Tutorial on SysML, Modelica, Eclipse and ModelicaML

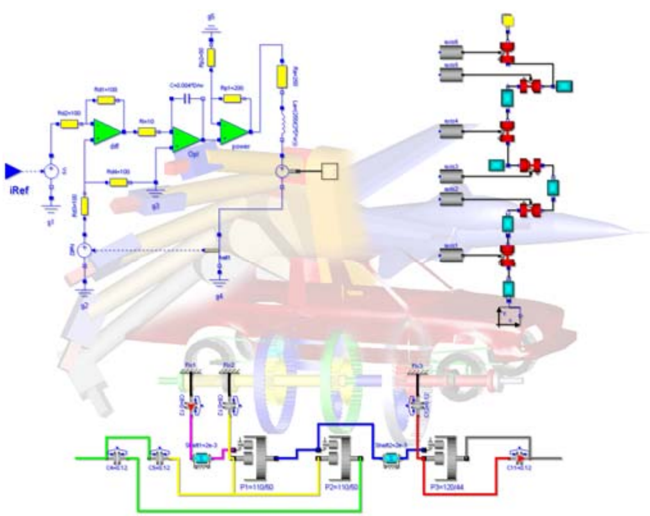
Adrian Pop

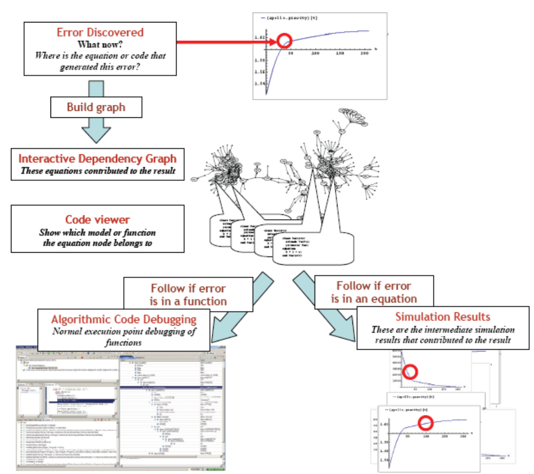
Open Source Modelica Consortium
Programming Environment Laboratory
Linköping University

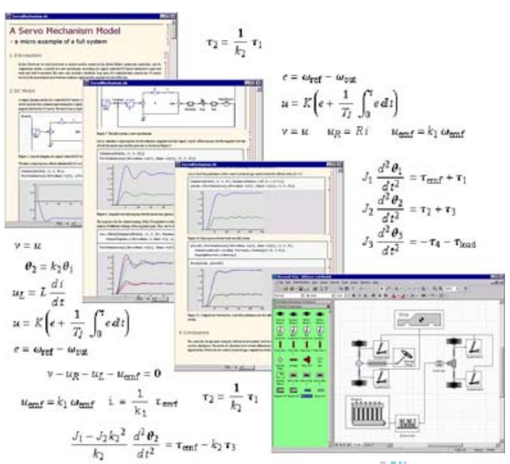
Erik Hertzog

Saab Aerosystems

ModProd'2009 2009-02-03







Outline

- Systems Engineering
  - Introduction and Background
- SysML
  - a UML profile for systems engineering
- Modelica
  - modeling and simulation of physical systems
  - equation-based object-oriented language
- ModelicaML
  - Modelica vs. SysML
  - a UML profile for Modelica based on SysML
- Eclipse
  - Integrated Environments for Modelica
  - Short Demo of ModelicaML Eclipse Environment

Systems Engineering

Introduction and Background

System engineers


Requirements owner


System designer


System analyst

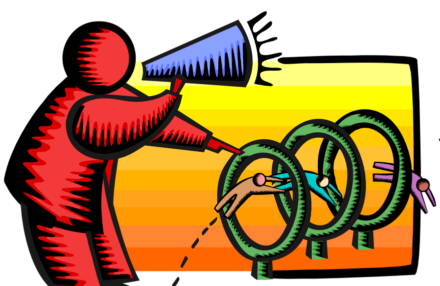

Verification & Validation

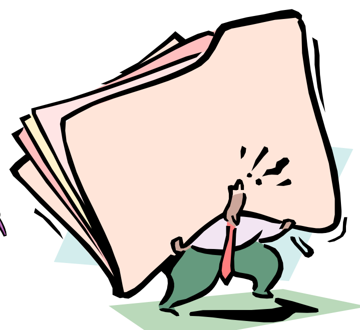

Logistics & Operation


Glue engineer


Customer interface


Co-ordinator


Technical manager

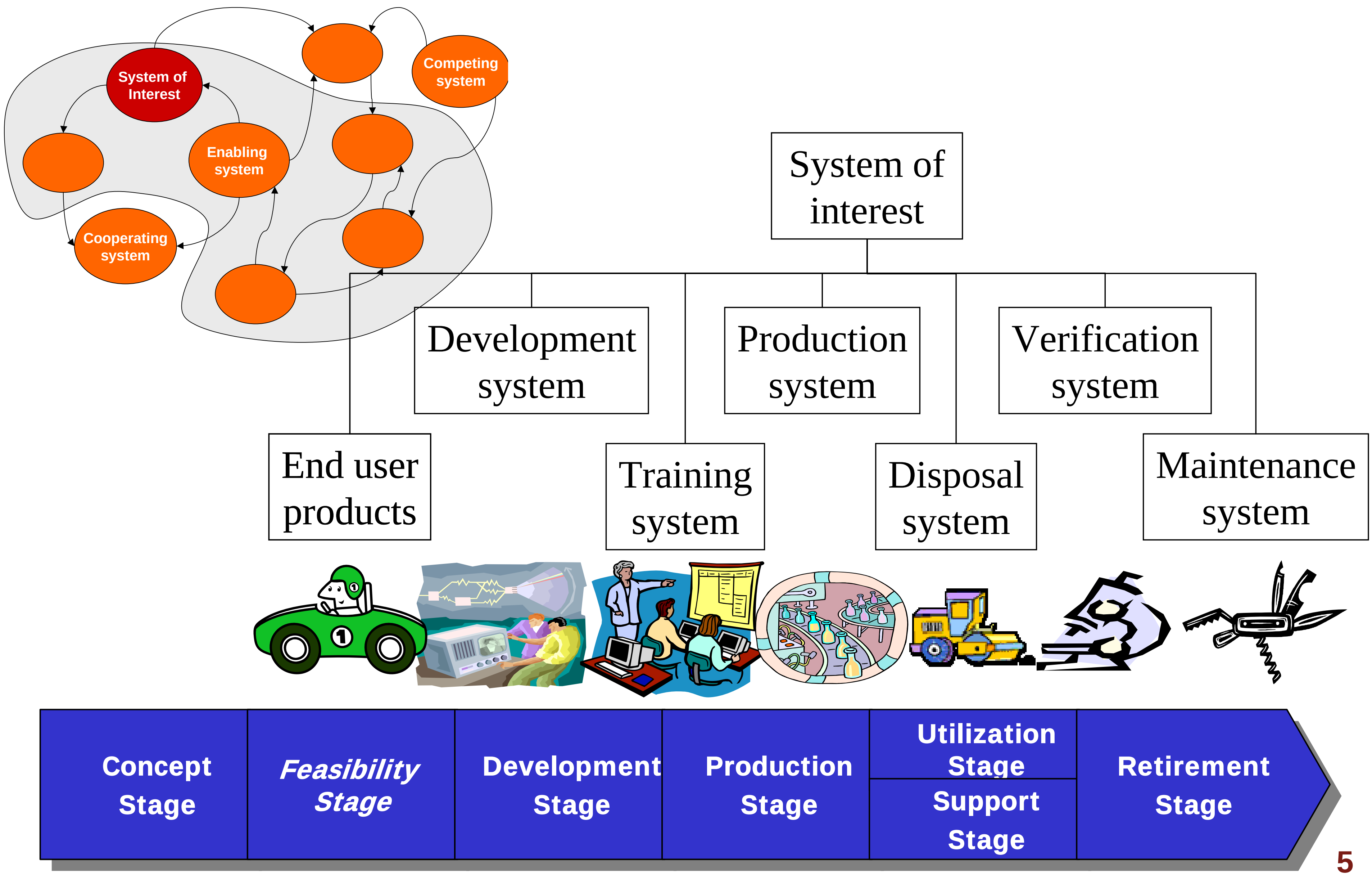

Information manager


Process engineer

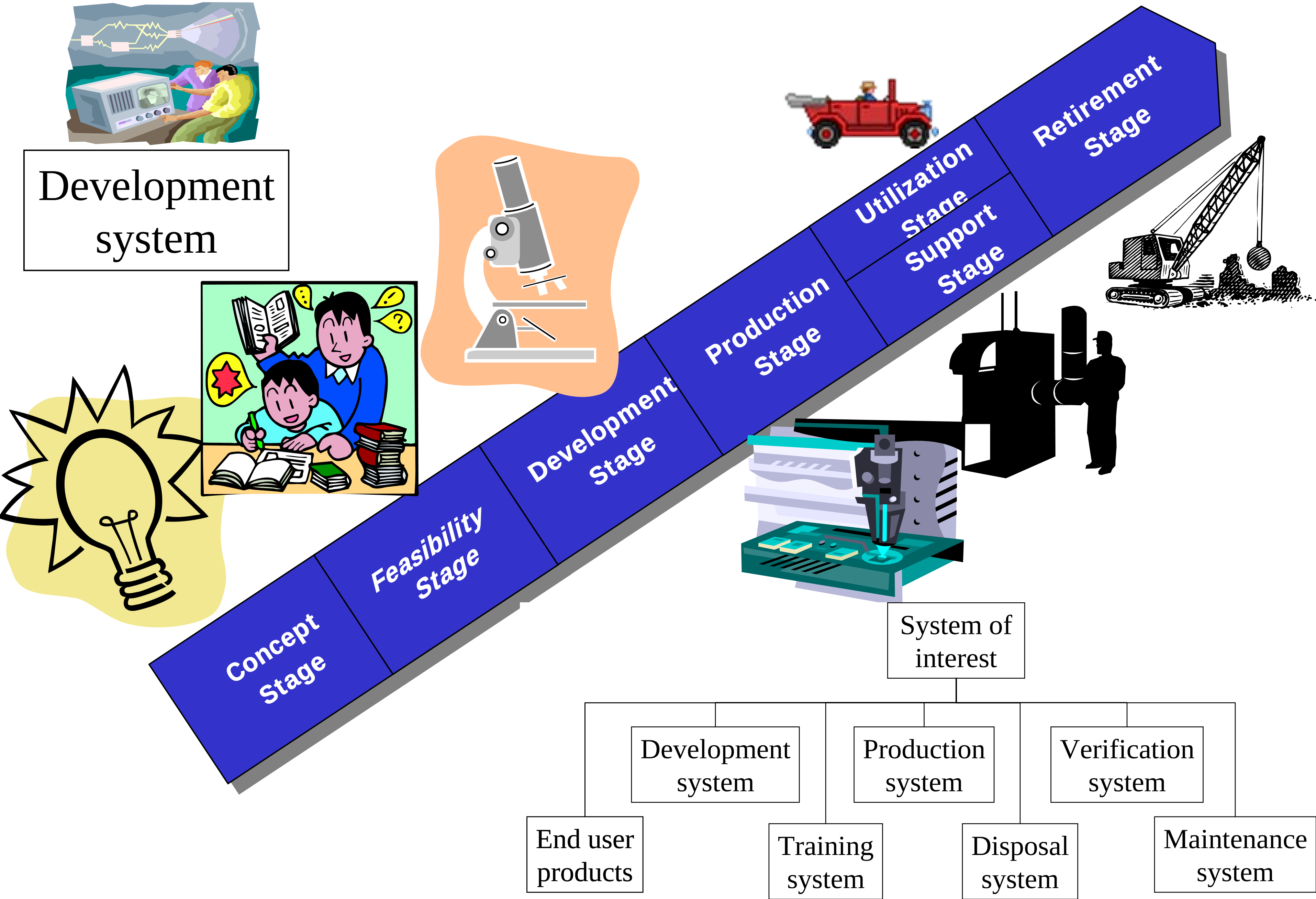

Classified ads engineer

S. Sheard,”12 Systems Engineering roles”, Proc of INCOSE 1996

Systems Engineering



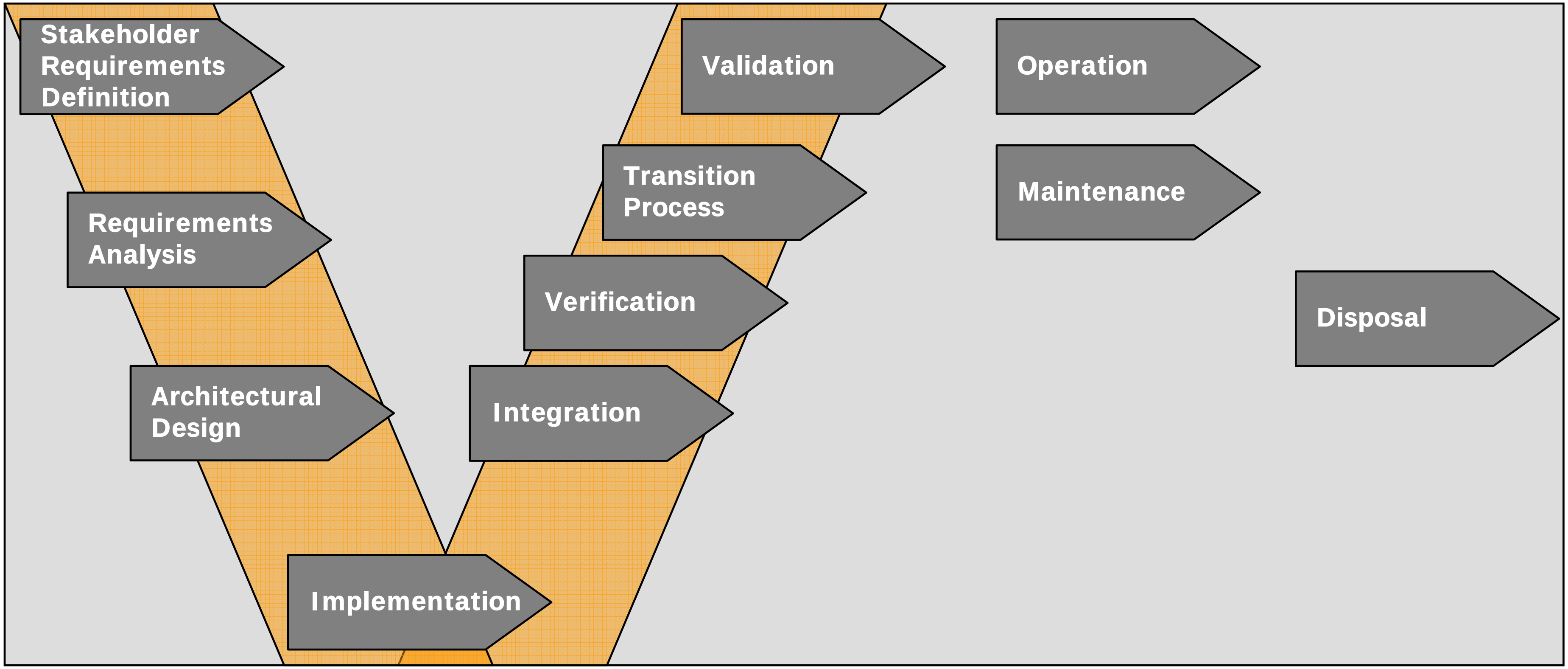
Example


Development system
Concept Stage
Feasibility Stage
Development Stage
Production Stage
Utilization Stage
Support Stage
Retirement Stage
System of interest
Development system
Production system
Verification system
End user products
Training system
Disposal system
Maintenance system

Observation

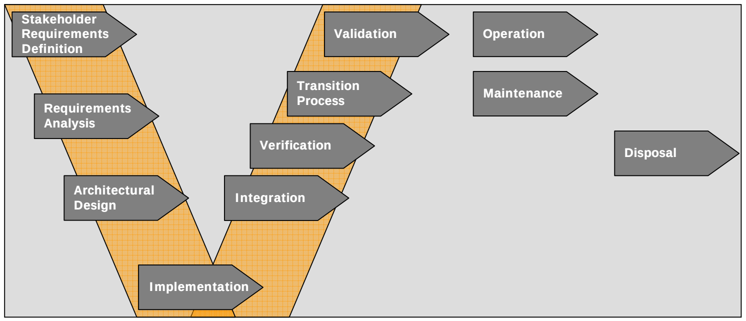
- For a domain engineer complexity can be bounded to a single engineering domain

- For a systems engineer complexity lies in the interactions between multiple systems/domains
  - Multiple technologies
  - Inter-technology interference
  - Multiple components
  - Complex interfaces

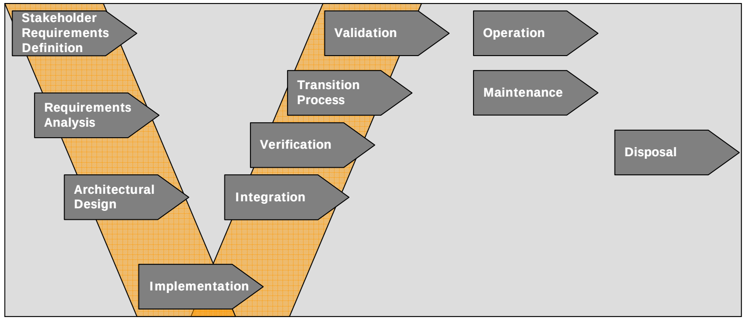

Stakeholder Requirements Definition
Requirements Analysis
Architectural Design
Implementation
Integration
Verification
Transition Process
Validation
Operation
Maintenance
Disposal

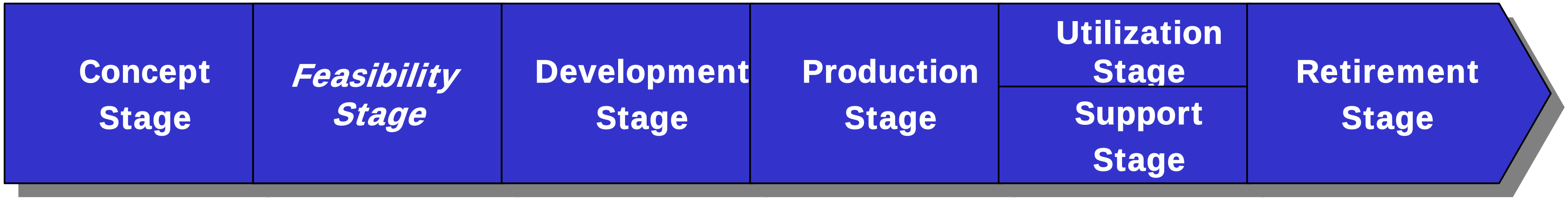
Process over lifecycle

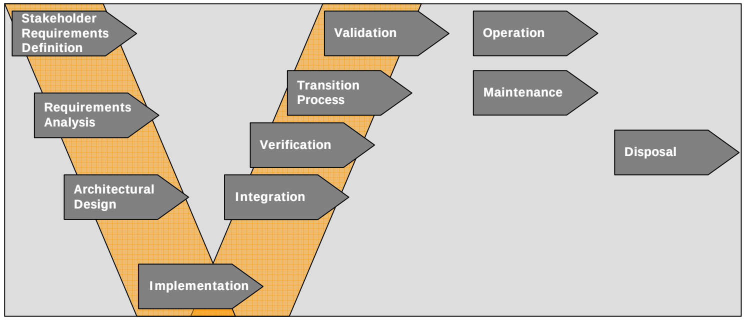
Concept system



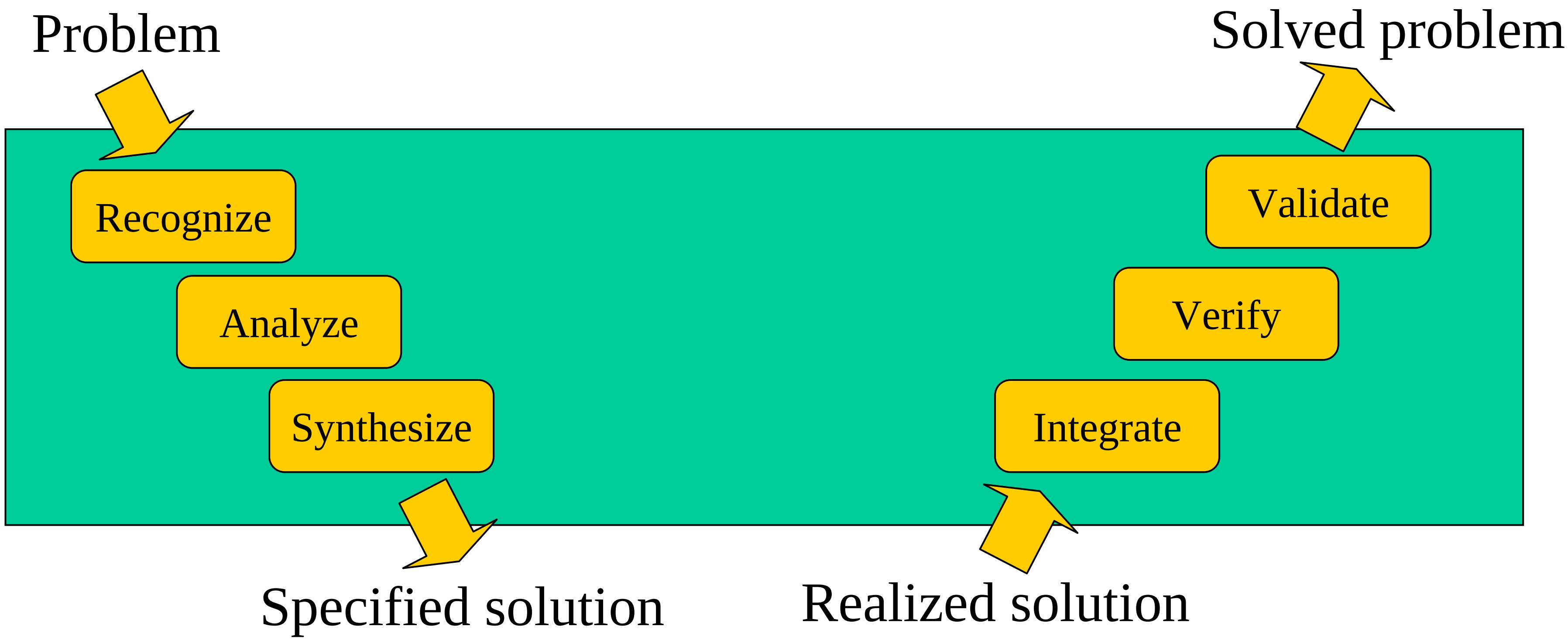
Development system



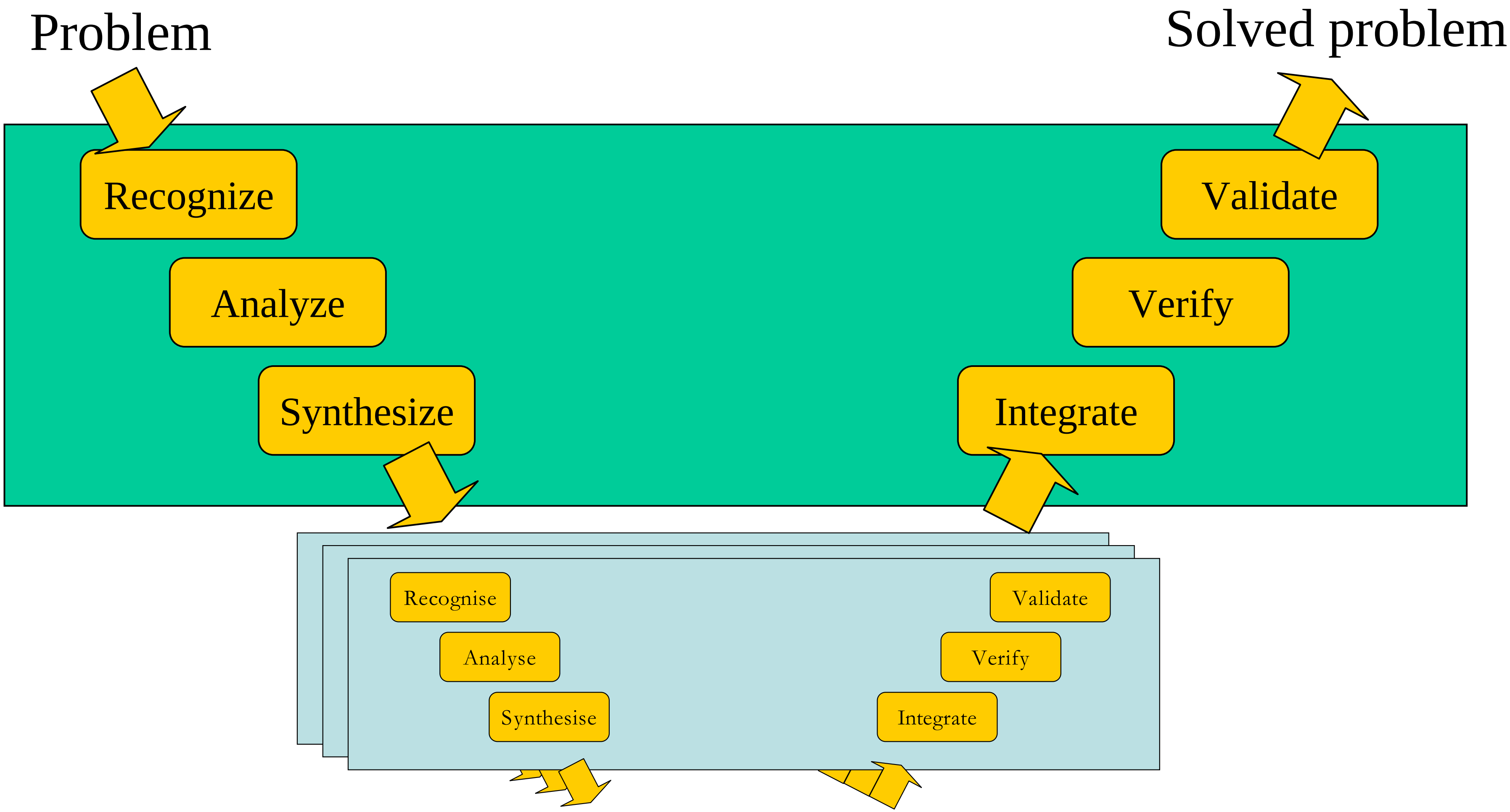




Feasibility system


Problem
Solved problem
Recognize
Analyze
Synthesize
Validate
Verify
Integrate
Specified solution
Realized solution

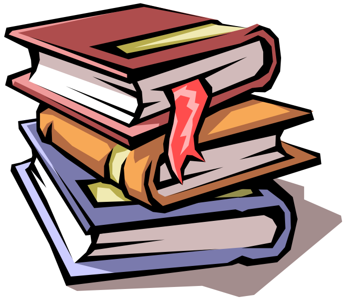
Process to manage complexity

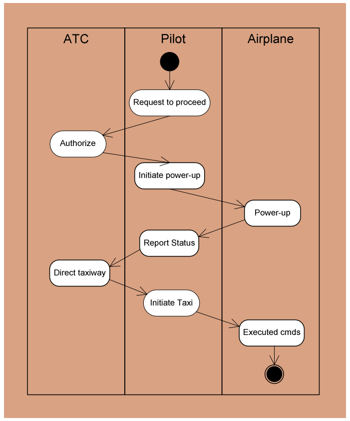


Challenges in the life of a Systems Engineer

- Specification ambiguity
- Specification coherency
- Information availability
- Traceability
- Verification and validation

- SE deliverables
  - Specifications
  - System design
  - Analysis and trade-off
  - Test plans
  - etc.





- Evolution
  - Document-based > Model-based

Why model-based development?

- Advantages
  - Improved communication
  - More rigorous and precise, less ambiguous, less defects
  - More complete representation
  - Less maintenance cost
  - Easier to preserve the competence

- Disadvantages
  - May stand for a high learning curve
    - due to new methods and notations (such as SysML)

What is UML?

- *“The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.”* (OMG UMG 2.0 Superstructure Specification)

- Object-oriented, visual modeling language
  - = notation (language, representation) + semantics (meaning)
  - UML is a language, not a method

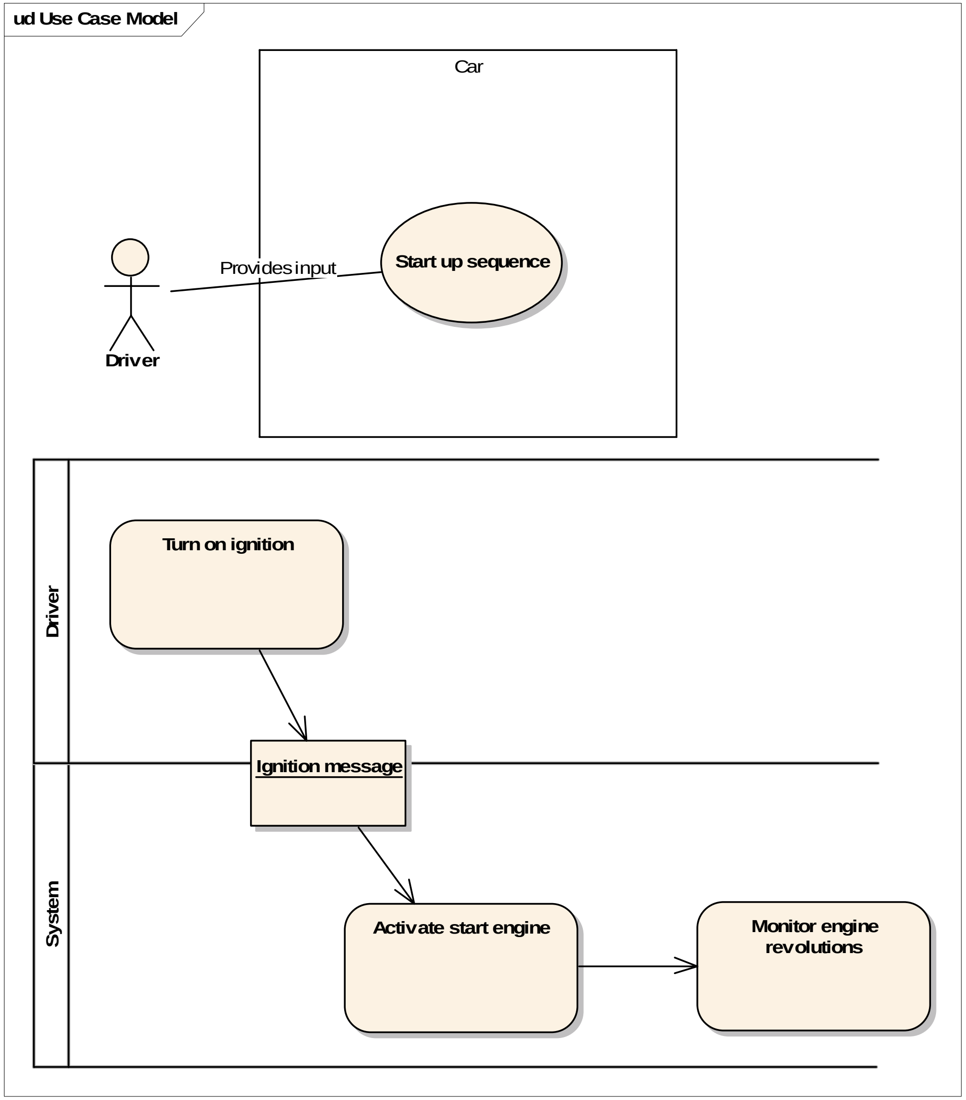
- De-facto standard
  - Software Engineering: Applications and components
  - Human activity systems: Industry sector, enterprises, business processes

- Brief history (most important versions)
  - **1.0** 1997, **1.4** 2001, **2.0** 2003

UML diagram concept

- UML is defined around a number of diagram types
  - Each with a specific purpose and a specific symbol set
  - Each symbol has a well defined meaning (semantics)

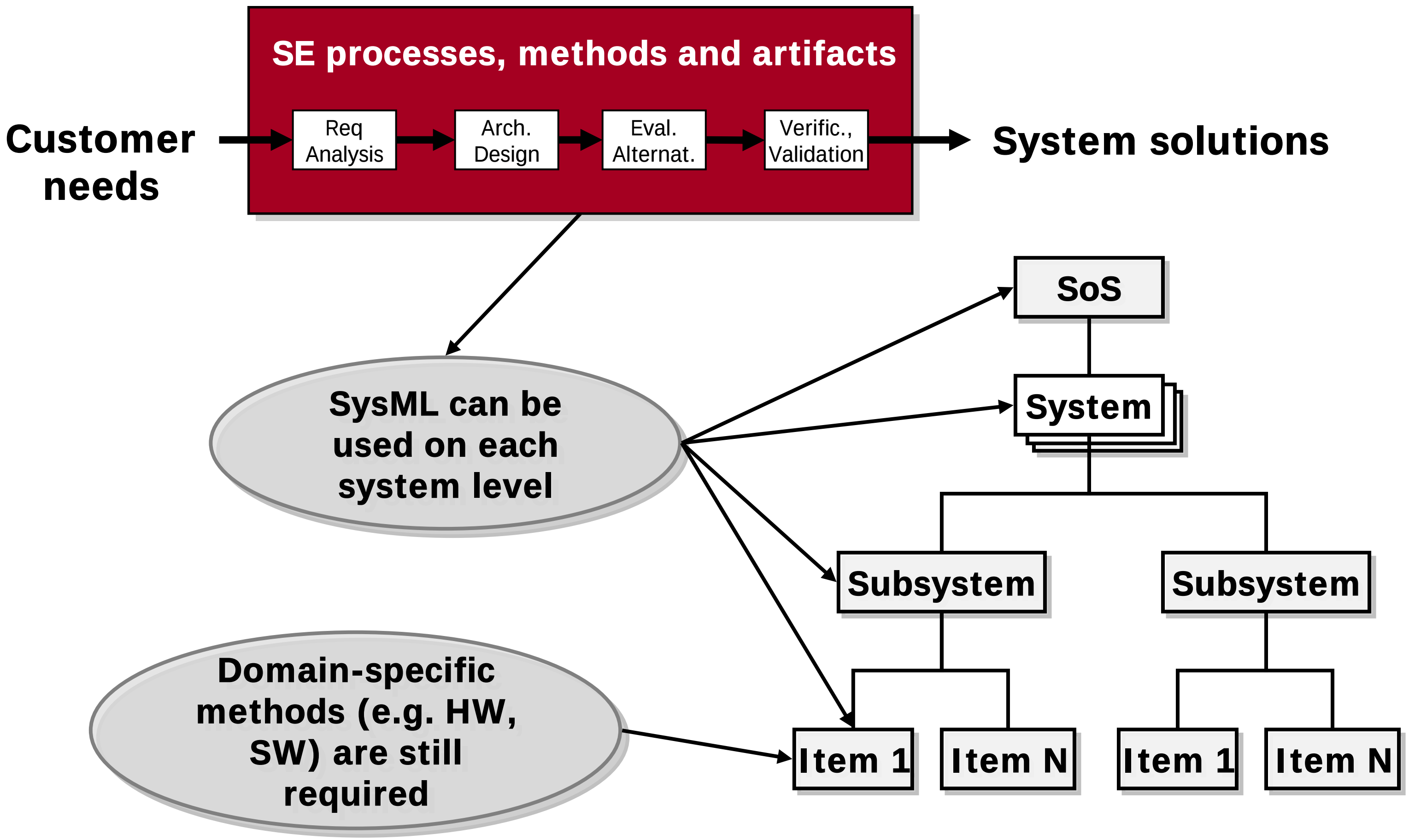
- Diagram elements are not tied to a specific diagram type
  - Allows for smart combinations of views on a system within a single diagram



UML Mechanisms for Extensions

- The language UML is defined using a language/toolbox named MOF
  - MOF = Meta Object Facility
- UML is defined to allow extensions to the semantics of language elements
  - Stereotypes: Modification to original element semantics, potentially with an associated attribute set (as defined by tagged values)
  - Tagged values: A name-value combination that is used to define properties of an element
- The stereotype concept is used extensively within SysML to define the language elements of interest to Systems Engineers
  - Any number of stereotypes can be applied to a base UML objects
- Extensions for a specific purpose can be summarized in a “UML profile”

SysML Scope



Customer needs
SE processes, methods and artifacts
Req Analysis
Arch. Design
Eval. Alternat.
Verific., Validation
System solutions
SysML can be used on each system level
SoS
System
Subsystem
Subsystem
Domain-specific methods (e.g. HW, SW) are still required
Item 1
Item N
Item 1
Item N

SysML Scope

- Adheres to the Systems Engineering tradition to model a system in terms of
  - Requirements
  - Functionality
  - Architecture
  - Verification

SysML

A visual modeling language
for Systems Engineering

System Modeling Language (SysML™)

- Designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems
- Effective in specifying requirements, structure, behavior, allocations, and constraints on system properties to support engineering analysis
- Intended to support multiple processes and methods such as structured, object-oriented, etc.

What is SysML?

- A graphical modeling language for Systems Engineering
  - a UML Profile that represents a subset of UML 2 with extensions
- Supports the specification, analysis, design, verification, and validation of systems that include hardware, software, data, personnel, procedures, and facilities.
- Supports model and data interchange via XMI and the evolving AP233 standard.

What is SysML?

Is a visual modeling language that provides

- Semantics = meaning
- Notation = representation of meaning

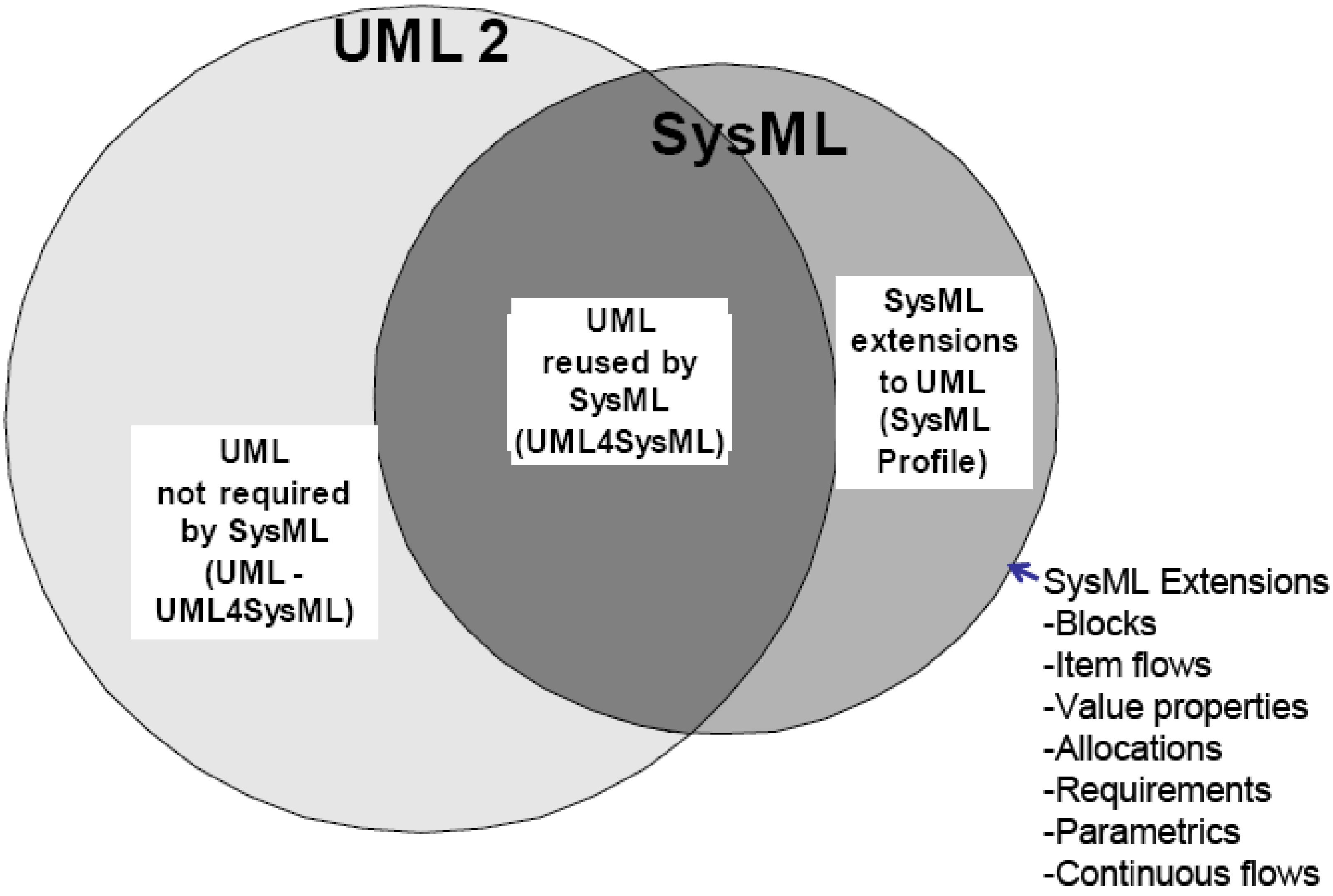
Is **not** a methodology or a tool

- SysML is methodology and tool independent

SysML Vendors

- Commercial
  - Artisan (Studio)
  - EmbeddedPlus (SysML Toolkit)
    - 3rd party IBM vendor
  - No Magic (Magic Draw)
  - Sparx Systems (Enterprise Architect)
  - IBM / Telelogic (Tau and Rhapsody)
  - Visio SysML template
- Open Source based on Eclipse
  - TopCased and Papyrus

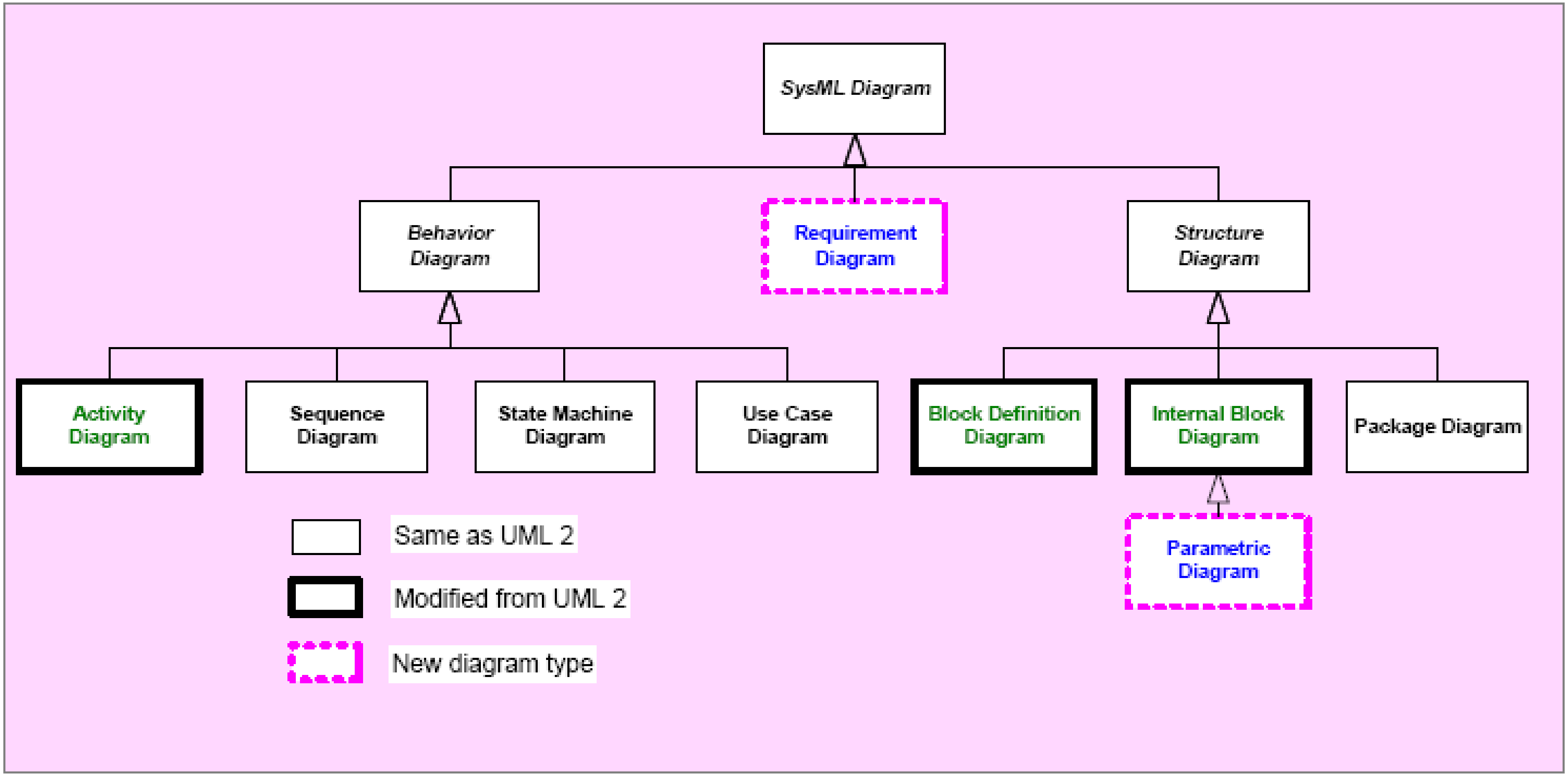
SysML vs. UML



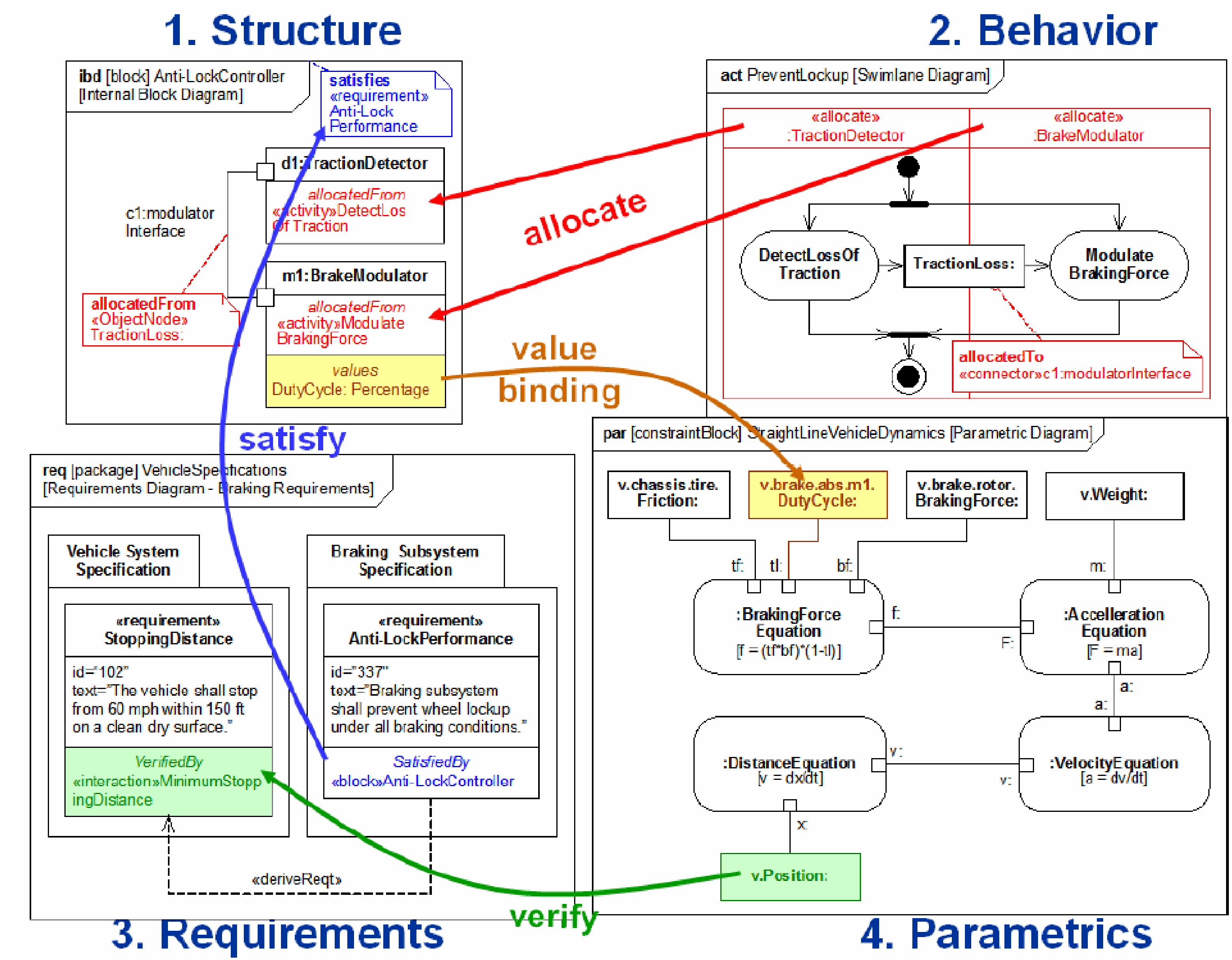
from OMG SysML tutorial

- A UML model can be sufficiently detailed for creation of products out of the model
- A SysML model is just an **abstraction** of the final system to be delivered
- Production drawings etc. will reside in other tools/environments

SysML Diagrams

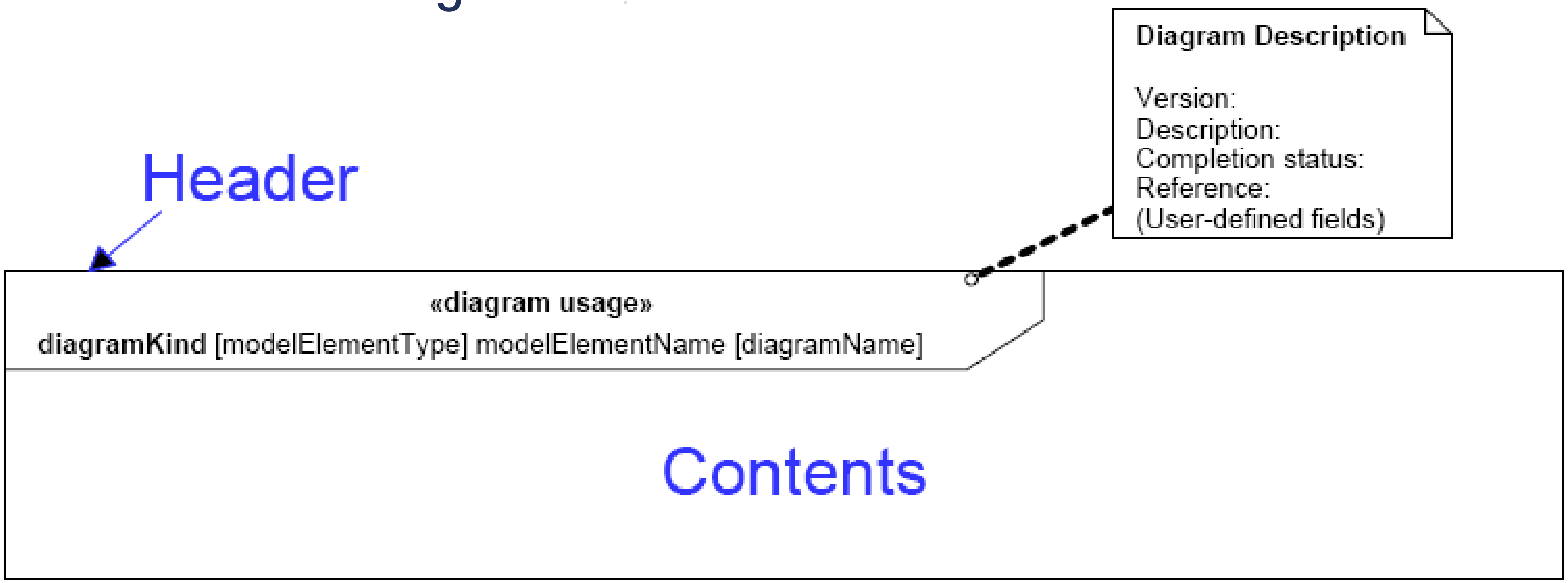


SysML pillars



A SysML Diagram
- represents a model element
- must have a Diagram Frame

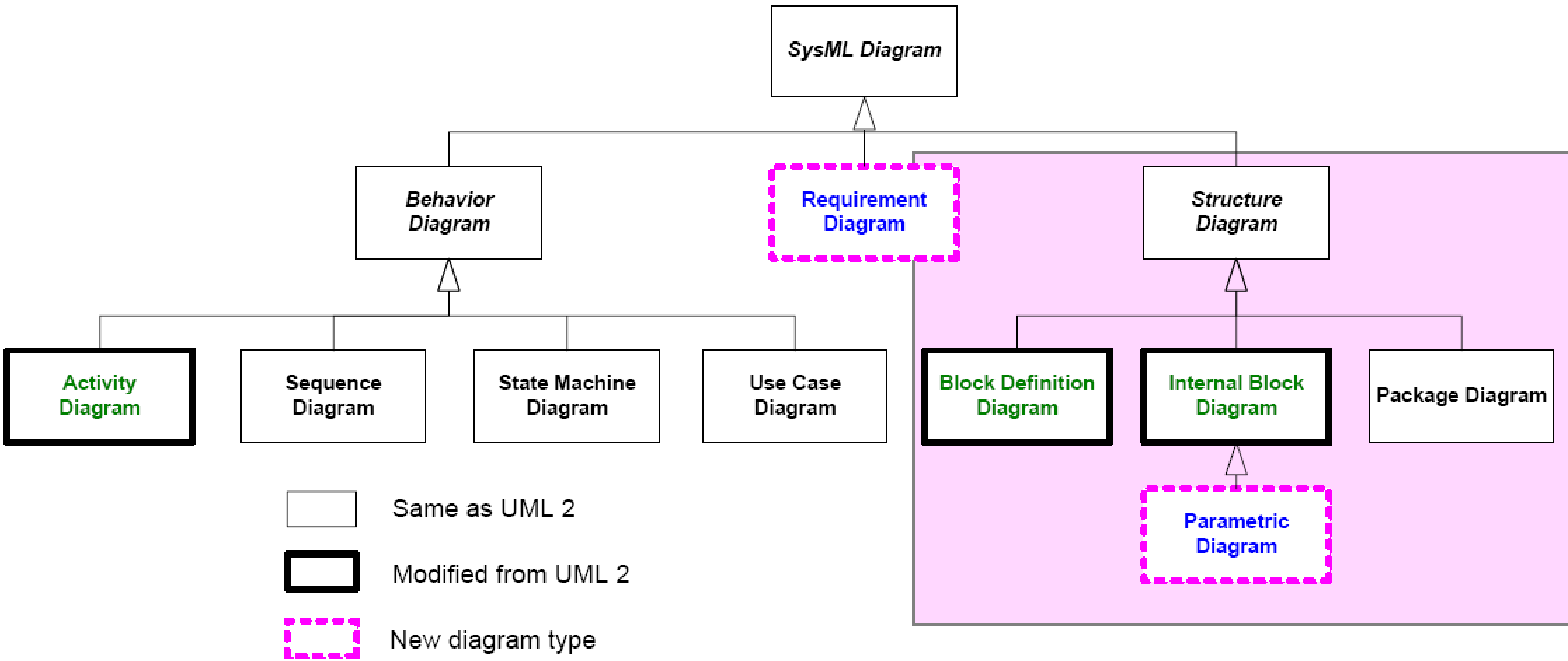
Diagram context defined in the header
- Diagram kind (act, bdd, ibd, sd, etc.)
- Model element type (package, block, activity, etc.)
- Model element name
- User defined diagram name or view name



SysML

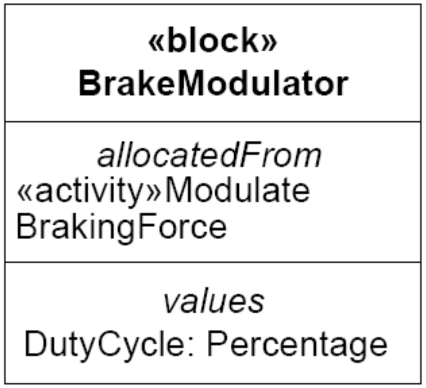
Specifying System Architecture

- Used to specify System Architecture



SysML Blocks

- «block» stereotype provides a common root for user-defined or domain-specific hierarchies of system component types
  - Hardware
  - Software
  - Data
  - Procedure
  - Facility
  - Person



- Blocks provide the backbone of the “system hierarchy” or “system of systems” architecture which drives much of modern systems engineering
  - Blocks do not represent the parts view/product structure of a product
    - Rather it is an abstraction of the system under specification

- Block definition diagram
  - Composition may be handled to any number of levels within a single diagram
    - Using the white diamond aggregation relationship
  - Based on the UML class diagram

- Internal block diagram
  - Composition is captured in a single level per diagram
  - Interfaces are captured explicitly

About blocks

- Based on UML Class from UML Composite Structure
  - Eliminates association classes, etc.

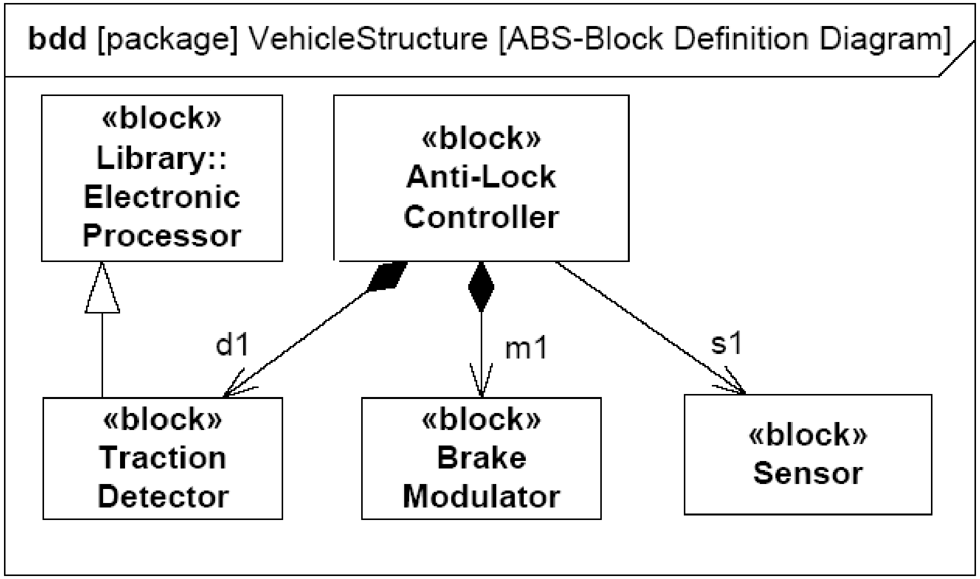
- Differentiates value properties from part properties
- Block interfaces
  - Service port – traditional SW service architecture
  - Flow port – for continuous or discrete signals

- Block definition diagram describes the relationship among blocks (e.g., composition, association, classification)

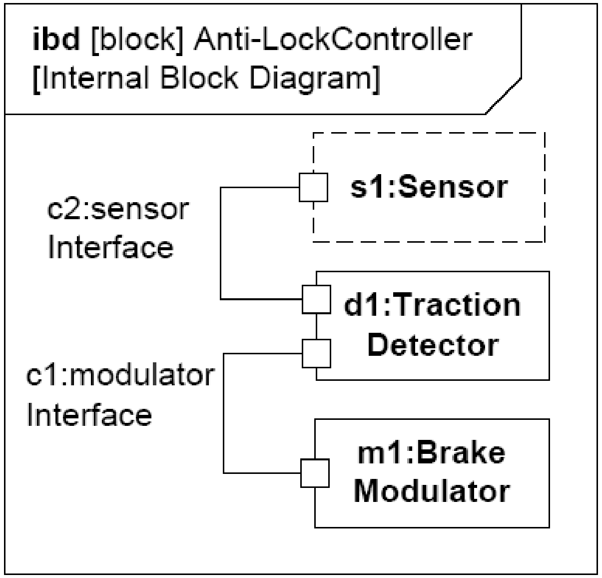
- Internal block diagram describes the internal structure of a block in terms of its properties and connectors

- Requirements and Behavior can be allocated to blocks
- Block subtypes may be created using stereotypes or through classification

Block Definition Diagram



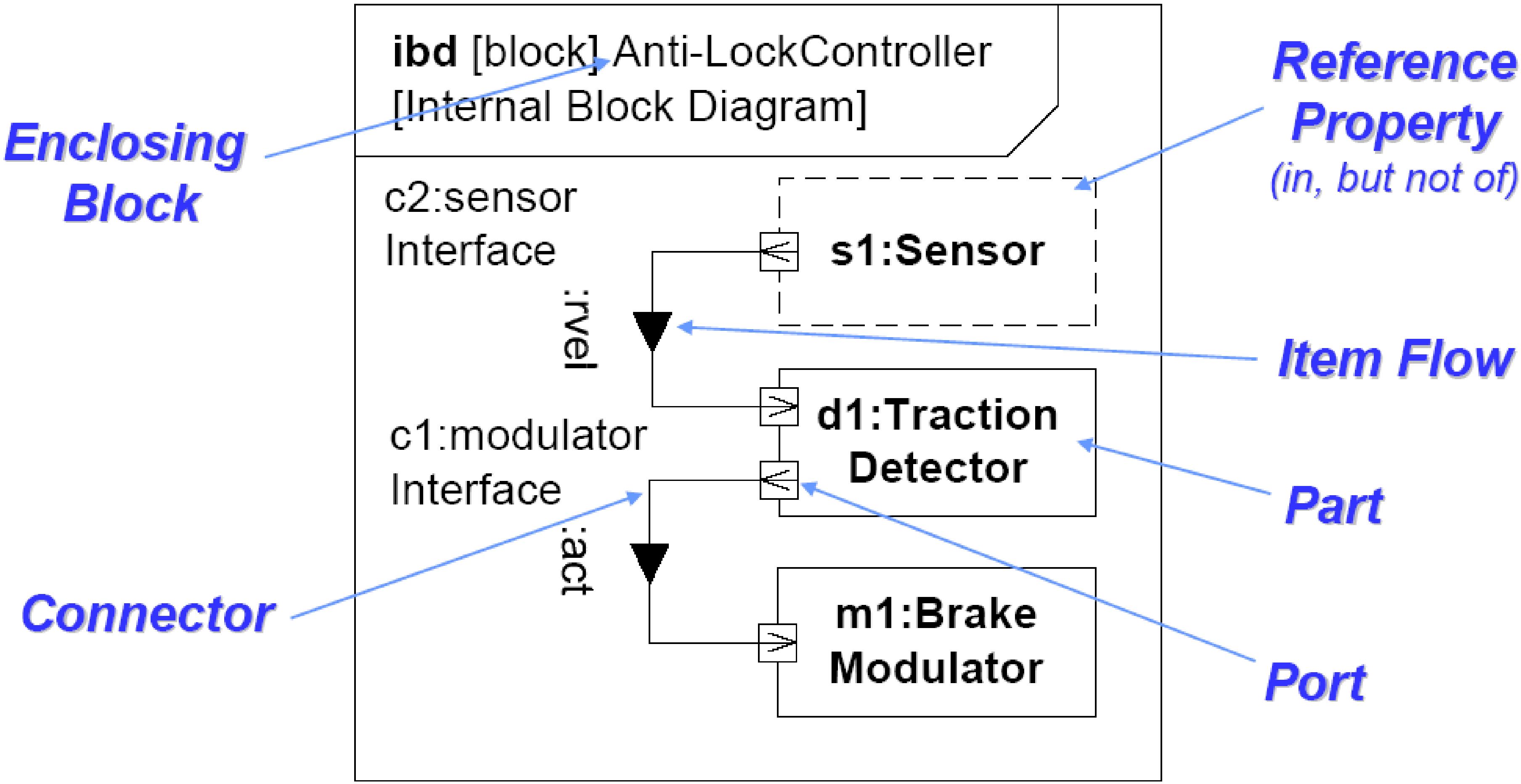
Internal Block Diagram



- Definition of "building" blocks
- Capture properties
- Can be used in multiple contexts
- Block relationships

- A "part" indicate the usage of a particular block
- Interfaces are visible

Blocks, Parts, Ports, Connectors & Flows

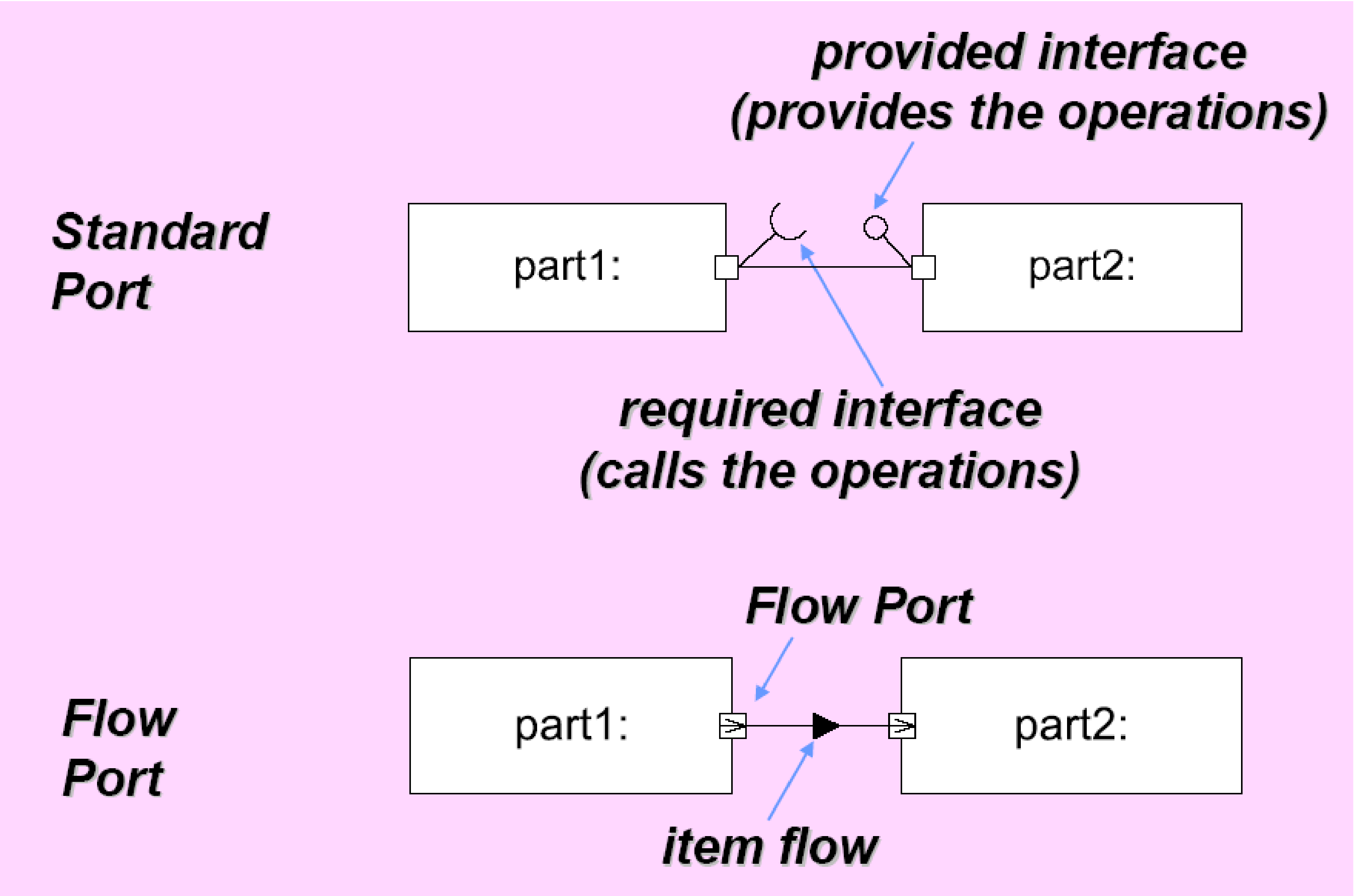


Standard (UML) port

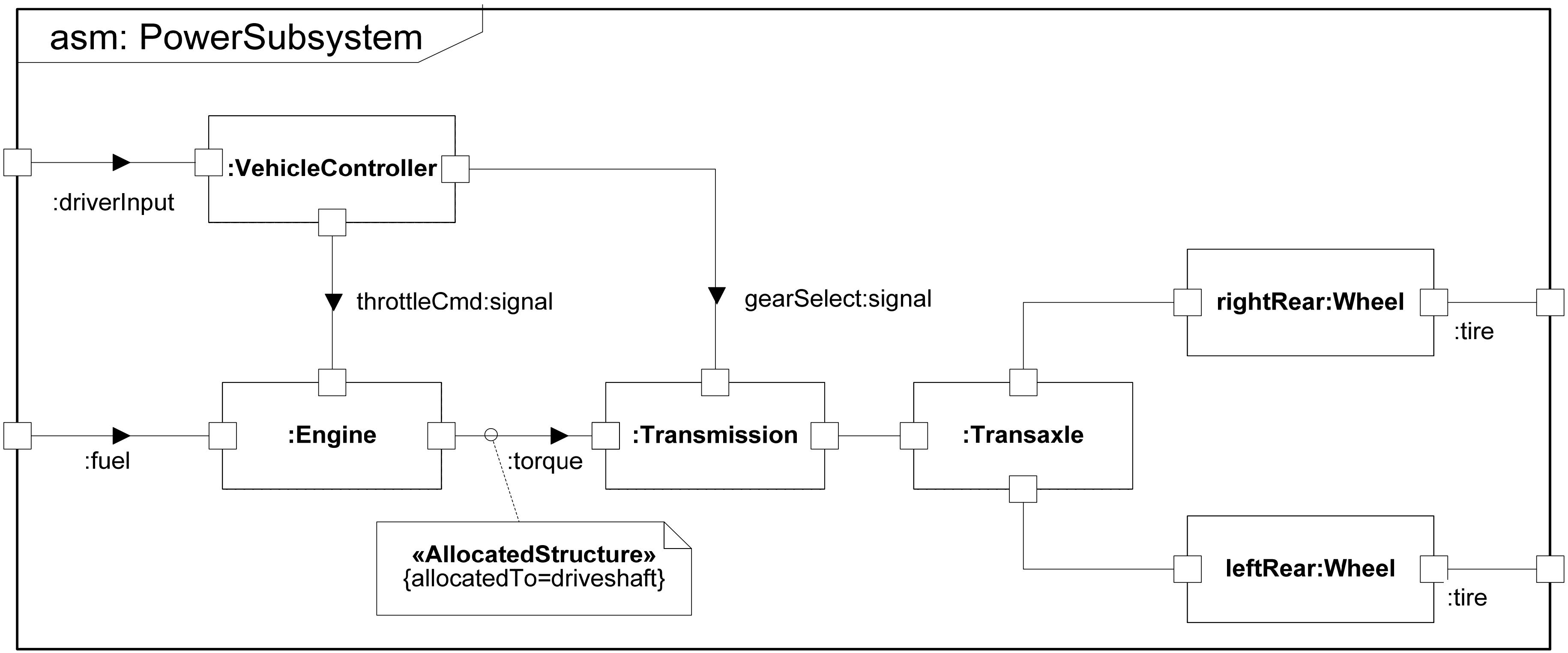
- The port indicate the existence of a service interface which external blocks may call (as in software)
- Interaction is as defined for the individual operation made available through the interface

Flow ports

- Specifies what can flow in or out of a component
- Has a specified direction and content
  - May be bi-directional


provided interface
(provides the operations)
Standard
Port
part1:
part2:
required interface
(calls the operations)
Flow Port
Flow
Port
part1:
part2:
item flow

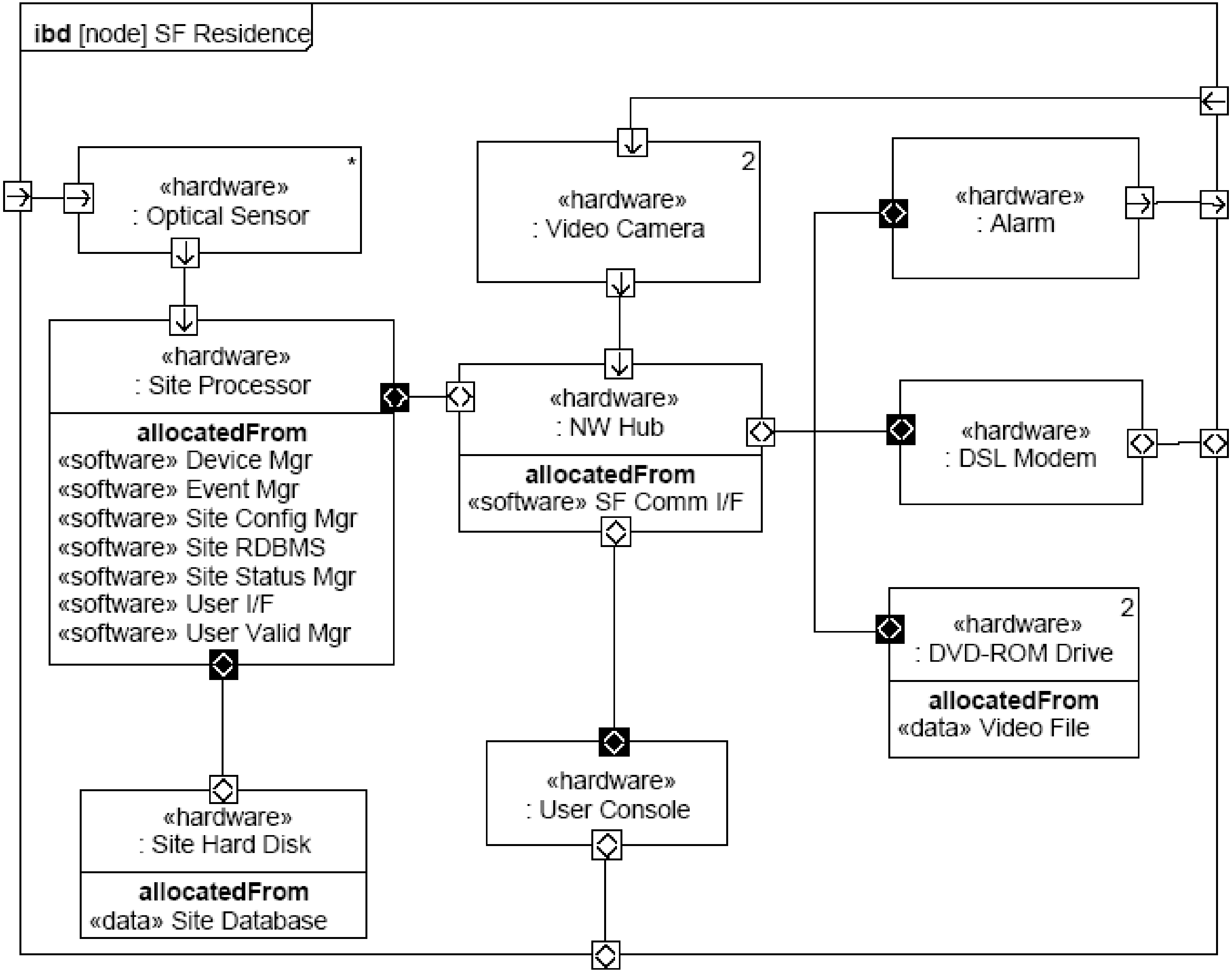
Internal Block Diagram Example



Allocation

- SysML provides 3 mechanisms for **representing the allocation** of functional or physical elements to other physical elements
  - Via Swimlanes in activity diagrams
    - Elegant
  - Via the addition of a separate compartment in the block structure
  - Via relationships directly on diagrams

Allocation example



SysML

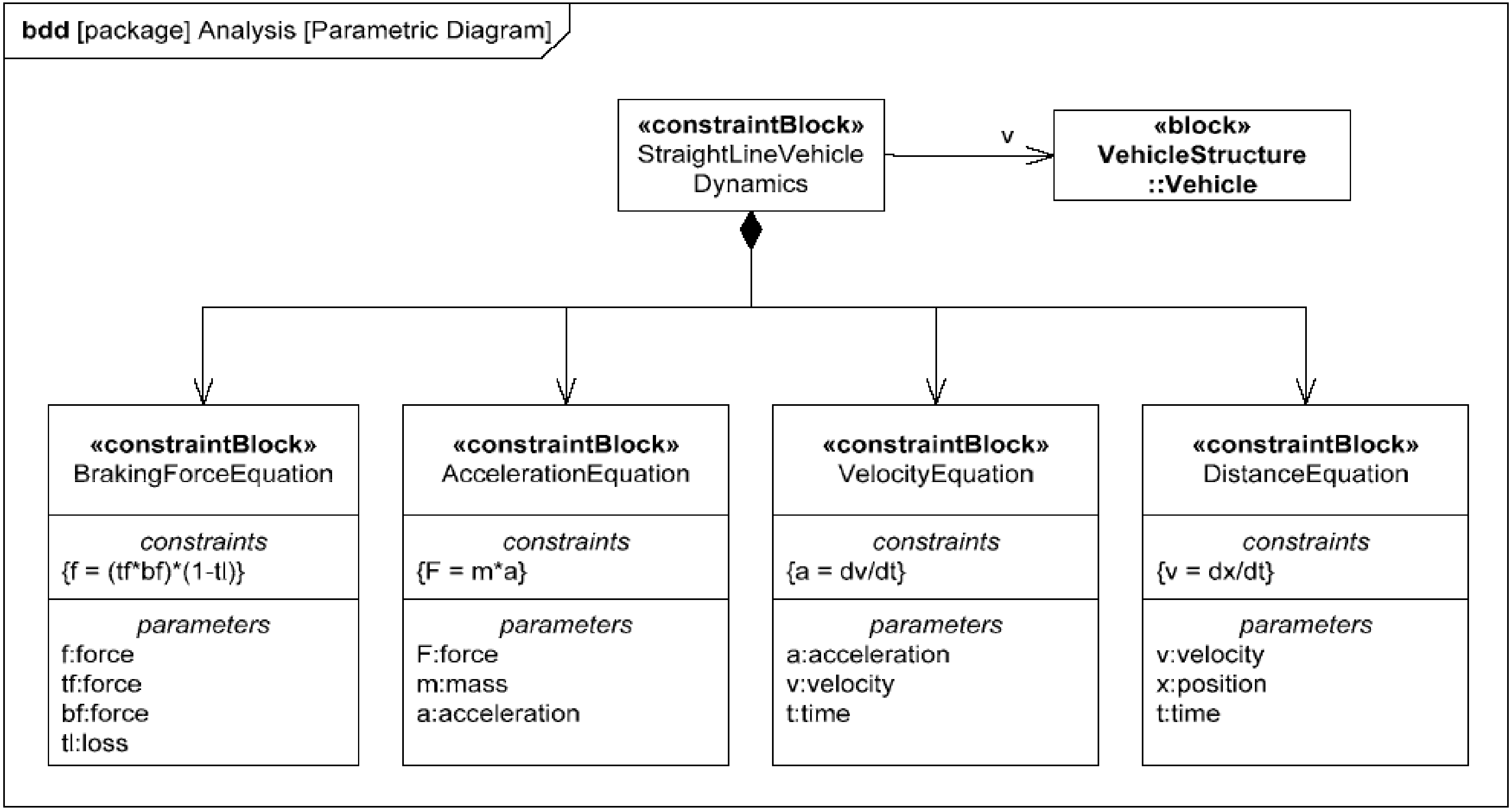
Parametric Constraints

Parametric Constraint

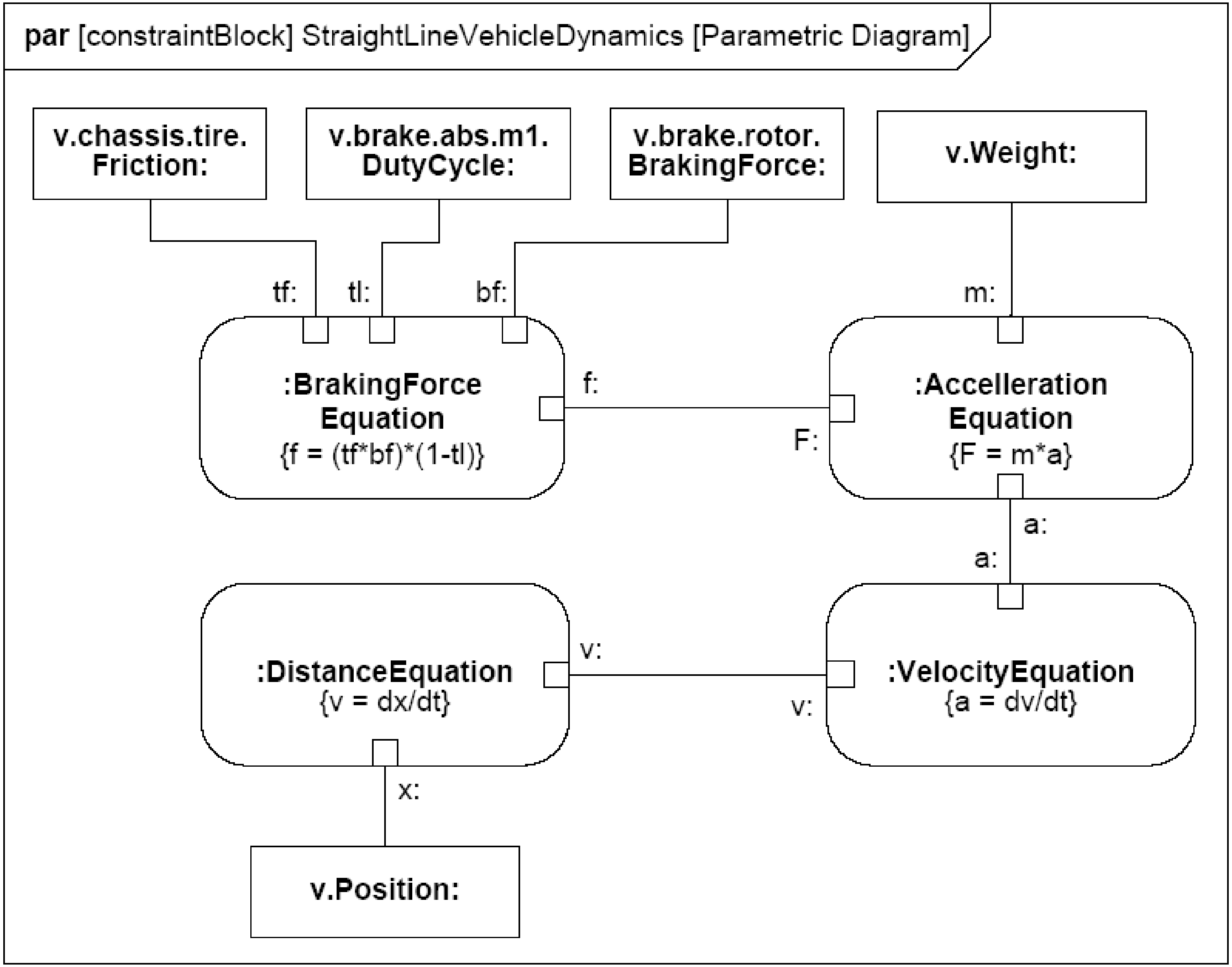
- Used to express constraints between quantifiable properties (aka non-functional characteristics) of assemblies and their decomposition
  - Reusable
  - Non-causal (i.e. declarative statement of the invariant without specifying dependent/independent variables)

- Defined as a stereotype
  - Expression: text string specifies the constraint
  - Expression language can be formal (e.g. MathML, OCL …) or informal
  - Computational engine is defined by applicable analysis tool and not by SysML

- Usage
  - Used in the context of a SysML assembly
  - Notation: parametric diagram distinguishes the parametric constraints from other parts of a containing assembly
  - Properties of parts connected to parameters of relation
  - Value binding connector declares that parameter and property are bound to the same value

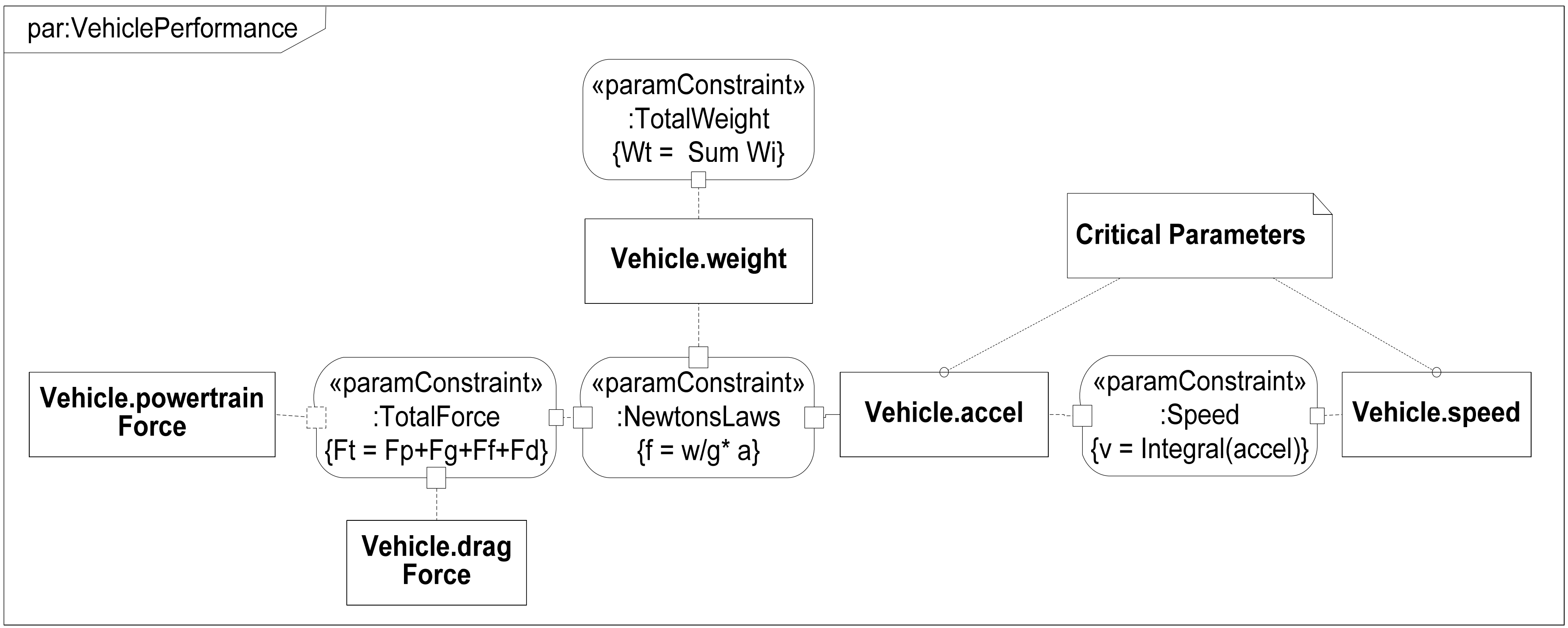

bdd [package] Analysis [Parametric Diagram]
«constraintBlock»
StraightLineVehicle
Dynamics
v
«block»
VehicleStructure
::Vehicle
«constraintBlock»
BrakingForceEquation
constraints
{f = (tf*bf)*(1-tl)}
parameters
f:force
tf:force
bf:force
tl:loss
«constraintBlock»
AccelerationEquation
constraints
{F = m*a}
parameters
F:force
m:mass
a:acceleration
«constraintBlock»
VelocityEquation
constraints
{a = dv/dt}
parameters
a:acceleration
v:velocity
t:time
«constraintBlock»
DistanceEquation
constraints
{v = dx/dt}
parameters
v:velocity
x:position
t:time

Defining variable binding



Parametric Diagram showing Vehicle Performance Par.

- Rounded rectangles are parametric constraints
- Rectangles are properties (parameters)



SysML Properties

- SysML Extension for Property, to address:
  - Quantity - Values, Units, and Dimensions
  - Probability Distribution
  - Example for a vehicle that weighs 1000 pounds with a uniform probability distribution:

Vehicle
totalWeight : VehicleMass=(value=1000.0, unit=pounds, distribution=(Uniform)(min=990.0,max=1010.0))

- New predefined data types
  - Real
  - Complex

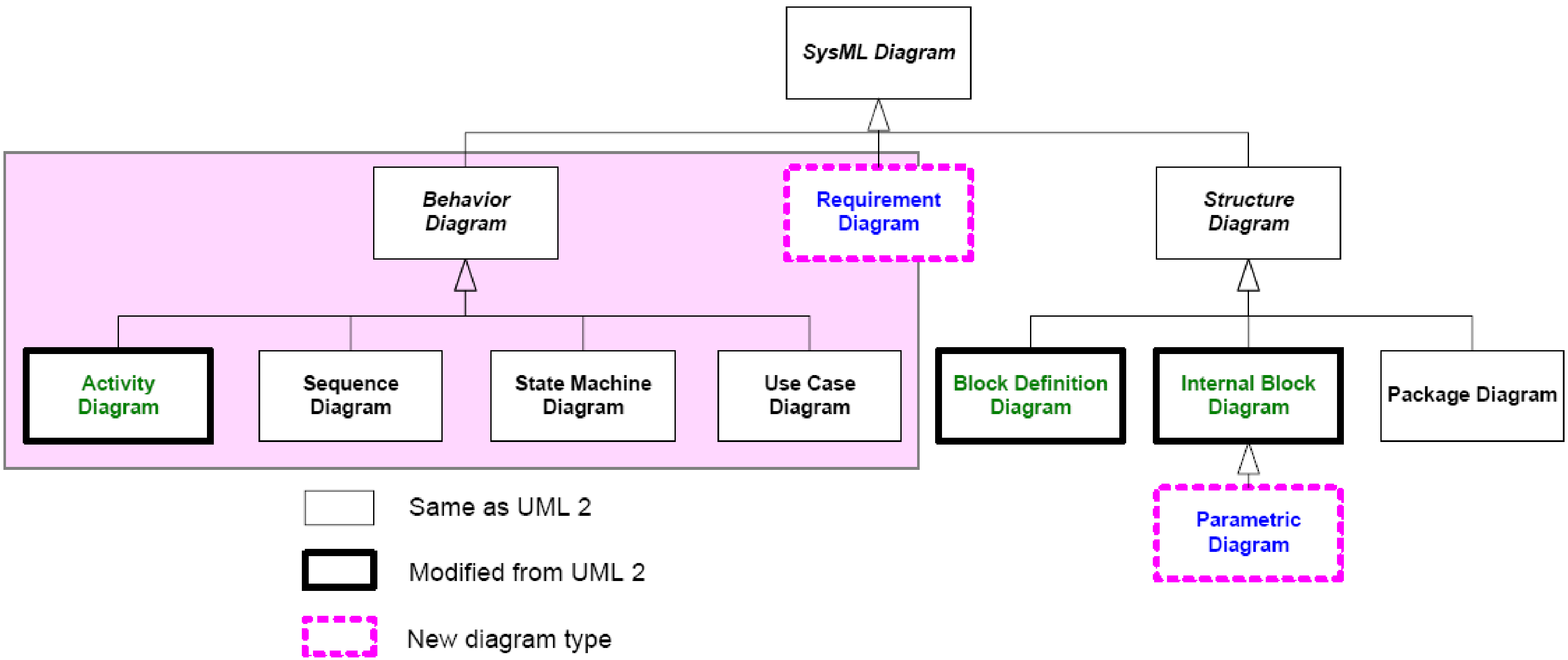
Trade-off & Parametrics

- Parametric relation can be used to support evaluation of alternatives (trade-off analysis)
  - Alternatives represented by different models
  - Objective function specified as a parametric relationship in terms of:
    - Criteria, weighting
    - Probability distributions can be applied to properties
    - Used to optimize based on measures of effectiveness
  - Can be represented in typical table format

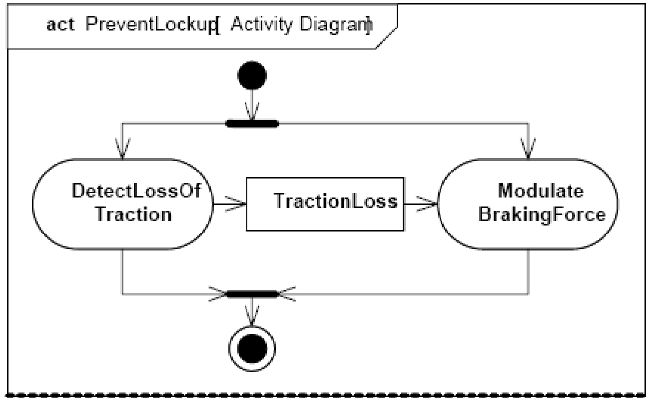
- Methods for trade-offs are not part of SysML

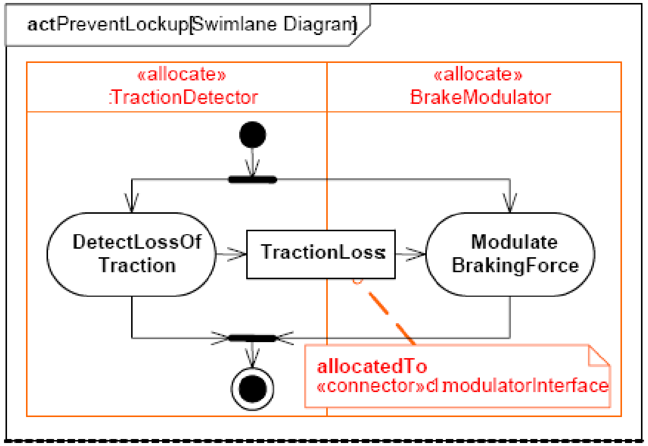
SysML

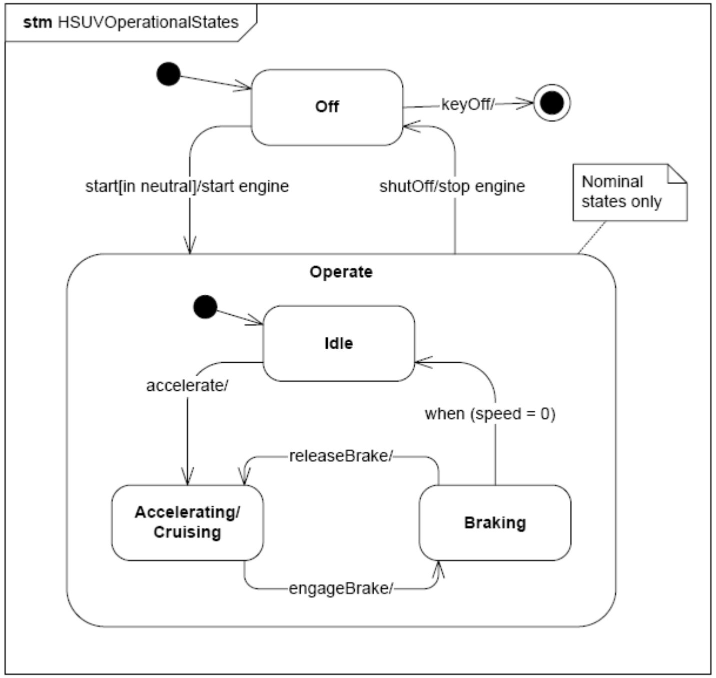
Specifying Behavior


SysML Diagram
Behavior Diagram
Requirement Diagram
Structure Diagram
Activity Diagram
Sequence Diagram
State Machine Diagram
Use Case Diagram
Block Definition Diagram
Internal Block Diagram
Package Diagram
Parametric Diagram
Same as UML 2
Modified from UML 2
New diagram type

Activity Diagrams and State Machine Diagrams







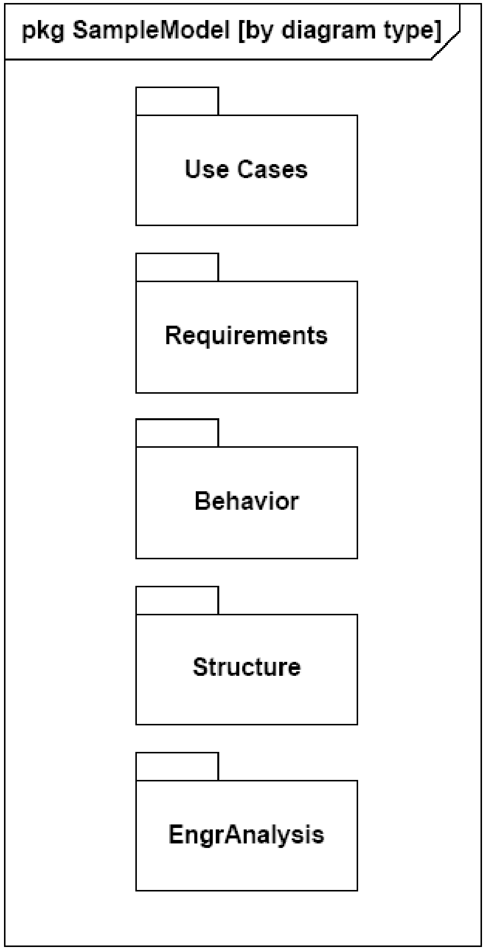
SysML

Specifying Requirements

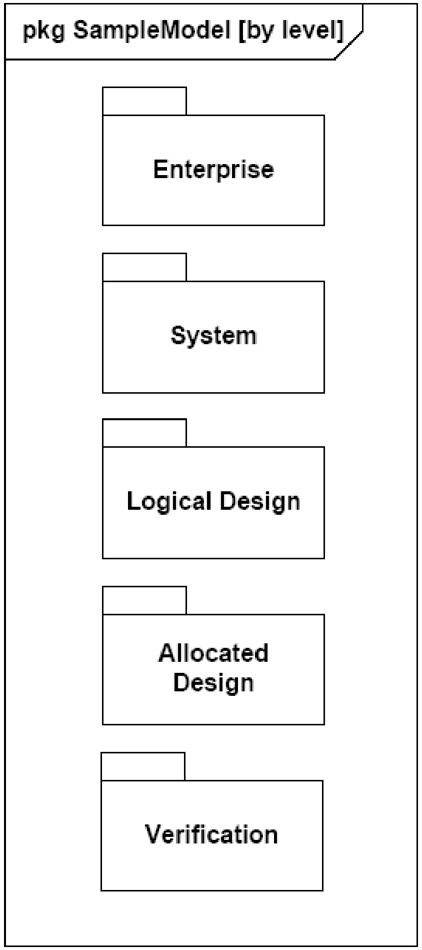
Information management in UML/SysML

- All design elements will reside in exactly one package
- But can be used on many different diagrams
  - Each diagram is located in a package
- A design element is defined by all UML artifacts related to the element
  - Regardless of diagram distribution
  - The complete picture may be distributed over multiple diagrams

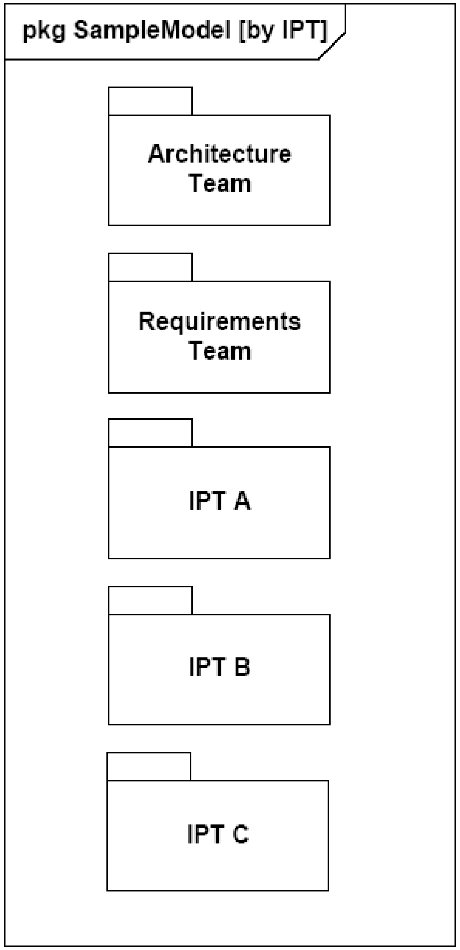
Potential package structures



By Diagram Type



By Hierarchy

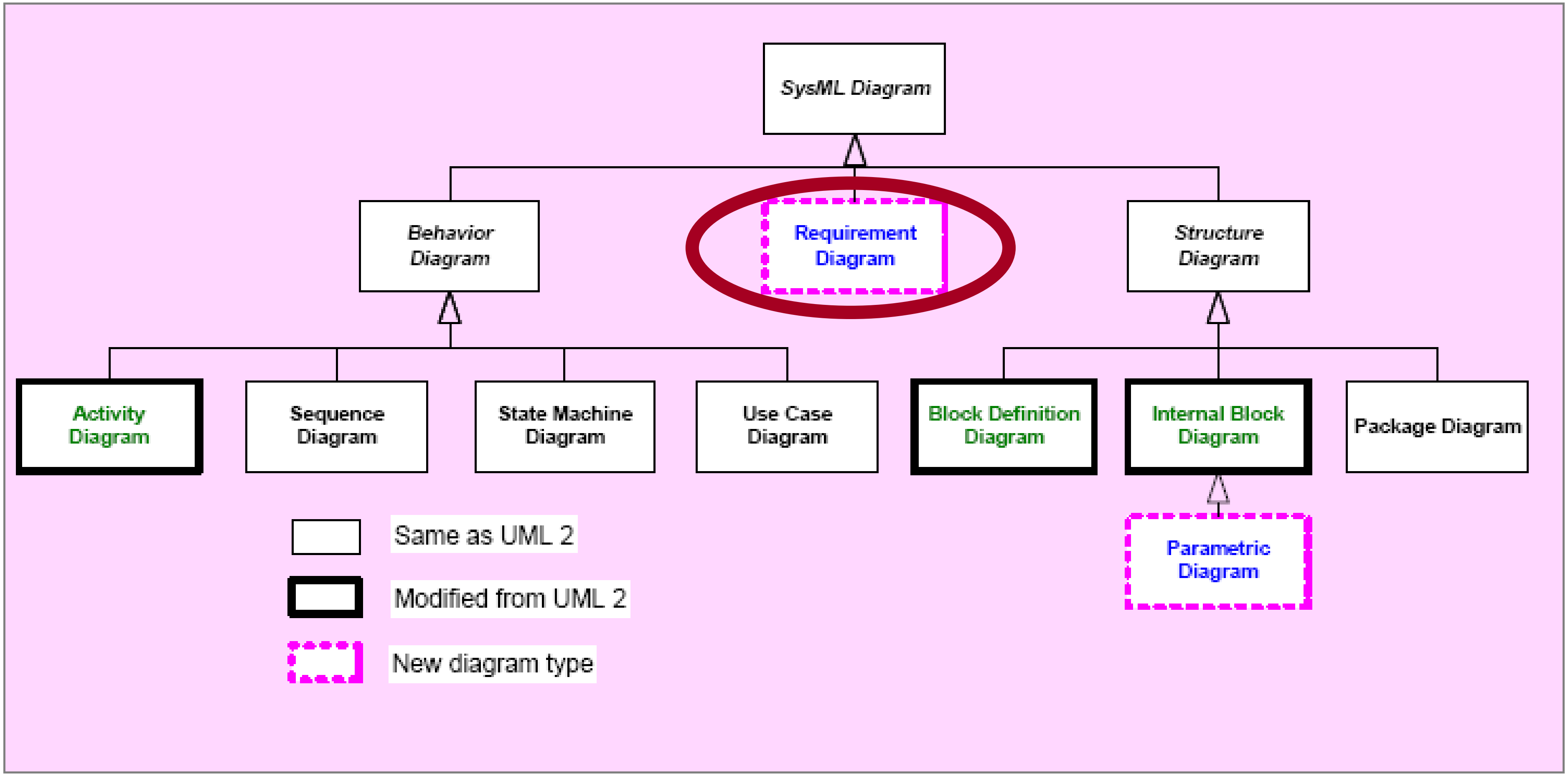


By IPT

What is a requirement?

- Obviously any element in SysML specification is expressing some kind of requirement on a system
- In SysML's terminology a requirement is a **textual statement**
- **No assumptions** are made on the introduction of Requirement elements in the process
- Other model element can be used to identify requirements

- A requirement is a cross-cutting construct



SysML Requirements Overview

SysML provides the following features

- Representation of requirements
  - Representation of individual <u>requirements</u>
  - Requirement <u>composition</u>
  - Requirements can be sub-classed using specialization
- Requirement relationships
  - <u>derive</u> relationship between derived and source requirements
  - <u>satisfy</u> relationship between design models and requirements
  - <u>verify</u> relationship between requirements and test cases
  - generalized <u>trace</u> relationship between requirements and other model elements
  - <u>rationale</u> for requirements traceability, satisfaction, etc
- Alternative graphical, tabular and tree representations
  - Supported by the standard, but currently not implemented in any tools

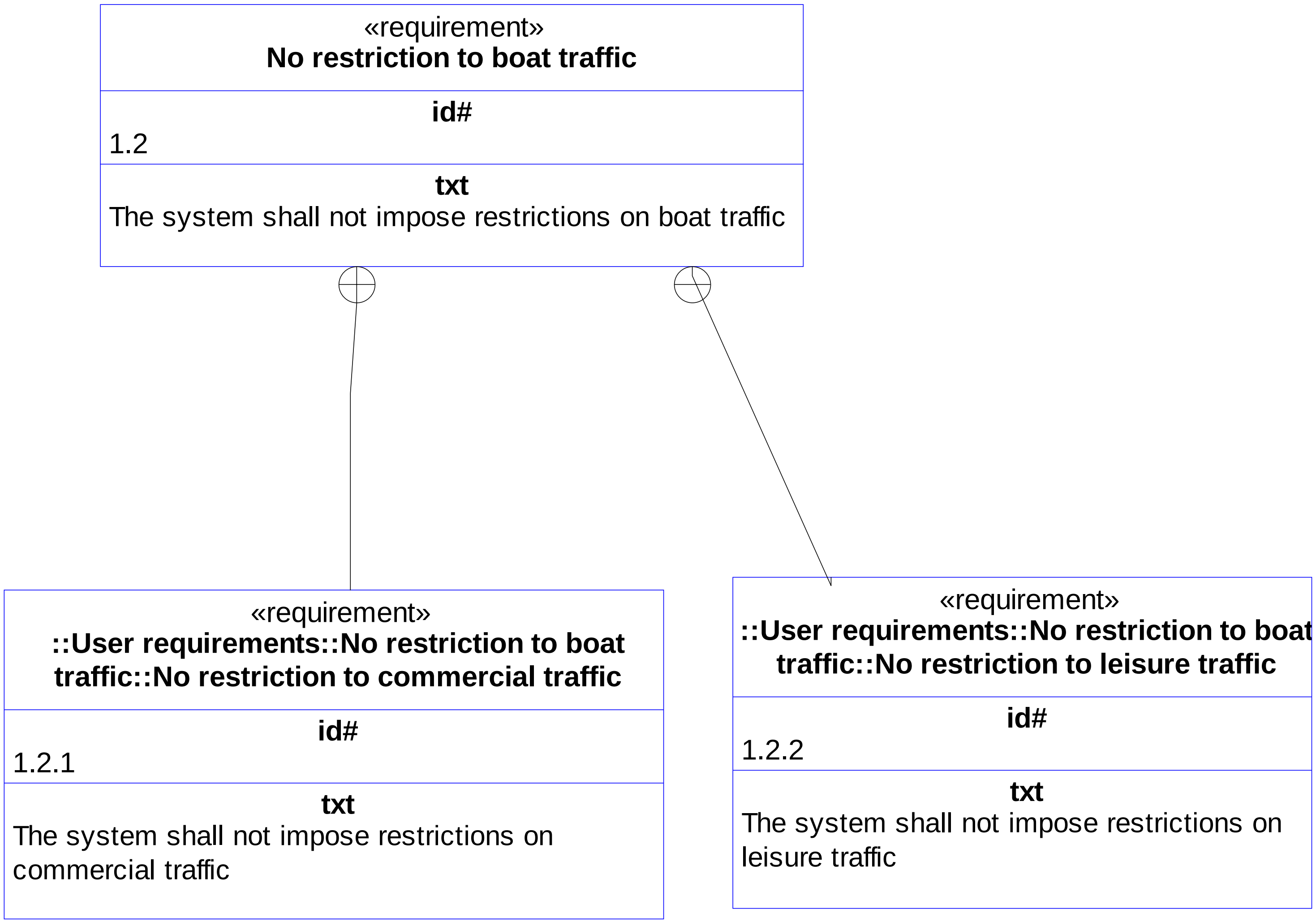
Requirement Representation

«requirement» **::No leisure traffic restriction::Capacity**
id#   1.1
txt   The system shall transport up to 15 passengeres and 1000 kg of cargo under all weather conditions

- Requirement is a stereotyped class
  - Multiple stereotypes can be combined
    - Possible to combine a requirement and safety critical stereotype to form attribute set for a safety critical requirement
- A requirement object has two mandatory attributes:
  - Id
  - Text
- Possible to add new attributes
- A class object is created for each individual requirement

Requirement composition

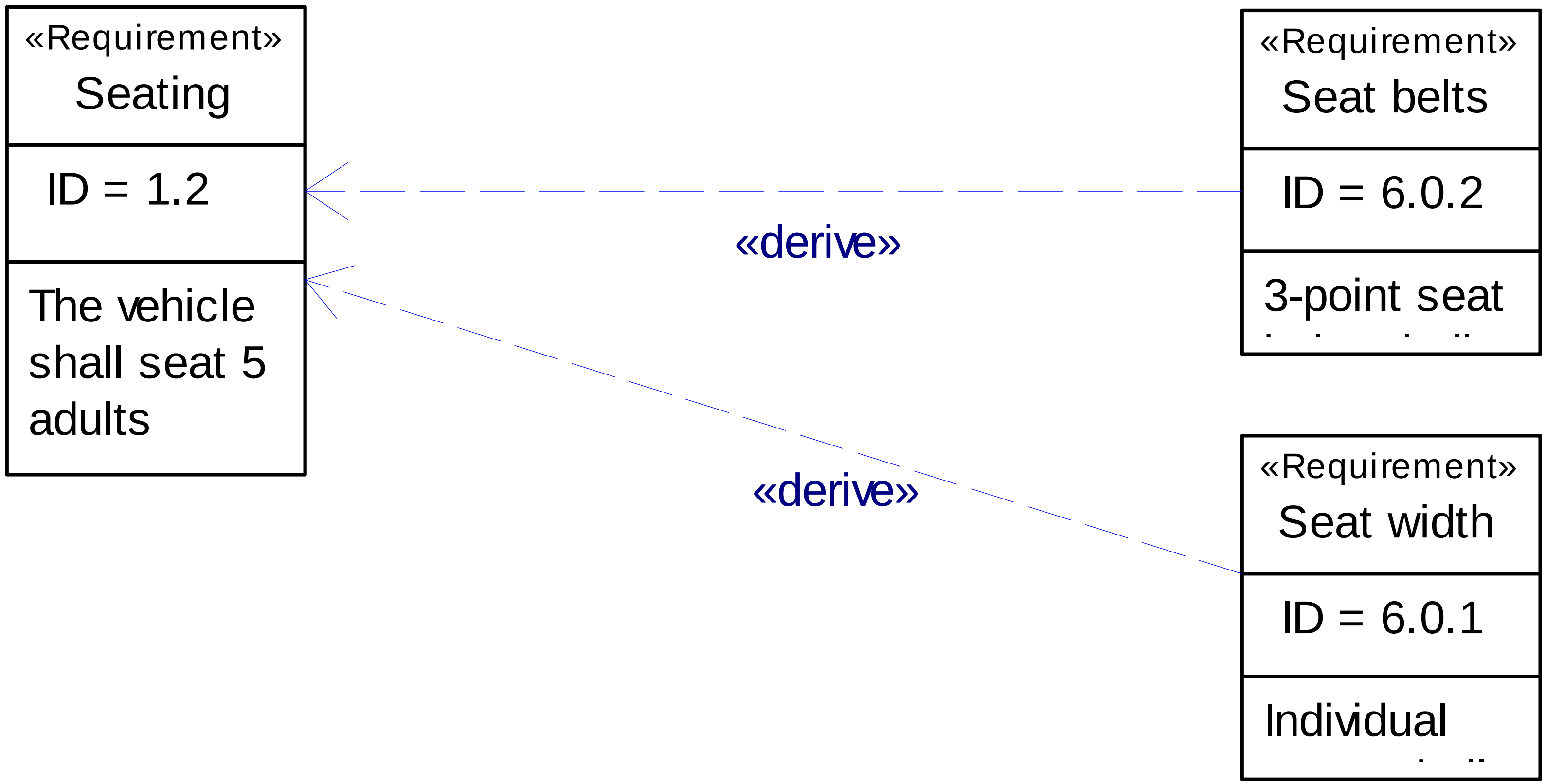
- Composition structure can be of arbitrary depth



Predefined requirement relationships

- **derive** relationship between derived and source requirements
  - The derived requirement is mandated by the source requirement(s)
- **satisfy** relationship between design models and requirements
  - Identified model element(s) are in existence because of the identified requirement
- **verify** relationship between requirements and test cases
  - A verification case may verify one or more requirements, or
  - Multiple cases may be defined for verification of a single requirement
- generalized **trace** relationship between requirements and other model elements
  - For identification of relationships other than those identified above

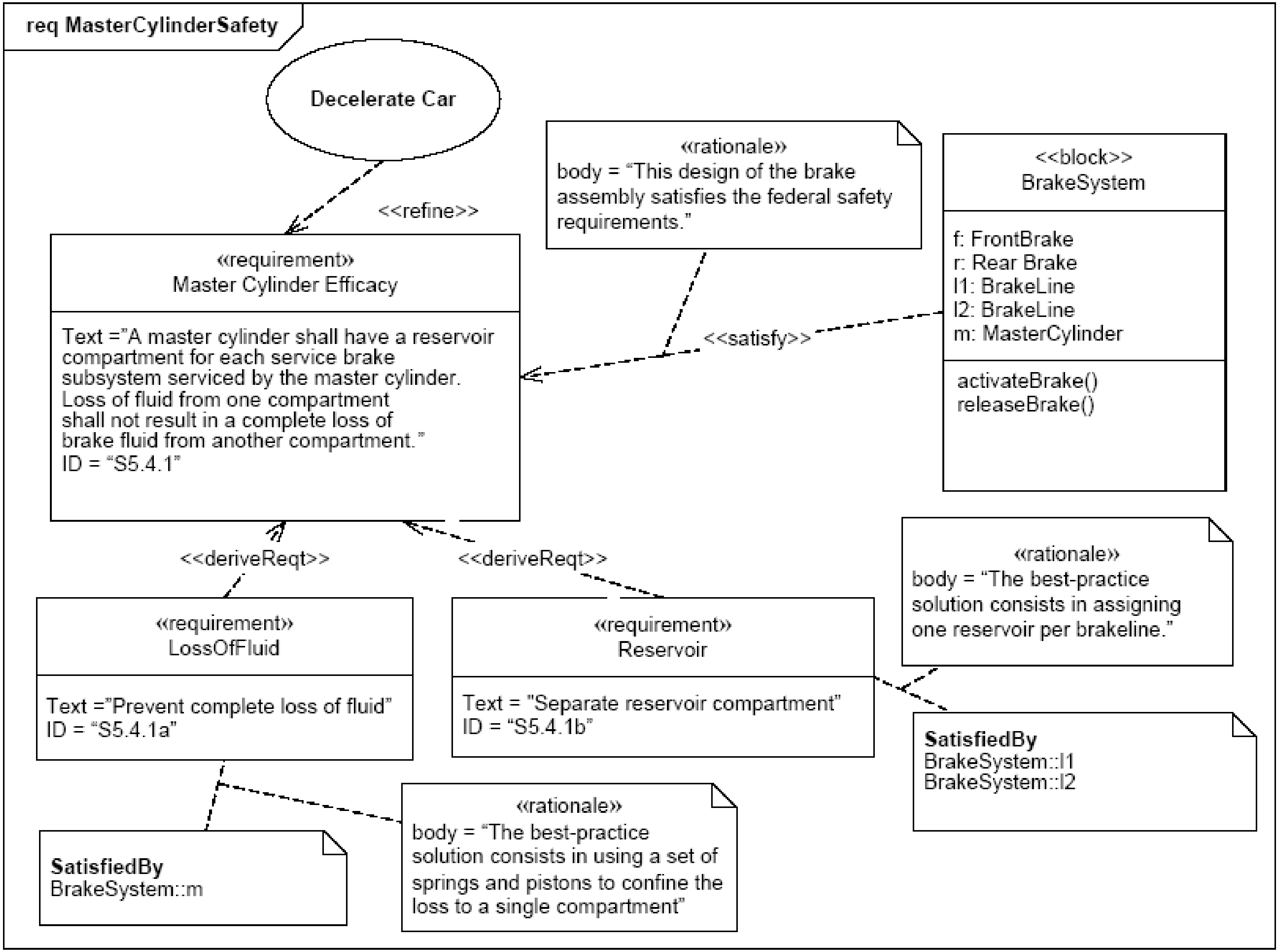
Derive relationship example



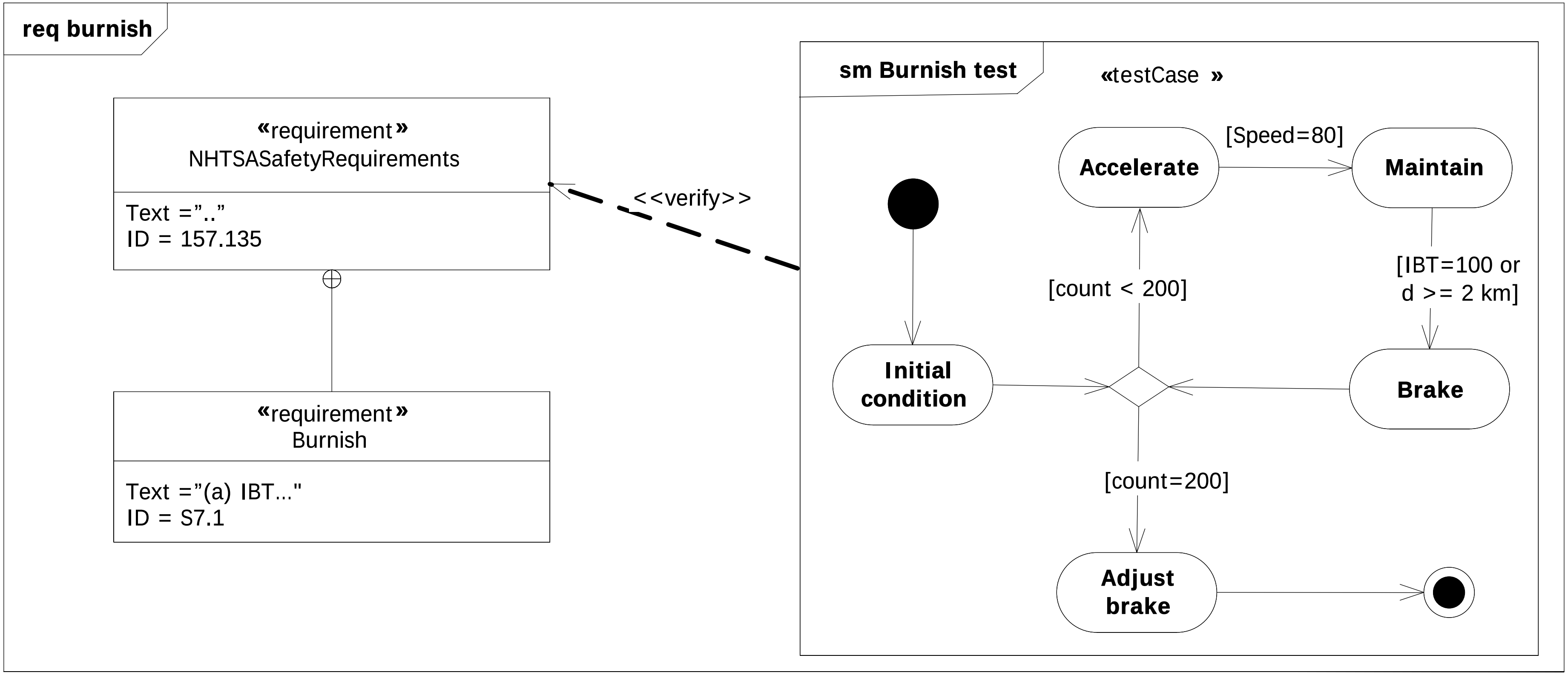
Managing requirements

- Packages – UML concept for grouping elements for some purpose can be used to
  - Separating requirements with different origins
  - Grouping requirements into packages is independent to grouping on diagrams
- Nested packages supported
- A single requirement may appear on multiple requirements diagrams but resides in a single package

Requirement relationships



Linking to verification



SysML Requirements Evaluation

- Requirements use a lot of diagram real-estate
  - Approach does not scale up – unable to efficiently handle projects with several hundreds of requirements
  - The traditional (graphical) UML view does not lend itself well to requirements representation
    - A tabular view would be more appropriate (as used in traditional requirements management tools)
- Requirements modeling is performed on class definition basis
  - Each requirement is actually a new class object

Workarounds

- Distribute requirements over multiple diagrams
- Create diagram exclusively for allocation and traceability
  - Risk for loosing overview
- Perform requirements management in separate tool
  - Do the traceability in SysML
  - Difficult to maintain consistency

SysML

Verification

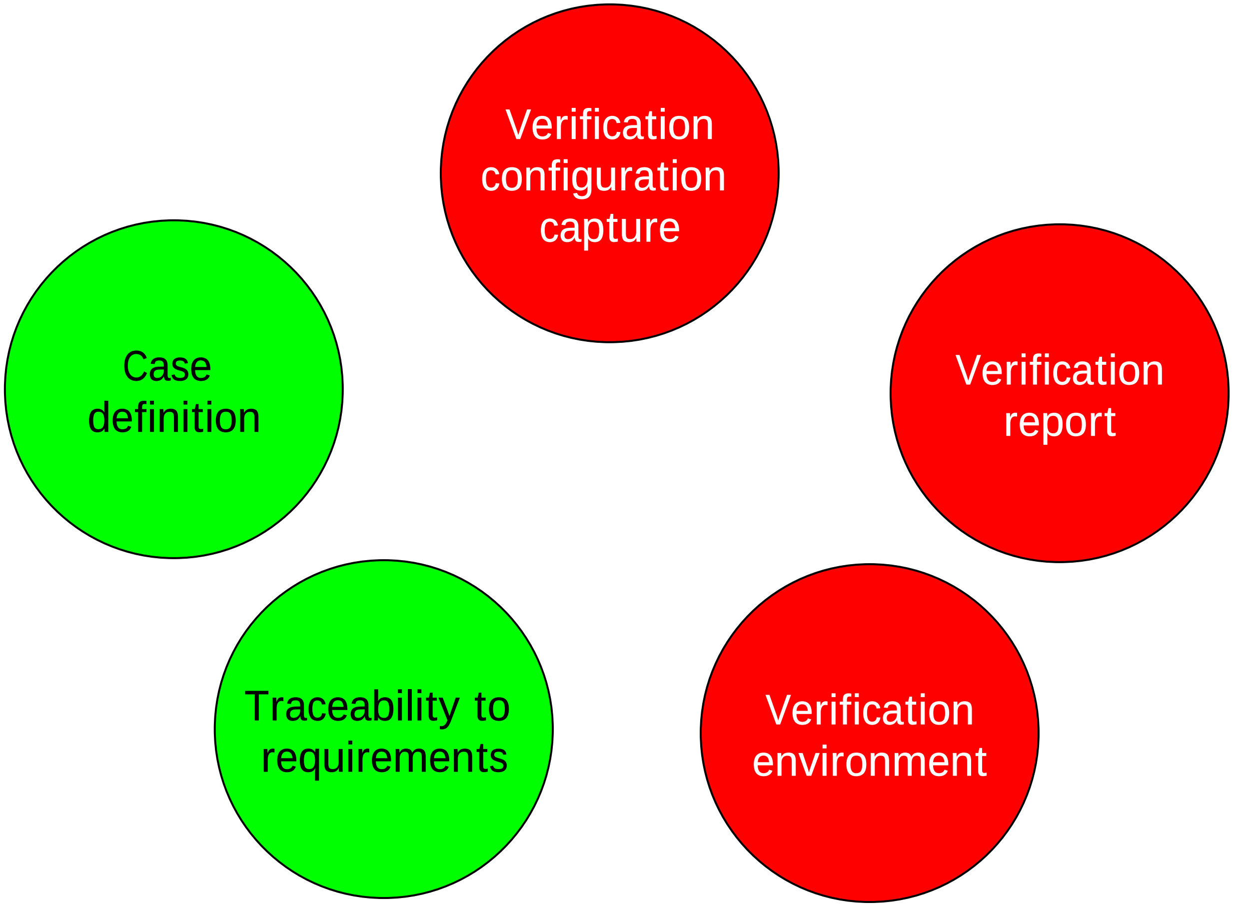
Principles

- Develop a model that defines the verification conditions and procedure
  - Excellent for software where tests can be run within the tool
  - Not necessary applicable when the model shall depict a real world condition
- Primary application for systems verification is the capture of the verification procedure
  - Can not completely replace the traditional verification documentation
- At the present SysML does not support the representation realized system elements
  - Not possible to represent the configuration and exact properties of a unit under test

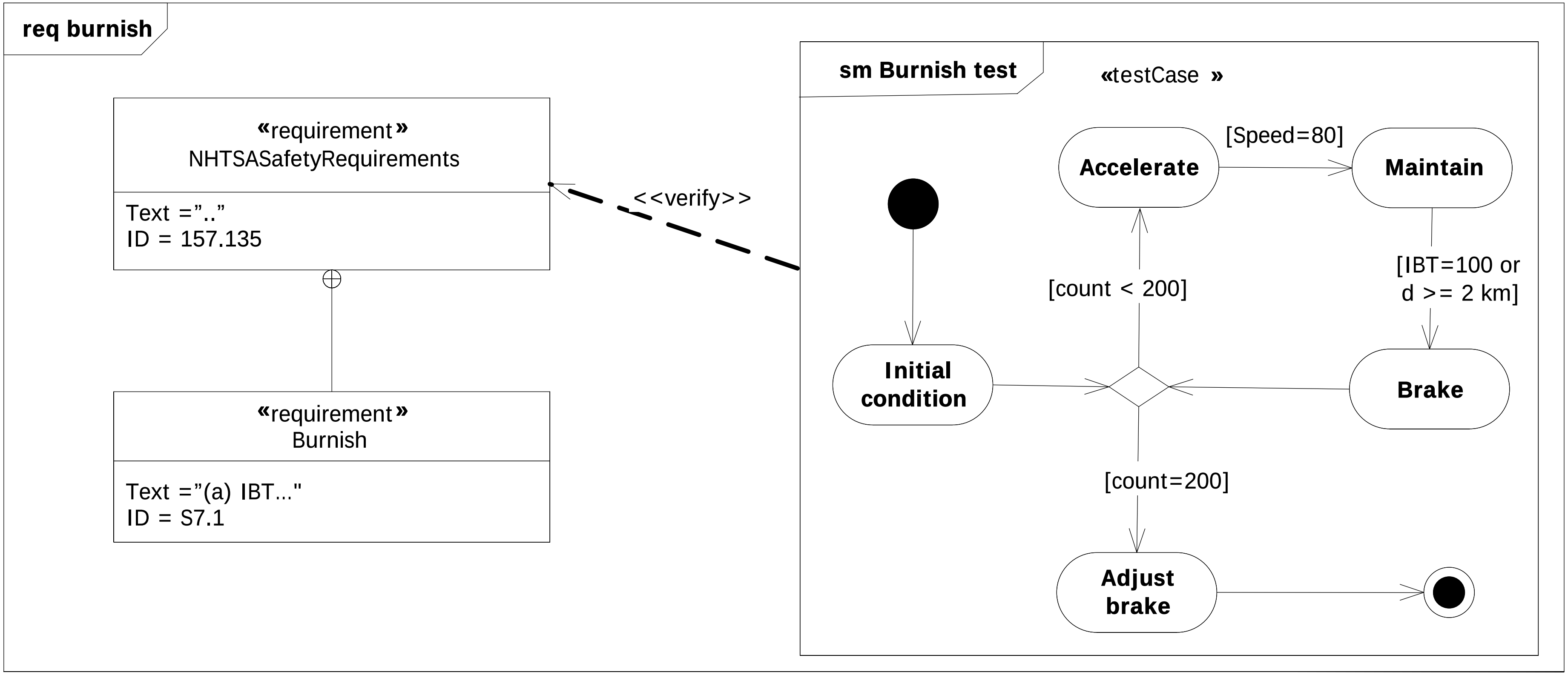
Verification case development

- Any set of model elements can be used to define the verification environment for a requirement
- The verification procedure can be captured in detail
- Textual elements can be captured using requirement objects with extra stereotypes
- Verification cases may be stored in dedicated packages

SysML Support for verification



Definition of verification case



SysML

Application in the development process

Applying SysML in the development process

- SysML is process independent
  - Any use is per definition correct
  - Model fidelity will increase over time
- SysML does not define a strict top down modeling method
  - Multiple viewpoints are supported via packages
  - Viewpoint integration must be considered
    - Which diagrams apply for a specific viewpoint?
    - What are the relationships between identified viewpoints
- The complete system specification will not be available in a single diagram

SysML in the design process

- Requirements
  - Requirement Diagrams
- Behavior
  - Activity Diagrams
  - Sequence Diagrams
  - State Machine Diagrams
  - Use Case Diagrams
- Architecture
  - Block Diagrams
  - Parametric Diagrams

Model management

- Tools often have links to standard version management systems
- Individual elements can be under version control
- Configuration control (of hierarchical structures) is typically not supported

Integration into the document centric paradigm

- All system relevant information does not lend itself to modeling
  - Traditional documents will still exist
- For good or bad we know how to manage documents
  - Readability
  - CM support
- SysML tools typically have
  - Report generators
  - Links to requirements management tools, e.g., DOORS
- Need to add textual element to create fully readable documents
- All information on a system will not reside in the SysML tool

SysML

Summary

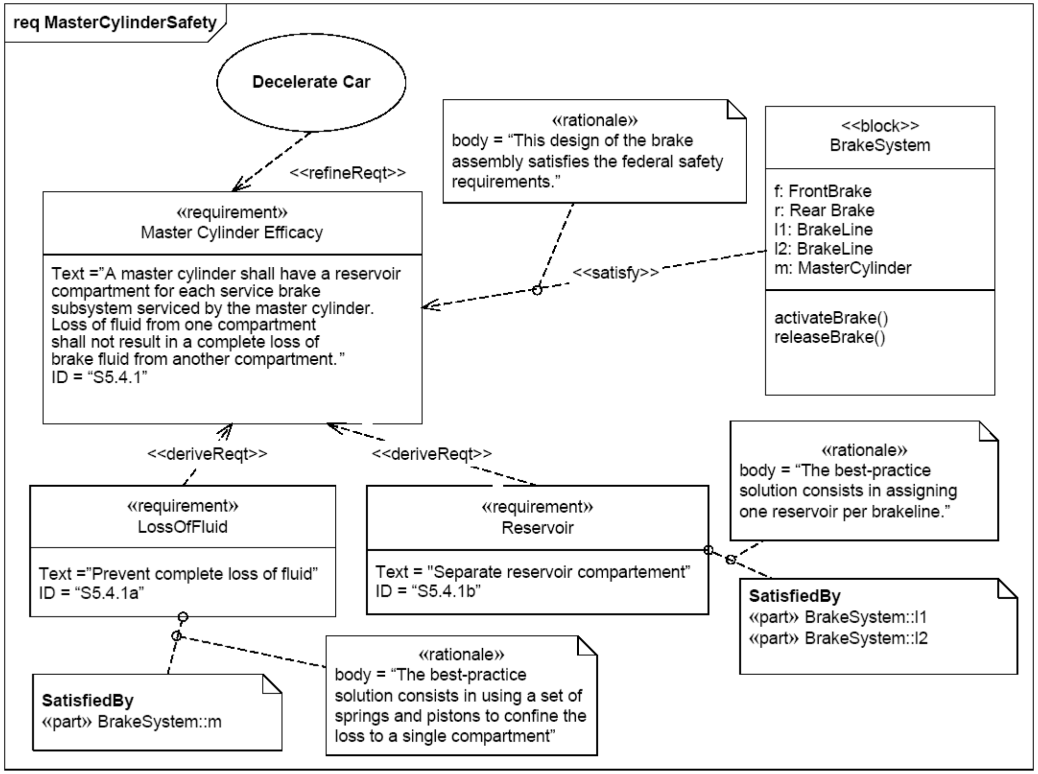
SysML – the good

- It is here, it is available
- Support from multiple vendors
- Broad user base
- It is UML – but simpler
- Excellent software engineering integration
    - Most SysML implementations are actually on top on UML tools
- XMI, promise for data portability

SysML – the bad (1)

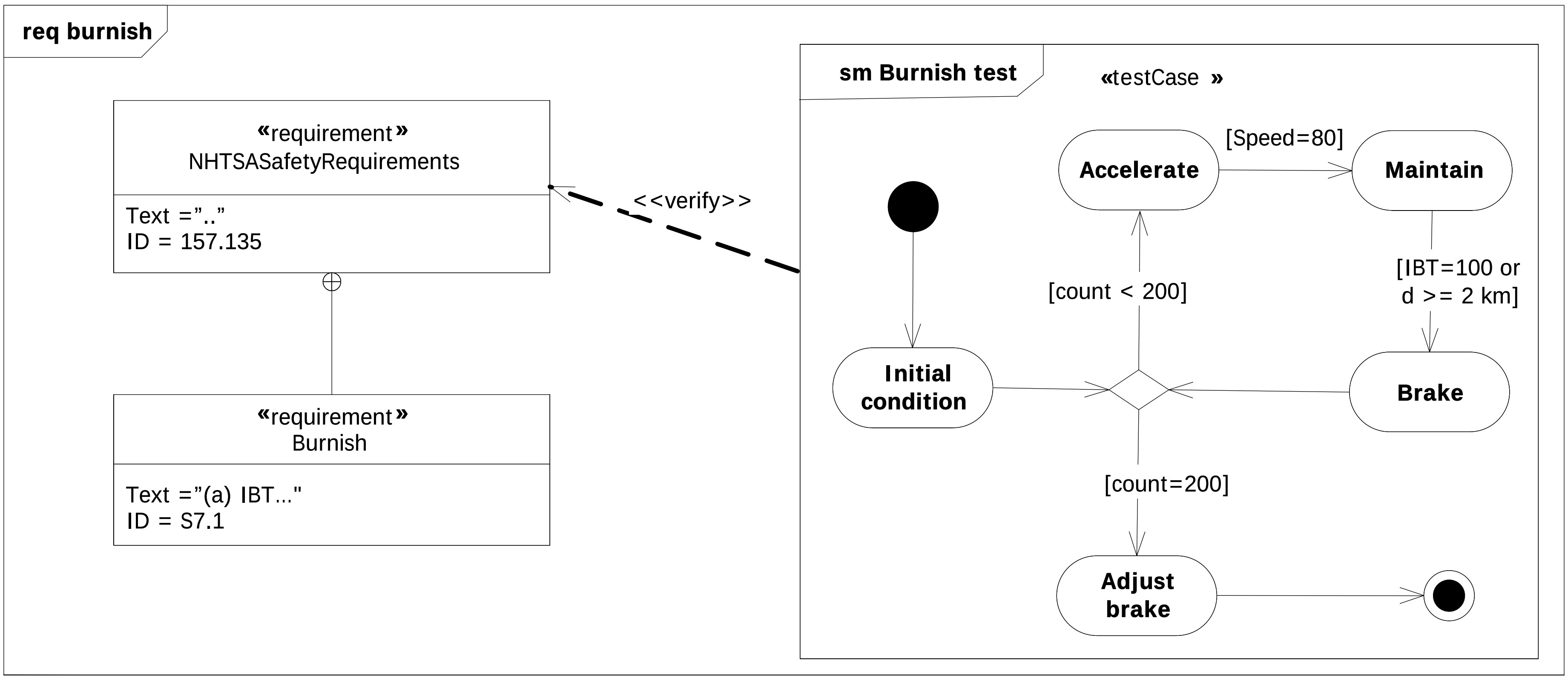
- It is an adoption of UML
  - Ad hoc implementation
- Contrived activity diagram semantics
  - Inherited from UML
- Manual management of allocation relationships
- Minimal verification support

SysML – the bad (2)



- Problem
  - The user must to manage all allocation relationships manually
  - Leads to cluttered diagrams
- The elegant solution
  - Automatic management of relationships

- Verification support



- How do I capture the product verified?

SysML Adoption Strategies

- Minimal cost
  - Use SysML notation in Powerpoint or Visio
- Hybrid MBSE
  - Use SysML tool to model key elements of a specification/design
  - But maintain document paradigm for deliverables
- True MBSE
  - Full SysML adoption
- The Alternative
  - Use existing SE tool with proprietary notation

SysML – positive aspects

- SysML is far better than PowerPoint!
- Can be highly valuable for highlighting core elements of a specification
- Is perfectly suited for modeling of Software intensive systems
  - Tight coupling to UML outweighs negative aspects identified herein
- Is the future
  - We must just ensure that SysML is modified and extended over time such that the core problems are addressed!
    - Integrated configuration and change management support
    - Connection to the complete system lifecycle
    - Connection to domain engineering disciplines

SysML Conclusions

- SysML is an admirable product considering
  - Its ancestry
  - The limited resources used in its creation
- There are a number of weak areas in the language as outlined in this presentation
- The overarching problem is that SysMLs failure to address the core issues
  - Through life traceability
  - Configuration management
- This is a problem inherited from the UML framework
  - And not addressed in contemporary SE tools
- These problems are challenges for development system vendors to overcome
  - With guidance and assistance from the user community

SysML

Evaluation Summary

- SysML and UML tools have different target groups
  - Systems engineers will probably not gain from code generation and all related functionality
  - Systems engineers will probably not modify the underlying notation
  - Systems engineers will probably not modify the tool to fit the problem

- Tool vendors need to simplify the user interfaces
  - minimize actions and manipulations for using the tool
  - hide the extension mechanisms

An ideal Vision …

- A development environment that allows for maintaining an overall traceability from the initial ideas to the realized product

- Traceability …
  - … from requirements to the realized product
  - … from and to software and hardware elements
  - … across different variants of a product line
  - … across different configurations
  - … across time (history)
  - … between every individual element

… and the reality

- The creators of SysML have been driven by a less ambitious vision
  - i.e. more realistic vision
- SysML lacks support for versions & configurations
- SysML has limited support for specific individuals
  - an individual realized product
- SysML has a clear heritage of software development language

Risks

- UML tool vendors have good understanding for software-related system development
  - but lack understanding for SE in a broader perspective
- There is a risk that the future development of SysML (tools) will be predominantly influenced by software engineering
  - And increased resources on “code refactoring” do not deliver any value to systems engineers
- Systems Engineers risk to become yet another customer of tools that are basically domain-specific
  - e.g. the lack of integrated support for configuration management

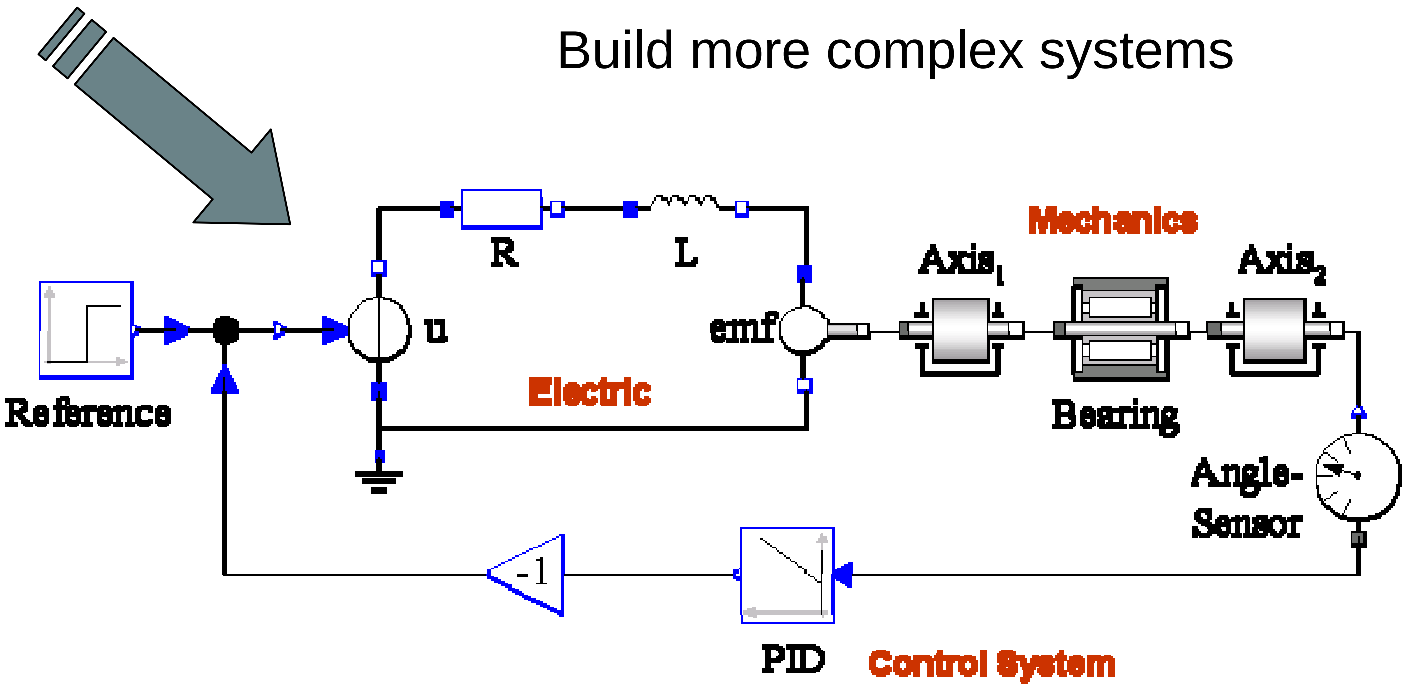
Modelica

An equation-based object-oriented language for modeling and simulation of physical systems

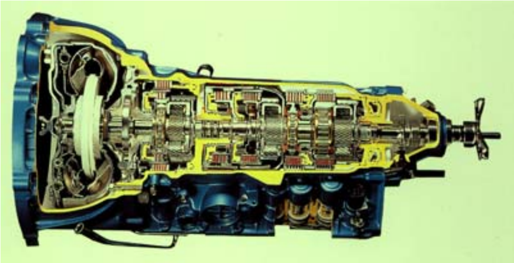
Why Modeling & Simulation?

- Increase understanding of complex systems
- Design and optimization
- Virtual prototyping
- Verification

Build more complex systems



- Robotics
- Automotive
- Aircrafts
- Satellites
- Biomechanics
- Power plants
- Hardware-in-the-loop, real-time simulation
- etc

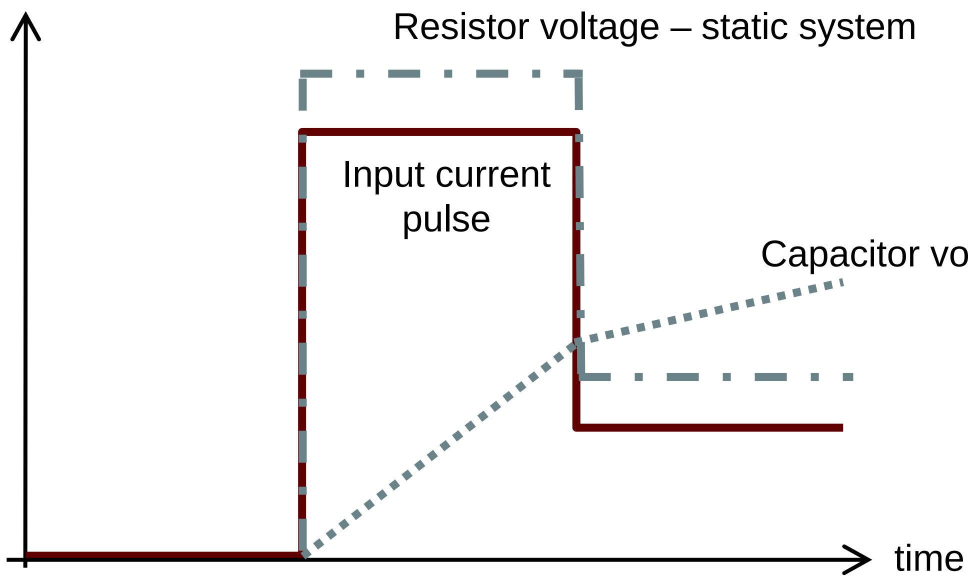


Kinds of Mathematical Models

- Dynamic vs. Static models
- Continuous-time vs. Discrete-time dynamic models
- Quantitative vs. Qualitative models

A **dynamic** model includes *time* in the model

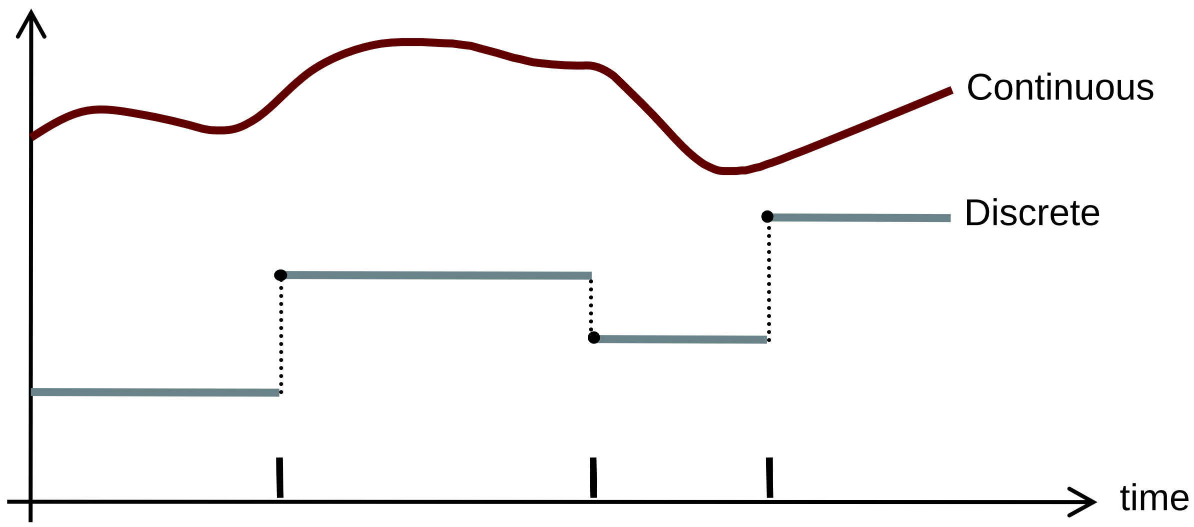
A **static** model can be defined *without* involving *time*



Continuous vs. Discrete-Time Dynamic Models

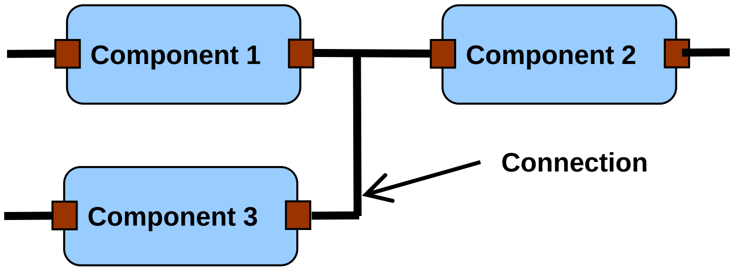
Continuous-time models may evolve their variable values *continuously* during a time period

Discrete-time variables change values a *finite* number of times during a time period



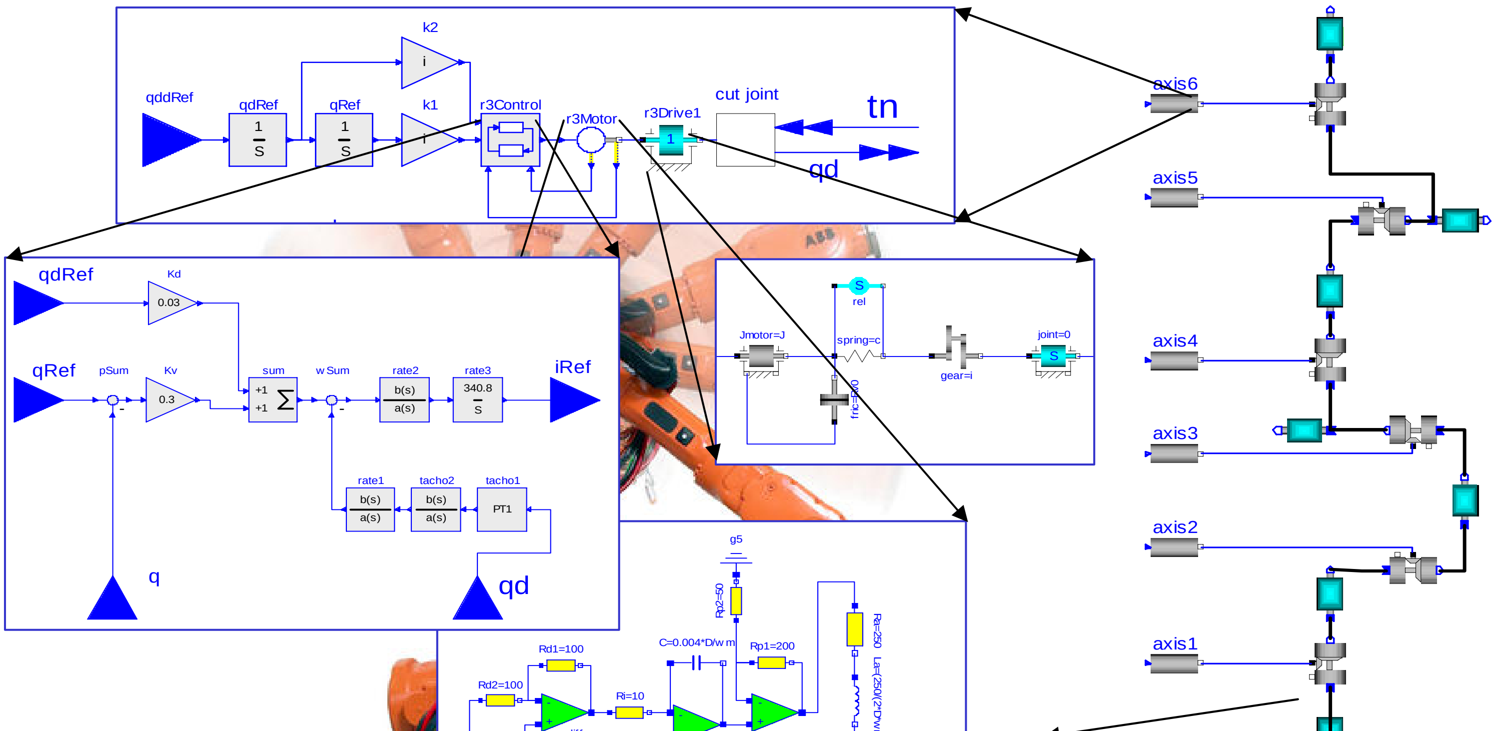
Principles of Graphical Equation-Based Modeling

- Each icon represents a physical component i.e. Resistor, mechanical Gear Box, Pump
- Composition lines represent the actual physical connections i.e. electrical line, mechanical connection, heat flow



- Variables at the interfaces describe interaction with other component
- Physical behavior of a component is described by equations

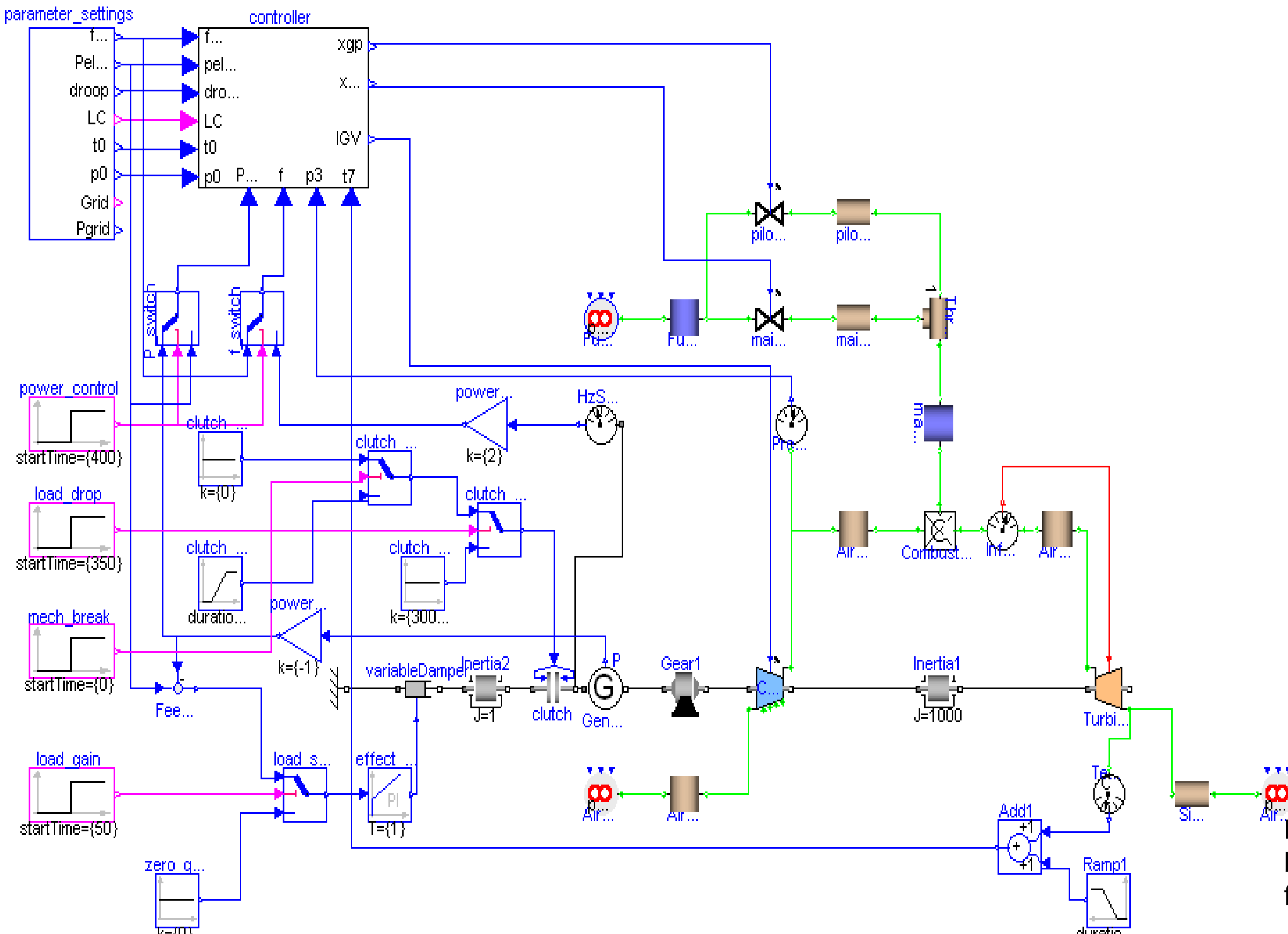
Application Example – Industry Robot




```
Srel = n*transpose(n)+(identity(3)- n*transpose(n))*cos(q)-
skew(n)*sin(q);
wrela = n*qd;
zrela = n*qdd;
Sb = Sa*transpose(Srel);
r0b = r0a;
vb = Srel*va;
wb = Srel*(wa + wrela);
ab = Srel*aa;
zb = Srel*(za + zrela + cross(wa, wrela));
```

Courtesy of Martin Otter

GTX Gas Turbine Power Cutoff Mechanism



Developed by MathCore for Siemens

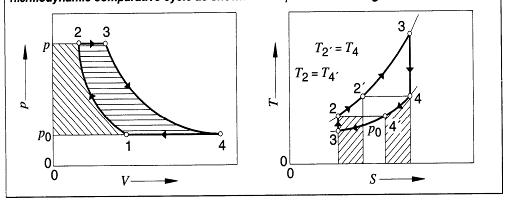
Courtesy of Siemens Industrial Turbomachinery AB

Modelica

The Next Generation Modeling Language

Model knowledge is stored in books and human minds which computers cannot access

Thermodynamic comparative cycle as shown in the *p–V-* und *T–S* diagrams



from T_2 to $T_{2'}$, supplied by the heat exchanger is coupled with a thermal discharge (4 → 4′). If heat is completely exchanged, the quantity of heat to be added per unit of gas is reduced to

$$q_{in} = c_p \cdot (T_3 - T_{2'}) = c_p \cdot (T_3 - T_4)$$

and the quantity of heat to be removed is

$$q_{out} = c_p \cdot (T_{4'} - T_1) = c_p \cdot (T_2 - T_1).$$

The maximum thermal efficiency for the gas turbine with heat exchanger is:

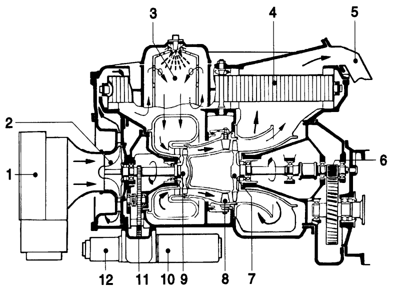
$$\eta_{th} = 1 - Q_{out}/Q_{in} = 1 - (T_2 - T_1)/(T_3 - T_4)$$

Where $p_2/p_1 = (T_2/T_1)^{\frac{\chi}{\chi-1}} = (T_3/T_4)^{\frac{\chi}{\chi-1}}$ and $T_4 = T_3 \cdot (T_1/T_2)$ thus

$$\eta_{th} = 1 - (T_2/T_3)$$

Current gas-turbine powerplants achieve thermal efficiencies of up to 35 %.
Advantages of the gas turbine: clean exhaust without supplementary emissions-control devices; extremely smooth running; multifuel capability; good static torque curve; extended maintenance intervals.
Disadvantages: manufacturing costs still high; poor transitional response; higher fuel consumption; less suitable for low-power applications.

Gas turbine *1 Filter and silencer, 2 Radial-flow compressor, 3 Burner, 4 Heat exchanger, 5 Exhaust port, 6 Reduction gearset, 7 Power turbine, 8 Adjustable guide vanes, 9 Compressor turbine, 10 Starter, 11 Auxiliary equipment drive, 12 Lubricating oil pump.*

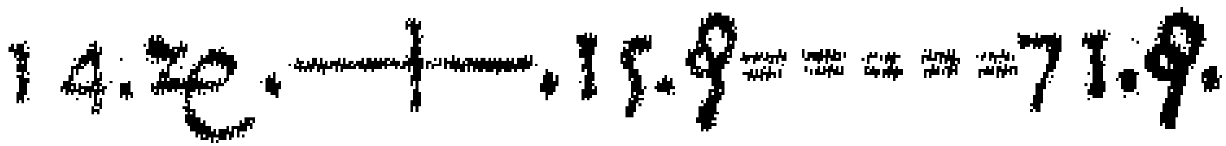


"The change of motion is proportional to the motive force impressed"
– Newton

Lex. II.

Mutationem motus proportionalem esse vi motrici impressæ, & fieri secundum lineam rectam qua vis illa imprimitur.

The Form – Equations

- Equations were used in the third millennium B.C.
- Equa  corde in 15

Newton still wrote text (Principia, vol. 1, 1686)

“The change of motion is proportional to the motive force impressed ”

CSSL (1967) introduced a special form of “equation”:

```
variable = expression
v = INTEG(F)/m
```

Programming languages usually do not allow equations!

Modelica – The Next Generation *Modeling* Language

- *Declarative language*
  - Equations and mathematical functions allow acausal modeling, high level specification, increased correctness
- *Multi-domain modeling*
  - Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...
- *Everything is a class*
  - Strongly typed object-oriented language with a general class concept, Java & Matlab like syntax
- *Visual component programming*
  - Hierarchical system architecture capabilities
- *Efficient, nonproprietary*
  - Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations

- The static *declarative structure* of a mathematical model is emphasized
- OO is primarily used as a *structuring concept*
- OO *is not* viewed as dynamic object creation and sending messages
- *Dynamic model* properties are expressed in a *declarative way* through equations.
- Acausal classes supports *better reuse of modeling and design knowledge* than traditional classes

What is *acausal* modeling/design?

Why does it increase *reuse*?

The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

Example: a resistor *equation*:

R*i = v;

can be used in three ways:

i := v/R;
v := R*i;
R := v/i;

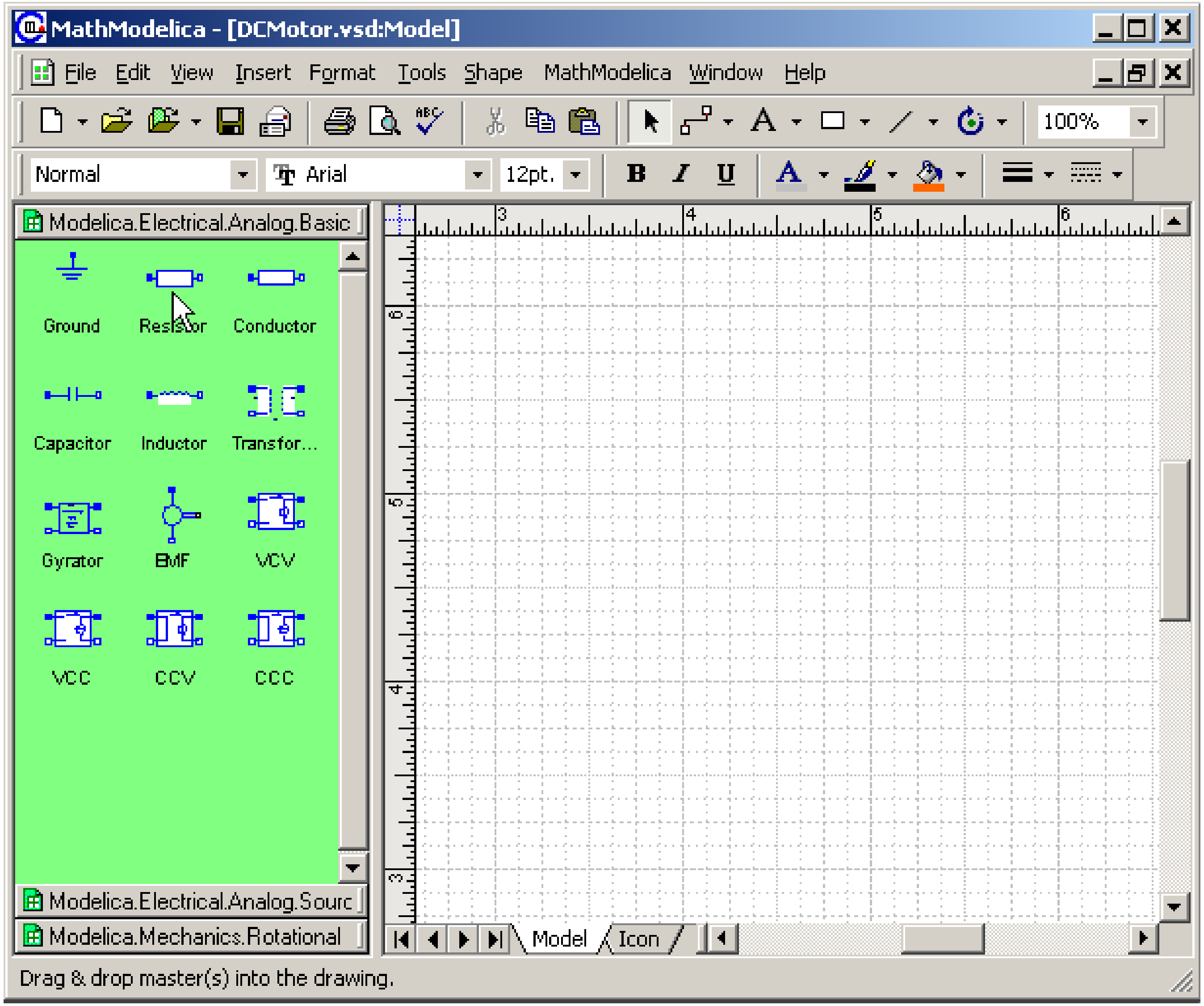
Brief Modelica History

- First Modelica design group meeting in fall 1996
  - International group of people with expert knowledge in both language design and physical modeling
  - Industry and academia

- Modelica Versions
  - 1.0 released September 1997
  - 2.0 released March 2002
  - 2.2 released March 2005
  - 3.0 released September 2007

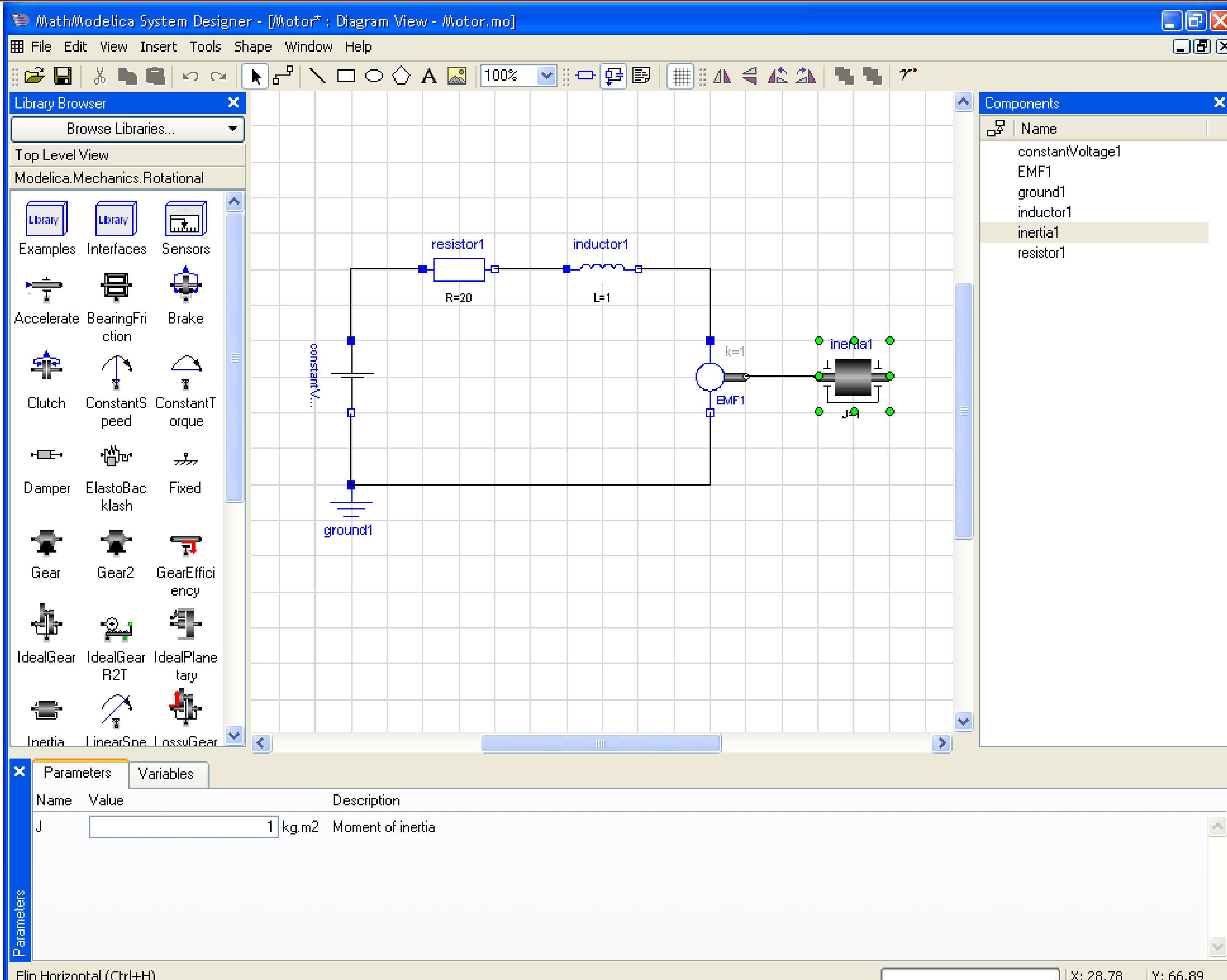
- Modelica Association established 2000
  - Open, non-profit organization

Graphical Modeling Using Drag and Drop Composition



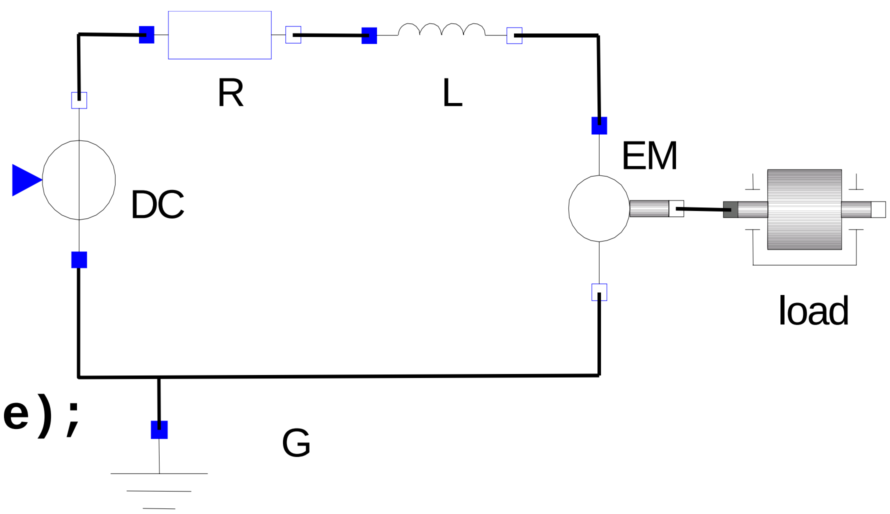
Courtesy MathCore Engineering AB

Graphical Modeling - Drag and Drop Composition



- A DC motor can be thought of as an electrical circuit which also contains an electromechanical component

```
model DCMotor
   Resistor R(R=100);
   Inductor L(L=100);
   VsourceDC DC(f=10);
   Ground G;
   ElectroMechanicalElement EM(k=10,J=10, b=2);
   Inertia load;
equation
   connect(DC.p,R.n);
   connect(R.p,L.n);
   connect(L.p, EM.n);
   connect(EM.p, DC.n);
   connect(DC.n,G.p);
   connect(EM.flange,load.flange);
end DCMotor
```



Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

```
0 == DC.p.i + R.n.i
DC.p.v == R.n.v

0 == R.p.i + L.n.i
R.p.v == L.n.v

0 == L.p.i + EM.n.i
L.p.v == EM.n.v

0 == EM.p.i + DC.n.i
EM.p.v == DC.n.v

0 == DC.n.i + G.p.i
DC.n.v == G.p.v
```

```
EM.u == EM.p.v - EM.n.v
0 == EM.p.i + EM.n.i
EM.i == EM.p.i
EM.u == EM.k * EM.ω
EM.i == EM.M / EM.k
EM.J * EM.ω == EM.M - EM.b * EM.ω

DC.u == DC.p.v - DC.n.v
0 == DC.p.i + DC.n.i
DC.i == DC.p.i
DC.u == DC.Amp * Sin[2 π DC.f * t]
```

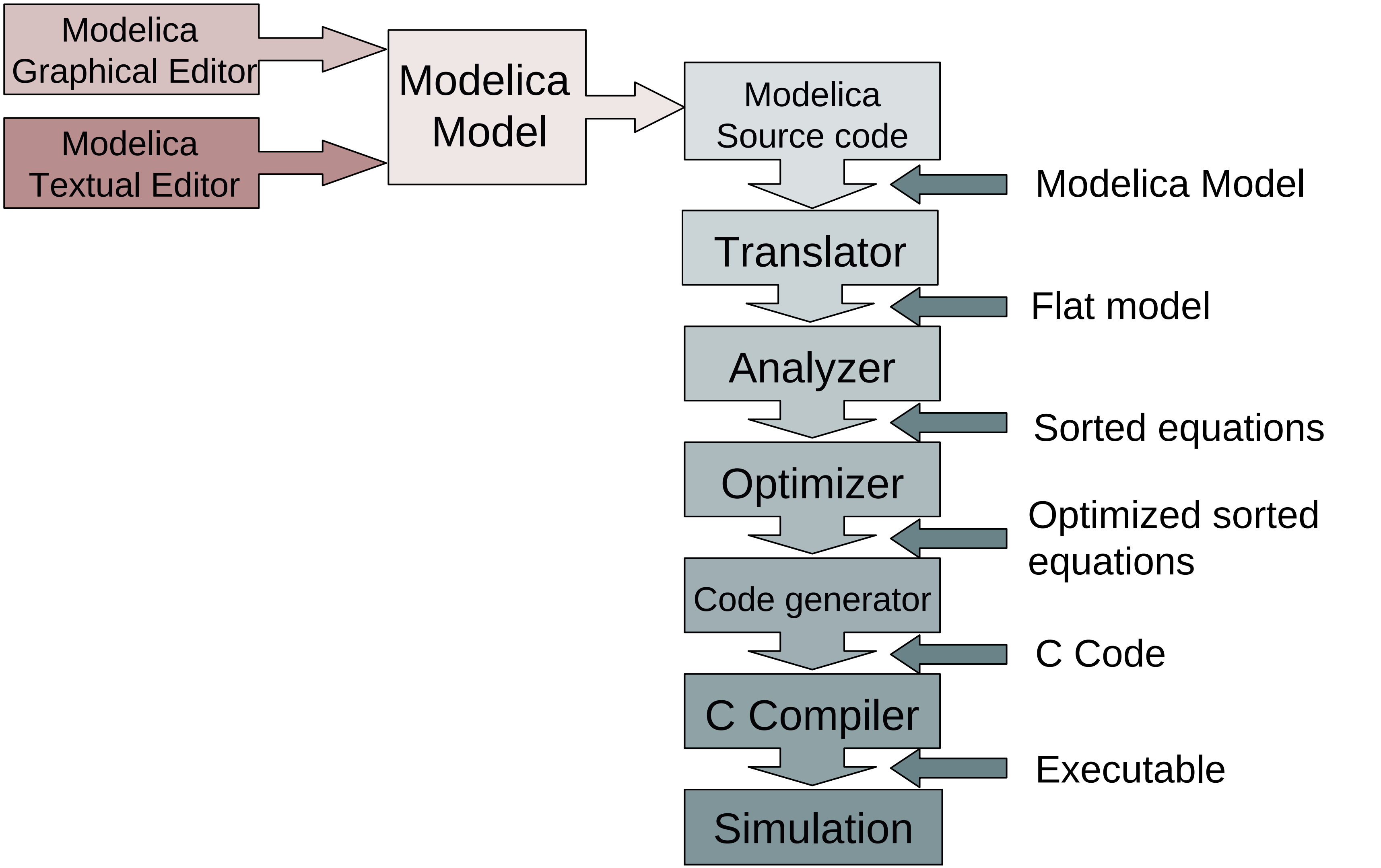
```
R.u == R.p.v - R.n.v
0 == R.p.i + R.n.i
R.i == R.p.i
R.u == R.R * R.i

L.u == L.p.v - L.n.v
0 == L.p.i + L.n.i
L.i == L.p.i
L.u == L.L * L.i'
```

(`load` component not included)

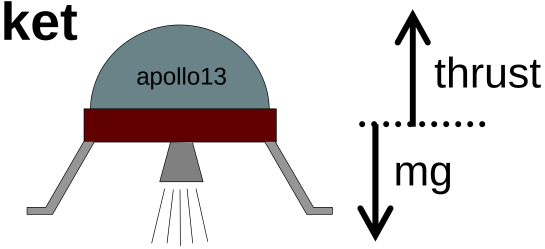
Automatic transformation to ODE or DAE for simulation:

$$\frac{dx}{dt} == f[x, u, t] \qquad\qquad g\left[\frac{dx}{dt}, x, u, t\right] == 0$$

Translation of Models to Simulation Code



Rocket



$$acceleration = \frac{thrust - mass \cdot gravity}{mass}$$

$$mass' = -massLossRate \cdot \text{abs}\left(thrust\right)$$

$$altitude' = velocity$$

$$velocity' = acceleration$$

```
class Rocket "rocket class"
  parameter String name;
  Real mass(start=1038.358);
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust;  // Thrust force on rocket
  Real gravity; // Gravity forcefield
  parameter Real massLossRate=0.000277;
equation
  (thrust-mass*gravity)/mass = acceleration;
  der(mass)  = -massLossRate * abs(thrust);
  der(altitude) = velocity;
  der(velocity) = acceleration;
end Rocket;
```

- **new model** ← `class Rocket`
- **declaration comment** ← `"rocket class"`
- **parameters (changeable before the simulation)** ← `parameter String`
- **floating point type** ← `Real`
- **start value** ← `start= -2003`
- **name + default value** ← `massLossRate=0.000277;`
- **mathematical equation (acausal)** ← `(thrust-mass*gravity)/mass = acceleration;`
- **differentiation with regards to time** ← `der(velocity)`

Celestial Body Class

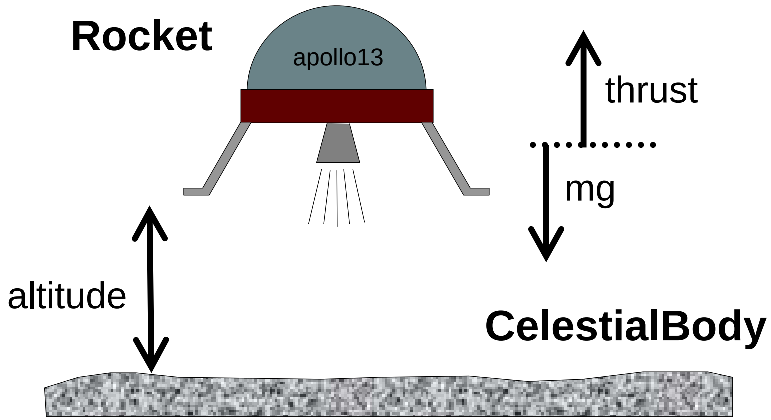
A class declaration creates a *type name* in Modelica

```
class CelestialBody
  constant  Real    g = 6.672e-11;
  parameter Real    radius;
  parameter String  name;
  parameter Real    mass;
end CelestialBody;
```

An *instance* of the class can be declared by *prefixing* the type name to a variable name

```
...
CelestialBody moon;
...
```

The declaration states that **moon** is a variable containing an object of type **CelestialBody**



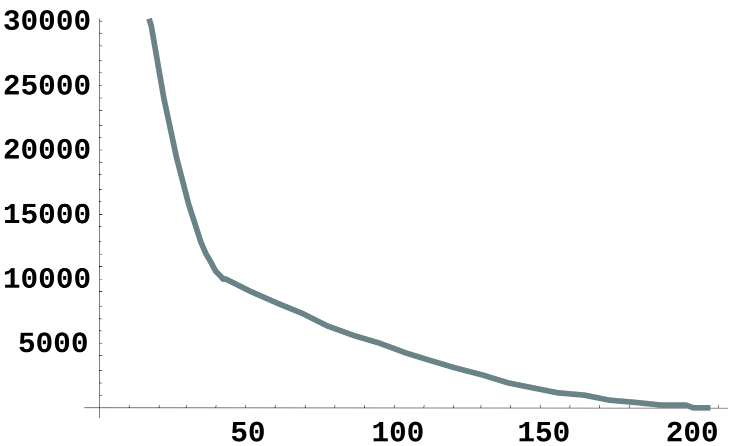
$$apollo.gravity = \frac{moon.g \cdot moon.mass}{(apollo.altitude + moon.radius)^2}$$

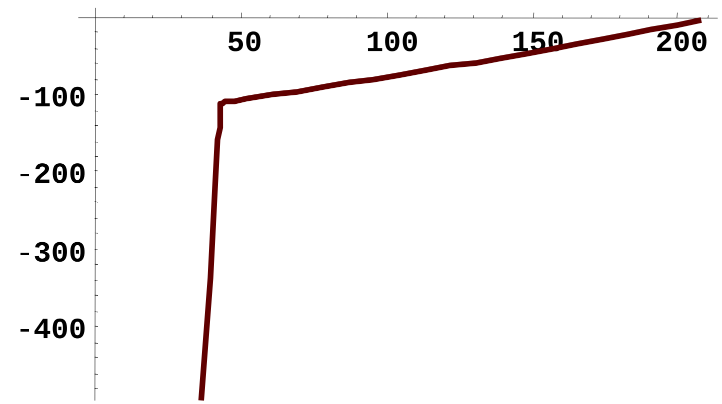
only access inside the class

access by dot notation outside the class

```
class MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
protected
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
public
  Rocket        apollo(name="apollo13");
  CelestialBody   moon(name="moon",mass=7.382e22,radius=1.738e6);
equation
  apollo.thrust = if (time < thrustDecreaseTime) then force1
                  else if (time < thrustEndTime) then force2
                  else 0;
  apollo.gravity=moon.g*moon.mass/(apollo.altitude+moon.radius)^2;
end MoonLanding;
```

```
simulate(MoonLanding, stopTime=230)
plot(apollo.altitude, xrange={0,208})
plot(apollo.velocity, xrange={0,208})
```





It starts at an altitude of 59404 (not shown in the diagram) at time zero, gradually reducing it until touchdown at the lunar surface when the altitude is zero

The rocket initially has a high negative velocity when approaching the lunar surface. This is reduced to zero at touchdown, giving a smooth landing parent class to Color

restricted kind of class without equations

child class or subclass

keyword denoting inheritance

```
record ColorData
  parameter Real red = 0.2;
  parameter Real blue = 0.6;
  Real   green;
end ColorData;

class Color
  extends ColorData;
equation
  red + blue + green = 1;
end Color;
```

```
class ExpandedColor
  parameter Real red=0.2;
  parameter Real blue=0.6;
  Real green;
equation
  red + blue + green = 1;
end ExpandedColor;
```

Data and behavior: field declarations, equations, and certain other contents are copied into the subclass

Inheriting definitions

```
record ColorData
  parameter Real red = 0.2;
  parameter Real blue = 0.6;
  Real   green;
end ColorData;

class ErrorColor
  extends ColorData;
  parameter Real blue = 0.6;
  parameter Real red = 0.3;
equation
  red + blue + green = 1;
end ErrorColor;
```

Legal!
Identical to the inherited field blue

Inheriting multiple identical definitions results in only one definition

Illegal!
Same name, but different value

Inheriting multiple different definitions of the same item is an error

Inheritance of Equations

```
class Color
  parameter Real red=0.2;
  parameter Real blue=0.6;
  Real green;
equation
  red + blue + green = 1;
end Color;
```

```
class Color2  // OK!
  extends Color;
equation
  red + blue + green = 1;
end Color2;
```

```
class Color3  // Error!
   extends Color;
equation
  red + blue + green = 1.0;
  // also inherited: red + blue + green = 1;
end Color3;
```

`Color` is identical to `Color2`

Same equation twice leaves one copy when inheriting

`Color3` is overdetermined

Different equations means two equations!

Multiple Inheritance is fine – inheriting both geometry and color

```
class Color
  parameter Real red=0.2;
  parameter Real blue=0.6;
  Real green;
equation
  red + blue + green = 1;
end Color;
```

```
class Point
  Real x;
  Real y,z;
end Point;
```

```
class ColoredPoint
  extends Point;
  extends Color;
end ColoredPoint;
```

multiple inheritance

Equivalent to

```
class ColoredPointWithoutInheritance
  Real x;
  Real y, z;
  parameter Real red = 0.2;
  parameter Real blue = 0.6;
  Real green;
equation
  red + blue + green = 1;
end ColoredPointWithoutInheritance;
```

Only one copy of multiply inherited class `Point` is kept

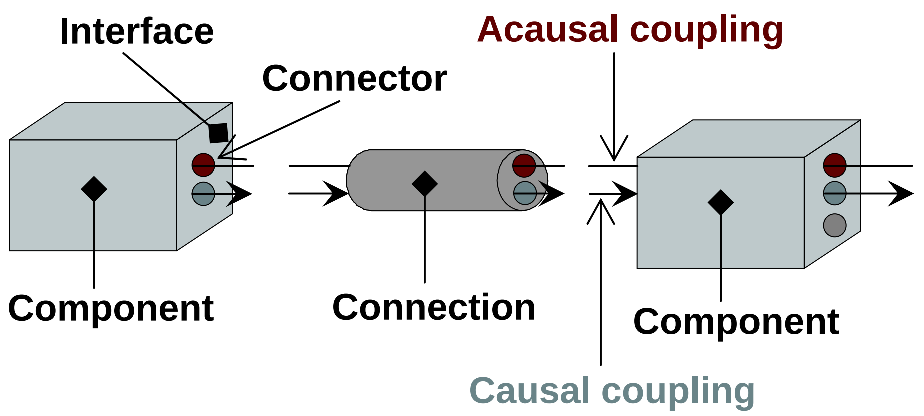
```
class Point
  Real x;
  Real y;
end Point;
```

```
class VerticalLine
  extends Point;
  Real vlength;
end VerticalLine;
```

Diamond Inheritance

```
class HorizontalLine
  extends Point;
  Real hlength;
end HorizontalLine;
```

```
class Rectangle
  extends VerticalLine;
  extends HorizontalLine;
end Rectangle;
```



A component class should be defined *independently of the environment,* very essential for *reusability*

A component may internally consist of other components, i.e. *hierarchical* modeling

Complex systems usually consist of large numbers of *connected* components

Connectors are instances of *connector classes*

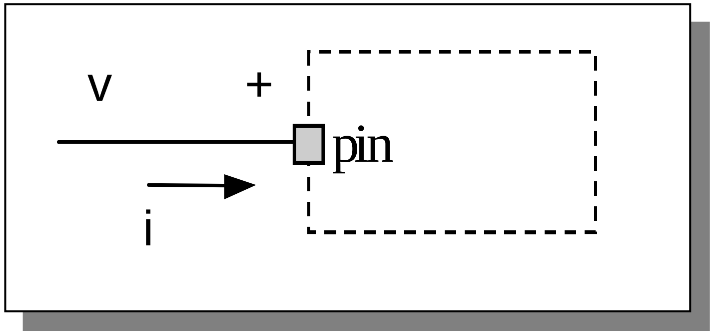
electrical connector

connector class

keyword `flow` indicates that currents of connected pins sum to zero.

an instance `pin` of class `Pin`

```
connector Pin
  Voltage      v;
  flow Current i;
end Pin;

Pin pin;
```

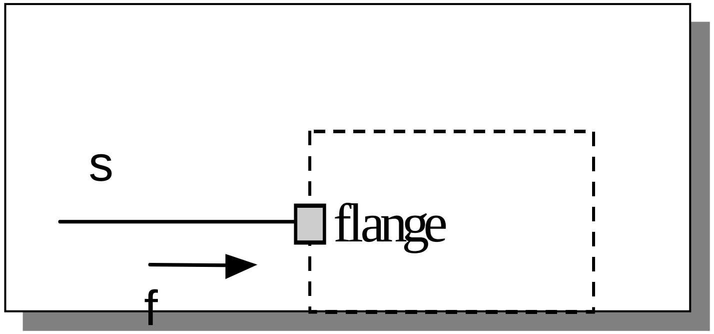


mechanical connector

connector class

an instance `flange` of class `Flange`

```
connector Flange
  Position    s;
  flow Force  f;
end Flange;

Flange flange;
```



Two kinds of variables in connectors:

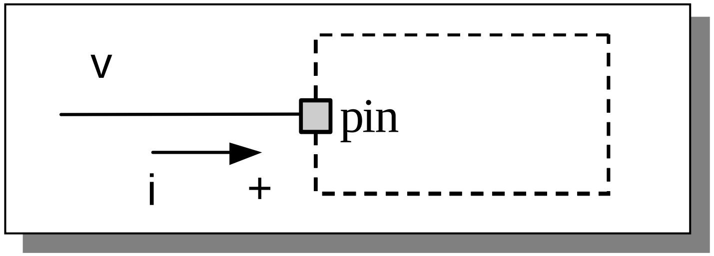
- *Non-flow variables potential* or energy level
- *Flow variables* represent some kind of flow

Coupling

- *Equality coupling*, for non-`flow` variables
- *Sum-to-zero coupling*, for `flow` variables

The value of a `flow` variable is *positive* when the current or the flow is *into* the component

positive flow direction:



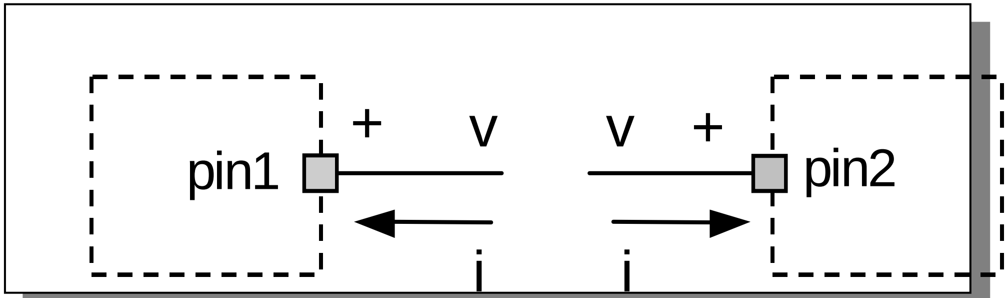
Classes Based on Energy Flow

Domain Type	Potential	Flow	Carrier	Modelica Library
Electrical	Voltage	Current	Charge	`Electrical.Analog`
Translational	Position	Force	Linear momentum	`Mechanical.Translational`
Rotational	Angle	Torque	Angular momentum	`Mechanical.Rotational`
Magnetic	Magnetic potential	Magnetic flux rate	Magnetic flux	
Hydraulic	Pressure	Volume flow	Volume	`HyLibLight`
Heat	Temperature	Heat flow	Heat	`HeatFlow1D`
Chemical	Chemical potential	Particle flow	Particles	Under construction
Pneumatic	Pressure	Mass flow	Air	`PneuLibLight`

Connections between connectors are realized as *equations* in Modelica

```
connect(connector1,connector2)
```

The two arguments of a `connect`-equation must be references to *connectors*, either to be declared directly *within* the *same class* or be *members* of one of the declared variables in that class




```
Pin pin1,pin2;
//A connect equation
//in Modelica:
connect(pin1,pin2);
```

Corresponds to

```
pin1.v = pin2.v;
pin1.i + pin2.i =0;
```

```
Pin pin1,pin2;
//A connect equation
//in Modelica
connect(pin1,pin2);
```

Corresponds to

```
pin1.v = pin2.v;
pin1.i + pin2.i =0;
```

Multiple connections are possible:

```
connect(pin1,pin2); connect(pin1,pin3); ... connect(pin1,pinN);
```

Each primitive connection set of nonflow variables is used to generate equations of the form:

$$v_1 = v_2 = v_3 = \ldots v_n$$

Each primitive connection set of flow variables is used to generate *sum-to-zero* equations of the form:

$$i_1 + i_2 + \ldots (-i_k) + \ldots i_n = 0$$

Acausal, Causal, and Composite Connections

Two *basic* and one *composite* kind of connection in Modelica

- Acausal connections
- Causal connections, also called signal connections
- Composite connections, also called structured connections, composed of basic or composite connections

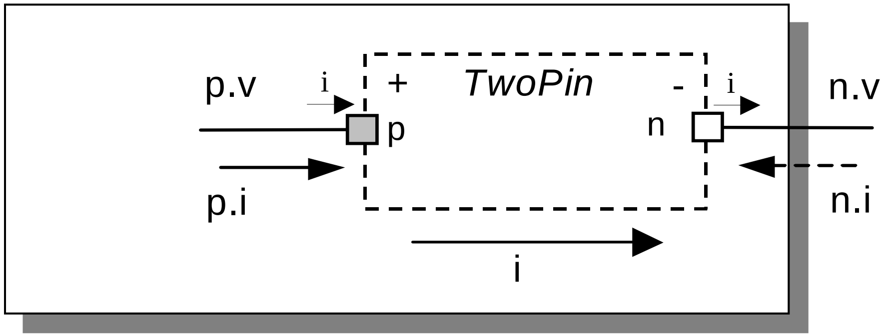
connector class

fixed causality

```
connector OutPort
  output Real signal;
end OutPort
```

Common Component Structure

The base class `TwoPin` has two connectors p and n for positive and negative pins respectively



partial class (cannot be instantiated)

positive pin

negative pin

```
partial model TwoPin
  Voltage     v
  Current     i
  Pin p;
  Pin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
// TwoPin is same as OnePort in
// Modelica.Electrical.Analog.Interfaces
```

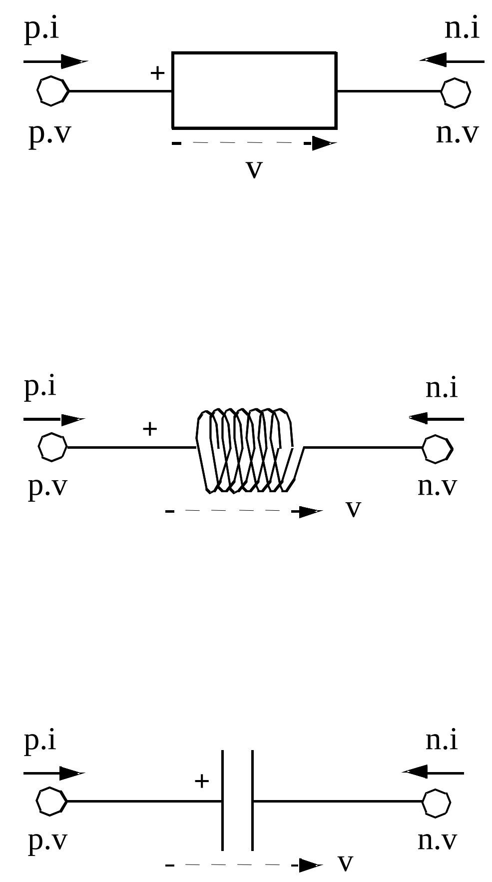
electrical connector class

```
connector Pin
  Voltage        v;
  flow Current   i;
end Pin;
```

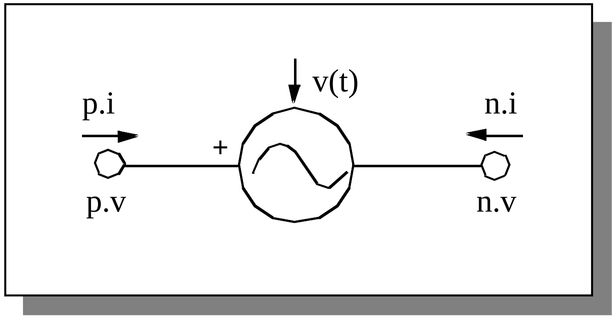
```
model Resistor ”Ideal electrical resistor”
  extends TwoPin;
  parameter Real R;
equation
  R*i = v;
end Resistor;
```

```
model Inductor ”Ideal electrical inductor”
  extends TwoPin;
  parameter Real L ”Inductance”;
equation
  L*der(i) = v;
end Inductor;
```

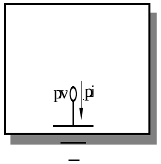
```
model Capacitor ”Ideal electrical capacitor”
  extends TwoPin;
  parameter Real C ;
equation
  i=C*der(v);
end Capacitor;
```

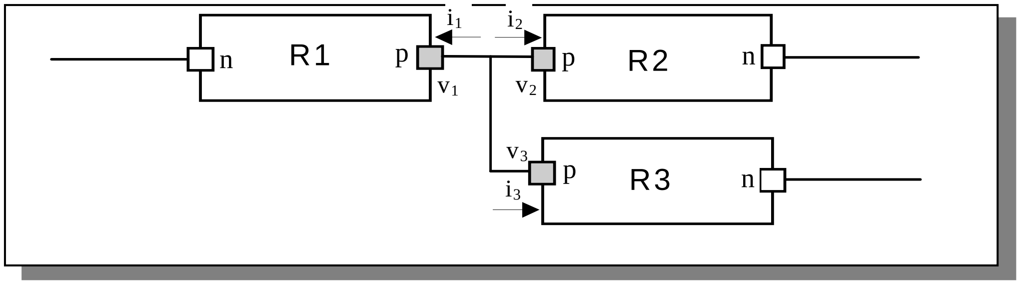


```
model Source
   extends TwoPin;
   parameter Real A,w;
equation
   v = A*sin(w*time);
end Resistor;
```




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






```
model ResistorCircuit
  Resistor R1(R=100);
  Resistor R2(R=200);
  Resistor R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end ResistorCircuit;
```

Corresponds to

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

Modelica Standard Library (called Modelica) is a standardized predefined package developed by Modelica Association

- It can be used freely for both commercial and noncommercial purposes under the conditions of *The Modelica License*.

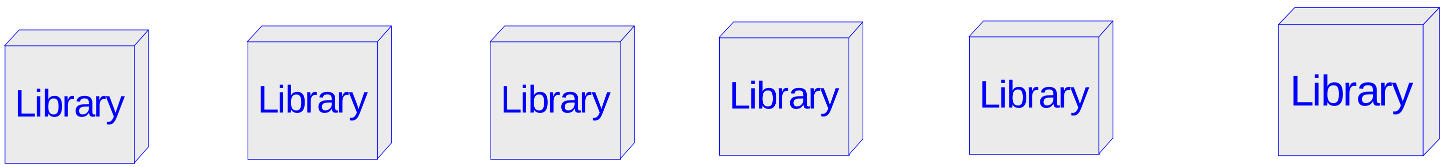
- Modelica libraries are available online including documentation and source code from http://www.modelica.org/library/library.html

Modelica Standard Library cont'

Modelica Standard Library contains components from various application areas, with the following sublibraries:

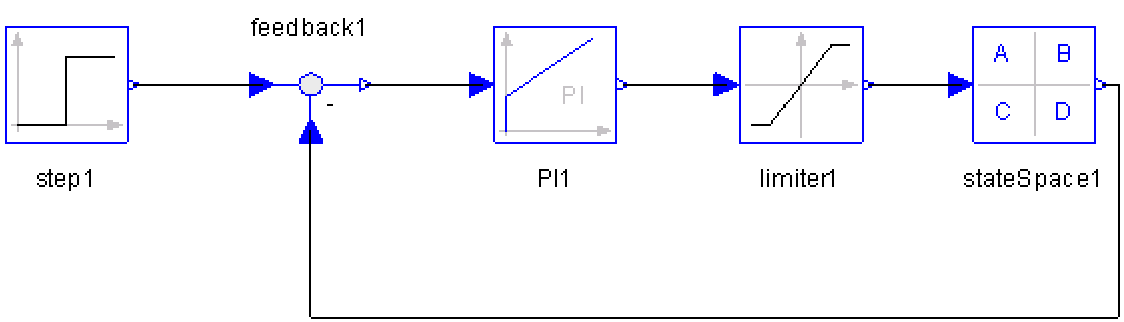
- Blocks Library for basic input/output control blocks
- Constants Mathematical constants and constants of nature
- Electrical Library for electrical models
- Icons Icon definitions
- Math Mathematical functions
- Mechanics Library for mechanical systems
- Media Media models for liquids and gases
- SIunits Type definitions based on SI units according to ISO 31-1992
- Stategraph Hierarchical state machines (analogous to Statecharts)
- Thermal Components for thermal systems
- Utilities Utility functions especially for scripting

This library contains input/output blocks to build up block diagrams.

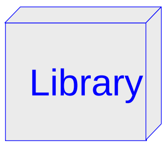


Continuous Sources Math Interfaces NonLinear Discrete

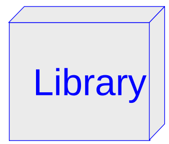
Example:



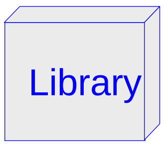
Electrical components for building analog, digital, and multiphase circuits



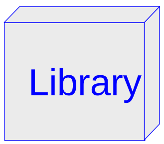
Analog



Digital

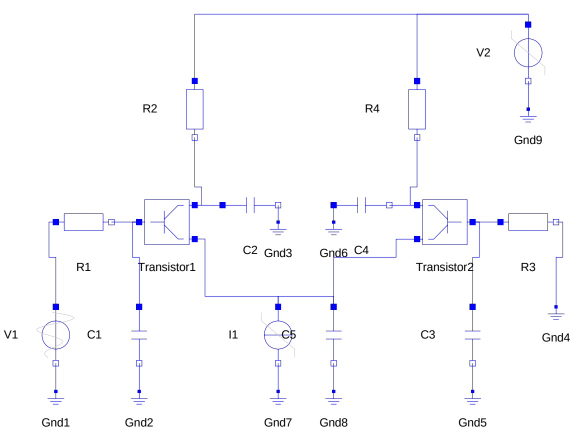


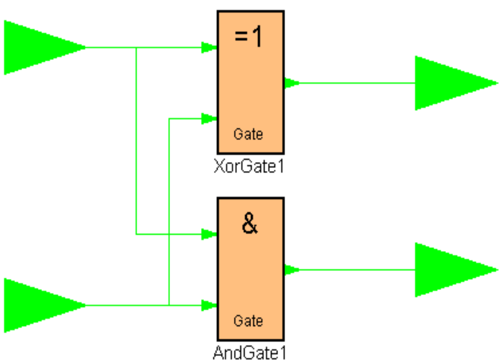
Machines

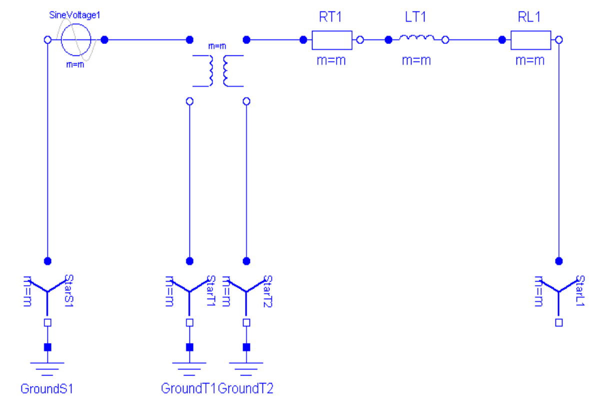


MultiPhase

Examples:



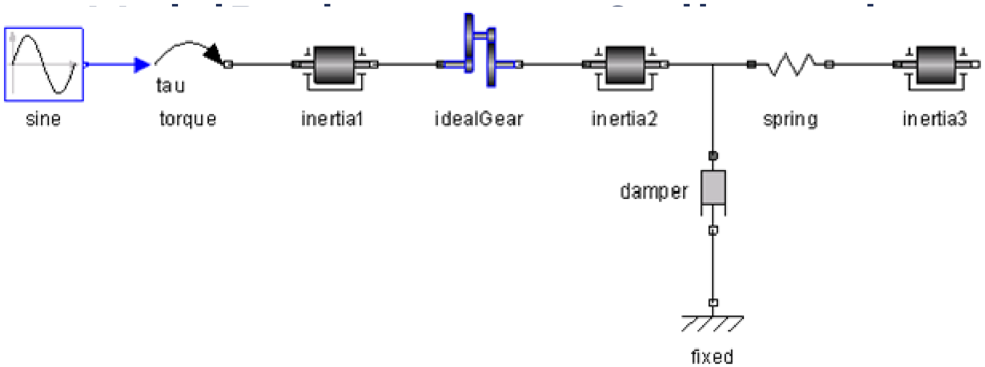


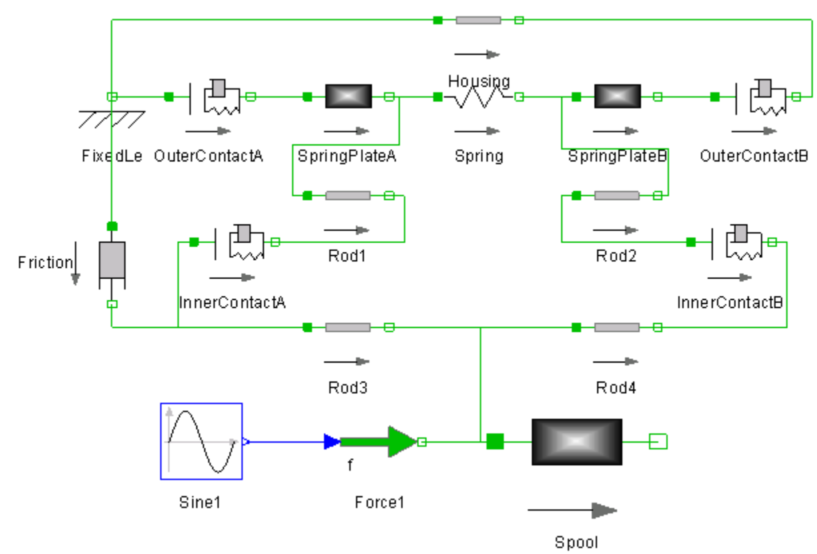


Package containing components for mechanical systems

Subpackages:

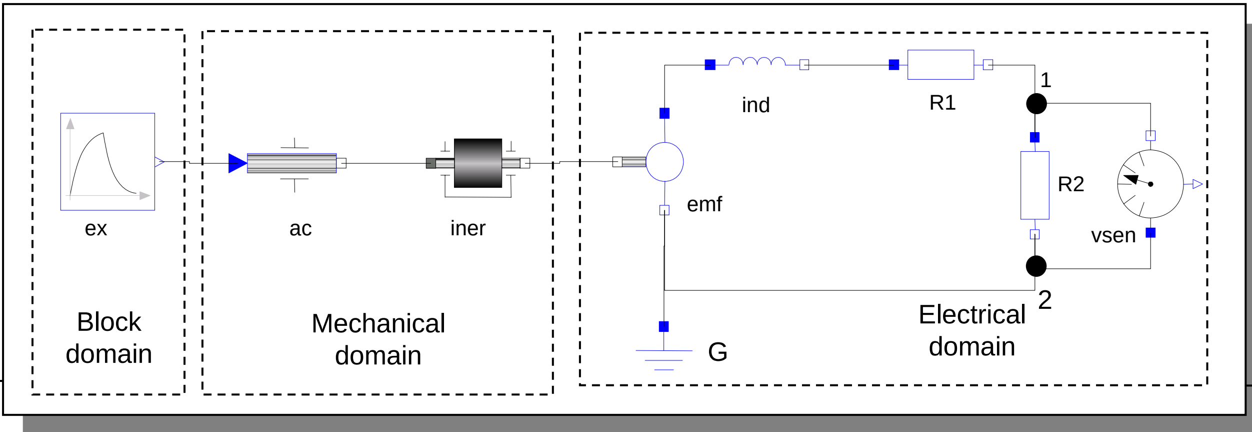
- Rotational 1-dimensional rotational mechanical components
- Translational 1-dimensional translational mechanical components





Connecting Components from Multiple Domains

- **Block domain**
- **Mechanical domain**
- **Electrical domain**

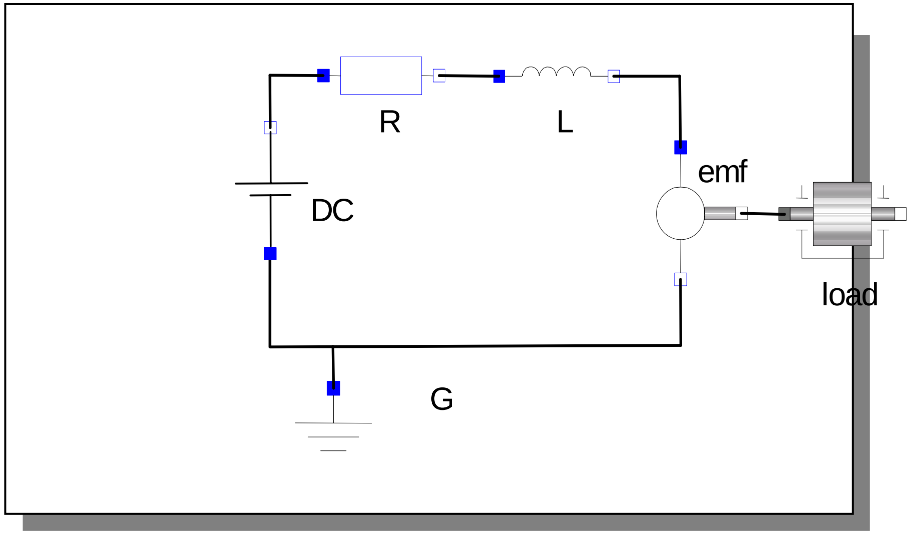



```
model Generator
  Modelica.Mechanics.Rotational.Accelerate ac;
  Modelica.Mechanics.Rotational.Inertia iner;
  Modelica.Electrical.Analog.Basic.EMF emf(k=-1);
  Modelica.Electrical.Analog.Basic.Inductor ind(L=0.1);
  Modelica.Electrical.Analog.Basic.Resistor R1,R2;
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.Sensors.VoltageSensor vsens;
  Modelica.Blocks.Sources.Exponentials ex(riseTime={2},riseTimeConst={1});
equation
  connect(ac.flange_b, iner.flange_a); connect(iner.flange_b, emf.flange_b);
  connect(emf.p, ind.p); connect(ind.n, R1.p); connect(emf.n, G.p);
  connect(emf.n, R2.n); connect(R1.n, R2.p); connect(R2.p, vsens.n);
  connect(R2.n, vsens.p); connect(ex.outPort, ac.inPort);
end Generator;
```

DCMotor Multi-Domain (Electro-Mechanical)

A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

```
model DCMotor
   Resistor R(R=100);
   Inductor L(L=100);
   VsourceDC DC(f=10);
   Ground G;
   EMF emf(k=10,J=10, b=2);
   Inertia load;
equation
   connect(DC.p,R.n);
   connect(R.p,L.n);
   connect(L.p, emf.n);
   connect(emf.p, DC.n);
   connect(DC.n,G.p);
   connect(emf.flange,load.flange);
end DCMotor;
```



ModelicaML

A UML profile for Modelica

ModelicaML - a UML profile for Modelica

- Supports modeling with all Modelica constructs i.e. restricted classes, equations, generics, discrete variables, etc.
- Multiple aspects of a system being designed are supported
  - system development process phases such as **requirements** analysis, design, implementation, verification, validation and integration.
- Supports mathematical modeling with equations (to specify system behavior). Algorithm sections are also supported.
- Simulation diagrams are introduced to configure, model and document simulation parameters and results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion and back.

SysML vs. Modelica

- SysML
  - Pros
    - Can model all aspects of complex system design
  - Cons
    - Precise behavior can be described *but not simulated (executed)*

- Modelica
  - Pros
    - Precise behavior *can be described and simulated*
  - Cons
    - Cannot model all aspects of complex system design, i.e. *requirements*, inheritance diagrams, etc

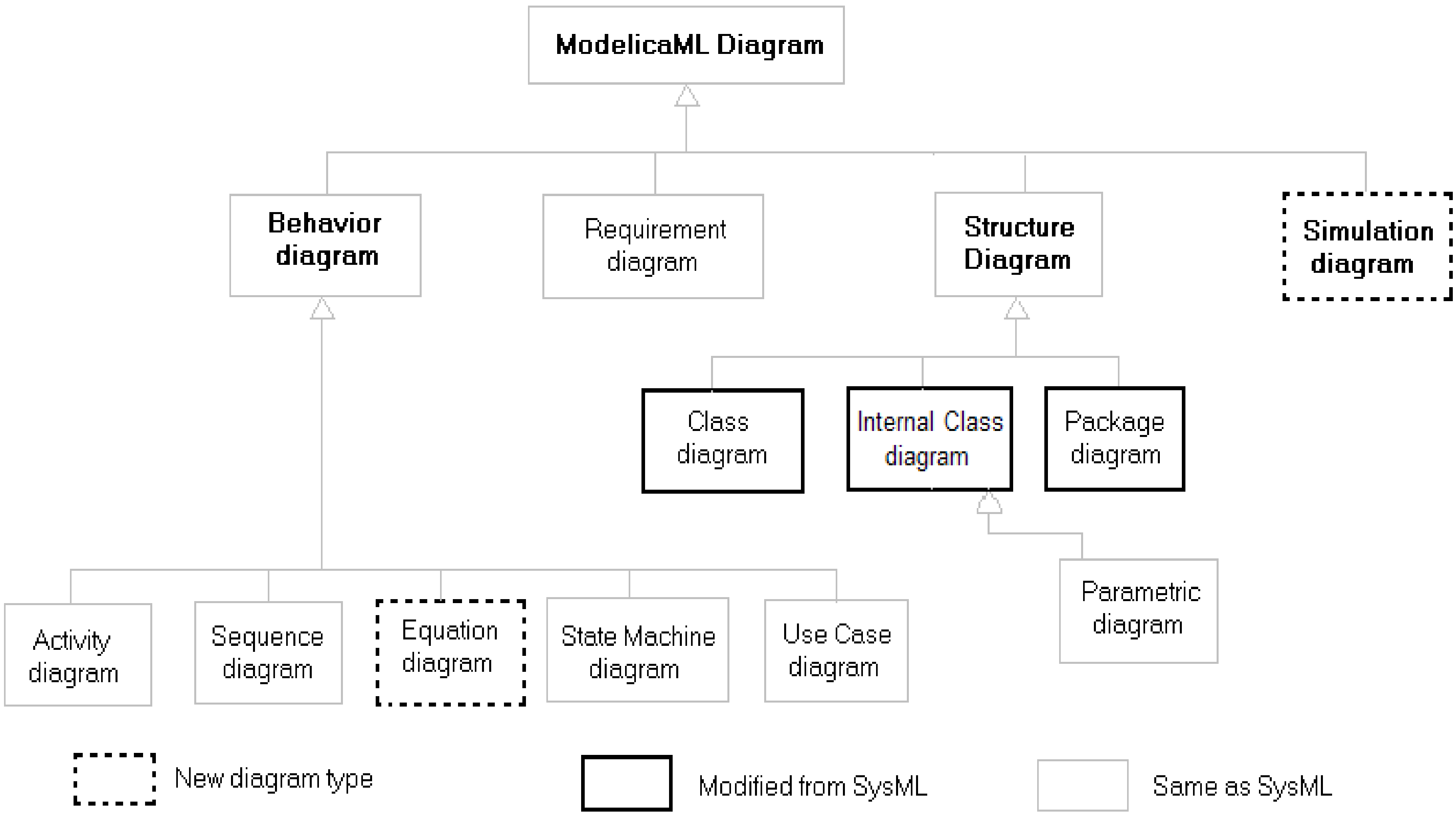
ModelicaML - Purpose

- Targeted to Modelica and SysML users

- Provide a SysML/UML view of Modelica for
  - Documentation purposes
  - Language understanding
  - Better software engineering

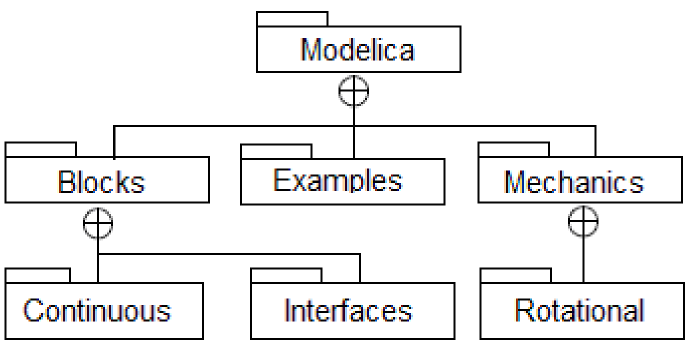
- To extend Modelica with additional design capabilities (requirements modeling, inheritance diagrams, etc)

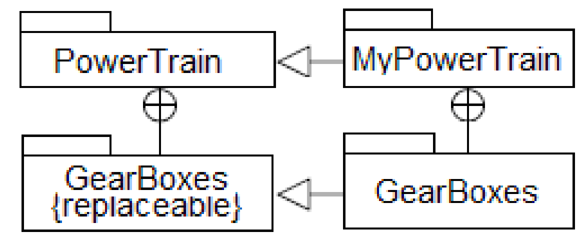
- To support translation between Modelica and SysML models via XMI

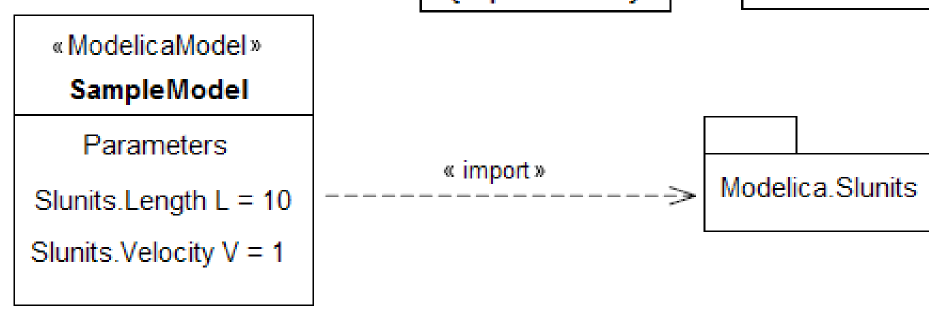

ModelicaML Diagram
Behavior diagram
Requirement diagram
Structure Diagram
Simulation diagram
Class diagram
Internal Class diagram
Package diagram
Parametric diagram
Activity diagram
Sequence diagram
Equation diagram
State Machine diagram
Use Case diagram
New diagram type
Modified from SysML
Same as SysML

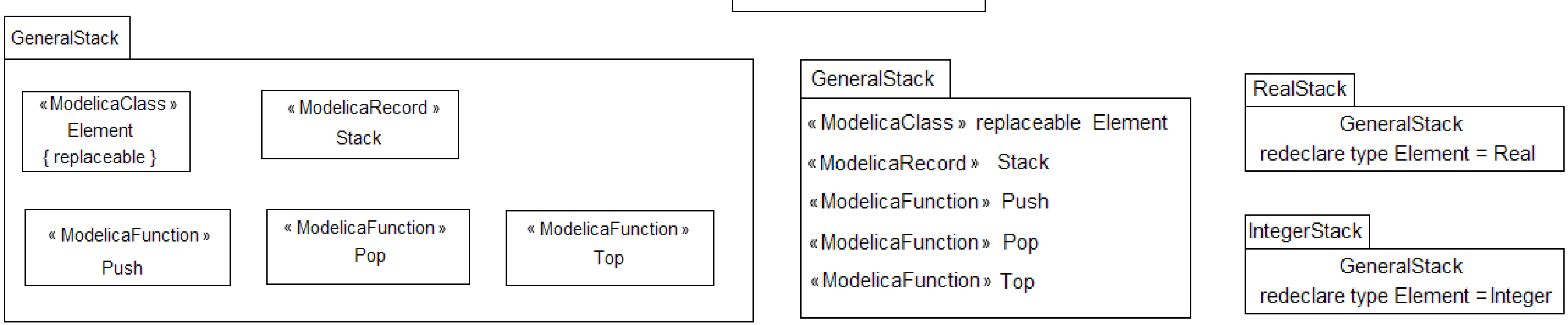
ModelicaML – Package Diagram

- The Package Diagram groups logically connected user defined elements into packages.
- The primarily purpose of this diagram is to support the specifics of the Modelica packages.



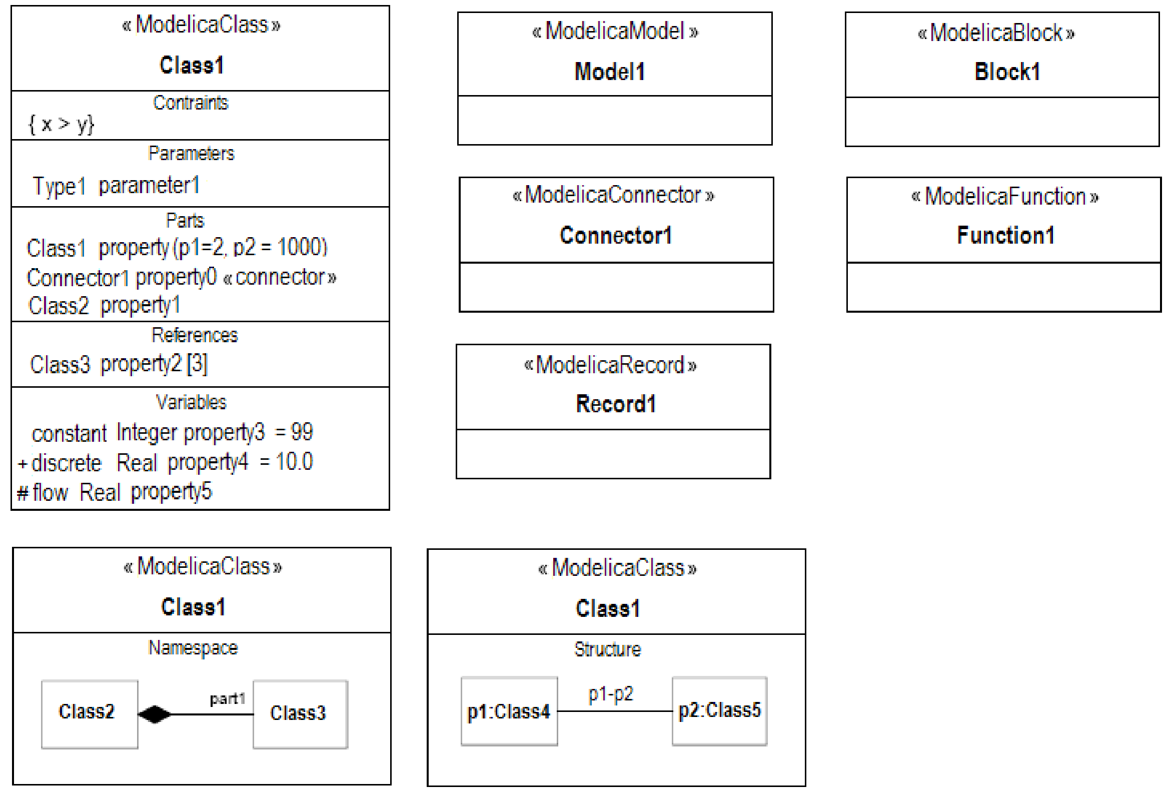






ModelicaML – Class Diagram

- ModelicaML provides extensions to SysML in order to support the full set of Modelica constructs.
- ModelicaML defines unique class definition types ModelicaClass, ModelicaModel, ModelicaBlock, ModelicaConnector, ModelicaFunction and ModelicaRecord that correspond to `class`, `model`, `block`, `connector`, `function` and `record` restricted Modelica classes.
- Modelica specific restricted classes are included because a modeling tool needs to impose their semantic restrictions (for example a record cannot have equations, etc).

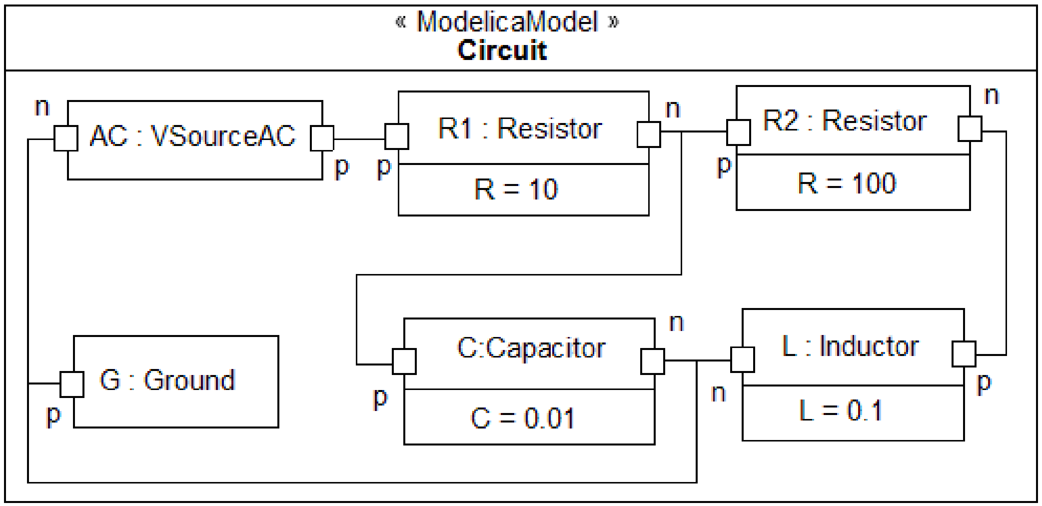


Class Diagram defines Modelica classes and relationships between classes, like generalizations, association and dependencies

ModelicaML - Internal Class Diagram

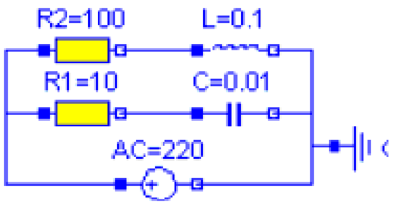
- Internal Class Diagram shows the internal structure of a class in terms of parts and connections

ModelicaML Internal Class Diagram



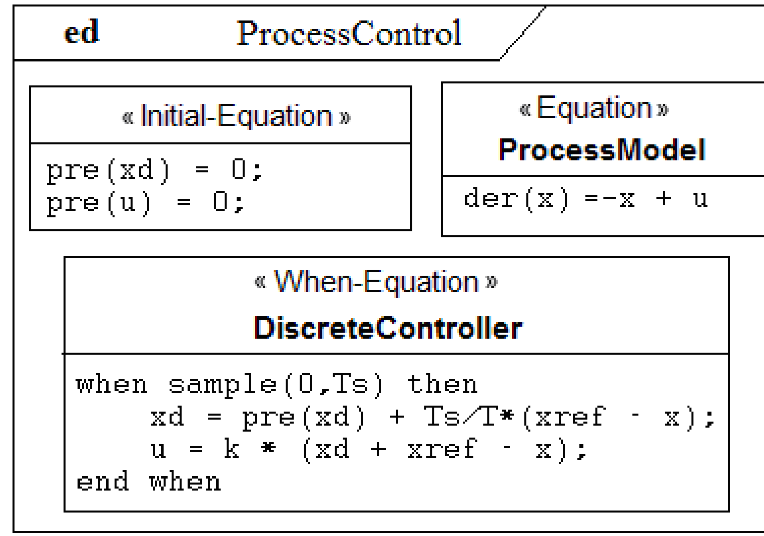

```
model Circuit
   Resistor  R1(R=10);
   Capacitor C(C=0.01);
   Resistor  R2(R=100);
   Inductor  L(L=0.1);
   VSourceAC AC;
   Ground    G;
equation
   connect (AC.p, R1.p);
   connect (R1.n, C.p);
   connect (C.n, AC.n);
   connect (R1.n, R2.p);
   connect (R2.n, L.p);
   connect (L.n,  C.n);
   connect (AC.n, G.p);
end Circuit;
```

Modelica Connection Diagram

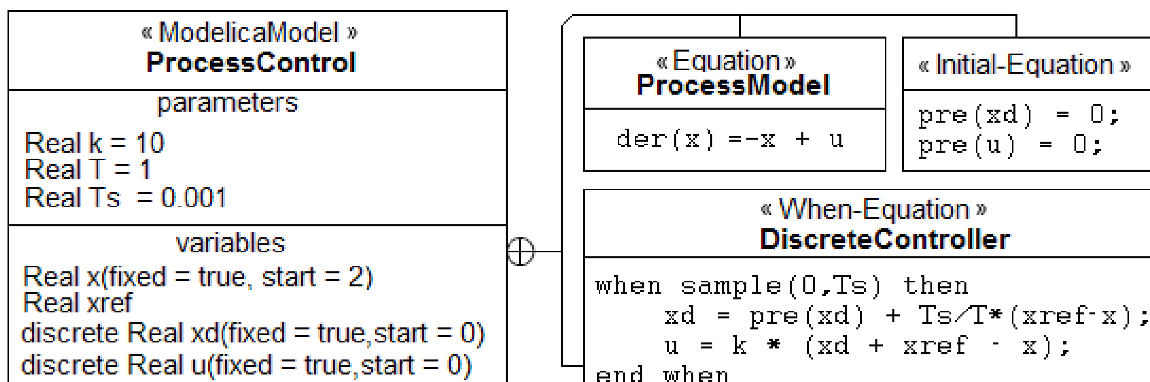


ModelicaML – Equation Diagram

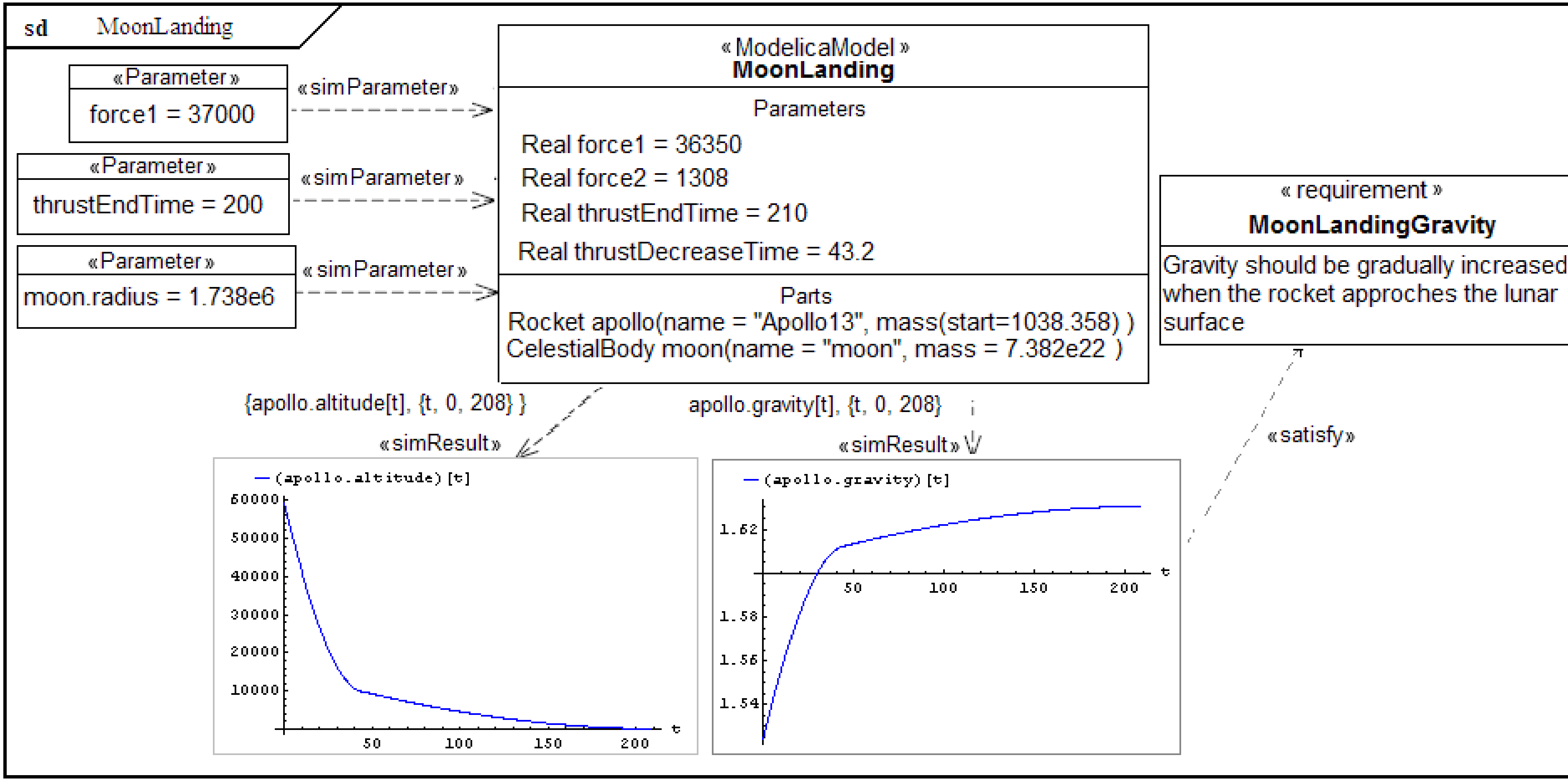
- behavior is specified using Equation Diagrams
- all Modelica equations have their specific diagram:
  - initial, when, for, if equations




```
model ProcessControl
  parameter Real k=10,T=1;
  parameter Real Ts=0.001;
  Real x(fixed=true,start=2);
  Real xref;
  discrete Real xd(fixed=true,start=0);
  discrete Real u(fixed=true,start=0);
equation
  der(x) = =-x + u; // Process model
  // Discrete PI Controller
  when sample(0,Ts) then
    xd=pre(xd)+Ts/T*(xref-x);
    u=k*(xd + xref - x);
  end when;
initial equation
  pre(xd) = 0; pre(u) = 0;
end ProcessControl;
```



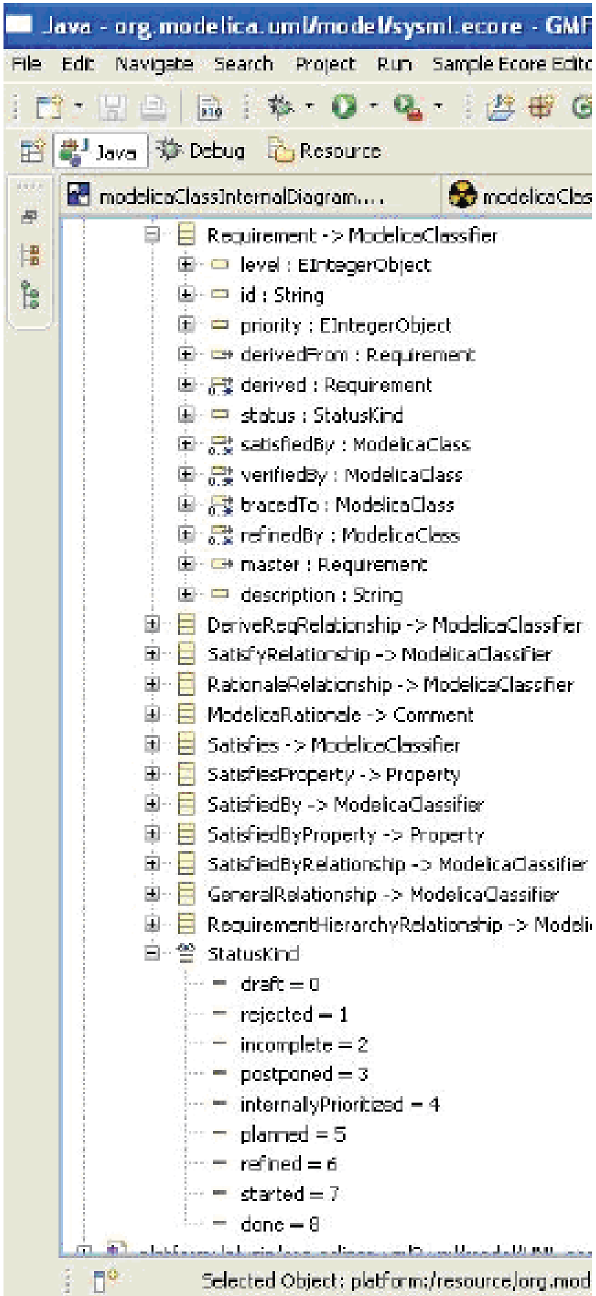
ModelicaML – Simulation Diagram



- Used to model, configure and document simulation parameters and results
- Simulation diagrams can be integrated with any Modelica modeling and simulation environment (OpenModelica)

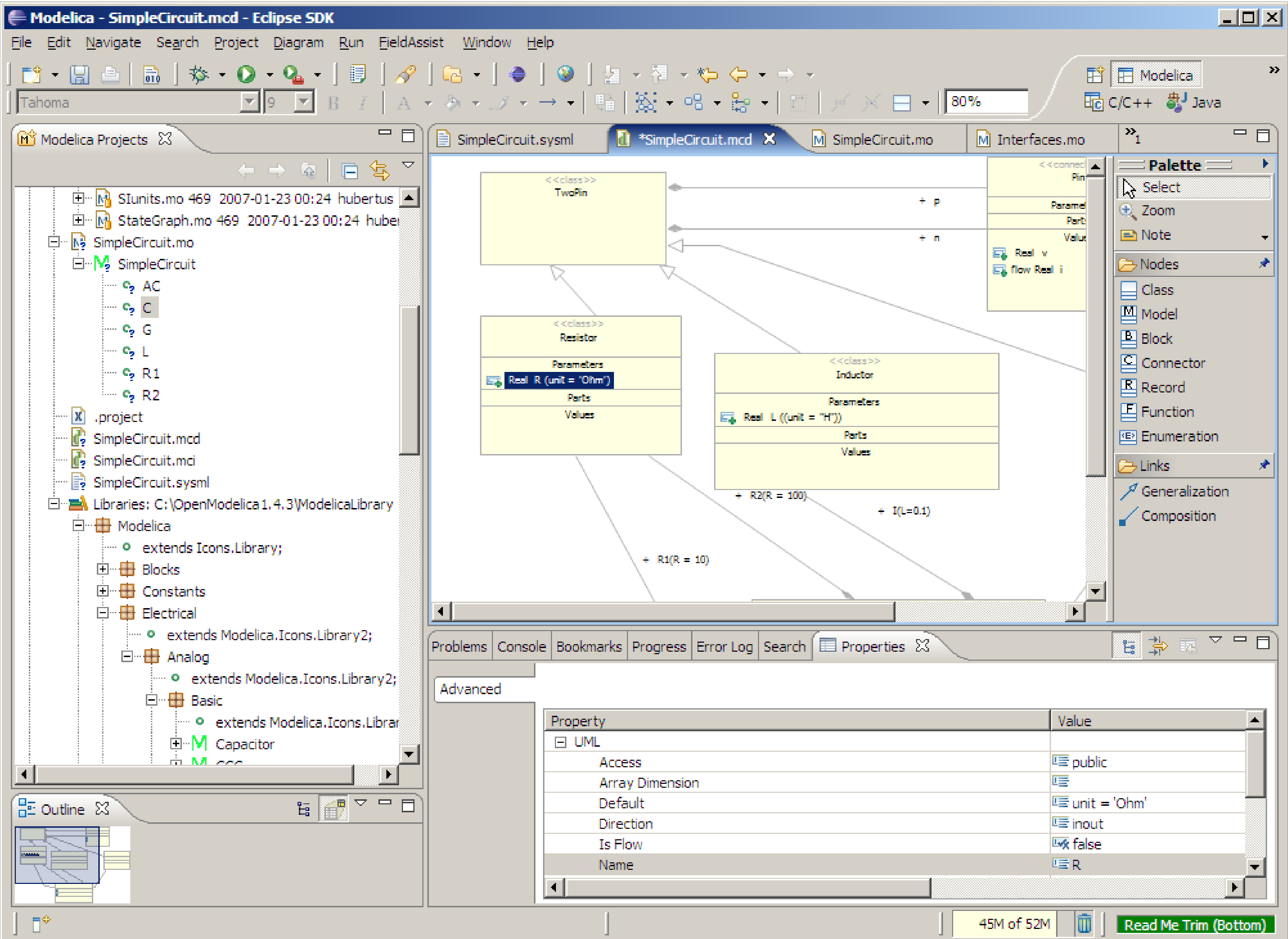
Eclipse

Integrated Environments for Modelica

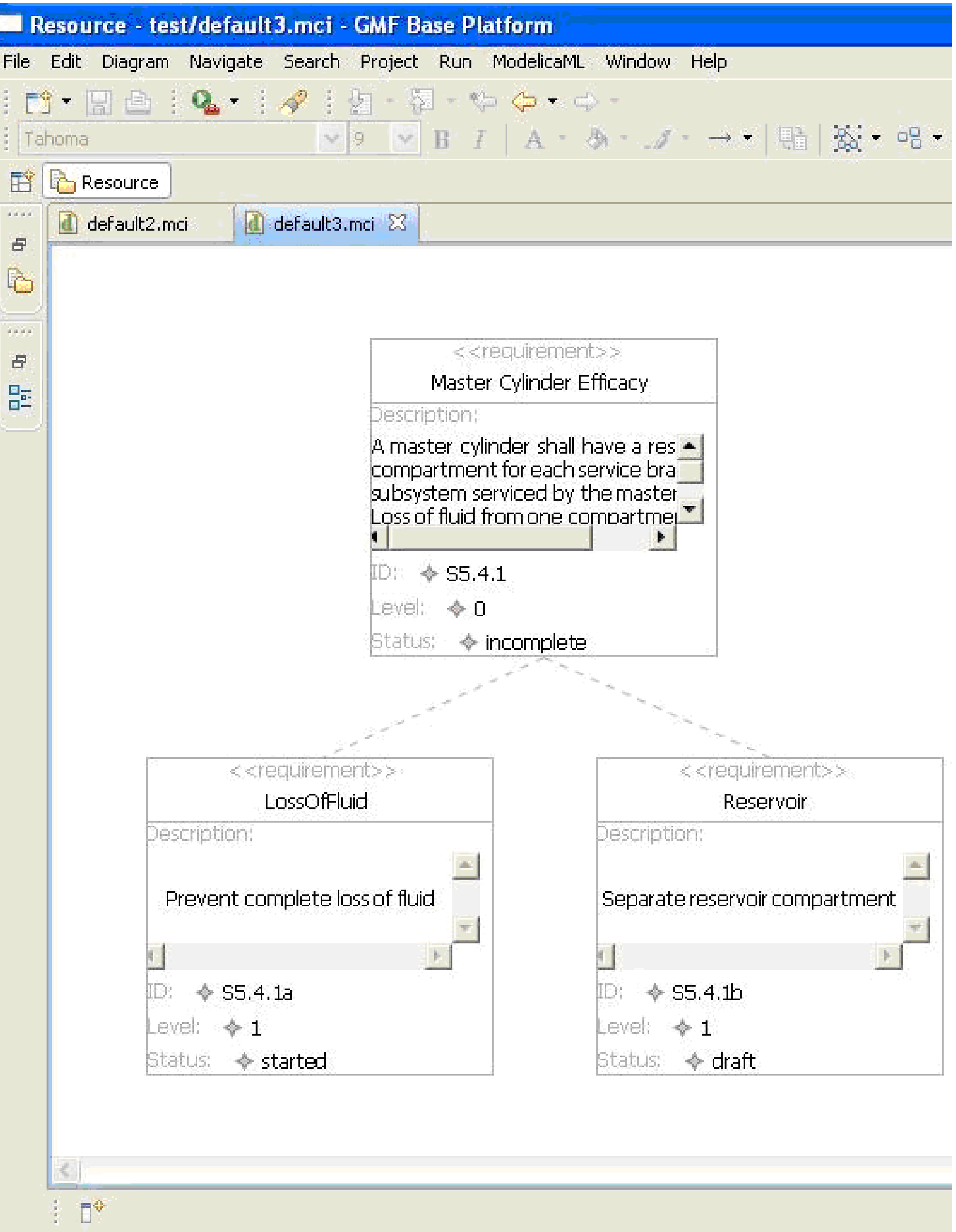


- EMF – Eclipse Modeling Framework
- GMF – Graphical Modeling Framework
- The UML2 Eclipse meta-model implementation

Eclipse environment for ModelicaML

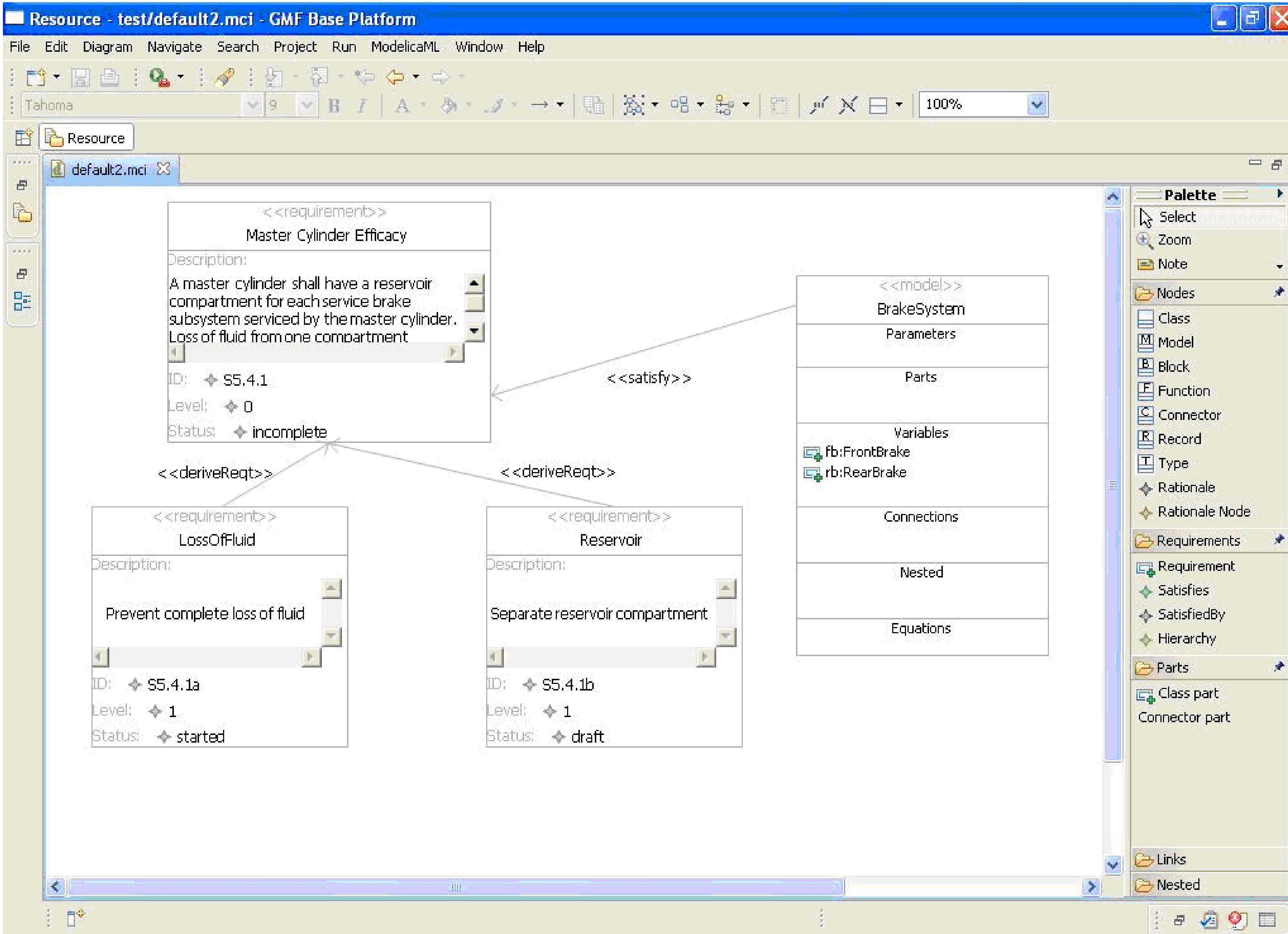


Requirements Modeling

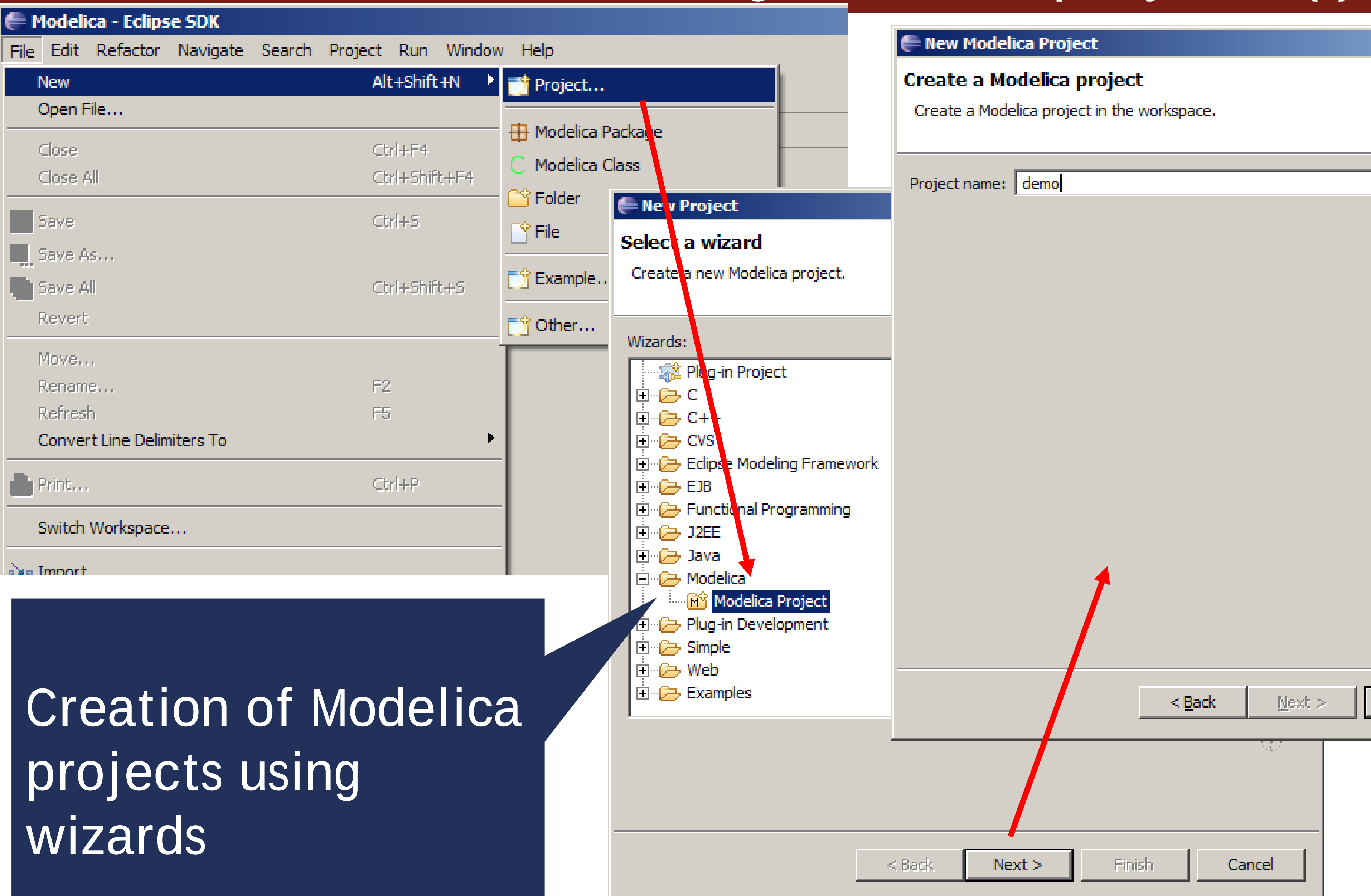


- Requirements
  - can be *modeled hierarchically*
  - can be *traced*
  - can be *linked* with other ModelicaML models
  - can be *queried* with respect of their attributes and links (coverage)

Requirements Modeling in Eclipse

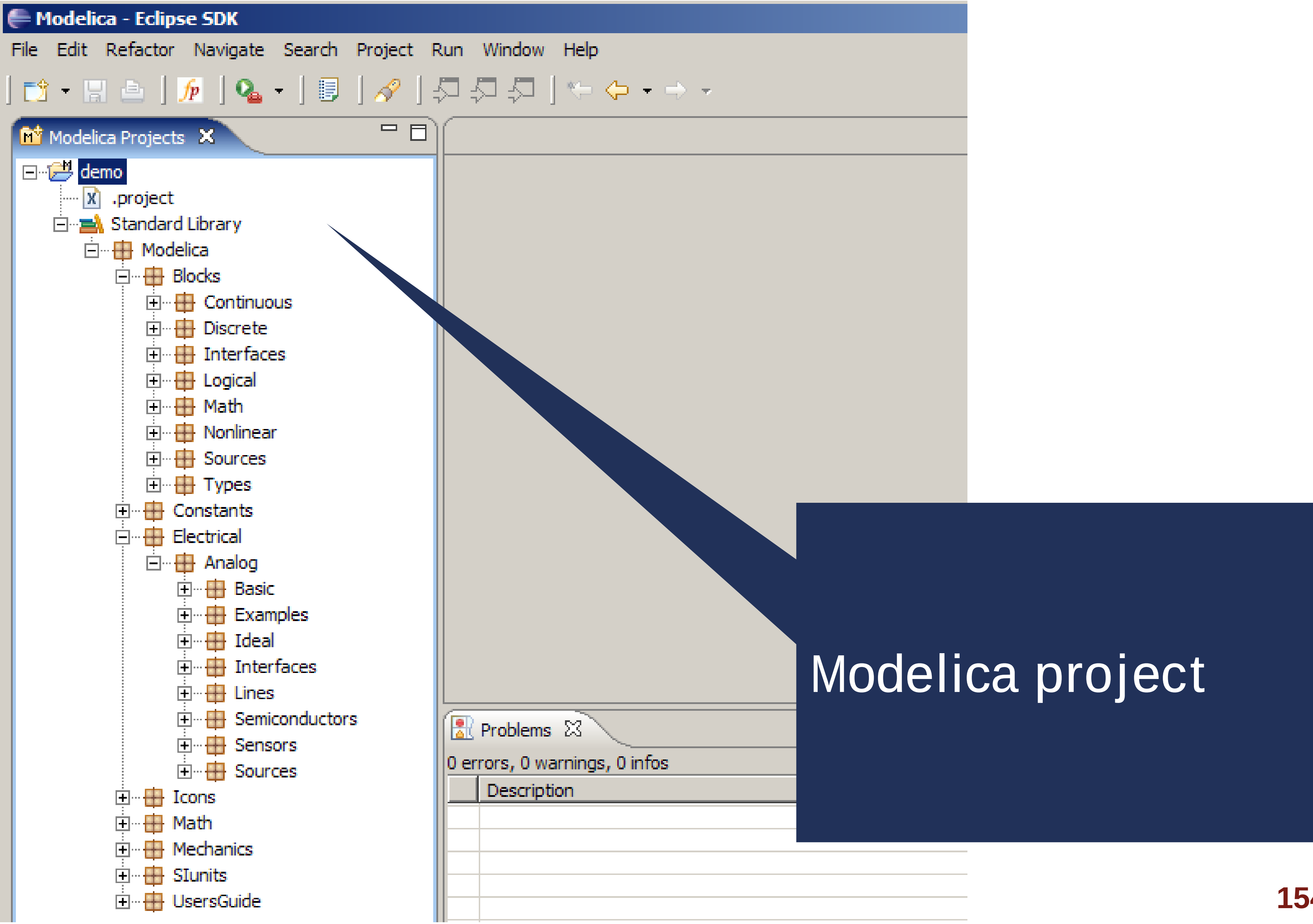


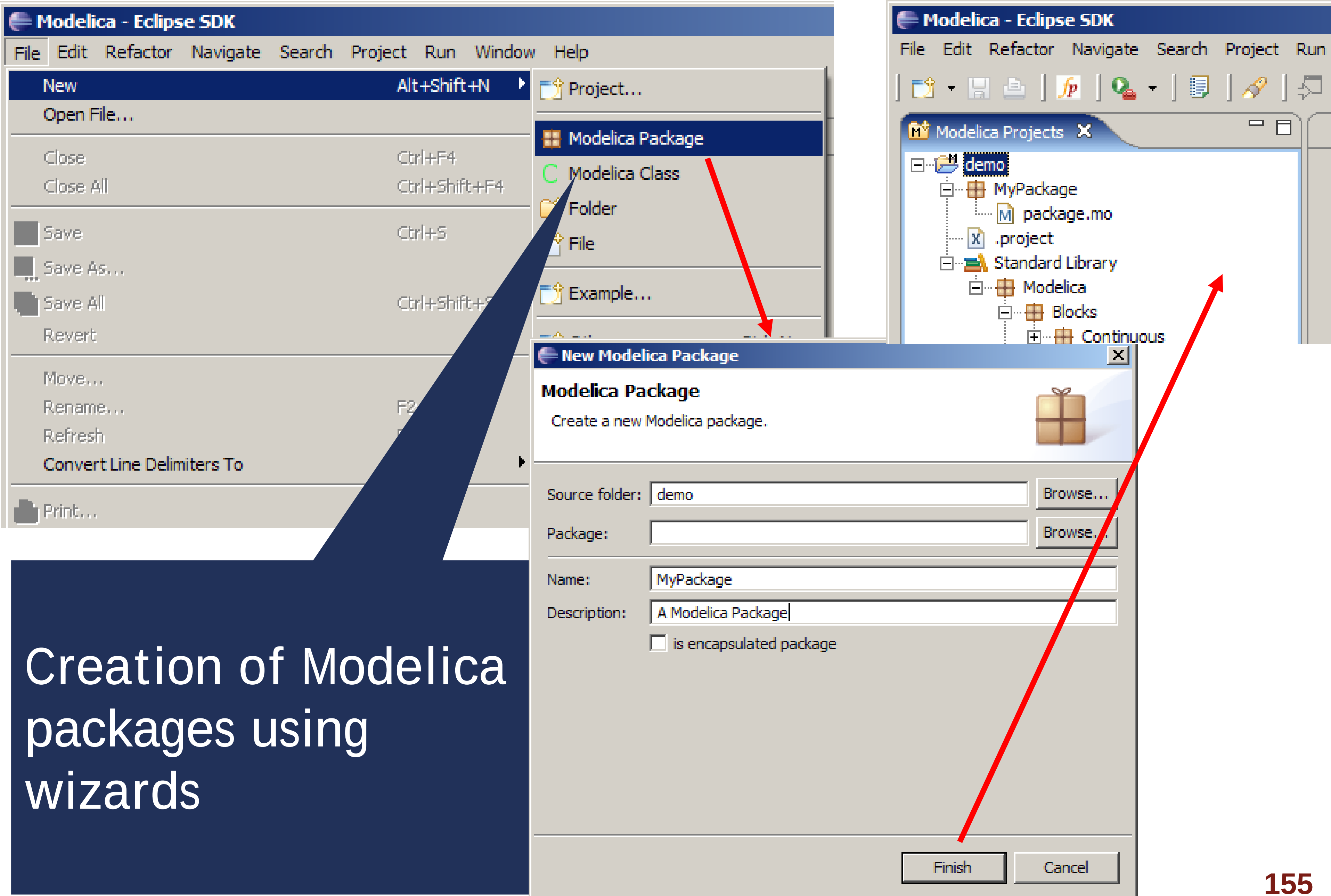
Creating Modelica projects (I)



Creation of Modelica projects using wizards

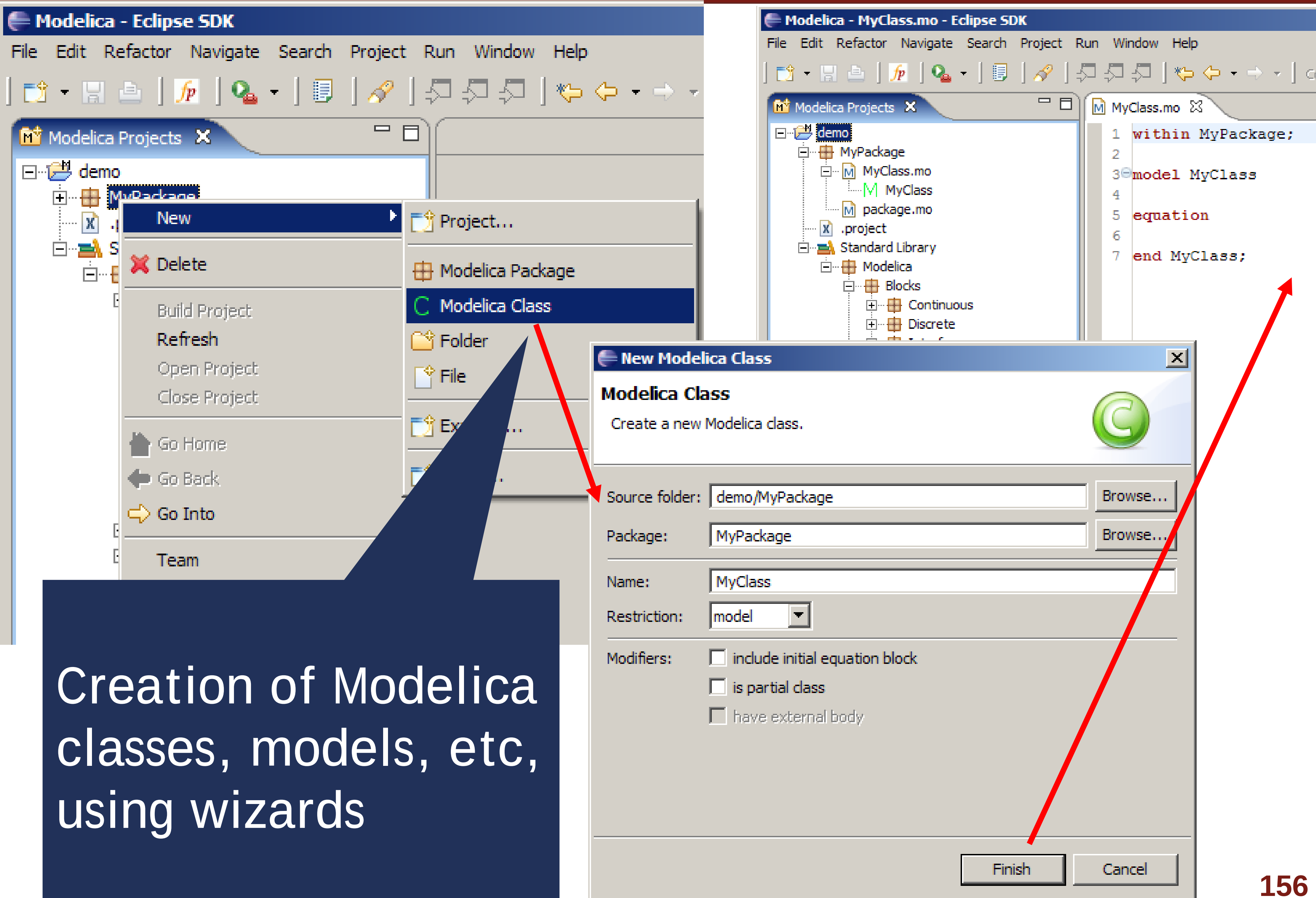
Creating Modelica projects (II)





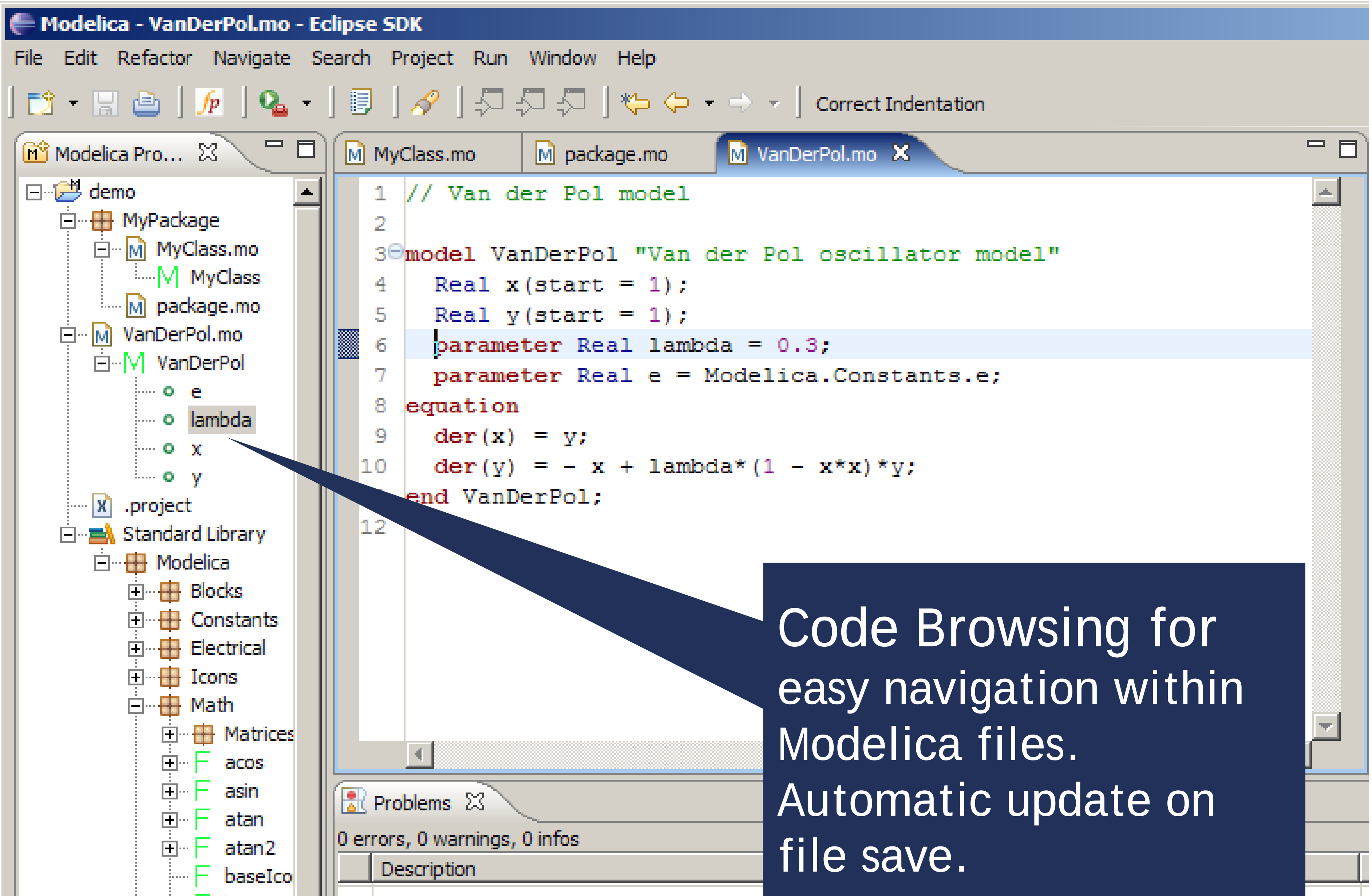
Creation of Modelica packages using wizards

Creating Modelica classes

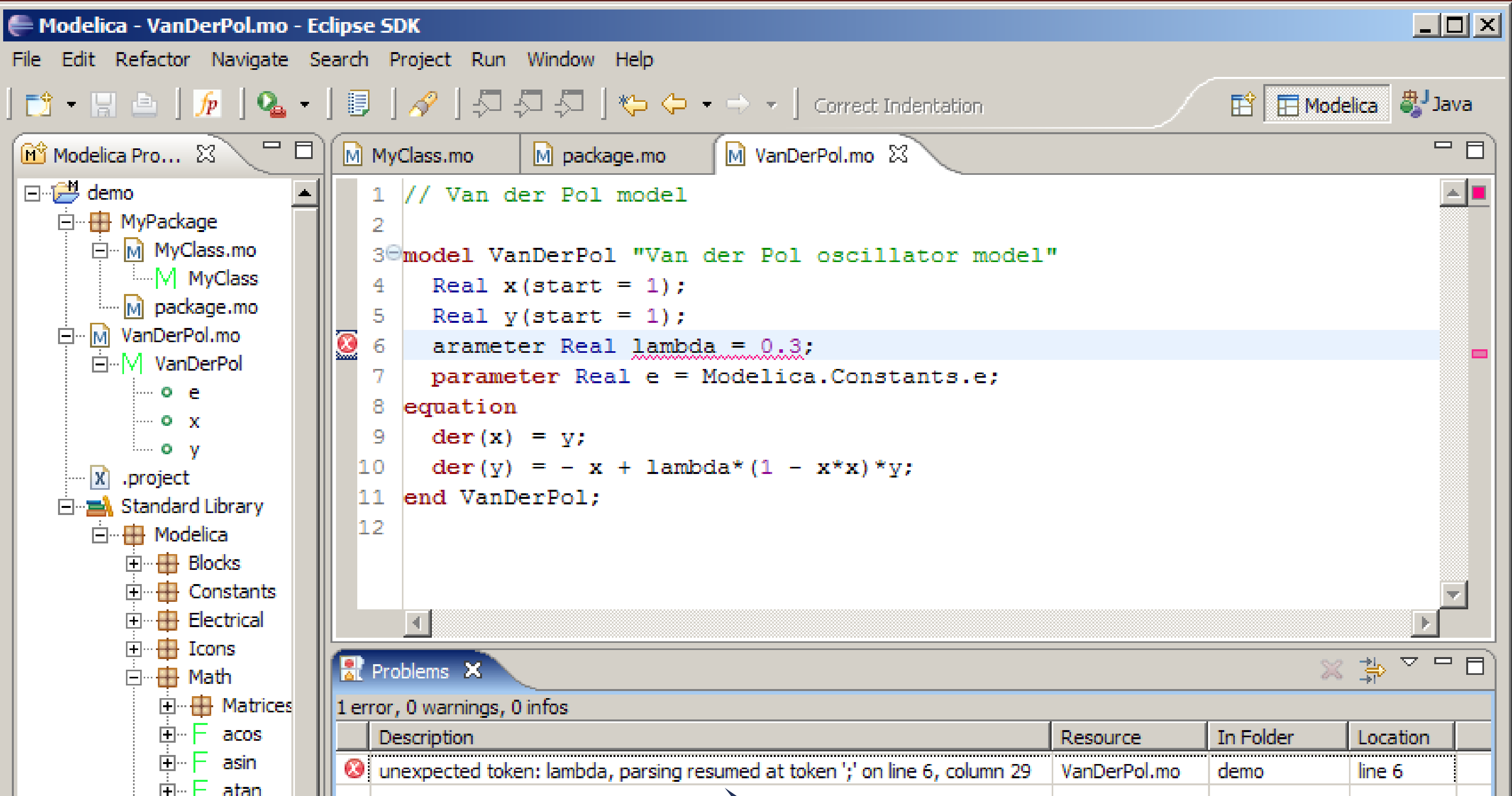


Creation of Modelica classes, models, etc, using wizards

Code browsing

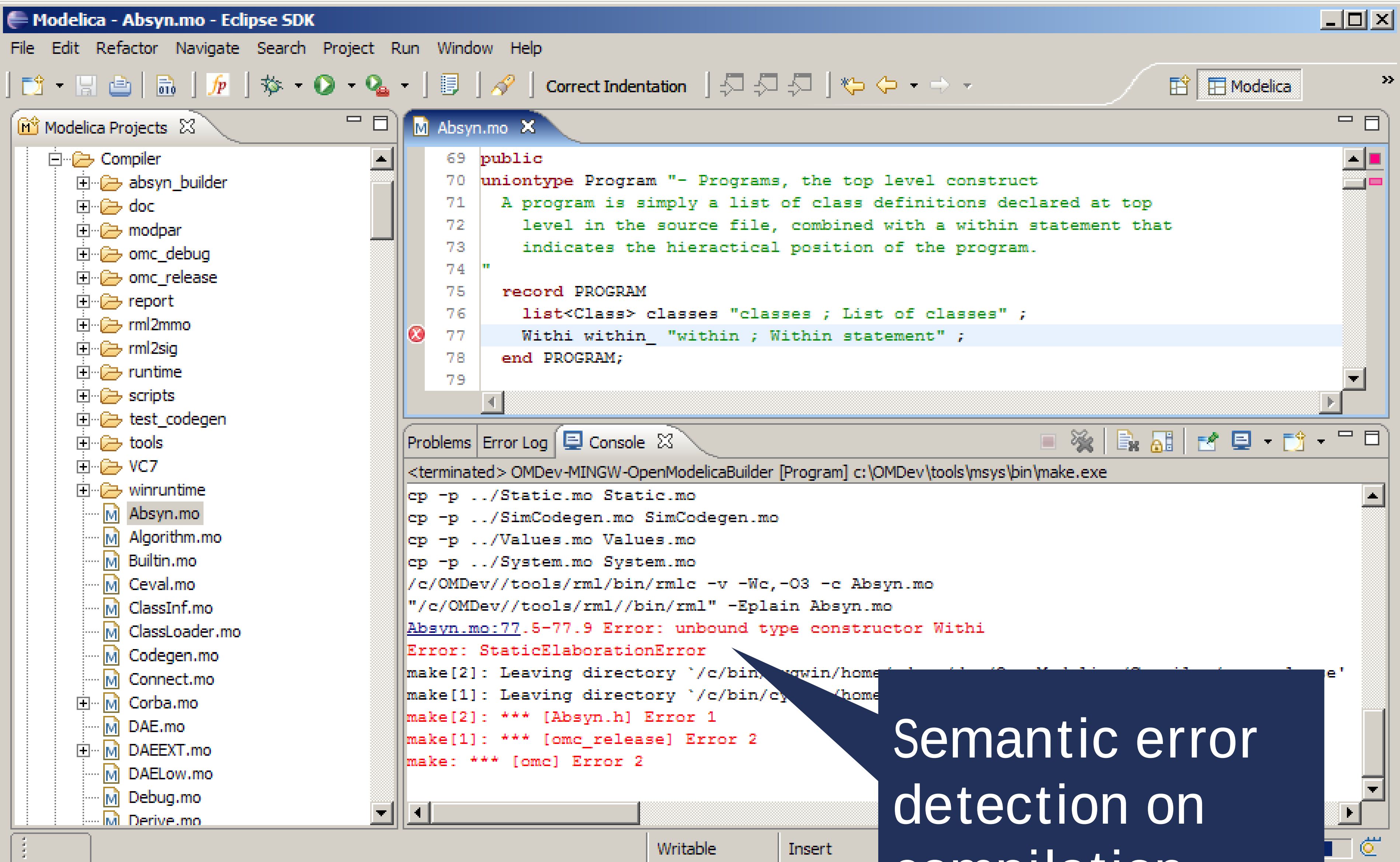


Code Browsing for easy navigation within Modelica files. Automatic update on file save.



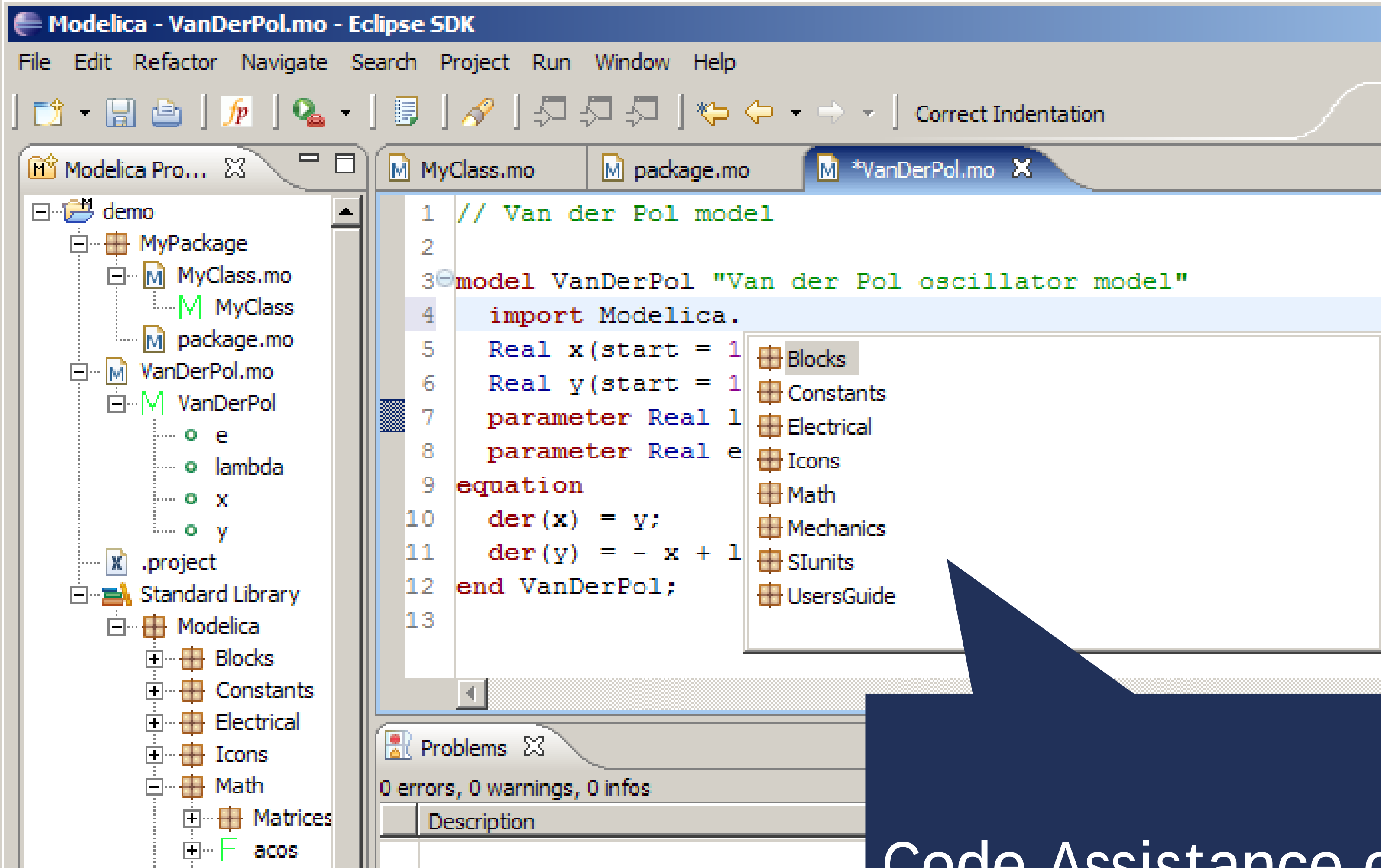
Parse error detection on file save

Error detection (II)

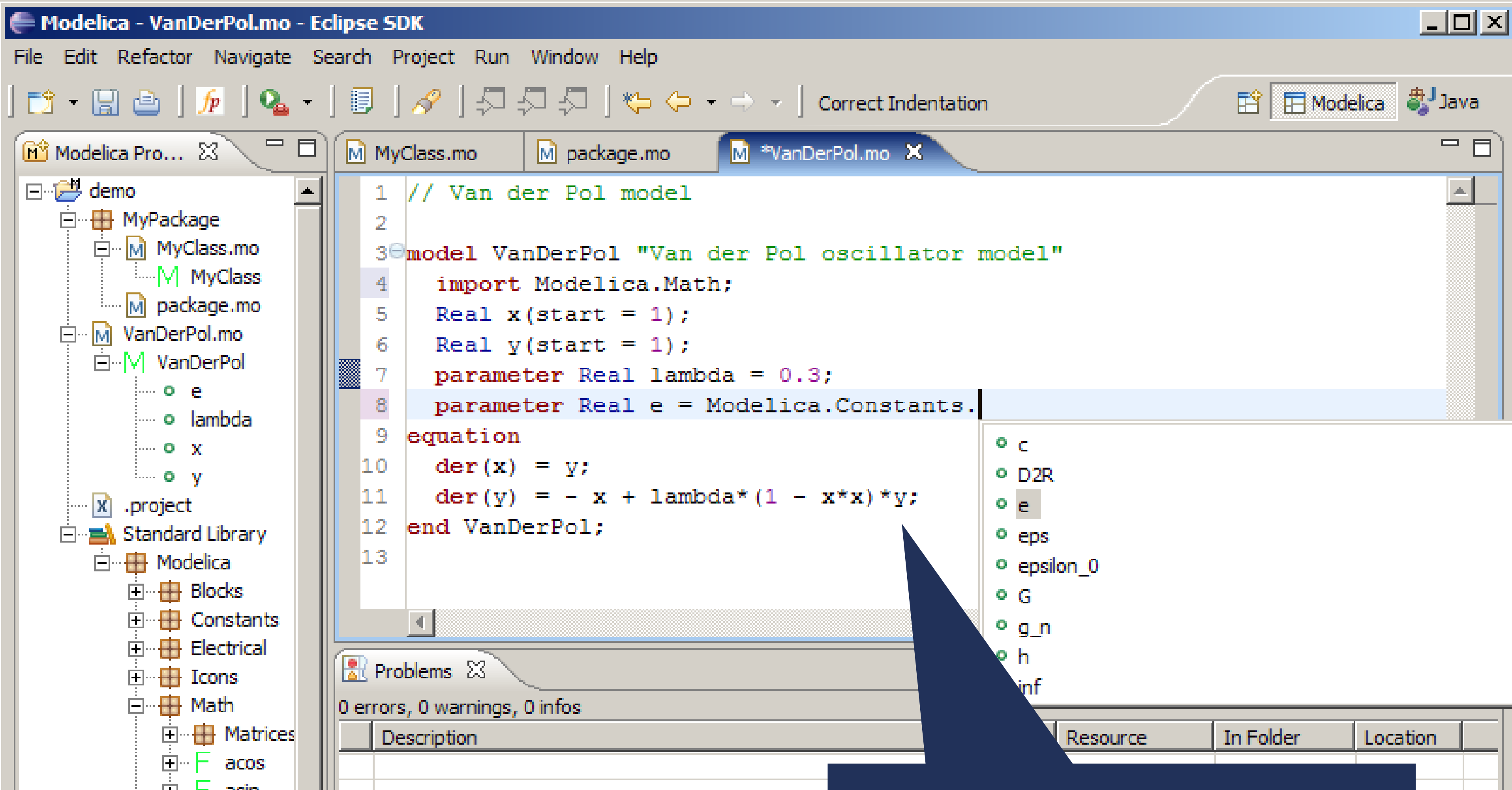


Semantic error detection on compilation

Code assistance (I)

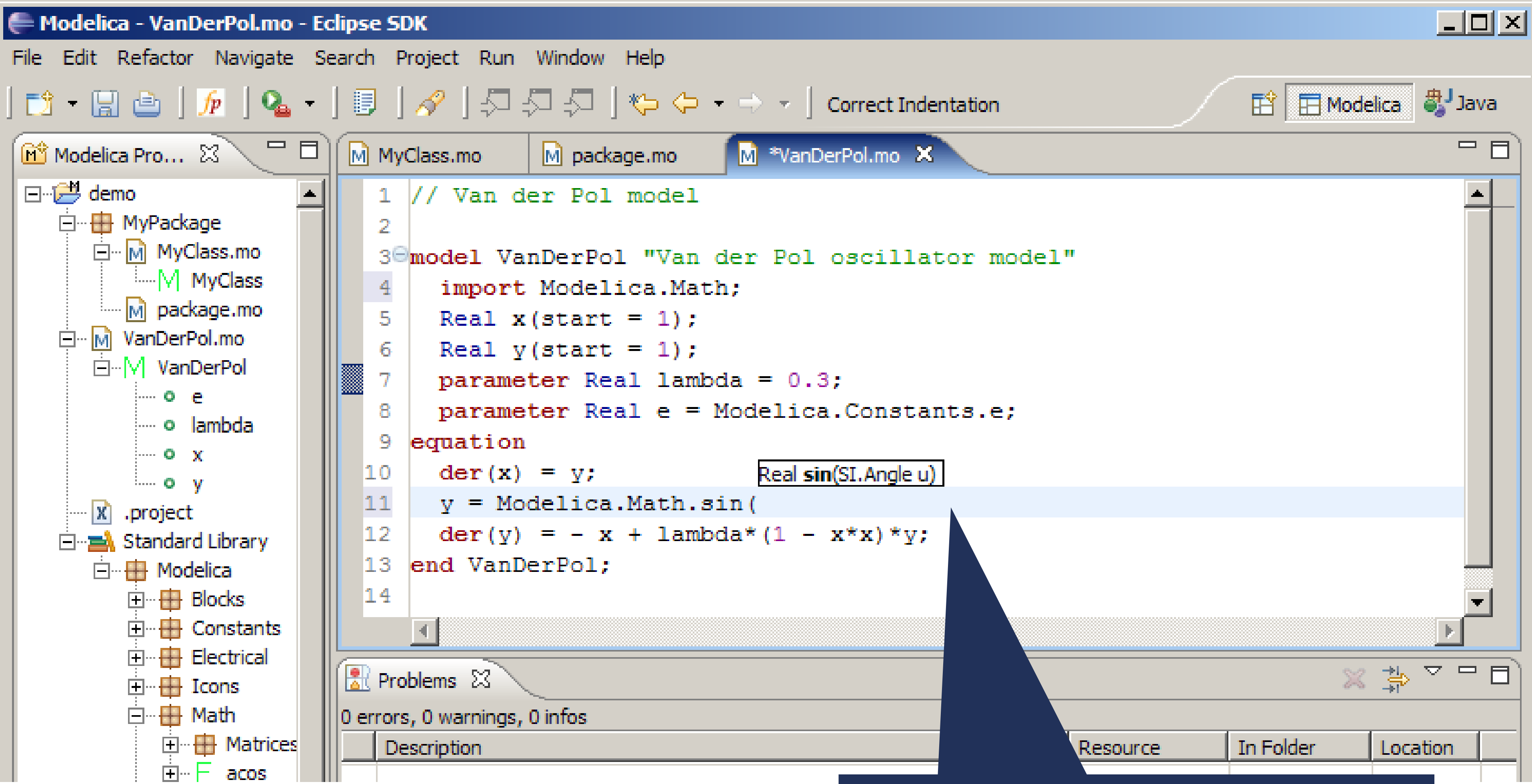


Code Assistance on imports



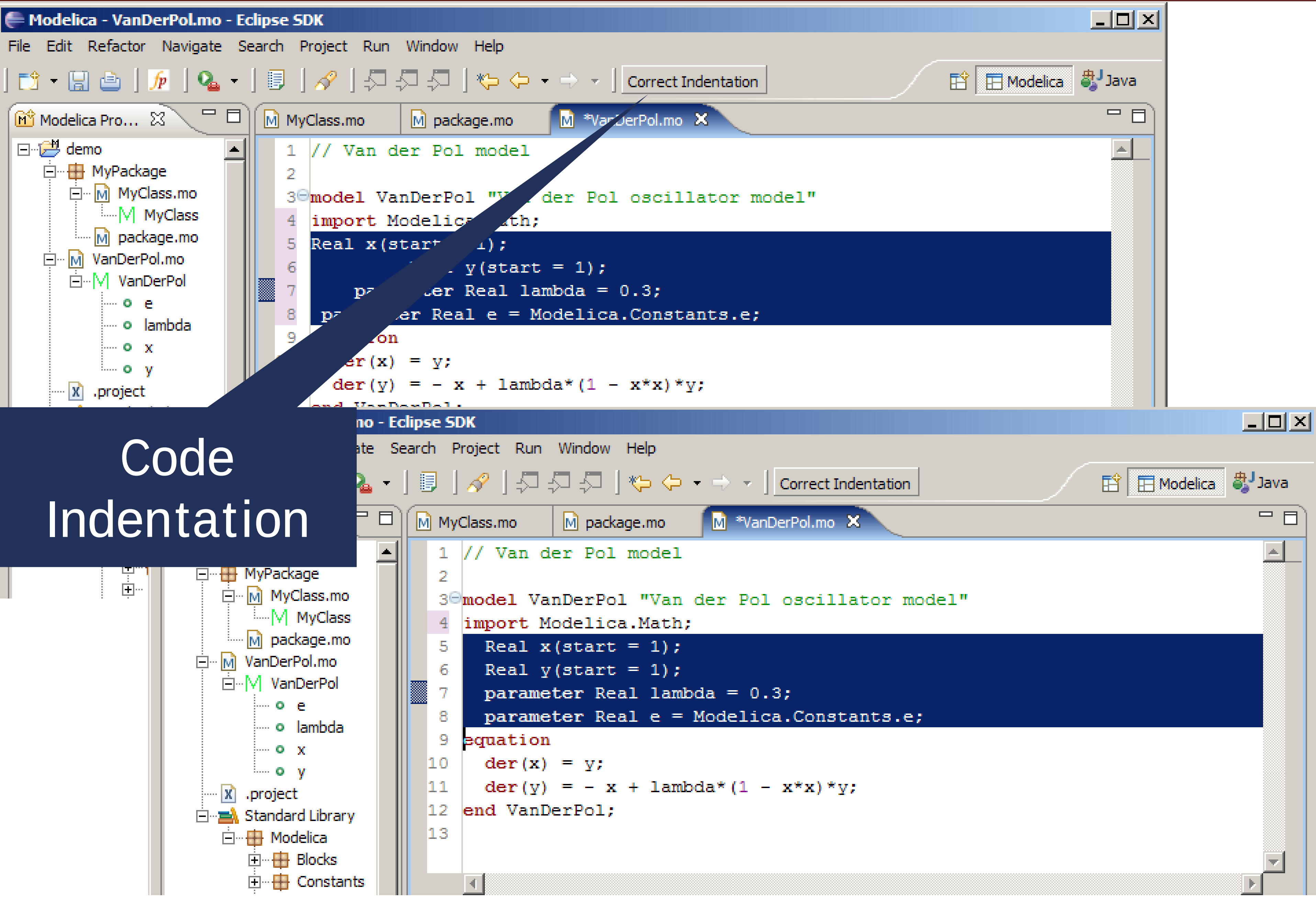
Code Assistance on assignments

Code assistance (III)

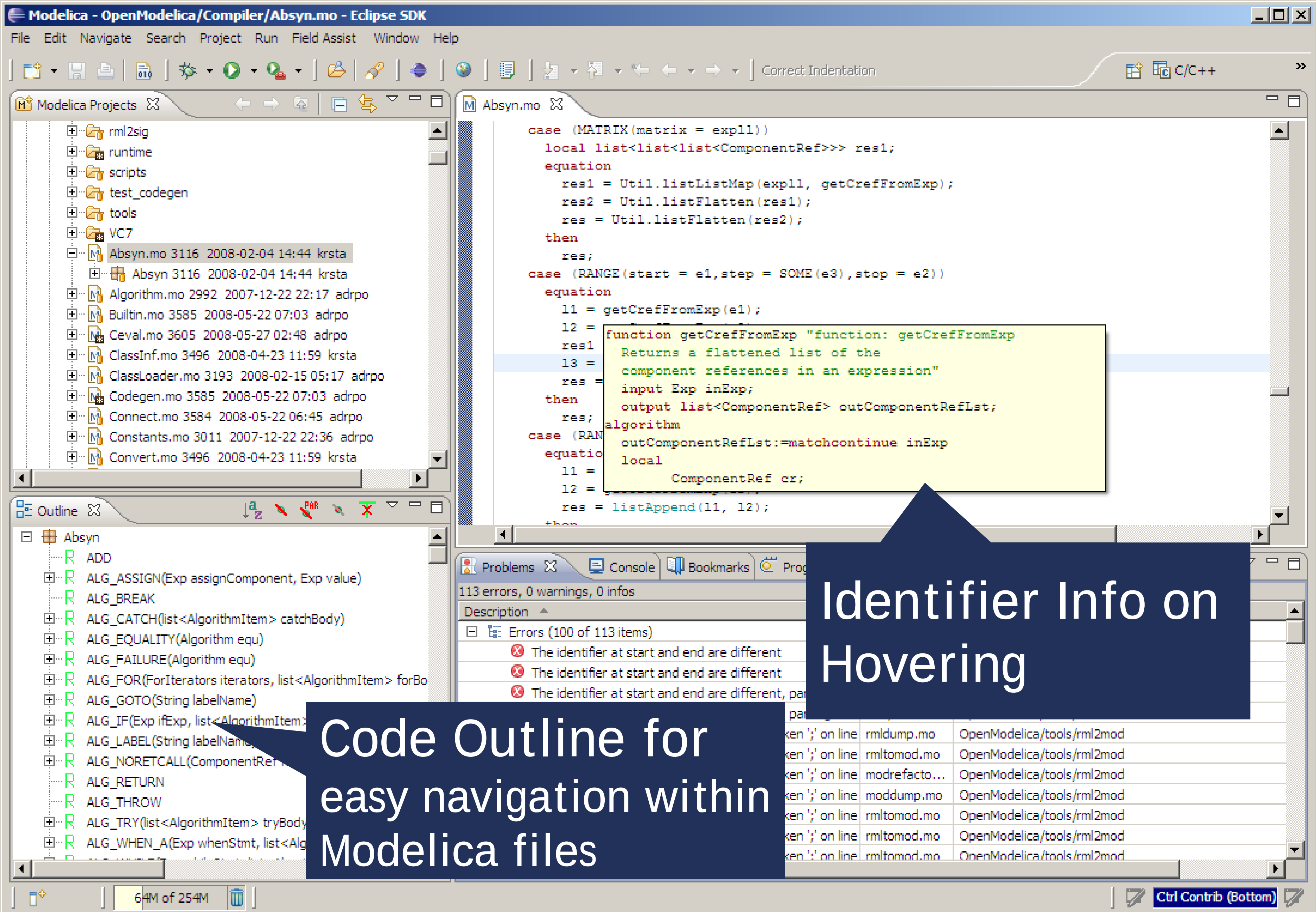


Code Assistance on function calls

Code indentation



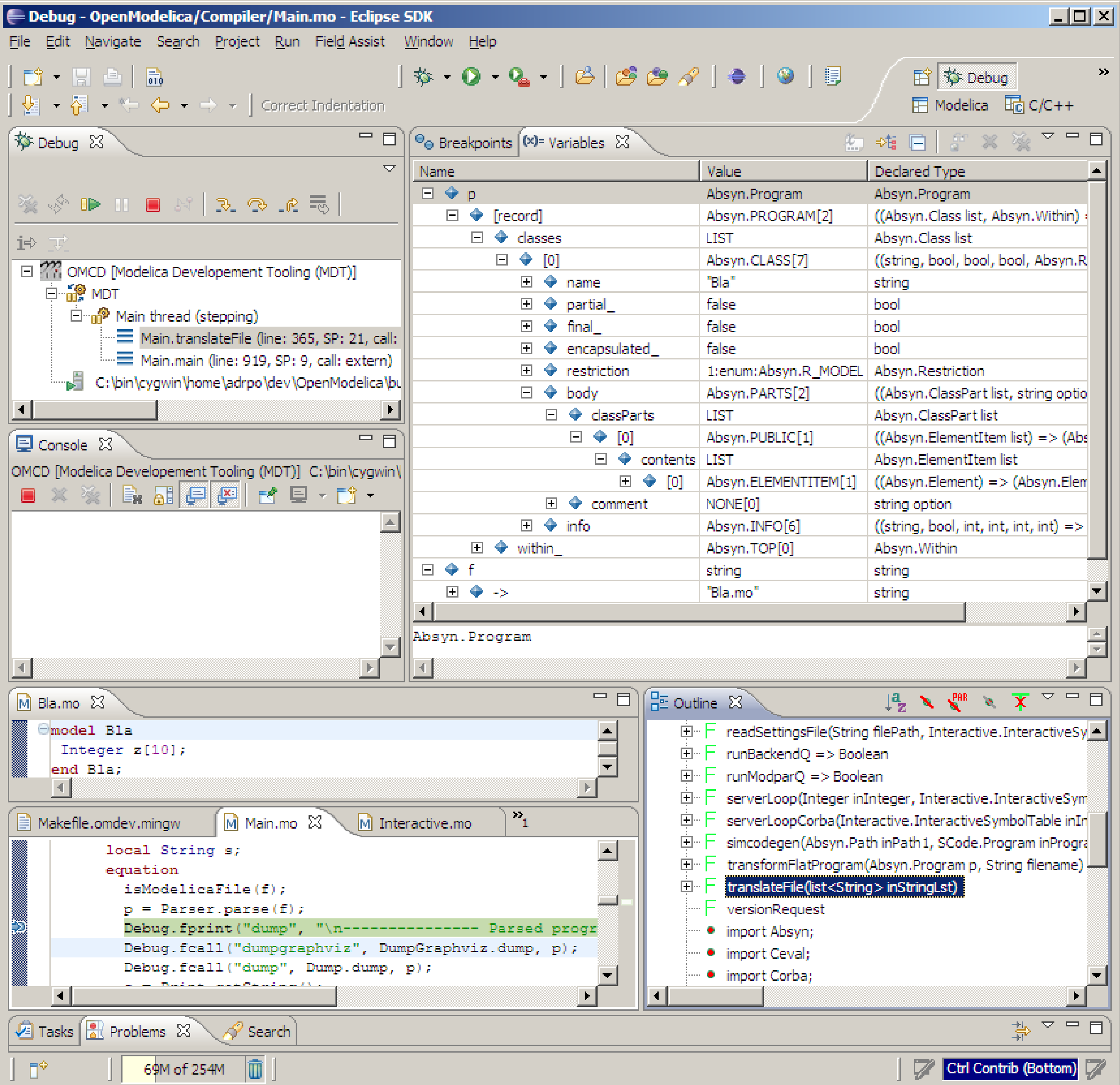
Code Outline and Hovering Info



Identifier Info on Hovering

Code Outline for easy navigation within Modelica files

Eclipse Debugging Environment



- Type information for all variables
- Browsing of complex data structures

- Conclusions
  - Eclipse environment supporting ModelicaML
    - Supports Requirements Engineering
    - Transformation between ModelicaML and Modelica
    - Extends Modelica with additional design capabilities (requirements modeling, inheritance diagrams, etc)
    - Supports translation between Modelica and SysML
- Future Work
  - Finalize the ModelicaML Eclipse environment
  - Better integrate Modelica with SysML

Short Demo

Modelica Development Tooling (MDT)

http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/

OpenModelica Project

http://www.OpenModelica.org

Thank You!

Questions?

Modelica Development Tooling (MDT)

http://www.ida.liu.se/~pelab/modelica/OpenModelica/MDT/

OpenModelica Project

http://www.OpenModelica.org

Encoding Requirements in Modelica

- Using annotations
- Pros: directly supported by Modelica
- Cons:
  - can be present only at specific places
  - is hard to keep track of them

```
type RequirementStatus =
  enumeration(Incomplete, Draft, Started);
annotation(
  Requirement(
    id="S5.4.1",
    level=0,
    status=RequirementStatus.Incomplete,
    name="Master Cylinder Efficacy",
    description="A master cylinder…"));
```

Encoding Requirements in Modelica

- Using restricted class: requirement
- Pros:
  - direct Modelica support for requirements
  - hierarchies of requirements supported by inheritance
  - easy linking with
- Cons:
  - Modelica specification needs to be extended

```
type RequirementStatus =
  enumeration(Incomplete, Draft, Started);

requirement R1
  String name="Master Cylinder Efficiency";
  String id="S5.4.1";
  Integer level=0;
  RequirementStatus status=
     RequirementStatus.Incomplete;
  String description=”A master cylinder
                      shall have…”;
end R1;
```

```
requirement R2
  extends R1;
  String name"Loss Of Fluid";
  String id="S5.4.1a";
...
end R2;

model BreakSystem
  annotation(satisfy=R1);
...
end BreakSystem;
```