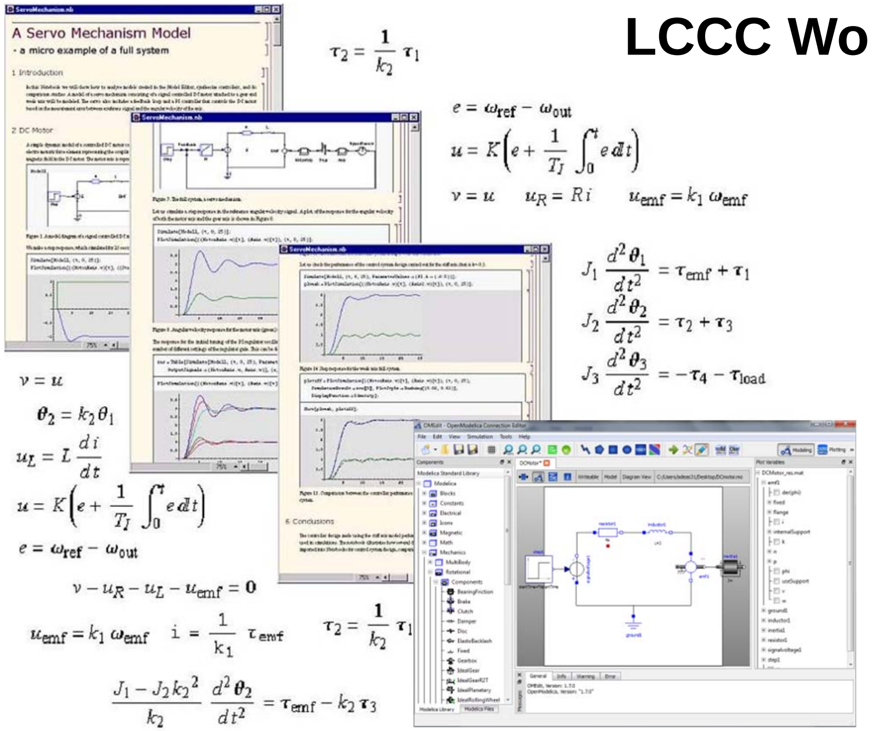
The OpenModelica Environment including Static and Dynamic Debugging of Modelica Models and Systems Engineering / Design Verification

LCCC Workshop in Lund



September 19-21, 2012

Peter Fritzson

Professor at Linköping University, Sweden
Vice Chairman of Modelica Association
Director of Open Source Modelica Consortium
peter.fritzson@liu.se

Main Contributors, these topics:

Wladimir Schamai, Martin Sjölund, Adrian Pop, Adeel Asghar & rest of OpenModelica team

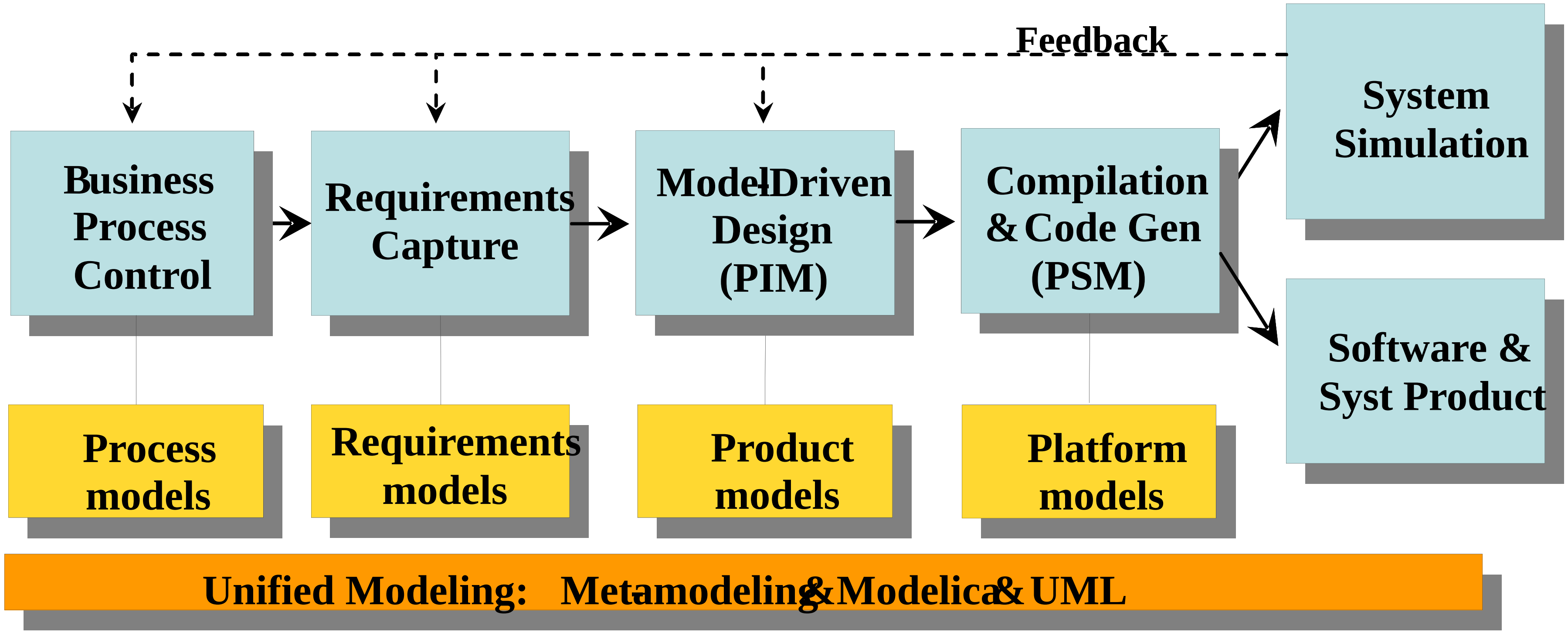
Linköpings universitet

MODELICA

Overview

- Background
- Debugging models
- Dynamic verification of requirements

Vision of Integrated Model-Based Development



Vision of unified modeling framework for model-driven product development from platform independent models (PIM) to platform specific models (PSM)

Formal Specification of Modelica Static Semantics

- First Structured Operational Semantics (SOS) Modelica subset formal specification
  - First version1998, main parts of Modelica static semantics
  - Primarily Big step semantics / Natural Semantics
  - Generating first version of the OpenModelica compiler
- Generating efficient compiler using RML tool
- 2005 converting rule-based syntax into MetaModelica syntax
- 2011 full integration with standard Modelica
  - Bootstrapping of the OpenModelica compiler

Main Language Extensions

- MetaModelica 2005
  - Recursive data structures, lists
  - Pattern matching
  - Failure/exception handling, backtracking
- ParModelica 2011
  - Dataparallel language constructs, multi-core, e.g. mapping to OpenCL
  - Memory hierarchy for data allocation
- Optimization extension 2012
  - Follow same syntax as Optimica in Jmodelica.org
- ModelicaML extension from 2007
  - Integrate UML/SysML graphical language and requirement handling
  - Separate tool, not yet integrated in Modelica and the OpenModelica compiler

OpenModelica – An Open Source Environment

Open Source Modelica Consortium, 43 org members Aug 2012

Founded Dec 4, 2007

Open-source community services

- **Website and Support Forum**
- **Version-controlled source base**
- **Bug database**
- **Development courses**
- **www.openmodelica.org**

Interactive Modelica compiler (OMC)

- **Compiles the Modelica Language**
- **Modelica and Python scripting**

Environment for creating models

- **OMShell – scripting commands**
- **OMNotebook – interactive notebook**
- **MDT –Eclipse plug-in**
- **OMEdit graphic Editor**
- **OMOptim optimization tool**
- **ModelicaML UML Profile**

MODELICA

Debugging Equation-Based Languages and Background

Problems

- Large Gap in Abstraction Level
  from Equations to Executable Code

- Example error message (hard to undestand)

  Error solving nonlinear system 132

      time = 0.002
      residual[0] = 0.288956
      x[0] = 1.105149
      residual[1] = 17.000400
      x[1] = 1.248448
      ...

Static vs Dynamic Debugging

- Static Debugging
  - Analyze the model/program at compile-time
  - Explain inconsistencies and errors, trace error dependencies
  - Example: Underconstrained/overconstrained systems of equations
  - Example: errors in symbolic transformations of models
- Dynamic Debugging
  - Find sources of errors at run-time, for a particular execution
  - **Declarative dynamic debugging** – compare the execution with a specfication and semi- automatically find the location of the error
  - **Traditional dynamic debugging** – interactively step through the program, set breakpoints, display and modify data structures, trace, stack inspection
- Goal: Integrated Static and Dynamic Debugging

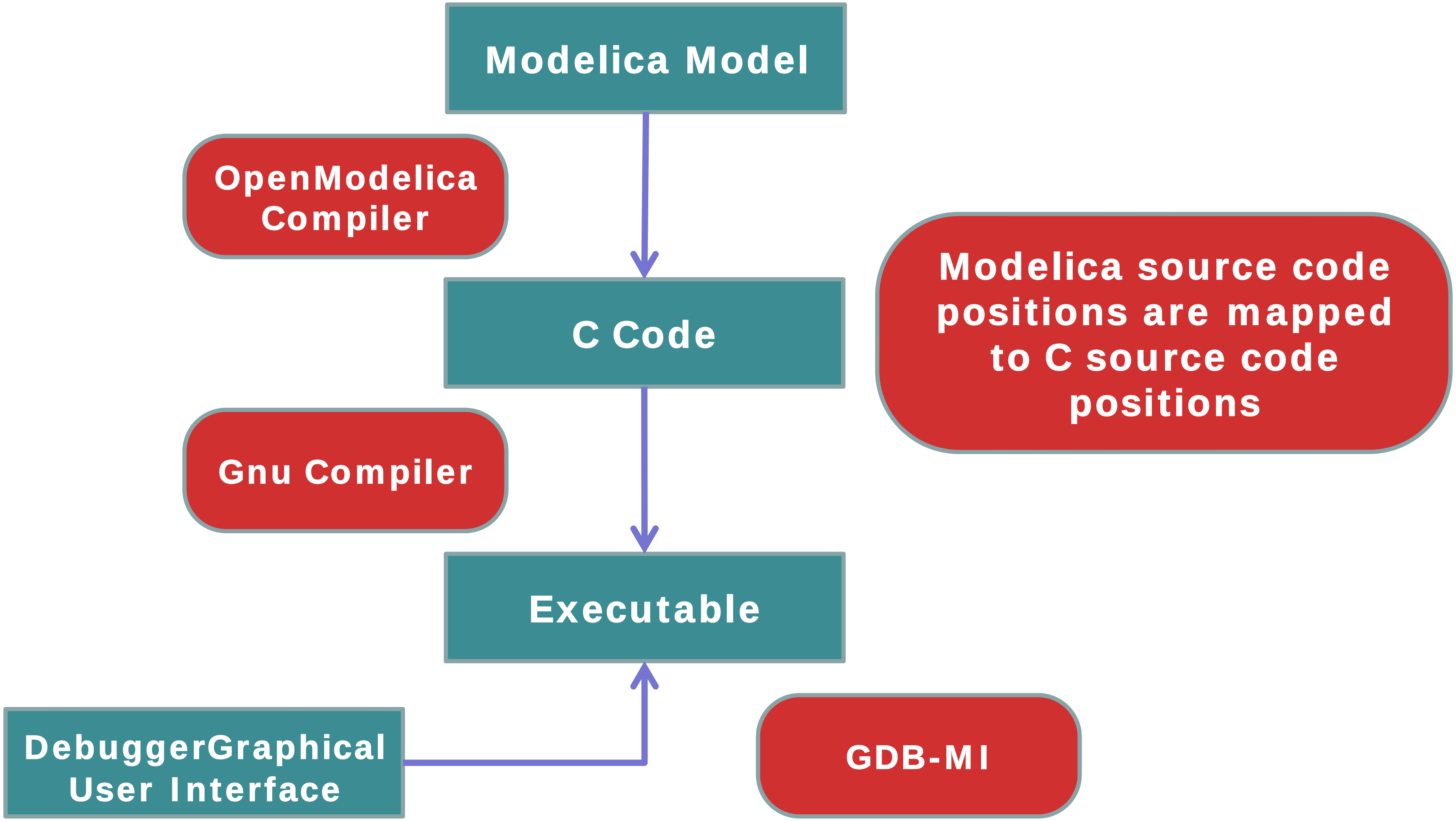
Previous PhD Theses on Dynamic/Static Debugging in Our Group

- *Dynamic*. Nahid Shahmeri(1991). Generalized Algorithmic Debugging
- *Dynamic*. Mariam Kamkar(1993). Interprocedural Dynamic Slicing with Applications to Debugging and Testing
- *Dynamic*. Henrik Nilsson(1998). Declarative Debugging for Lazy Functional Languages
- *Static/Dynamic*. Peter Bunus (June 2004). Debugging Techniques for Equation-Based Languages.
- *Dynamic*. Adrian Pop (June 5, 2008). Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages

Dynamic Debugging

Large Modelica Algorithmic Code Models

Tool Architecture and Communication



Example Mapping Modelica Postions to C Code

Convert Modelica code to C source code by adding Modelica line number references.

HelloWorld.mo

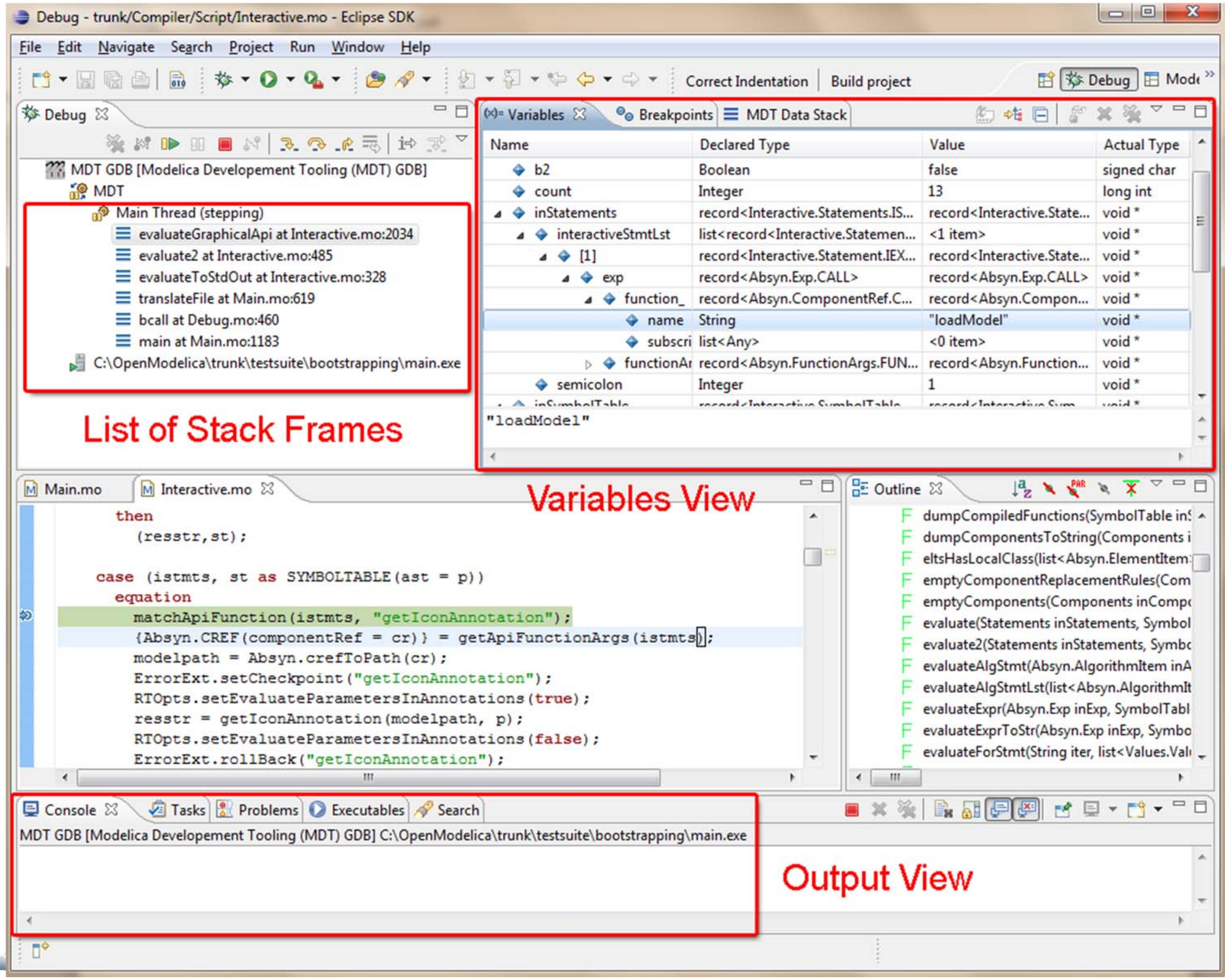
```
function HelloWorld
  input Real x;
  output Real y;
algorithm
  y := sin(x);
end HelloWorld;
```

HelloWorld.conv.c

```
#line 29 "HelloWorld.c"
  /* functionBodyRegularFunction: var inits */
#line 30 "HelloWorld.c"
  /* functionBodyRegularFunction: body */
#line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
  tmp2 = sin(_x);
#line 5 "/c/workspace/HelloWorld/HelloWorld.mo"
  _y = tmp2;
#line 35 "HelloWorld.c"
```

Debugger Integrated in Eclipse OpenModelica MDT Environment

- Eclipse plugin **MDT (Modelica Development Tooling)** is the integrated development environment
- Debugger is a debug plug-in within MDT



Static Debugging

Transformational Debugging of Equation-Based Models

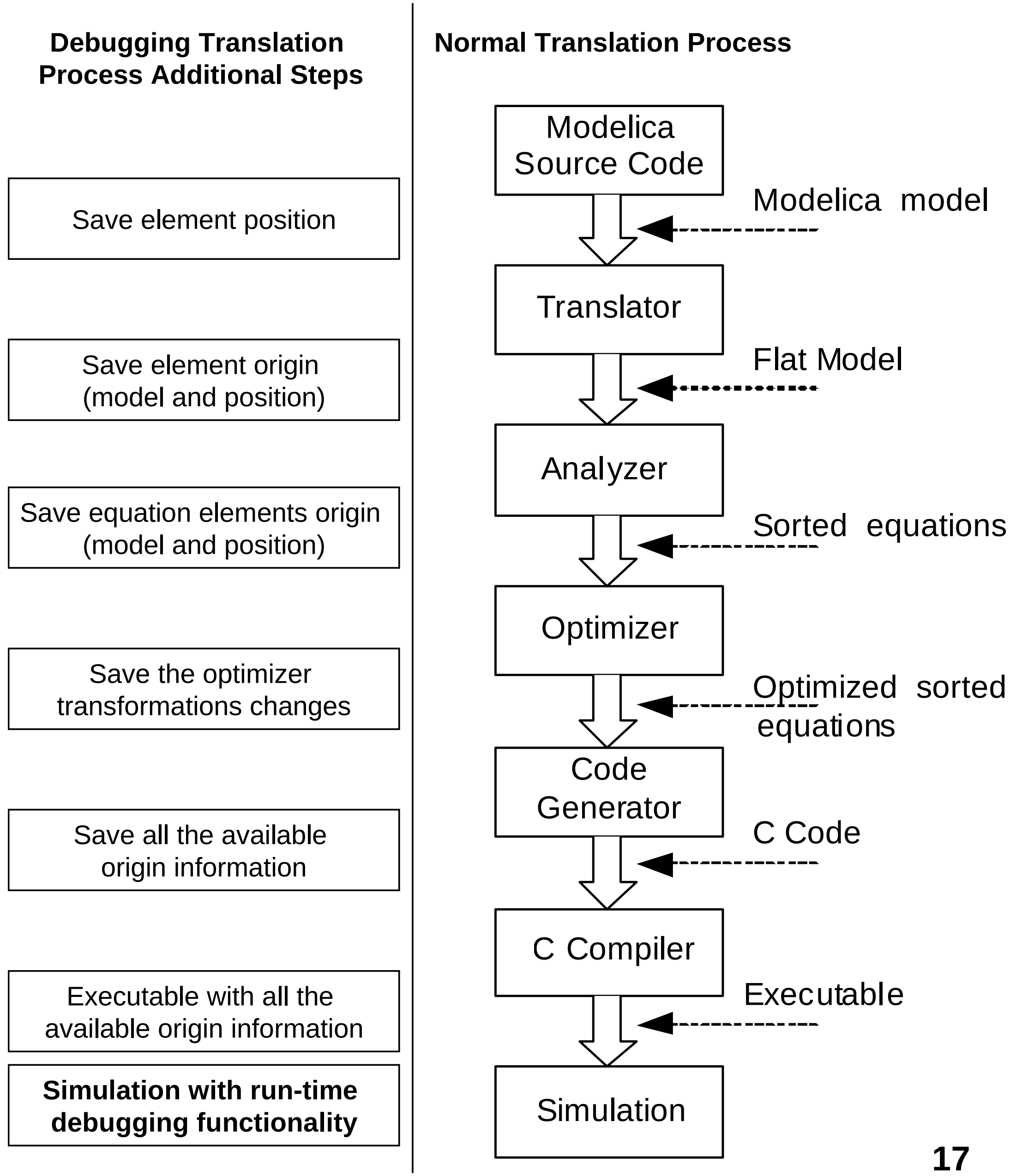
Debugging Equation Systems

Modelica Compiler Backend

- Complex mathematical transformations
- Hidden to users
- Users want to access this information
- Not intuitive, because
    - No explicit control flow
    - Numerical solvers
    - Linear/Non-linear blocks
    - Optimization
    - Events

Translation Phases with Model Debugging

- **Include debugging support within the translation process**



Input to Debugger: Modelica Model

```
class RC // 24 equations and variables
 …
equation
  …
  ground1.p.v = 0.0;
  0.0 = resistor1.p.i + resistor1.n.i;
  resistor1.i = resistor1.p.i;
  resistor1.T_heatPort = resistor1.T;
  capacitor1.i = capacitor1.C * der(capacitor1.v);
  capacitor1.v = capacitor1.p.v - capacitor1.n.v;
  0.0 = capacitor1.p.i + capacitor1.n.i;
  capacitor1.i = capacitor1.p.i;
  …
end RC;
```

Output from Compiler Frontend:
Sorted ODE or DAE (Differential Algebraic Equations)

```
class RC // 24 equations and variables
 …
equation
  …
  ground1.p.v = 0.0;
  0.0 = resistor1.p.i + resistor1.n.i;
  resistor1.i = resistor1.p.i;
  resistor1.T_heatPort = resistor1.T;
  capacitor1.i = capacitor1.C *
     der(capacitor1.v);
  capacitor1.v = capacitor1.p.v –
      capacitor1.n.v;
  0.0 = capacitor1.p.i + capacitor1.n.i;
  capacitor1.i = capacitor1.p.i;
  ...
end RC;
```

```
class RC // 5 equations and variables
 …
 // 14 alias variables 5 constants
equation
  sinevoltage1.signalSource.y =
sinevoltage1.signalSource.offset + (if time <
sinevoltage1.signalSource.startTime then 0.0
else sinevoltage1.signalSource.amplitude *
sin(6.28318530717959 *
(sinevoltage1.signalSource.freqHz * (time -
sinevoltage1.signalSource.startTime)) +
sinevoltage1.signalSource.phase));
  resistor1.v = capacitor1.v -
sinevoltage1.signalSource.y;
  capacitor1.i = -resistor1.v / resistor1.R_actual;
  resistor1.LossPower = -resistor1.v *
capacitor1.i;
  der(capacitor1.v) = capacitor1.i / capacitor1.C;
end RC;
```

Symbolic Transformations

- From source code to flat equations
  - Most of the structure remains
  - Few symbolic manipulations (mostly simplification/evaluation)
- Equation System Optimization
  - Changes structure
  - Strong connected components
  - Variable replacements
  - … and more

Tracing Symbolic Transformations

- Simple Idea
  - Store transformations as equation metadata
- Works best for operations on single equations
  - Alias Elimination (a = b)
  - Equation solving ($f_1(a,b) = f_2(a,b)$, solve for a)
- Multiple equations require special handling
  - Gaussian Elimination (linear systems, several equations)
  - ...

Tracing Overhead?

- OpenModelica compiler implementation is so fast that tracing is enabled by default
  - 1 extra comparison and/or cons operation per optimization
  - Not noticeable during normal compilation
  - Less than 1% time overhead for tracing
- No real overhead unless you output the trace

Substitution Example, Storing the Trace

a = b

c = a + b

d = a - b

c = a + b (subst a=b) =>

c = b + b (simplify) =>

c = 2 * b

d = a - b (subst a=b) =>

d = b - b (simplify) =>

d = 0.0

- The alias relation a=b stored in variable a
- The equations are e.g. stored as (lhs,rhs,list<ops>)

Debugging Using the Transformation Trace

- Text output
  - Initial implementation
  - Verify performance and correctness of the trace
- Structured output based on database storage
  - Graphical debugging
  - Cross-referencing equations (dependents/parents)
  - Ability to see why a variable is solved in a particular way
  - Requires a schema
  - Future work/work in progress

Trace Example (1)

0 = y + der(x * time * z);   z = 1.0;

(1) substitution:

y + der(x * (time * z))

=>

y + der(x * (time * 1.0))

(2) simplify:

y + der(x * (time * 1.0))

=>

y + der(x * time)

(3) expand derivative (symbolic diff):

y + der(x * time)

=>

y + (x + der(x) * time)

(**4) solve:**

0.0 = y + (x + der(x) * time)

=>

der(x) = ((-y) - x) / time

Trace Example (2)

differentiation:

d/dtime L ^ 2.0

=>

0.0

differentiation:

d/dtime x ^ 2.0 + y ^ 2.0

=>

2.0 * (der(x) * x + der(y) * y)

Substitution:

2.0 * (der(x) * x + der(y) * y)

=>

2.0 * ($DER.x * x + $DER.y * y)

=>

2.0 * (u * x + $DER.y * y)

=>

2.0 * (u * x + v * y)

=>

2.0 * (u * xloc[1] + v * xloc[0])

Readability of Transformation Trace

- Most equations have very **few** transformations on them
- Most of the interesting equations have a few
  - Still rather readable
- Some extra care to handle Modelica variable aliasing

MSL 3.1 MultiBody DoublePendulum

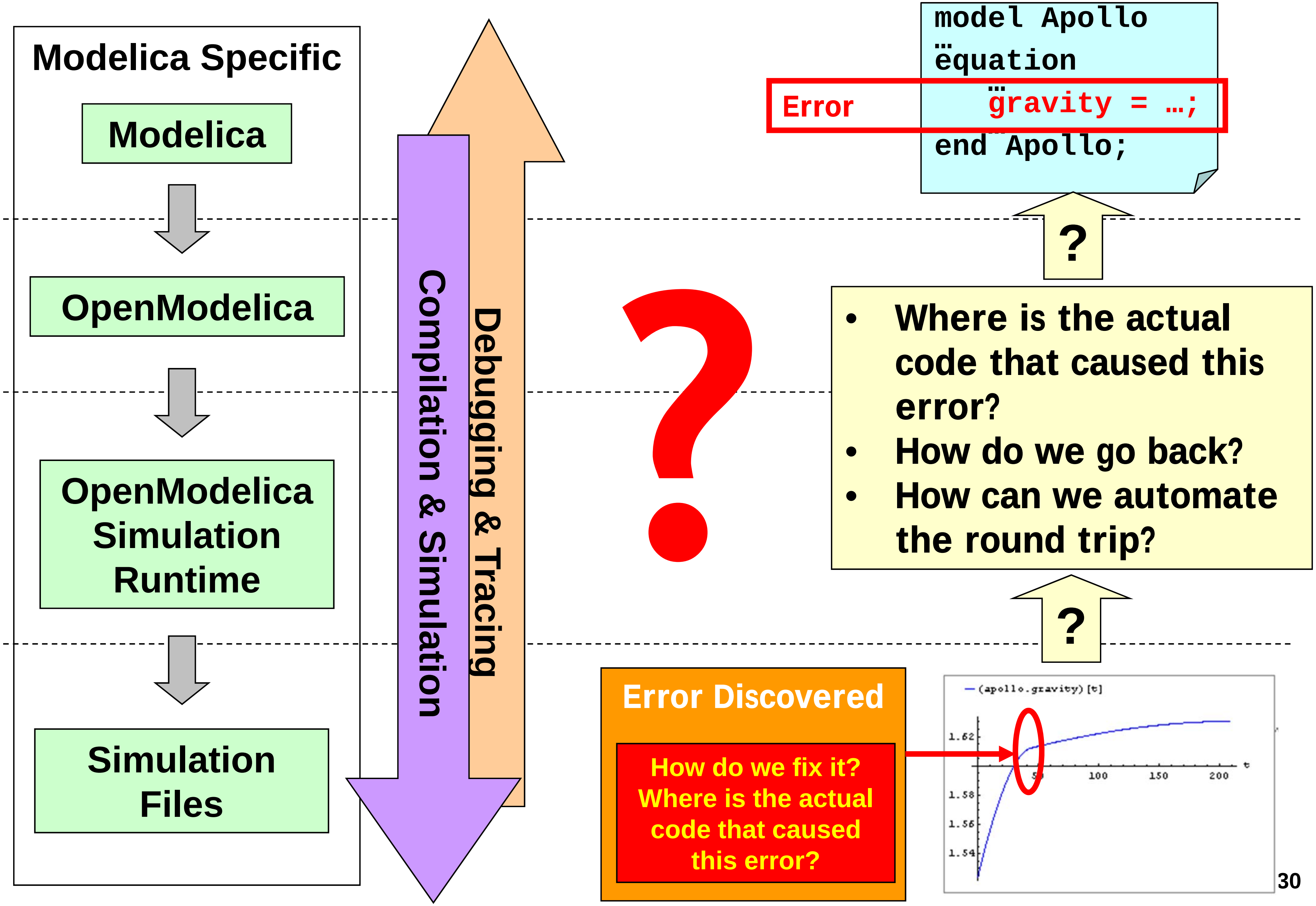
# Ops	Frequency	Comment
0	457	Parameters
1	89	Dummy eq & know var
2	720	Alias vars
3	479	Alias vars
4	124	Alias after simplify
5	25	Alias after simplify
6	99	Alias after simplify
7	55	Scalar eq
8	37	...
9	110	...
10	72	...
11	12	...
12	25	...
13	35	...
14	3	Known constant after many replacements
21	27	World object (3x3 matrix with many occurances of aliased vars)

Future Work on Transformational Debugging

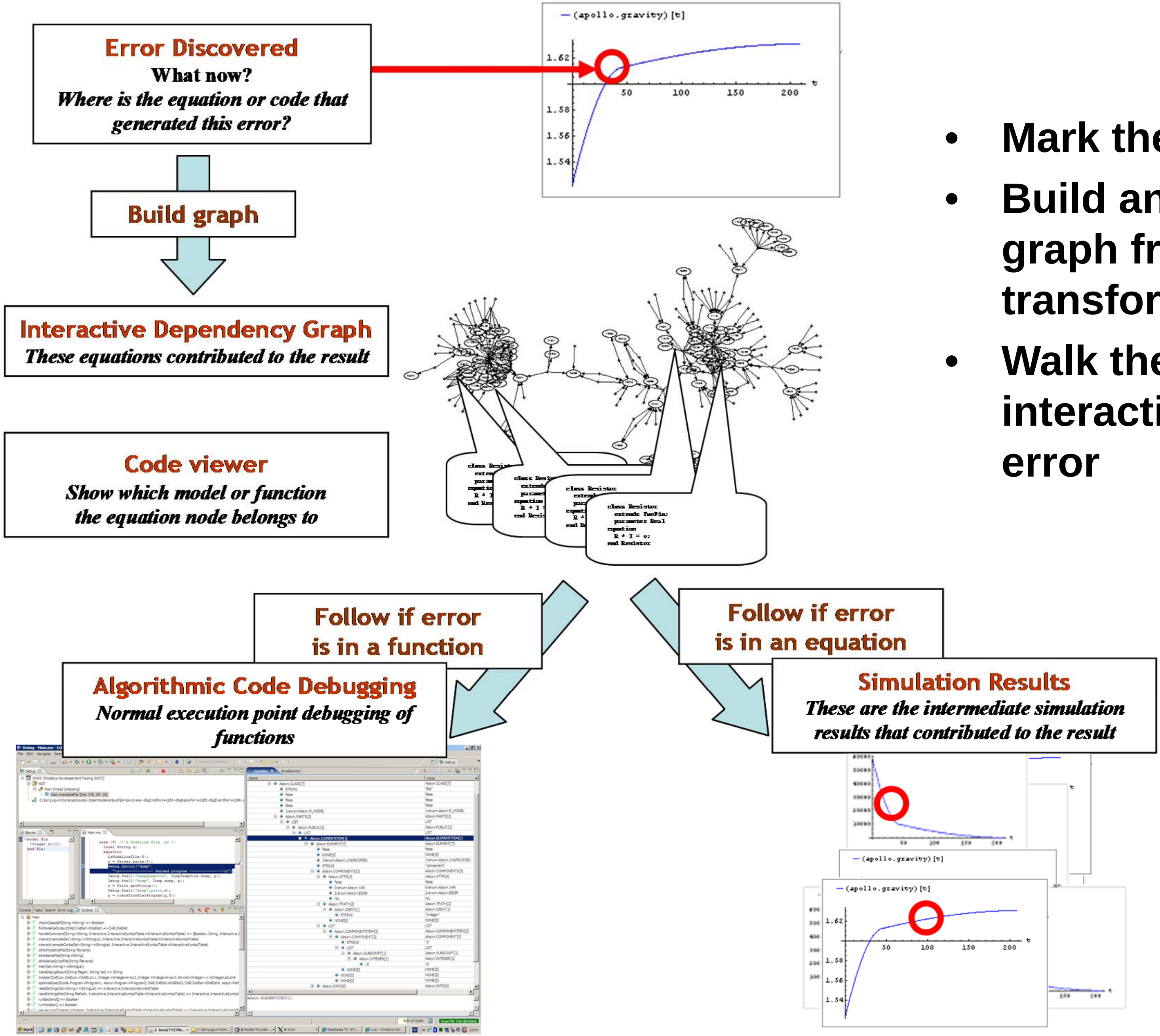
- Structural debug information queries based on a database
- Graphical debugger
- Simulation runtime uses database
- More operations recorded
  - Dead code elimination
  - Control flow and events
  - Forgotten optimization modules

Integrated Debugging

Need to Combine Approaches to Help the User



Integrated Debugging Approach



- **Mark the error**
- **Build an interactive graph from the transformation trace**
- **Walk the graph interactively to find the error**

Debugging Based on User Interaction

- The interactive dependency graph contains two types of edges:
  - *Calculation dependency edges*
  - *Origin edges from traced symbolic transformations*

- The user interacts with the dependency graph in several ways:
  - *Displaying simulation results* through selection of the variables
  - *Classifying a variable* as having wrong values
  - *Classifying an equation as correct*
  - *Building a new dependency graph* based on the new set of variables with wrong values (classified variables) or by modifying the equations or parameter values nodes.
  - *Displaying model code* by following origin edges
  - *Invoking the algorithmic code debugging subsystem*

32

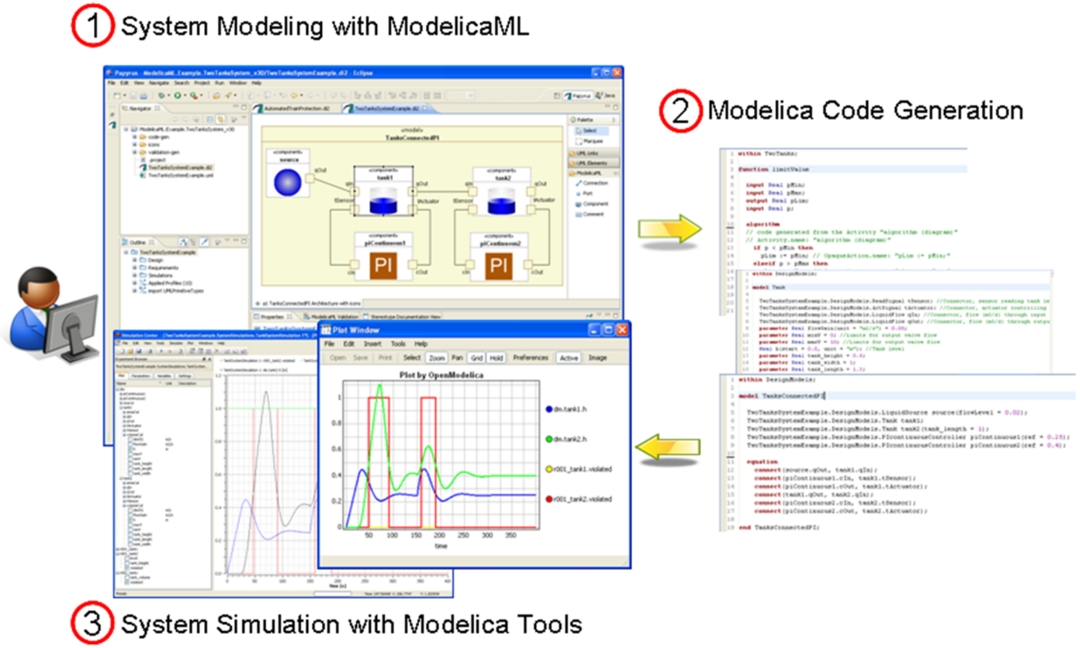
Debugging Summary

- Debugging **equation-based** models present new **challenges**
- **Equation** systems are **transformed** symbolically to a form hard for the user to recognize
- Static **transformational** debugging **explains** the transformations and maintains a mapping between the low level and the high level model
- **Dynamic debugging** helps to **walk** through a model/program and **inspect** data for an execution
- **Goal: integrated static/dynamic debugging approach**

Requirements traceability and dynamic model verification

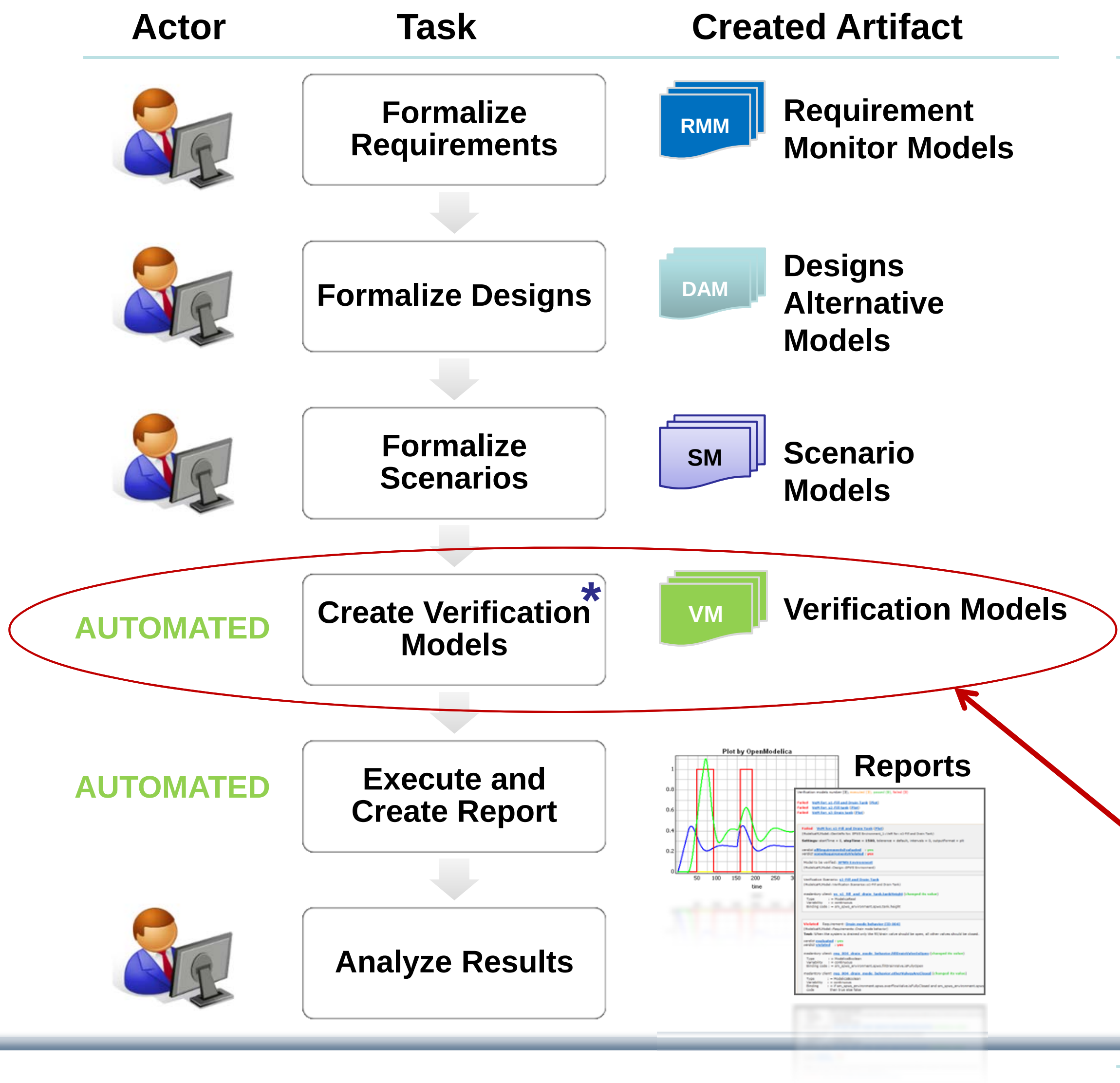
Introduction: ModelicaML Background

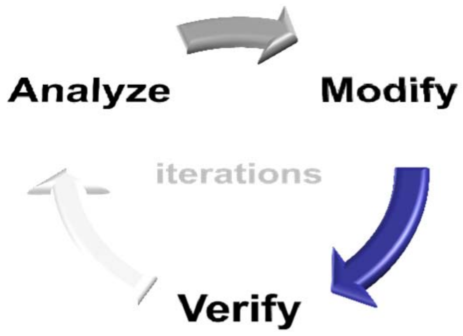
- **ModelicaML Eclipse plug-in Modelica/UML profile** integrates a subset of the UML and the Modelica language in order to leverage standardized graphical notations of UML for system modeling and the simulation power of Modelica
- ModelicaML enables engineers to describe
  - System **requirements**
  - System **design** (structure and behavior)
  - Usage-, test **scenarios**



- **vVDR (Virtual Verification of Designs against Requirements) is a method that enables a model-based design verification against requirements**
- **vVDR is supported in ModelicaML**

Introduction: vVDR Method





Goal: Enable on-demand verification of designs against requirements using automated model composition at any time during development.

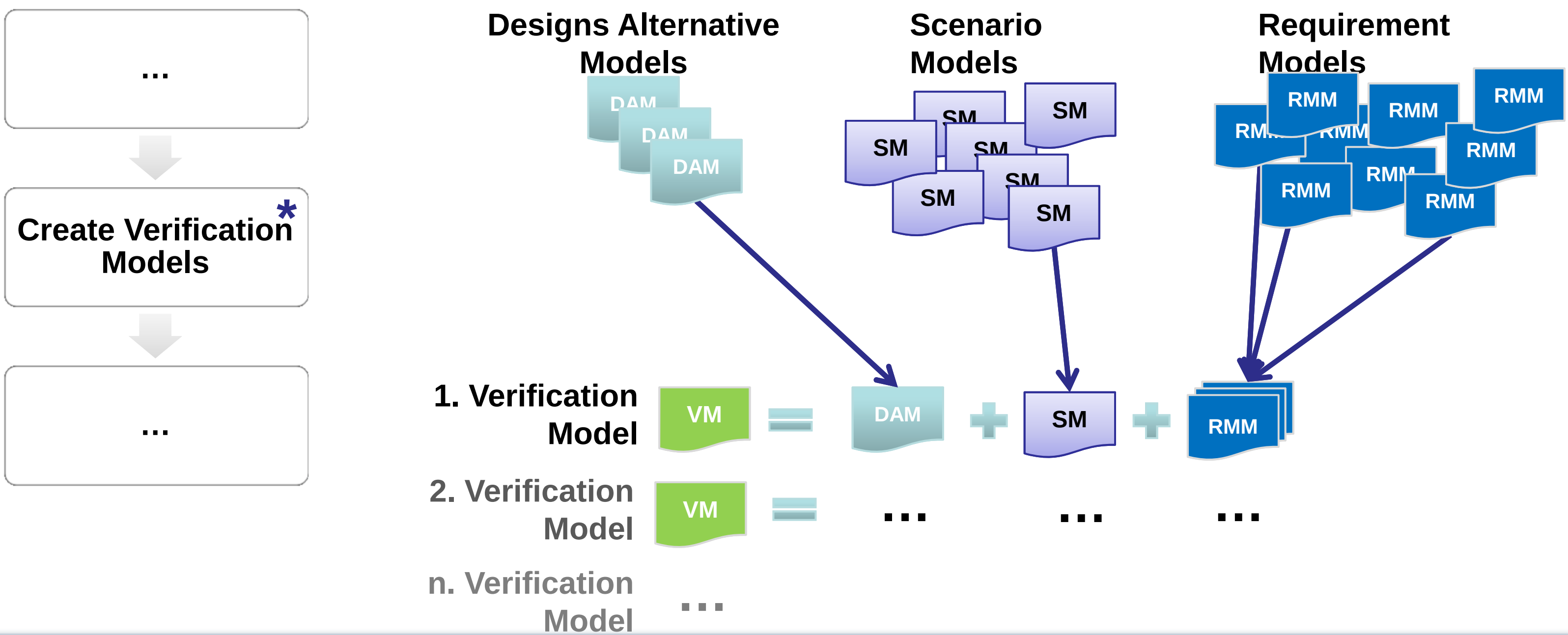
Focus of this presentation

Challenge

- We want to verify **different design alternatives** against **sets of requirements** using **different scenarios**. Issues:

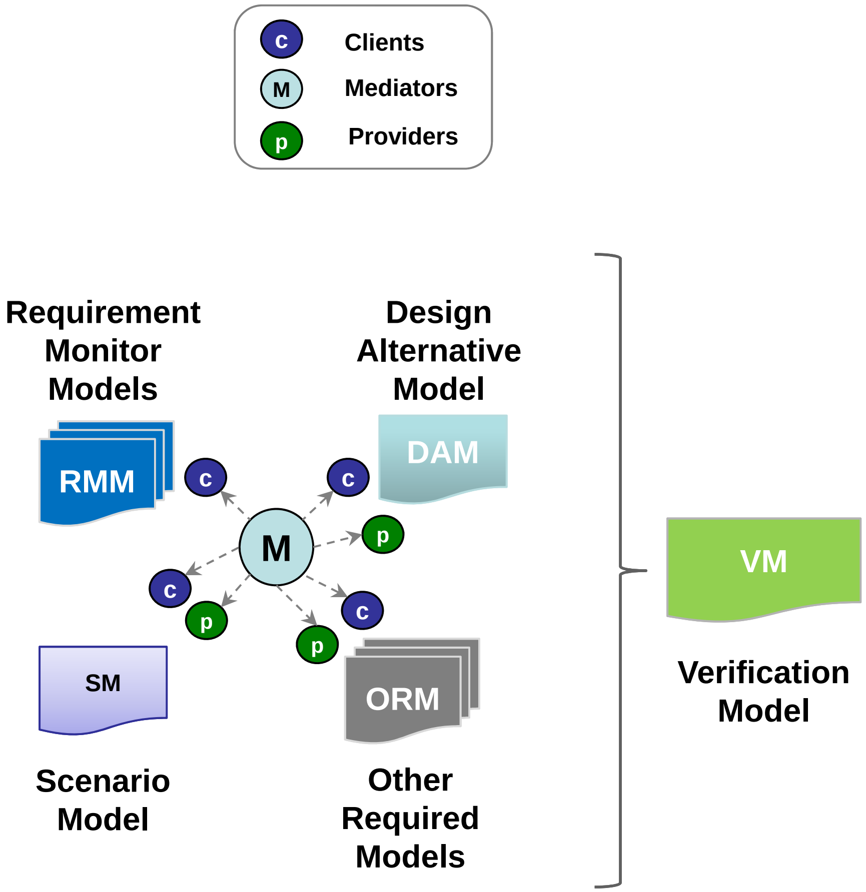
1) How to **find valid combinations** of **design alternatives**, **scenarios** and **requirements** in order to enable an automated composition of verification models?

2) Having found a valid combination: How to **bind all components correctly**?



Solution Proposal: Value Bindings

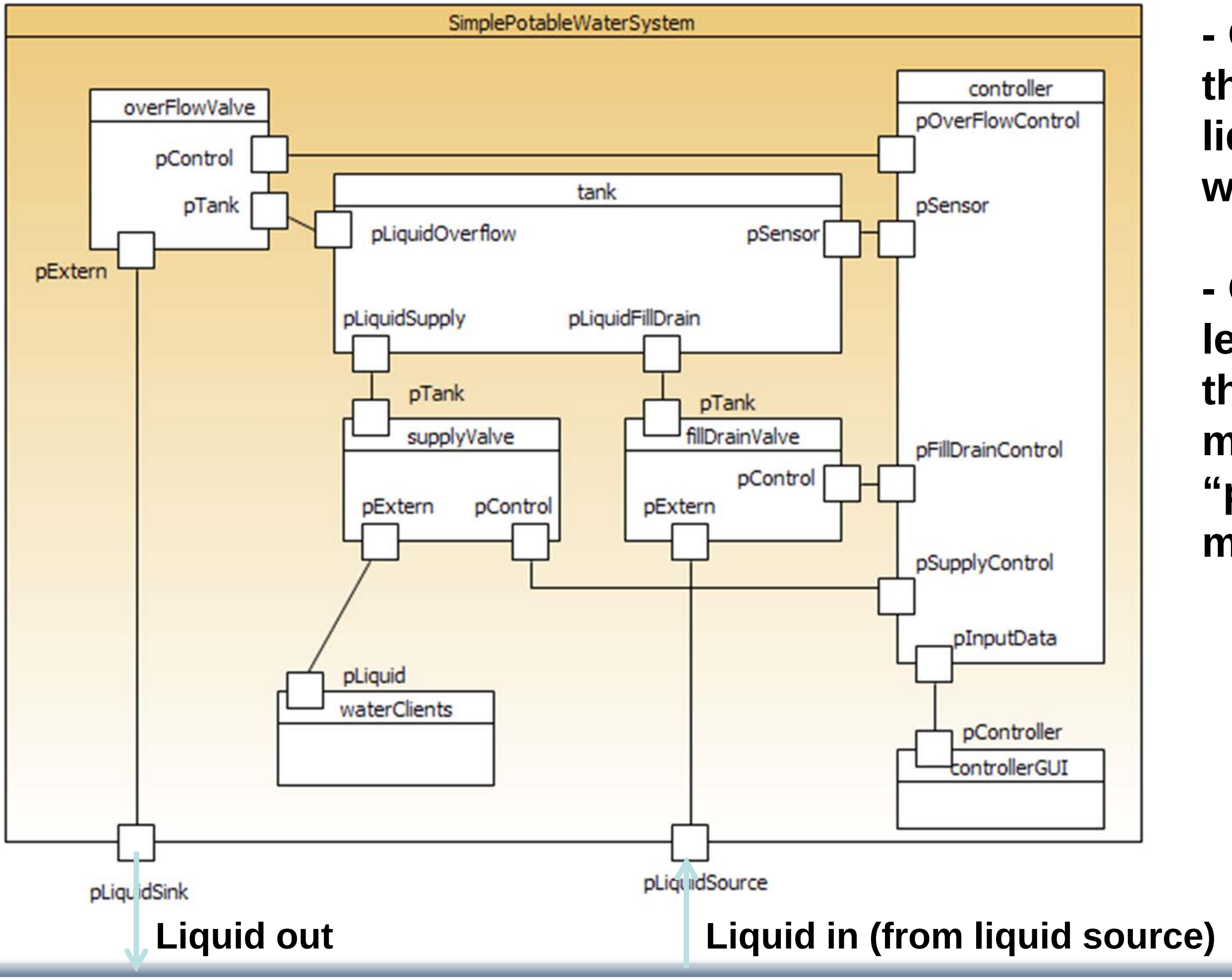
- **Value Binding** enables the automation of verification model composition
- Value Bindings include the definition of:
  - **Client** (component that requires data from other components)
  - **Provider** (component that provides data for other components)
  - **Mediator** (mediates between clients and providers)
- Depending on which mediators and providers are in place we can:
  - Determine which clients can be satisfied
  - **Find valid combinations** and generate verification models
  - Generate **binding** code for client components in verification models



MODELICA

Example: Design Alternative Model

- Simplified Aircraft Potable Water System



- Overhead tank system that can be filled using a liquid source from bottom with the aircraft on ground.

- Controller monitors the level of liquid and controls the valves according to its mode (e.g. “fill”-, “drain”-, “pre-selected value fill”-mode).

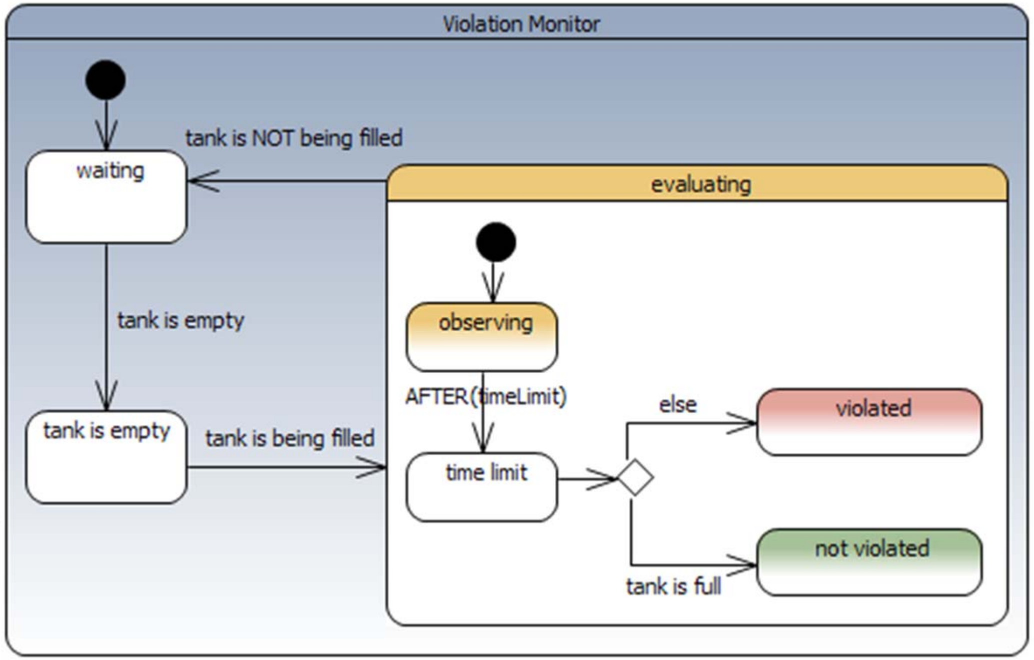
Example: Requirement Monitor Model

"The time to fill an empty tank shall be 300 sec. max."

Clients to get input values from design model providers

Rq 001 - Tank filling time

- input Boolean tankIsEmpty = false
- input Boolean tankIsBeingFilled = false
- input Boolean tankIsFull = false
- parameter Real timeLimit = 300
- output Integer status
- Violation Monitor



"status" is set by the violation monitor and indicates the following:

- **0 = not evaluated**
- **1 = evaluated and not violated**
- **2 = violated**

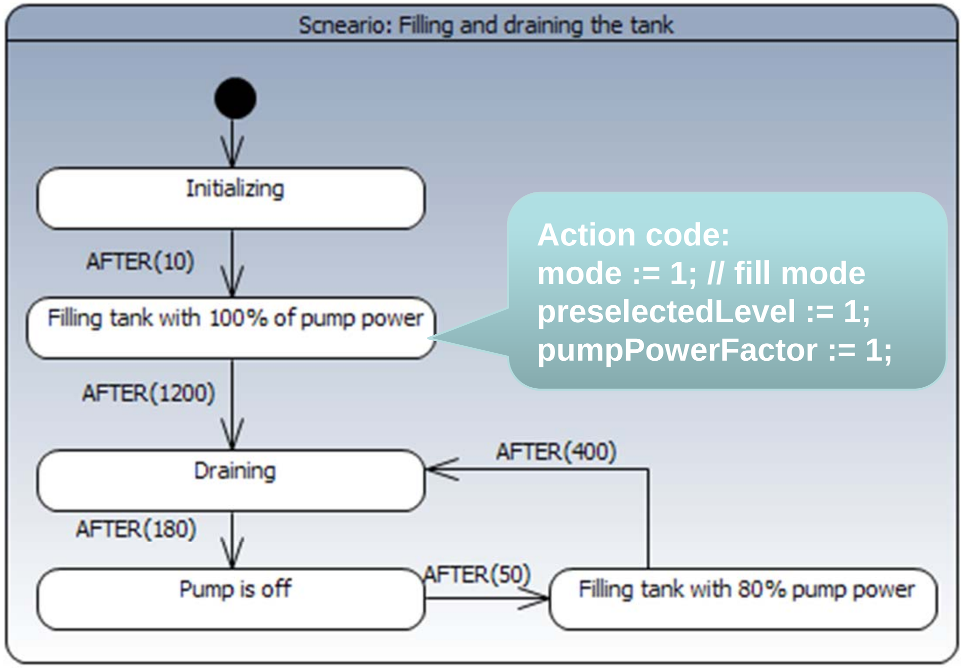
MODELICA

Example: Scenario Model “*Filling and draining the ta*

Providers for design model clients

s1-Fill and Drain Tank

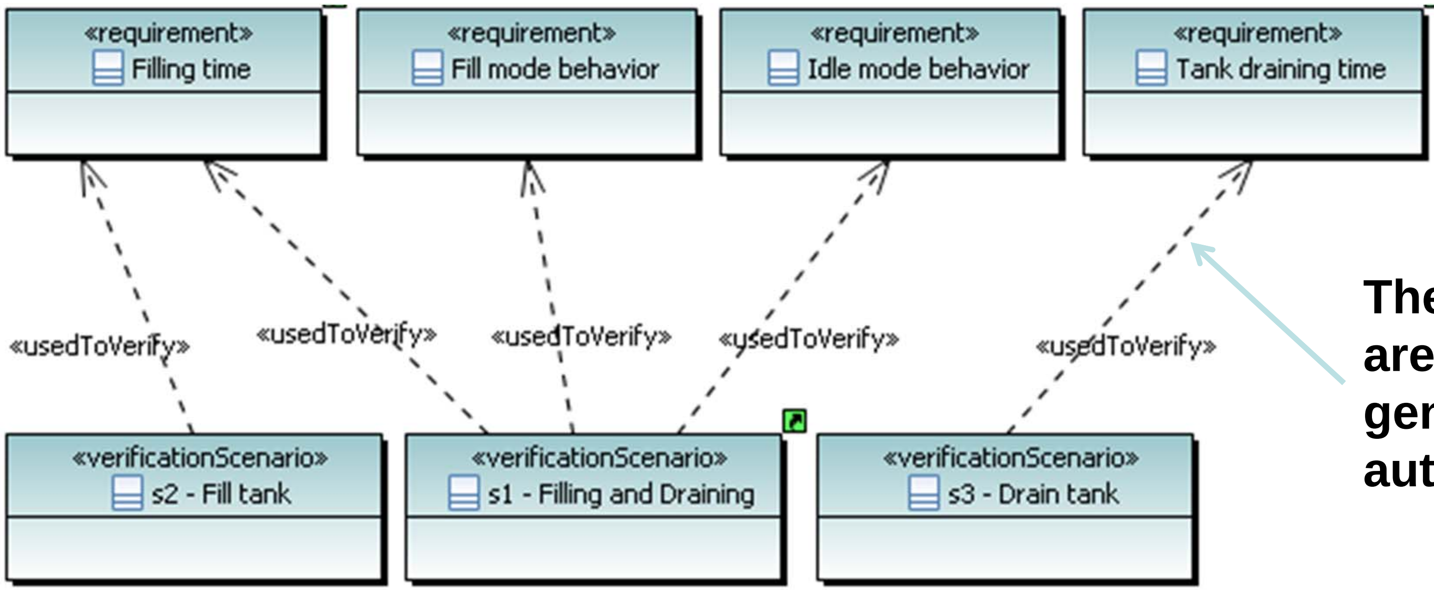
- parameter output Real ambPressure(..) = 101325
- output Integer mode(..)
- output Real pumpPowerFactor
- output Real preselectedLevel
- output Integer overflowValveStuckAt = 0
- output Integer fillDrainValveStuckAt = -1
- output Integer supplyVavleStuckAtPosition = 100
- input Real tankHeight = 1
- Sceneario: Filling and draining the tank



Example scenario: Tank cleaning by filling and draining the tank several times when the aircraft is on ground.

Example: Mapping Scenarios to Requirements

- Automatic generation/selection of which scenarios are appropriate to verify which requirements
  - **One scenario** can be used to **verify multiple requirements** (to increase requirements coverage and confidence in verification results)
  - Each requirement should be referenced by at least one scenario



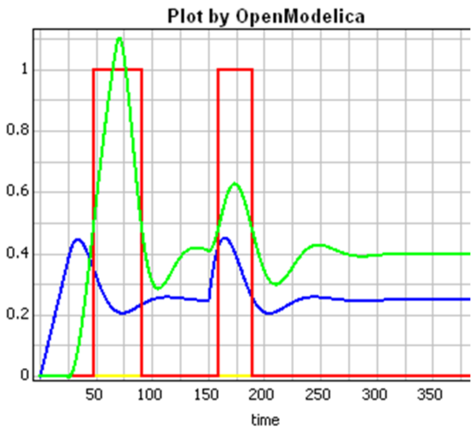
These relations are now generated automatically

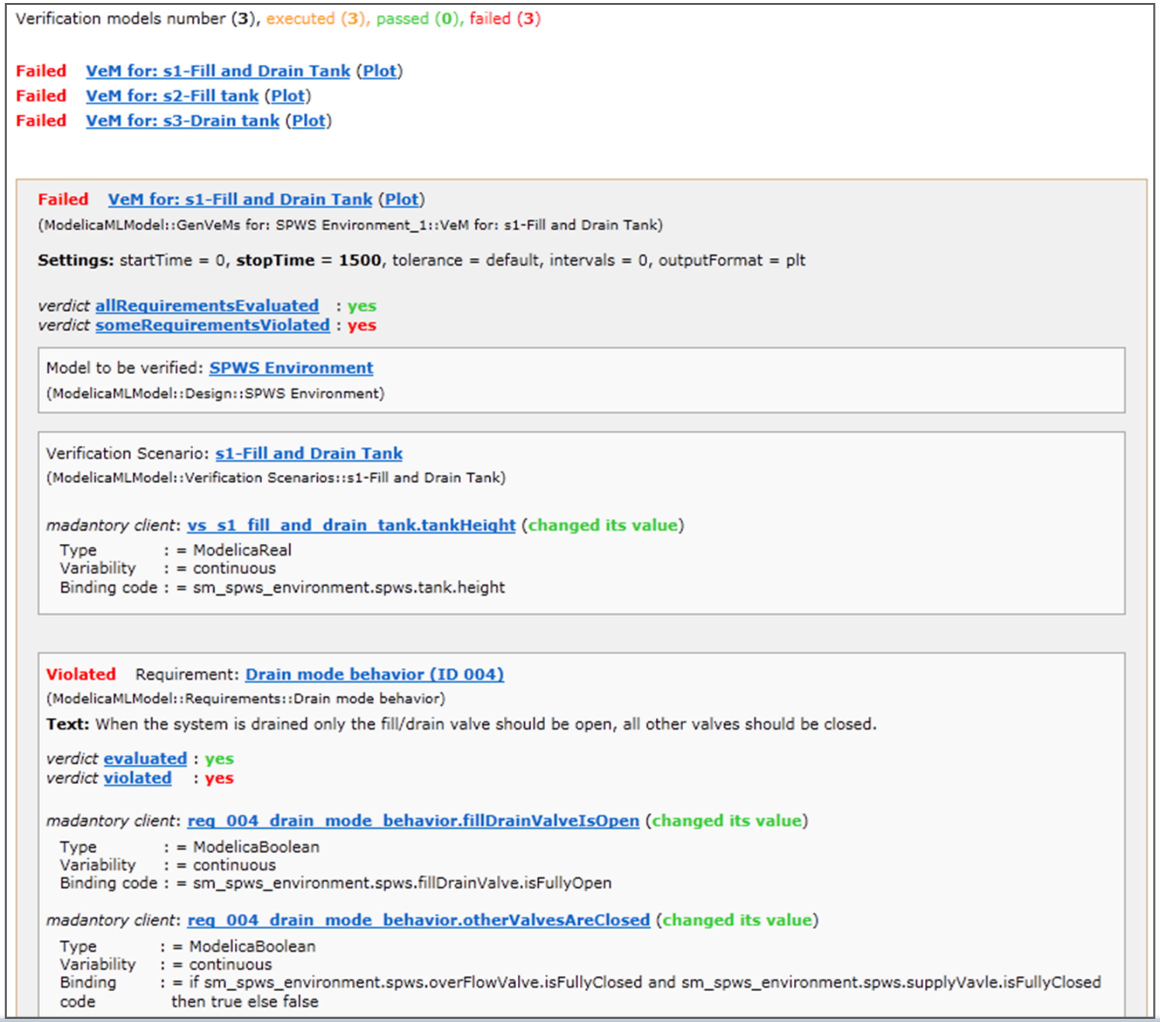
Simulation and Report Generation in ModelicaML

•Verification models are simulated.

The generated Verification Report is a prepared summary of:

- **Configuration, bindings**
- **Violations of requirements**
- **etc.**





Verification models number (3), executed (3), passed (0), failed (3)

Failed VeM for: s1-Fill and Drain Tank (Plot)
Failed VeM for: s2-Fill tank (Plot)
Failed VeM for: s3-Drain tank (Plot)

Failed VeM for: s1-Fill and Drain Tank (Plot)
(ModelicaMLModel::GenVeMs for: SPWS Environment_1::VeM for: s1-Fill and Drain Tank)

Settings: startTime = 0, **stopTime** = **1500**, tolerance = default, intervals = 0, outputFormat = plt

verdict allRequirementsEvaluated : yes
verdict someRequirementsViolated : yes

Model to be verified: SPWS Environment
(ModelicaMLModel::Design::SPWS Environment)

Verification Scenario: s1-Fill and Drain Tank
(ModelicaMLModel::Verification Scenarios::s1-Fill and Drain Tank)

madantory client: vs_s1_fill_and_drain_tank.tankHeight (changed its value)
Type : = ModelicaReal
Variability : = continuous
Binding code : = sm_spws_environment.spws.tank.height

Violated Requirement: Drain mode behavior (ID 004)
(ModelicaMLModel::Requirements::Drain mode behavior)
Text: When the system is drained only the fill/drain valve should be open, all other valves should be closed.

verdict evaluated : yes
verdict violated : yes

madantory client: req_004_drain_mode_behavior.fillDrainValveIsOpen (changed its value)
Type : = ModelicaBoolean
Variability : = continuous
Binding code : = sm_spws_environment.spws.fillDrainValve.isFullyOpen

madantory client: req_004_drain_mode_behavior.otherValvesAreClosed (changed its value)
Type : = ModelicaBoolean
Variability : = continuous
Binding code : = if sm_spws_environment.spws.overFlowValve.isFullyClosed and sm_spws_environment.spws.supplyVavle.isFullyClosed then true else false

Conclusion

- The ModelicaML Value Bindings approach enables automated model composition, which is used in ModelicaML for **automatic generate verification models**

- Bindings do not modify client or provider models (important when libraries are used)

- Using binding definitions we can find **valid combinations** and **automatically** generate **verification** models

- The generated **verification models** become artifacts that are **created automatically on-demand** and do not need to be maintained

Overall Summary

- Goal of integrated model-based development

This talk covers two aspects

- Integrated static/dynamic debugging of models
  - Dynamic debugging of large algorithmic models fully functional
  - Static Equation debugging prototype need to be integrated and scaled up for large models
- Requirements traceability and verification
  - Automated dynamic verification and generation of verification models
  - Need to be integrated in Modelica standard