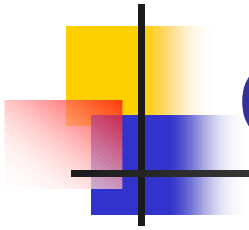
The Design of the MuModelica Compiler

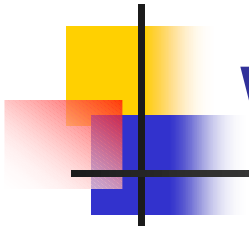
Steven Xu
@MSDL
School Of Computer Science
McGill University

Aug 27, 2004



Overview of Modelica

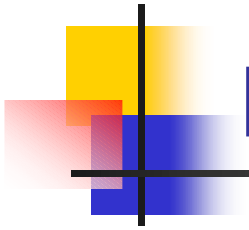
- Modelica is a freely available, object-oriented language for modeling of large, complex, and heterogeneous physical system.
- Built on non-causal modeling with mathematical equations and object-oriented constructs to facilitate reuse of modeling knowledge



Why Modelica Compiler

In order that the Modelica modeling language can be used to solve actual problems, a modeling and simulation environment (tool) is needed:

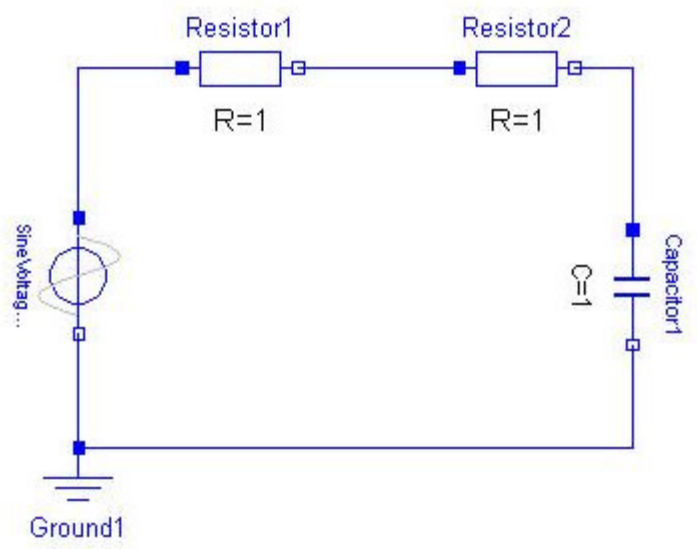
- to translate Modelica models into a form which can be efficiently simulated in an appropriate simulation environment
- to simulate the translated model with standard numerical integration methods and visualize the result

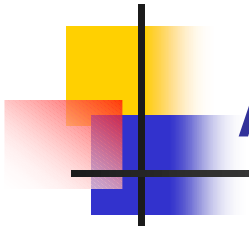


Basic Language Elements

- Basic components: Real, Integer, Boolean and String
- Structured components
- Component arrays, to handle real matrices, arrays of sub-models etc
- Equations and/or algorithms (assignment statements)
- Connections
- Functions

A simple electrical circuit





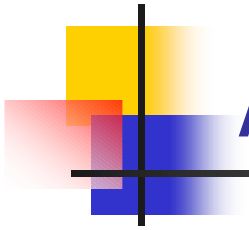
A simple electrical circuit

- Declaring two physical quantities

```
type Voltage = Real(quantity="Voltage", unit="V");
type Current = Real(quantity="Current", unit="A");
```

- Defining a connector

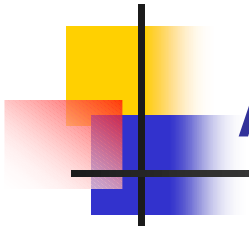
```
connector Pin "pin of an electric component"
    Voltage v "Potential at the pin";
    flow Current i "Current flowing into the pin";
end Pin;
```



A simple circuit in Modelica

- A connection `connect (Pin1, Pin2)`, connects the two pins such that they form one node
- The keyword `flow` is used to generate sum-to-zero equations
- A connection implies two equations:

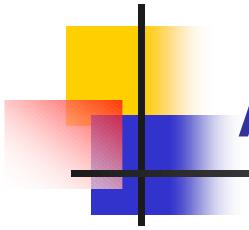
```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```



A simple circuit in Modelica

- An electrical port

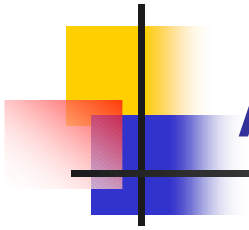
```
partial model OnePort "Superclass of Components with
two electrical pins p and n"
    Voltage v "Voltage drop between p and n";
    Current i "Current flowing from p to n";
    Pin p;
    Pin n;
equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
end OnePort;
```



A simple circuit in Modelica

- A resistor

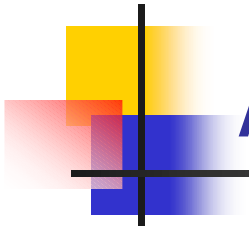
```
model Resistor "Ideal linear electrical resistor"
    extends OnePort;
    parameter Real r(unit="O") "Resistance"
equation
    r*i = v;      "Ohm's Law"
end Resistor;
```



A simple circuit in Modelica

- A capacitor

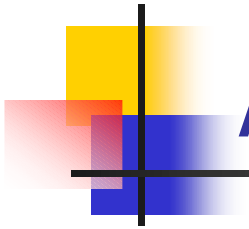
```
model Capacitor "Ideal electrical Capacitor"
    extends OnePort;
    parameter Real c(unit="F") "Capacitance"
equation
    c*der(v) = i;
end Capacitor;
```



A simple circuit in Modelica

- A sin-wave voltage source

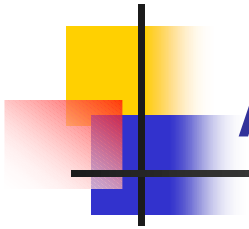
```
model VsourceAC "sin-wave voltage source"
  extends OnePort;
  parameter Voltage VA = 220 "Amplitude";
  parameter Real f (unit="Hz") = 50 "Frequency";
  constant Real PI = 3.14159265;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```



A simple circuit in Modelica

- A ground point

```
model Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;
```

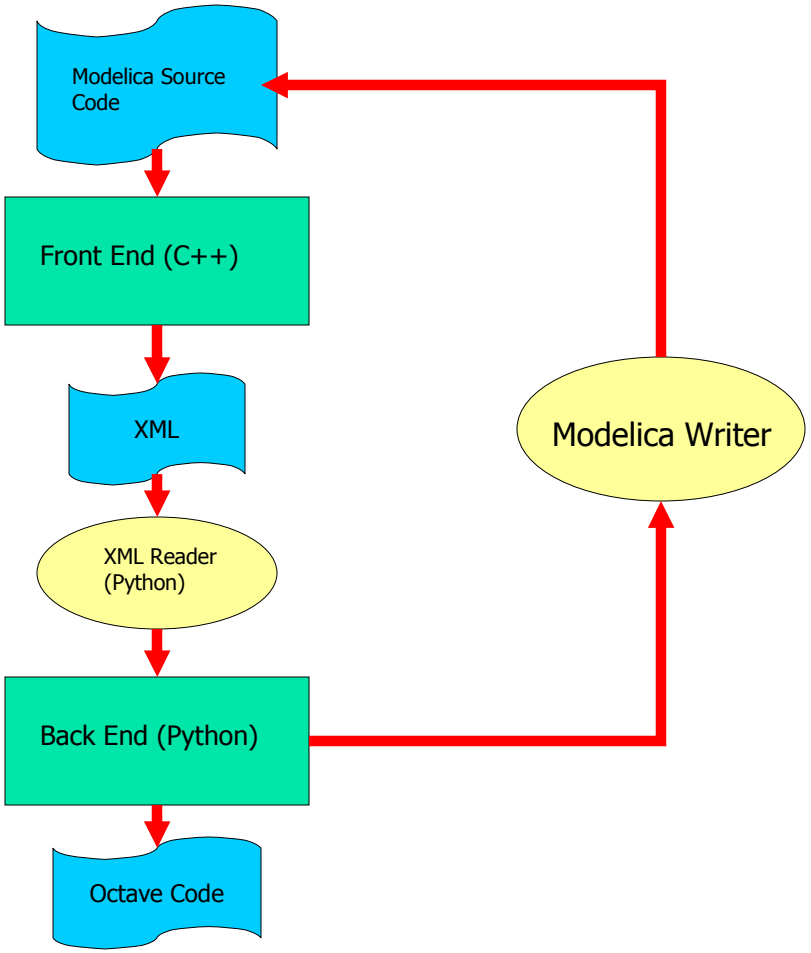


A simple circuit in Modelica

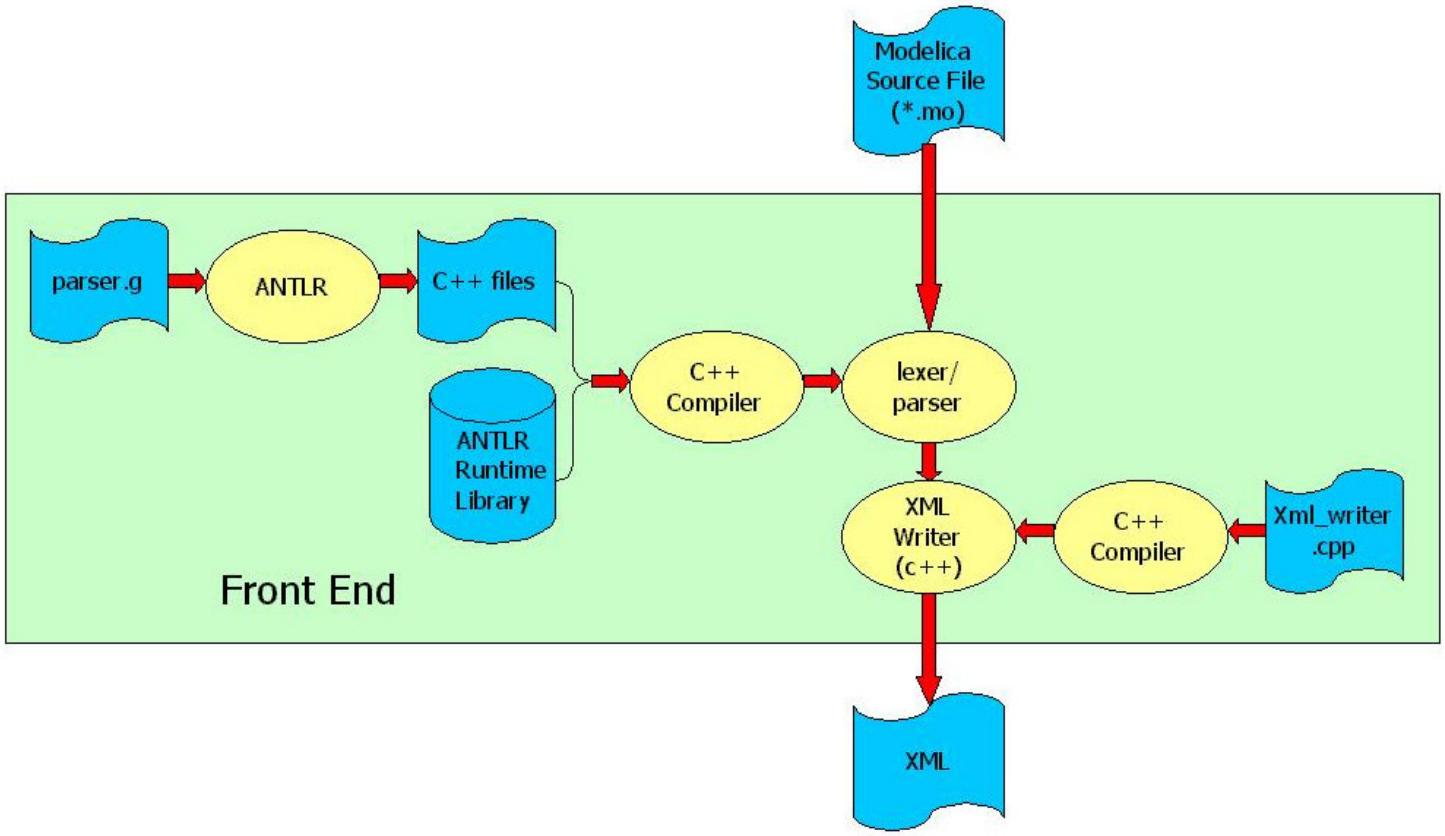
- The circuit

```
model circuit
     Resistor R1(r=10);
     Resistor R2(r=10);
     Capacitor C(c=1);
     VsourceAC AC;
     Ground G;
equation
     connect(AC.p, R1.p);
     connect(R1.n, R2.p);
     connect(R2.n, C.p);
     connect(C.n, AC.n);
     connect(AC.n, G.p);
end circuit;
```

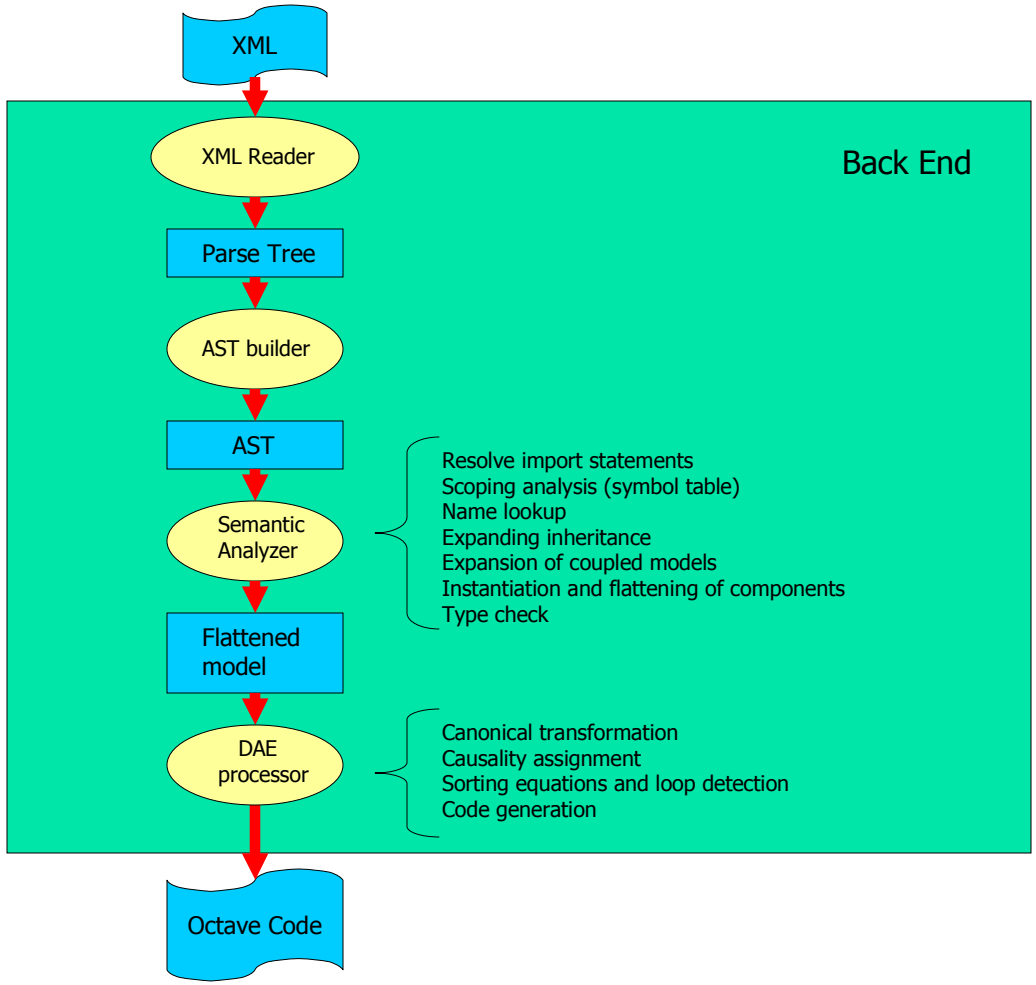
Overview of the MuModelica Compiler

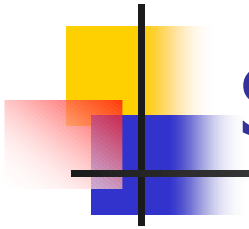


The Front End



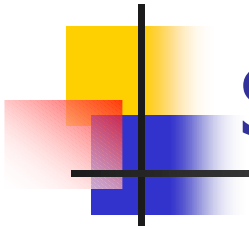
The Back End


XML
XML Reader
Back End
Parse Tree
AST builder
AST
Semantic Analyzer
Resolve import statements
Scoping analysis (symbol table)
Name lookup
Expanding inheritance
Expansion of coupled models
Instantiation and flattening of components
Type check
Flattened model
DAE processor
Canonical transformation
Causality assignment
Sorting equations and loop detection
Code generation
Octave Code



Scoping Analysis

- Scoping analysis is characterized by the introduction and maintenance of *symbol tables*
- A symbol table stores mappings of identifiers to their types and definitions.
- As class definitions and declarations are processed, bindings from identifiers to their meanings are added to the symbol tables.



Scoping Analysis

- Data structure

```
// example: A.mo
class A
  Real a;
  Real b;
equation
  a = b;
end A;
```

Scoping Analysis

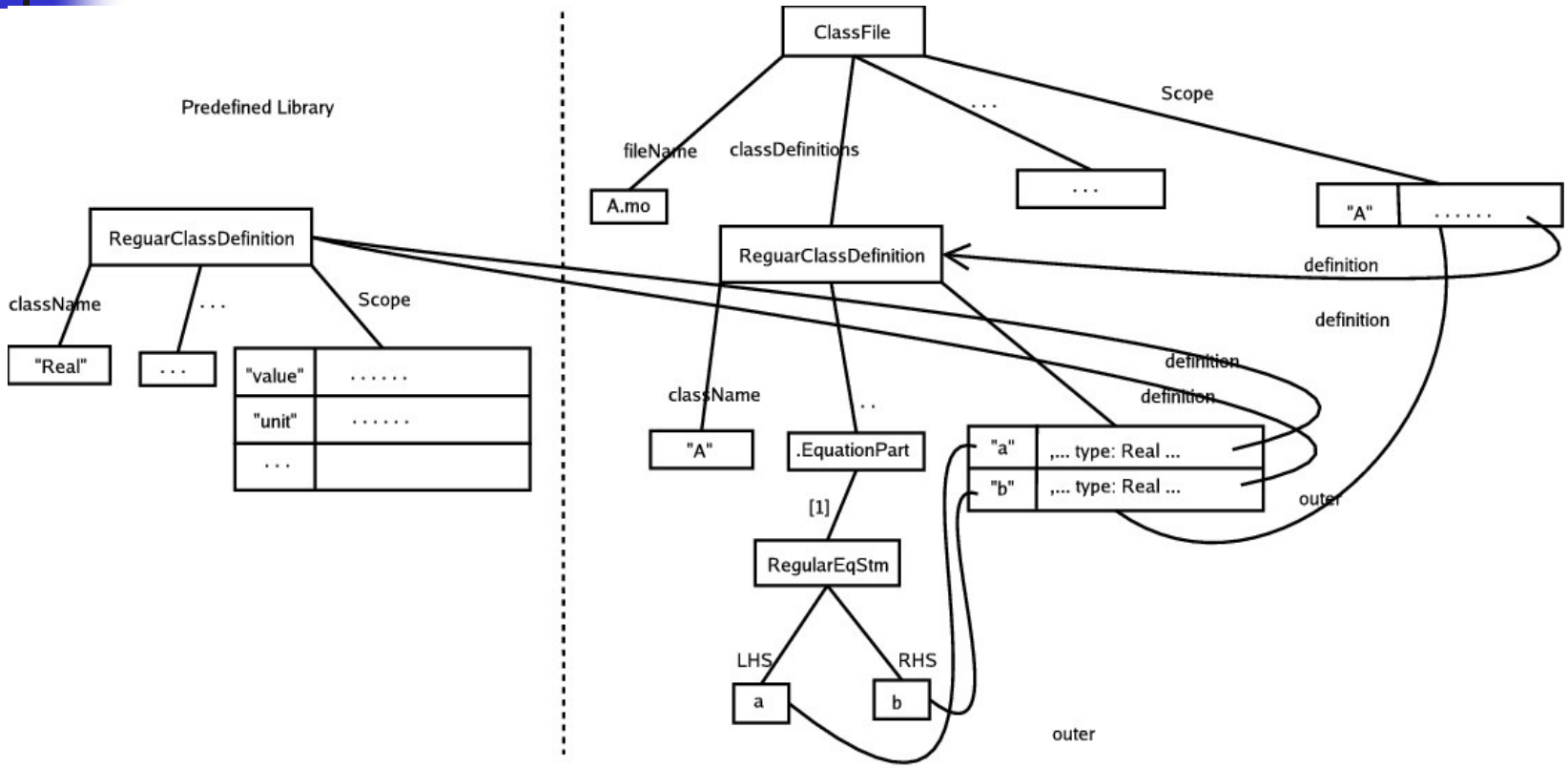
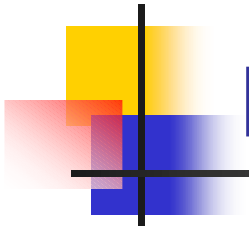


Figure. AST with scope nodes



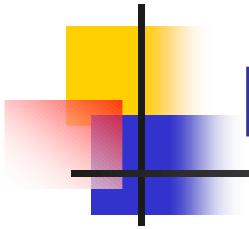
Expansion of Class Inheritance

- Expansion of inheritance in Modelica means copying all elements (both definitions and declarations) and equations from the base class.
- Inheritances need to be expanded before names are looked up.

Expansion of Class Inheritance

```
package P
  constant Real PI=3.14;
  class A
    Real a1, a2;
  equation
    a1 = a2;
  end A;
  class B
    A a(a2=PI);
  end B;
end P;
```

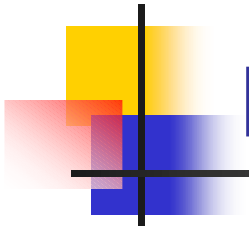
```
class C
  class C1
    Real c11;
  end C1;
end C;

model M
  extends C;
  extends P.B;
  C1 x;
end M;
```



Expansion of Class Inheritance

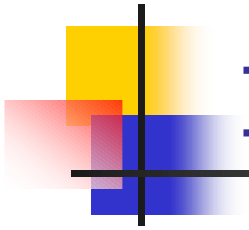
- Expanding the extends clauses in model M leads to the following expanded version of M

```
model M
  // inherited from C
  class C1
    Real c11;
  end C1;
  // inherited from P.B
  constant Real PI=3.14;
  A a(a2=PI);
 C1 x;
end M;
```



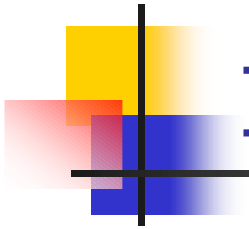
Name lookup

- When uses of identifiers are found, they are looked up in the symbol tables.
- The current Modelica language (version 2.0) allows **use-before-declare** (UBD).
- Multiple passes are required to support UBD in implementation.
- Lookup algorithm (refer to “report” online)



Instantiation

- Class modification creates a variant of the original class definition.
- The data structure has to be augmented to hold concrete instances for these variants when components are flattened and modifications are merged.
- An instance is a copy of its original definition, except that it carries additional information of modification.



Instantiation

```
class A
  Real a1;
  Integer a2=1;
end A;

class B
  Real b1(unit="N");
  Real b2=2.0;
end B;

model M
  extends A;
  B b(b1=1.5, b2=3.0);
end M;
```

Instantiation

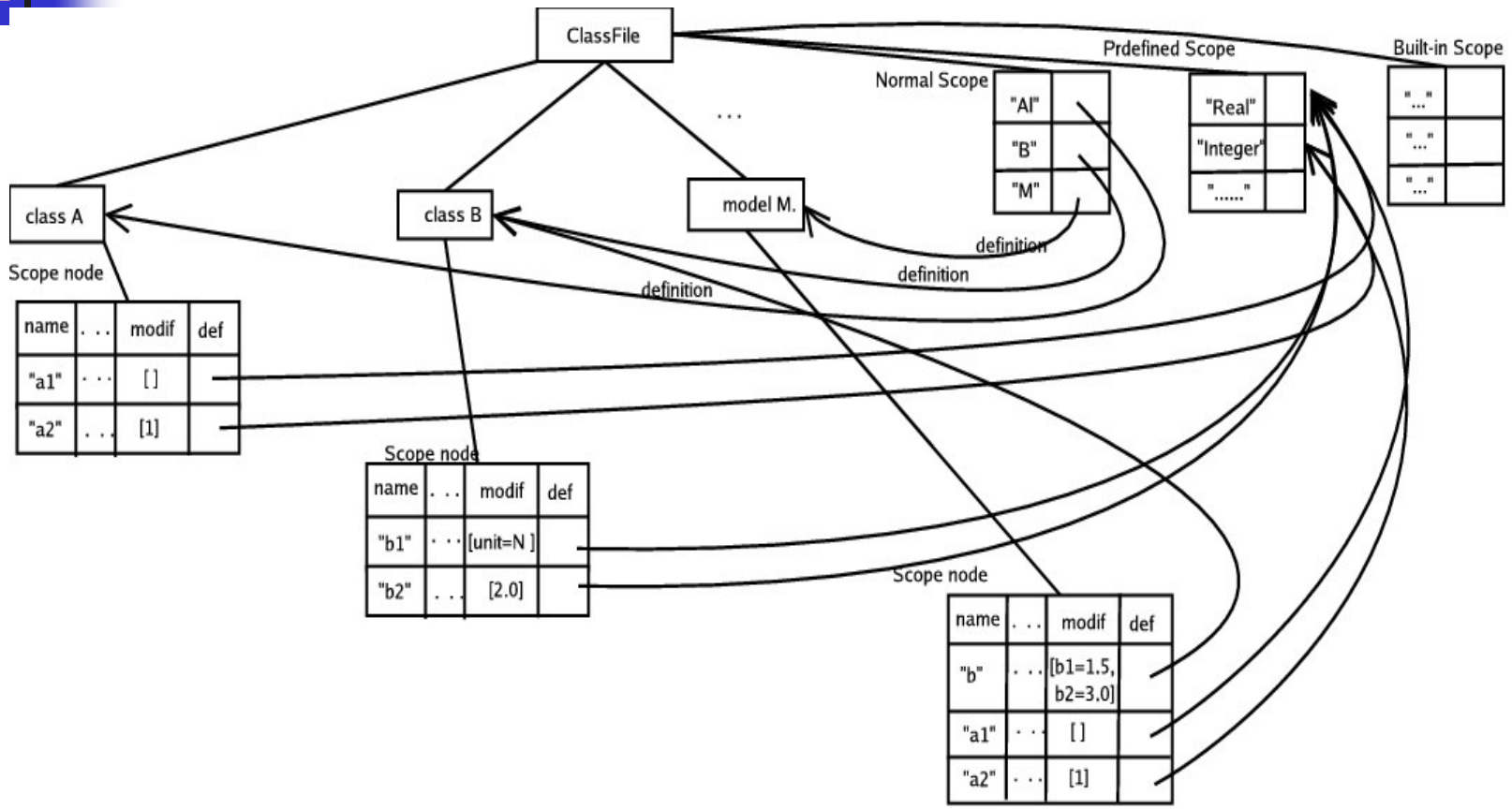


Figure. AST before Instantiation

Instantiation

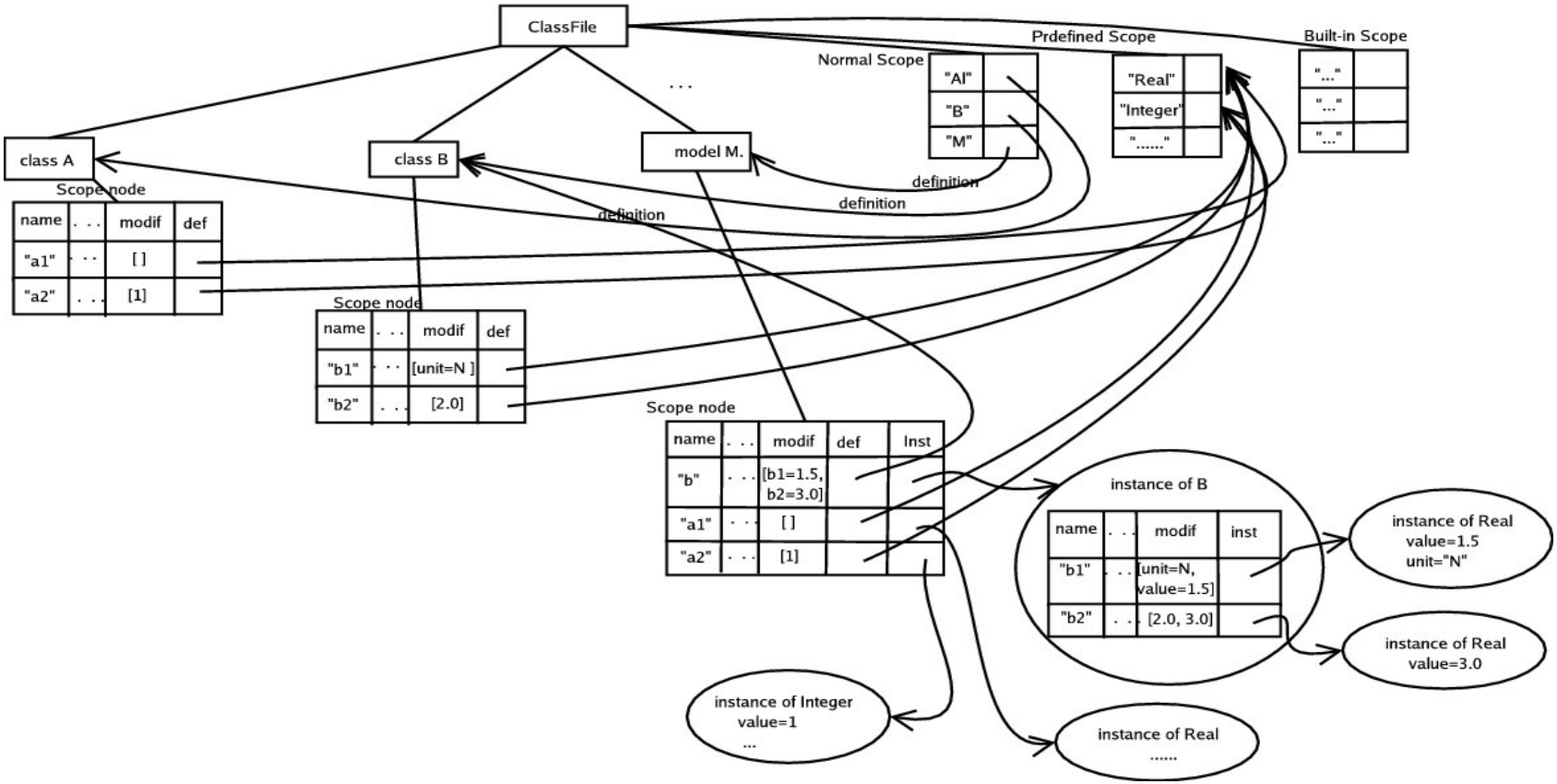
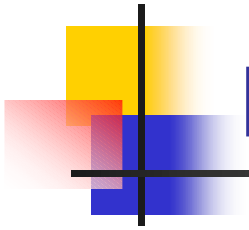
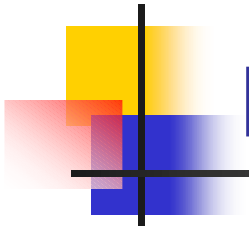


Figure. AST after Instantiation



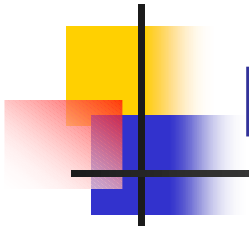
Flattening

- The goal of semantic analysis is to transform original Modelica models to flat DAE form
- A at DAE form of Modelica model consists of
  - Declarations of variables in predefined types, e.g., Real a
  - Equations from the equation section with names resolved.



Flattening

- Main design issues in flattening are:
  - Expand composite components to declarations in predefined types, and assign each of these declarations a globally unique name.
  - Merge modifications at built-in type level, and turn modifications into equations
  - Expand coupled models, that is, replace “connect” statements by connection equations

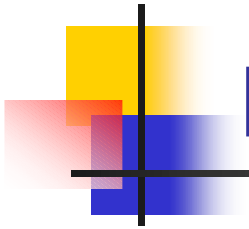


Flattening

```
class A
  Real a1;
  Integer a2=1;
end A;

class B
  Real b1(unit="N");
  Real b2=2.0;
end B;

model M
extends A;
B b(b1=1.5, b2=3.0);
end M;
```

flatten

```
model M
  Real a1;
  Integer a2=1;
  Real b_b1(unit="N")=1.5;
  Real b_b2=3.0;
end M;
```



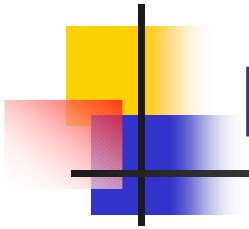
Flattening

- Expanding coupled components: components are coupled by `connect` statements. These statements imply additional equations, eg.

```
connect(AC.p, R1.p)
```

implies

```
AC.p.v = R1.p.v
AC.p.i + R1.p.i = 0
```



Flattening

- All flow variables sum to zero at each node. For example, flattening

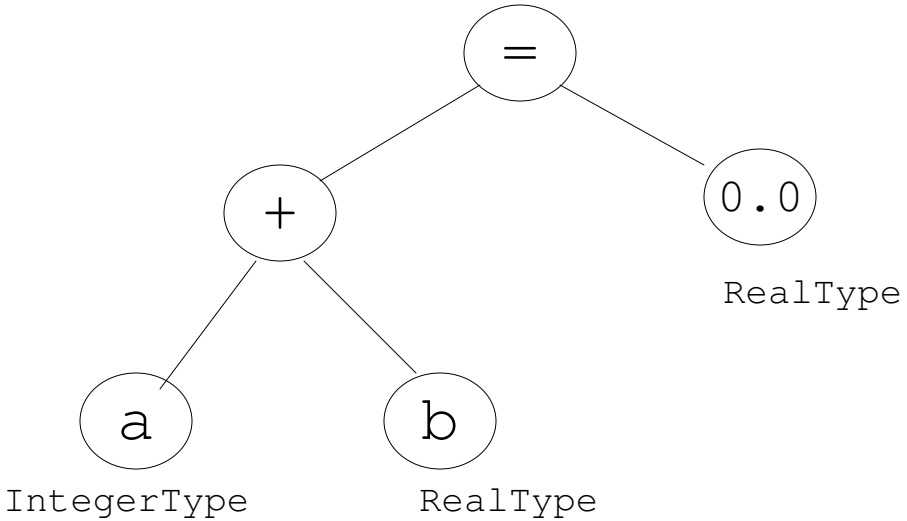
```
connect(C.n, AC.n);
connect(AC.n, G.p);
```

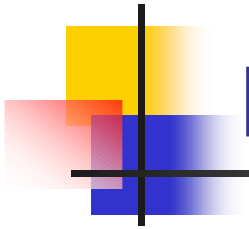
give equations

```
c.n.v = AC.n.v
AC.n.v = G.p.i
c.n.i + AC.n.i + G.p.i = 0
```

Type Check

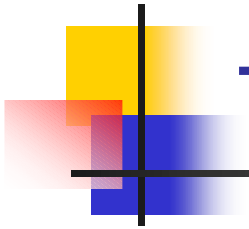
- Type checking: check that the types of operands are legally defined over each operation.
- Type check is performed at the built-in type level





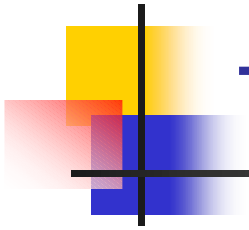
Flattening

DEMO



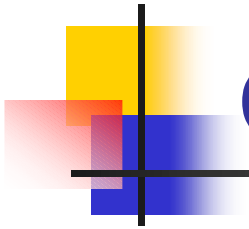
The DAE Processor

- DAE solvers are inefficient
- A far more efficient approach: DAE transformation
- In many cases, the DAE processor can transform the original DAE into explicit equations in a correct computation order



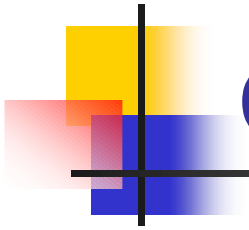
The DAE Processor

- Different transformation steps:
  - Eliminate aliases from equations (ie. get rid of equations of the type a=b)
  - Canonical transformation, including for example, constant folding
  - Causality assignment
  - Sort equations/detect algebraic loops
  - Inline integration
  - Solve algebraic loops (linear and non-linear)



Canonical Transformation

- An expression or equation is considered as a tree made up of operators and their operands
- Canonical representation: an expression or equation is stored internally in a particular, unique order. More specifically,
  - constants are folded
  - the operators and operands at every level of the tree are in a unique order



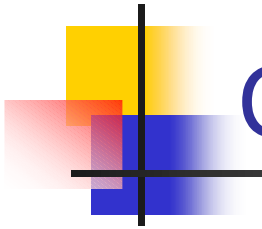
Canonical Transformation

- Why canonical transformation:
  - Easier formula manipulation
  - Runtime efficiency. For example, if constants are folded at compile time, there is no need to calculate the same operations on these constants at each time step at simulation runtime time
  - Equations like `x+x+y=0` cannot be transformed to causal form correctly (if `x` is to be calculated based on the value of `y`, i.e., `x` is in the LHS)

Canonical Transformation

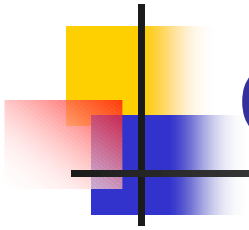
- Simplification rules:
  - Before sorting the equation tree into canonical order, the following rules should be implemented
    - The RHS of an equation is moved to the LHS, and the RHS is set to 0.0, e.g., `a=b` is transformed to `a-b=0.0`
    - Constants are rewritten as real numbers. Fractions are evaluated
    - A negative number or term is written as:
      - -c=+(-c), where c is a constant
      - -E=+(-1.0)*E, where E is a term
    - Expressions in reciprocal form (divisions) are rewritten in terms of negative powers, e.g., `x/y=x*y^(-1.0)`
    - Binary operators `+` and `*` are converted to n-ary operators since they are both commutative and associative. For example,

```
a+(b+c)=(a+b)+c=+(a,b,c)
a*(b*c)=(a*b)*c=*(a,b,c)
```



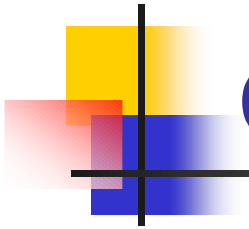
Canonical Transformation

- The following rules can now be repeatedly applied to the equation to transform it into canonical order
  - Constant folding, e.g., `x+3+2 = b -> x+5 = b`
  - remove superfluous zeros and ones, e.g., `0.0+E -> E`, `1.0*E -> E`
  - Like terms in a sum are collected and their constant coefficients are added, e.g., `a*x^(p)+b*x^(p) -> (a+b)*x^(p)`
  - …
- More detailed information about these rules can be found in[6].



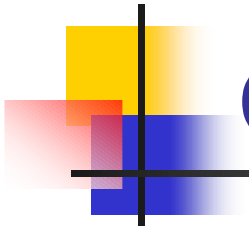
Canonical Transformation

DEMO



Causality assignment

- DAEs are represented in implicit form. The various unknowns cannot be solved directly by a simulator
- Need a mapping between equations and unknowns which determines what variable is solved by which equation
- DAEs will then be rewritten in explicit form



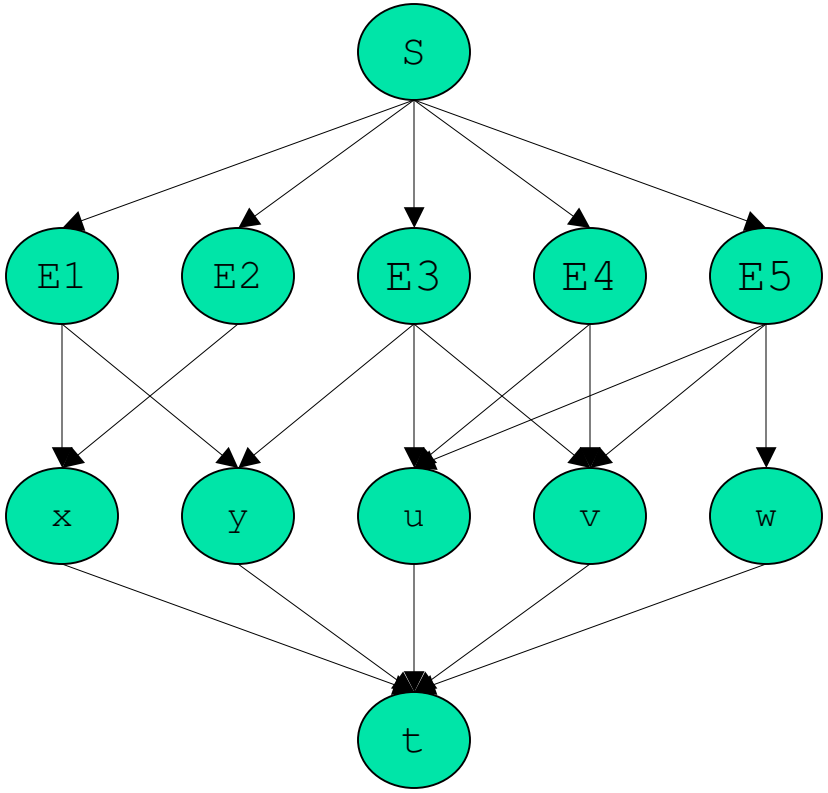
Causality assignment

- Dinic's algorithm for solving the maximum flow problem applies to this problem as well
- Both equations and variables are turned into nodes and their dependencies are turned into edges in a bipartite graph
- By adding a source node `s`, and a sink node `t` to the bipartite graph, and maximizing the flow from the source to the sink node, the causality assignment is carried out.

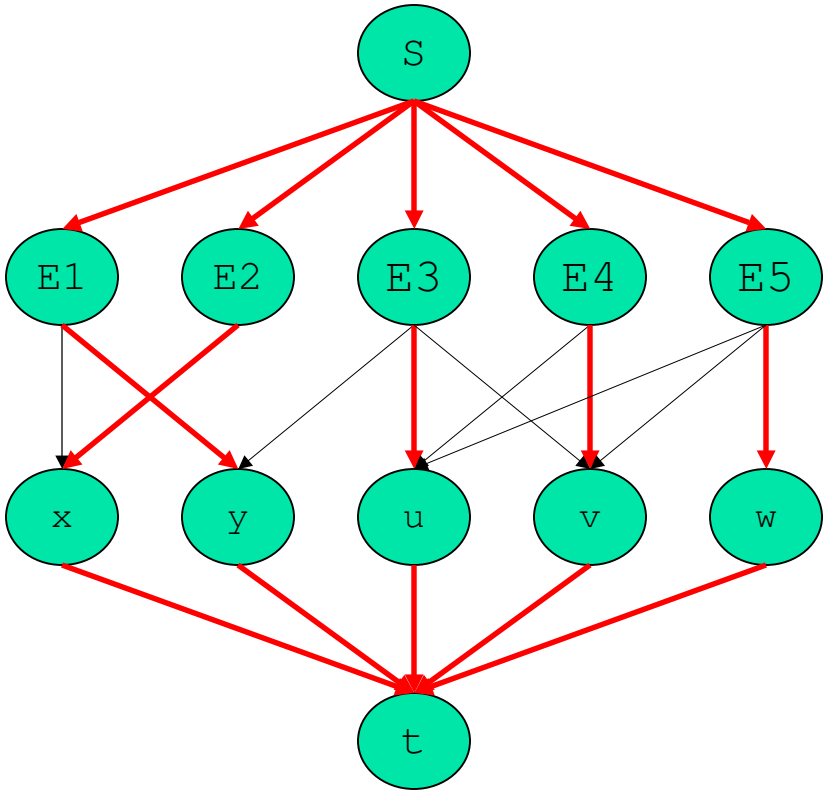
Causality assignment

- Example:

```
E1: x + y =1
E2: x =2
E3: u * v * y =4
E4: u – v =3
E5: w + v / u =1
```



Causality assignment



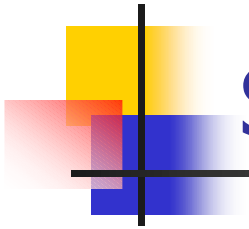
y by E1: x + y =1

x by E2: x =2

u by E3: u * v * y =4

v by E4: u − v =3

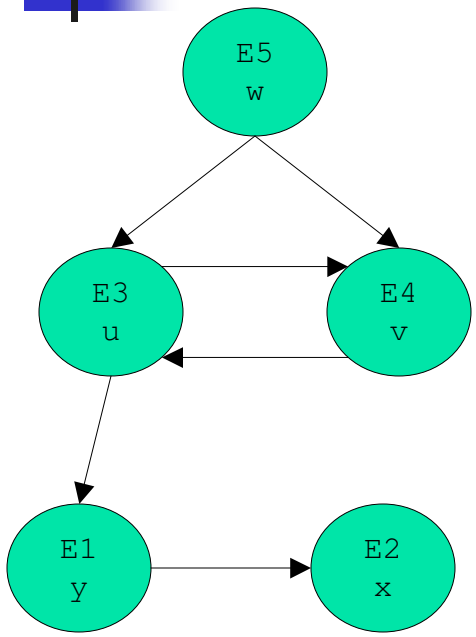
w by E5: w + v / u =1



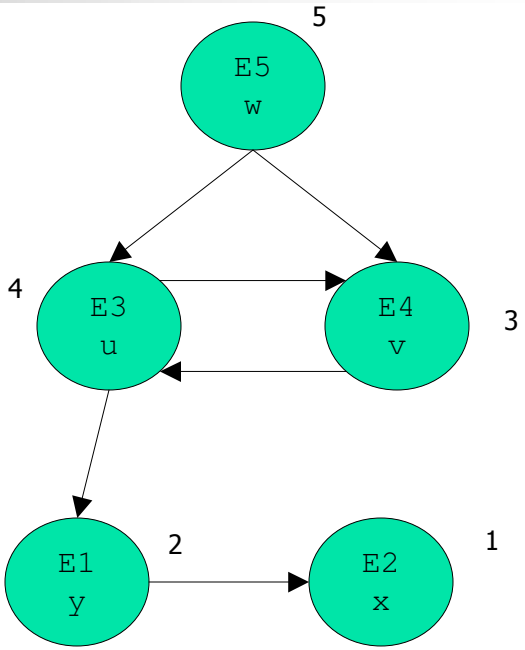
Sort equations

- The causality assignment gives pairing between equations and variables, but the equations are not yet in a correct computation order
- Equations must be sorted in the reversed order of their dependencies, i.e. if to calculate a variable is necessary to know the value of another, then the other has to be calculated first
- Based on the graph of computation dependency, this can be achieved by topological sort

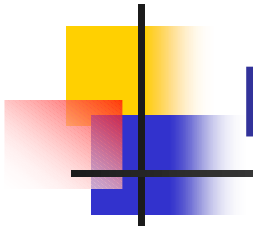
Sort equations



Topological Sort
(post-order numbering)

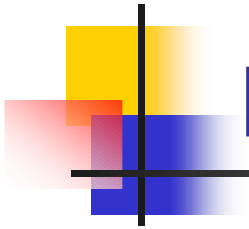



```
y by E1: x + y =1
x by E2: x =2
v by E4: u - v =3
u by E3: u * v * y =4
w by E5: w + v / u =1
```



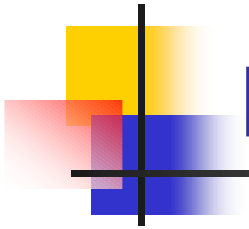
Detect algebraic loops

- We can observe that with the sorted equations, variables cannot be calculated correctly since variables `u` and `v` are mutually dependent
- These variables form an algebraic loop
- They need to be solved simultaneously, either with symbolic method or numeric method
- Therefore, in addition to sorting equations, detecting algebraic loops is also required



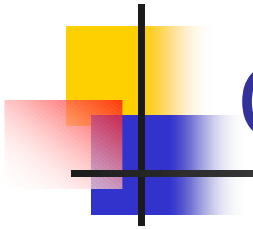
Detect algebraic loops

- Detecting algebraic loops (finding dependency cycles) is also known as locating *strongly connected components* in a graph
- A strongly connected component is a set of nodes in a graph whereby each node is reachable from each other node in the strongly connected component.
- Based on the topological sort result, the problem can be solved by producing a list of strong components
- If a node is not in a cycle, it will be in a strong component with only itself as a member.



Detect algebraic loops

DEMO

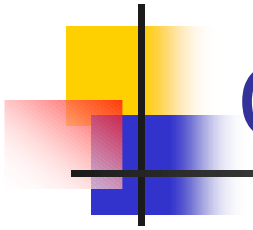


ODE

- Example

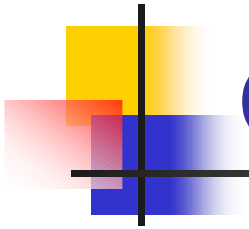
```
E1: y = sin(time)
E2: der(x) = y + z
E3: der(y) = z + x
```

- Alternatives in choosing causality:
  - All integral causality: `der(x), der(y), z`
  - some derivative causality: `x, der(y), z` or `der(x), y, z`
  - All derivative causality: `x, y, z`



ODE

- Prior option (default): all integral causality.
  - This scheme will give more accurate simulation result
  - Problem: causality assignment might fail
- Solution: more derivative causality step by step
  - Try all possible combination of derivative causality until a valid causality assignment is found
  - If no valid causality assignment is found, call a DAE solver



ODE

- Generated equations:

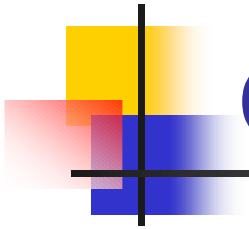
$$y = \sin(time)$$

$$der(y) = (y - y_{old}) / \triangle t$$

$$z = x - der(y)$$

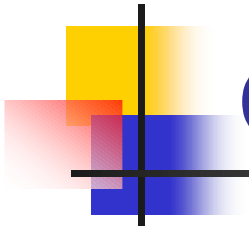
$$der(x) = y + z$$

$$x = integration(x_{old}, der(x))$$



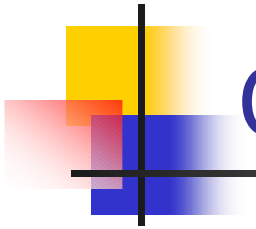
ODE

DEMO

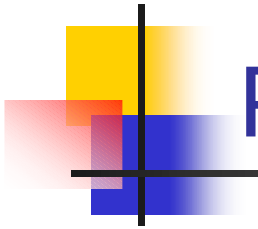


Code Generation

- In progress …
- In ideal cases, a Modelica model is finally transformed to a set of causal, sorted algebraic equations which may contain loops. Symbolic or numerical solvers are needed to solve algebraic loops, either in compile time or in runtime.
- If causality assignment fails, a DAE solver will be called in the simulation back end.



Questions?



References

[1] Modelica Association. *Overview article of Modelica.* Available at: http://www.modelica.org/

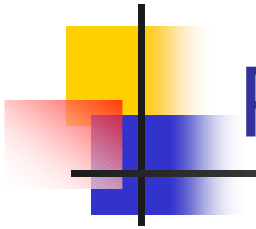
[2] Modelica Association. *Modelica Tutorial, version 1.4.* Available at: http://www.modelica.org/documents.shtml

[3] Modelica Association. *Modelica Specification, version 2.0*.

[4] EA International. *EcosimPro Mathematical Algorithms*. Dec. 1999.

[5] Michael Tiller. *Introduction to Physical Modeling with Modelica*. 2001

[6] Hans Vangheluwe, B. Sridharan, Indrani A.V. *An algorithm to implement a canonical representation of algebraic expressions in AToM³ .* Apr. 2003.



References

[7] F.E. Cellier, H. Elmqvist. *Automated Formula Manipulation Supports Object-Oriented Continuous-System Modeling*. Sep. 1993.

[8] H. Elmqvist, Martin Otter, F.E. Cellie. Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems. Jun. 1995.

[9] Kron, G. *Diakoptics – The Piecewise solution of Large-scale Systems*. 1962.