayEqn
- preOpt removeEqualRHS (`x = f(...); y = f(...) -> x = f(...); y = x;`)
- preOpt removeSimpleEquations (`x = -y;`, `x = 2.0;` removed and reconstructed during plotting)
- preOpt comSubExp (bad name)
- preOpt evalFunc
- preOpt encapsulateWhenConditions

Note Which optimizations run can be configured at runtime. And there are many flags to change algorithms used in them. Compilation may fail if some optimization phases are disabled.

Why is it OK that optimization changes the result?

Matching and sorting (different example)

```
class RC // 5 equations and variables
 // 14 alias variables 5 constants
algorithm // The equations are now ordered
  // 1
  sinevoltage.signalSource.y := sinevoltage.signalSource.of
  // 2
  r.v := c.v -sinevoltage.signalSource.y;
  // 3
  c.i := -r.v / r.R_actual;
  // 4
  r.LossPower := -r.v * c.i;
  // 5
  der(c.v) := c.i / c.C;
end RC;
```

Backend conversion phases (except initialization)

- postOpt lateInlineFunction (inline after matching)
- postOpt wrapFunctionCalls
- postOpt inlineArrayEqn
- postOpt constantLinearSystem
- postOpt simplifysemiLinear
- postOpt removeSimpleEquations
- postOpt simplifyComplexFunction
- postOpt solveSimpleEquations
- postOpt tearingSystem (method for solving nonlinear systems)
- postOpt inputDerivativesUsed
- postOpt calculateStrongComponentJacobians (extra runtime information)
- postOpt calculateStateSetsJacobians
- postOpt symbolicJacobian
- postOpt removeConstants
- postOpt simplifyTimeIndepFuncCalls
- postOpt simplifyAllExpressions (constant folding everywhere)
- postOpt findZeroCrossings
- postOpt collapseArrayExpressions
- sorting global known variables
- sort global known variables
- remove unused functions

Code generation (dump C-code)

- Create SimCode IR from backend IR (populate some hashtables, etc to make code generation easier)
- Templates (SimCode to C-code)

Intermediate representations

- Abstract syntax tree (from parser) including comments. Used in the API to update the program without changing the textual representation. Real numbers represented by strings.
- Exploded syntax tree. A canonical form of the AST used by the front-end.
- A single "New Frontend" data structure, including classes (such as functions) and the instantiated components. This is lowered step by step. At the final step, it is typed and a lot of language features no longer exist.
- The old frontend data structure. This is scalar (no arrays remain).
- The backend data structure is generated from the old frontend data structure. It is a set of time-dependent partitions and common known variables. It starts simple (preOpt modules), and after sorting/matching it contains an adjacency matrix (adjacency list) of variables/equations as well as having sorted all equations.
- The SimCode data structure is generated from the sorted equations and gives final indexes of everything, and more information.

Functions (algorithmic code)

Functions are similar to imperative programming languages, and a few optimization are performed:

- Function inlining
- Constant propagation
- Constant folding
- Dead code elimination
- Finding potential use of uninitialized variables

Some of these might only be implemented in the old frontend / bootstrapping.

The backend can also create specialized functions for some constant inputs to functions (eliminating a lot of branches, assertions, or allowing inlining).

More Reading

- `https://modelica.org`
- `https://openmodelica.org`

www.liu.se