# The Complexity Crisis

## Using Modeling and Simulation for System Level Analysis and Design

**Prof. Dr. François E. Cellier**
**Computer Science Department**
**ETH Zurich**
**Switzerland**







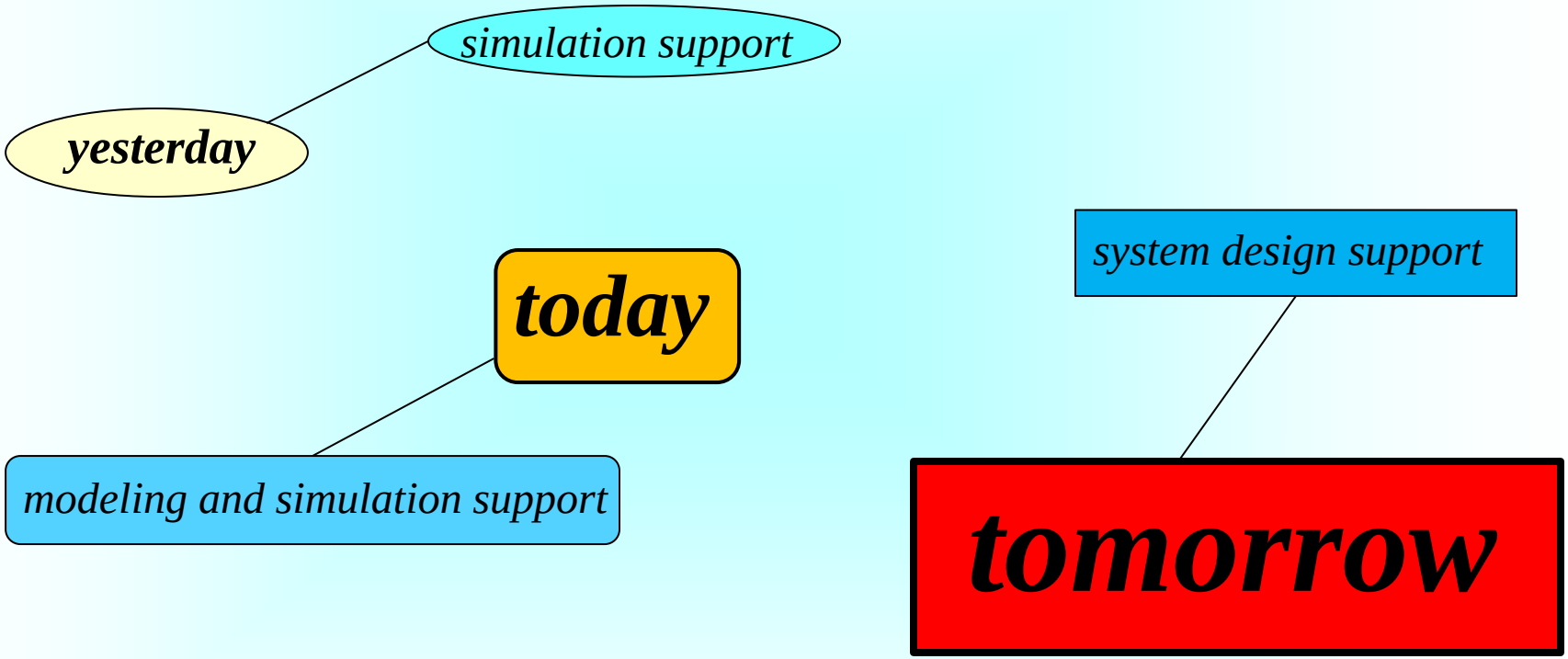
# Acknowledgments

***Dr. Ernesto Kofman*** (National University of Rosario, Argentina) is the originator of the *Quantized State System (QSS) Solvers* discussed in this presentation

***Xenofon Floros*** (ETH Zurich, Switzerland) is a Ph.D. student of mine working on multi-core implementations of QSS solvers and on making QSS solvers accessible from within Modelica

***Markus Andres*** and ***Thomas Schmitt*** (University of Applied Sciences of the Vorarlberg, Austria) were two MS students of mine who implemented free Modelica libraries for modeling tires and motorcycles

# Modeling and Simulation



# Modeling and Simulation

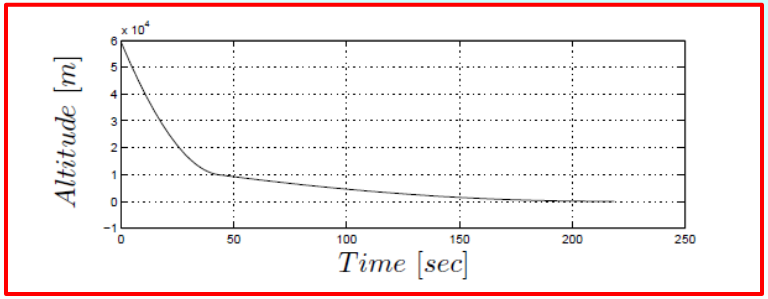
# The Past







# ACSL: The Advanced Continuous-system Simulation Language of the 1980s




```
PROGRAM Lunar Landing Maneuver
  INITIAL
    constant ...
      r = 1738.0E3,  c2 = 4.925E12,  f1 = 36350.0, ...
      f2 = 1308.0,  c11 = 0.000277,  c12 = 0.000277, ...
      h0 = 59404.0,  v0 = −2003.0,  m0 = 1038.358, ...
      tmx = 230.0,  tdec = 43.2,  tend = 210.0
    cinterval cint = 0.2
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      thrust = (1.0− step(tend)) * (f1 − (f1 − f2)*step(tdec))
      c1     = (1.0− step(tend)) * (c11 − (c11 − c12)*step(tdec))
      h      = integ(v, h0)
      v      = integ(a, v0)
      a      = (1.0/m) * (thrust − m * g)
      m      = integ(mdot, m0)
      mdot   = −c1*abs(thrust)
      g      = c2/(h + r) * *2
    END $ "of DERIVATIVE"
    termt (t.ge.tmx .or. h.le.0.0  .or. v.gt.0.0)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```







# ACSL: The Advanced Continuous-system Simulation Language of the 1980s

```
PROGRAM Lunar Landing Maneuver
  INITIAL
    constant ...
      r = 1738.0E3,  c2 = 4.925E12,  f1 = 36350.0, ...
      f2 = 1308.0,  c11 = 0.000277,  c12 = 0.000277, ...
      h0 = 59404.0,  v0 = -2003.0,  m0 = 1038.358, ...
      tmx = 230.0,  tdec = 43.2,  tend = 210.0
    cinterval cint = 0.2
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      thrust = (1.0- step(tend)) * (f1 - (f1 - f2)*step(tdec))
      c1     = (1.0- step(tend)) * (c11 - (c11 - c12)*step(tdec))
      h      = integ(v, h0)
      v      = integ(a, v0)
      a      = (1.0/m) * (thrust - m * g)
      m      = integ(mdot, m0)
      mdot   = -c1*abs(thrust)
      g      = c2/(h + r) * *2
    END $ "of DERIVATIVE"
    termt (t.ge.tmx .or. h.le.0.0  .or. v.gt.0.0)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

The typical model contained:

- up to half a dozen differential equations
- up to a few dozen algebraic equations

The simulation was performed using:

- a forth-order explicit Runge-Kutta solver (default)
- a stiff system implicit BDF solver was offered in case it should ever be needed







# Summary

- ✓ Simulation languages like ACSL offered very little modeling support.
- ✓ Once the models got more complex, the code became quickly as unreadable as if it had been coded in Fortran.
- ✓ The primary success of languages like ACSL was based on the fact that they protected the user from having to understand how the simulation was done.
- ✓ The modeler did not need to understand how numerical ODE solvers work.
- ✓ Most models could be simulated successfully using the default Runge-Kutta solver. Low-order models are rarely stiff. However, a stiff system solver was usually offered as an option in case it should ever be needed.
- ✓ Later versions of ACSL and similar tools offered primitive versions of event handling for dealing with small numbers of discontinuous functions in the model.




# Modeling and Simulation

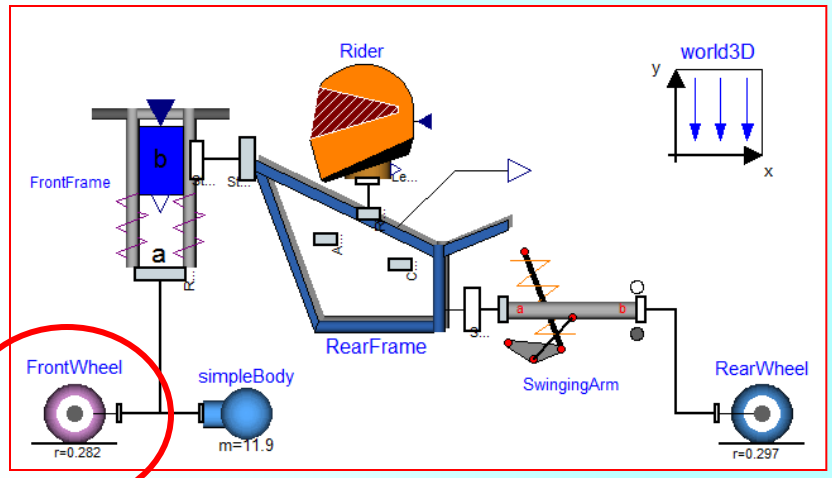
# The Present

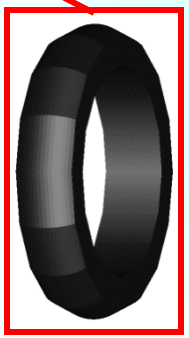


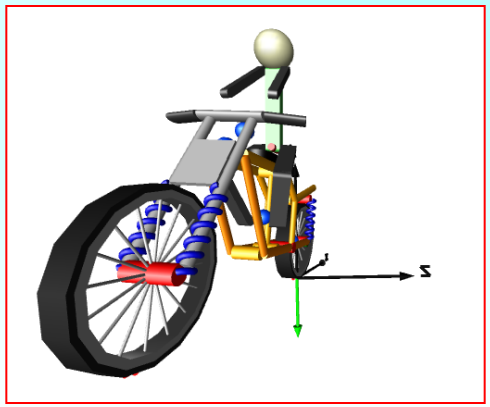




# Model-based Design of Systems






What does it buy us?

# Some Common Myths …

- Model-based design saves time and money, because it is much simpler to design a model than to build a real system.
  - ➢ To design a reliable model of a complex system is just as difficult and time-consuming as designing a real system.
- Models are much cheaper than real systems. Models of most system components are readily available and can be downloaded from the web for free.
  - ➢ You get what you pay for. There may indeed exist generic models for many system components on the web, but will they be compatible with your overall system model, and will they even be compatible among each other? Models of specific components are rarely free. They are proprietary and can be as expensive as the component they represent.




# Advantages of Model-based Design

- Compatible models are much quicker to assemble than real systems. Thus, time and money is not saved in the design of the component models themselves, but rather in their composition when building models of more complex systems.
- Experimentation with models is much easier and faster than with real systems. Thus, model optimization can lead to better system designs.
- Models can be duplicated for free, i.e., different designers can use the same model in parallel.
- Models can be made (but are not necessarily) more easily reusable than real system components. Adapting an existing model (if designed well) to modified needs is easier and cheaper than modifying a real system component for the same purpose.
- Models can be made self-documentary, whereas real system components always require separate spec sheets that may or may not provide correct information about the component.




# Principles of Good Model-based Design

- Since the design of good component models can be quite expensive, ***model reusability*** should be given top priority in model design.
- Model reusability is enhanced by ***self-documentation*** of models. Make each model as easily understandable as possible.
- ***Graphical models*** are better readable than equation models due to their two-dimensional nature. Try to model graphically down as far as possible. Ultimately, there most be added an equation layer, but that layer may be made so generic that it can be designed once and for all.
- ***Small is beautiful!*** If a model doesn't fit on a single computer screen, it is poorly designed and will likely fail when run with unforeseen parameter combinations.
- ***Don't ever write spaghetti code!*** The irreplaceable programmer should be fired at once.
- The principles of good ***modular model design*** are no different from the principles of good modular system design.




# Principles of Model-based Design II

- The design of a clean and fully documented ***model interface*** is even more important than the design of the model itself.
- A good modeling environment should allow a modeler to modify or even redesign from scratch any component model. As long as the interface remains unchanged, the new component model should be able to replace the former implementation without any need to change the embedding model.
- ***Model interface errors*** are more common than errors caused by incorrect model specifications. They are usually caused by unwritten limitations on a model interface.
- It is thus important to ***design the model interfaces around sound physical concepts***, such as energy flows. A model that connects to a neighboring model using an interface that represents an energy interchange port is unlikely to fail.




# The *Modelica* Modeling Environment

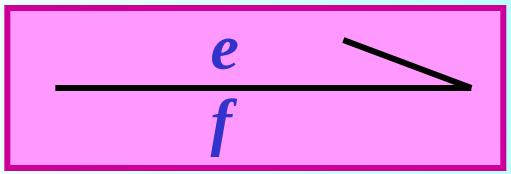
- ***Modelica*** is currently the best physical systems modeling environment on the market.
- ***Modelica*** supports object-oriented modeling featuring ***encapsulation*** (information hiding), ***decomposition*** (hierarchical modeling), ***inheritance*** (each modeling concept needs to be coded only once), and ***abstraction*** (a graphical user interface).
- The ***Modelica Standard Library (MSL)*** offers a large set of (generic) component models from a wide variety of physical domains.
- Whereas Modelica implementations may be proprietary, the MSL is maintained by a standard committee and has been placed in the public domain.
- There exist a large number of additional (free as well as commercial) Modelica libraries accompanying the MSL.
- Modelica is used widely both in academia and by industry.





# Bond Graphs

- Bond graphs offer a generic (domain-independent) approach to object-oriented graphical modeling of physical systems.
- The bond represents energy flow in a physical system. It carries two variables, an effort, ***e***, and a flow, ***f***, the product of which represents power.




$$P = e \cdot f$$

***e: Effort***
***f: Flow***

***Examples:***

$$P_{el} = u \cdot i$$
$$P_{mech} = f \cdot v$$

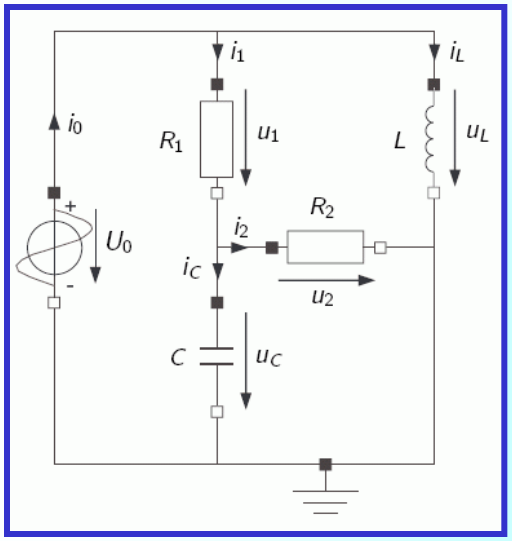
$$[W] = [V] \cdot [A]$$
$$= [N] \cdot [m/s]$$
$$= [kg \cdot m^2 \cdot s^{-3}]$$

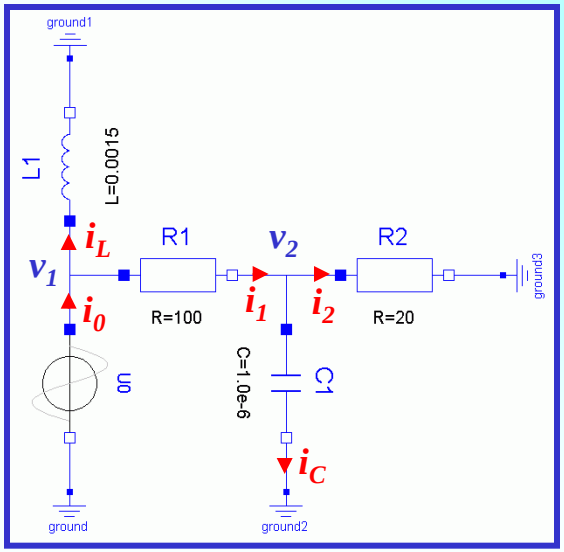


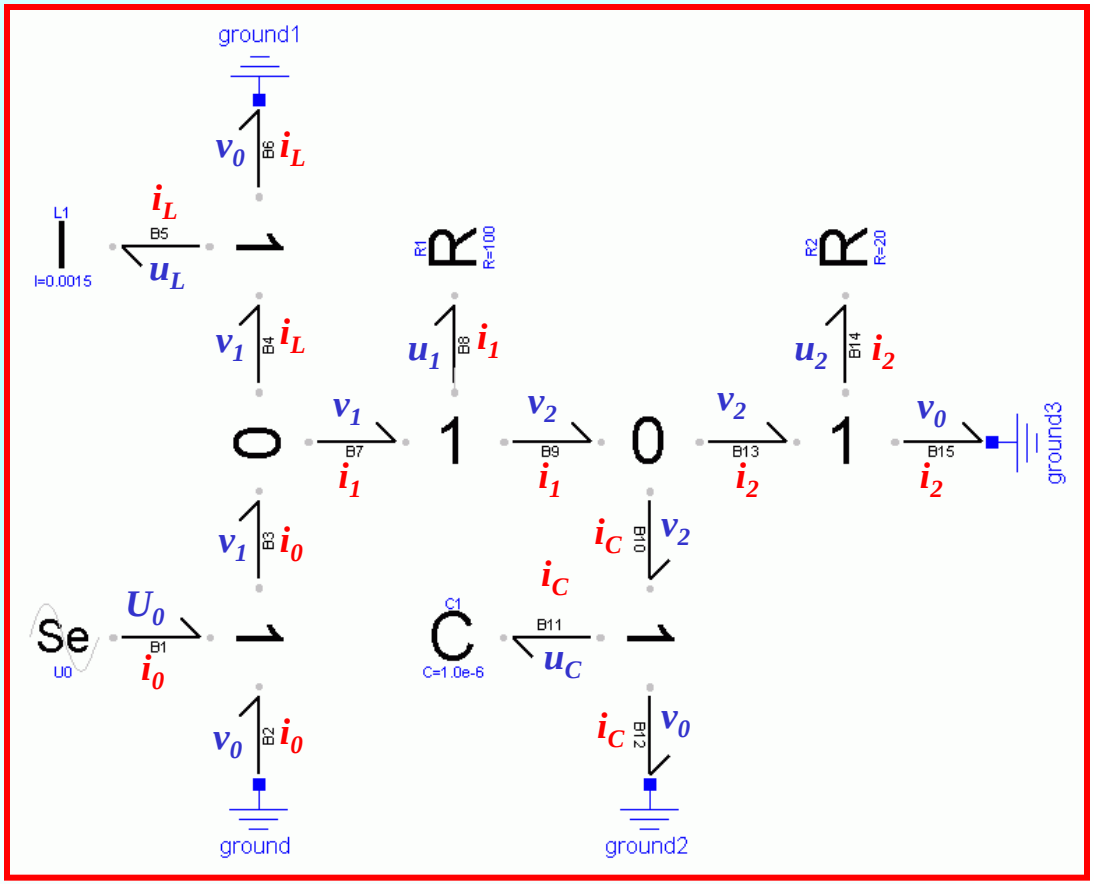




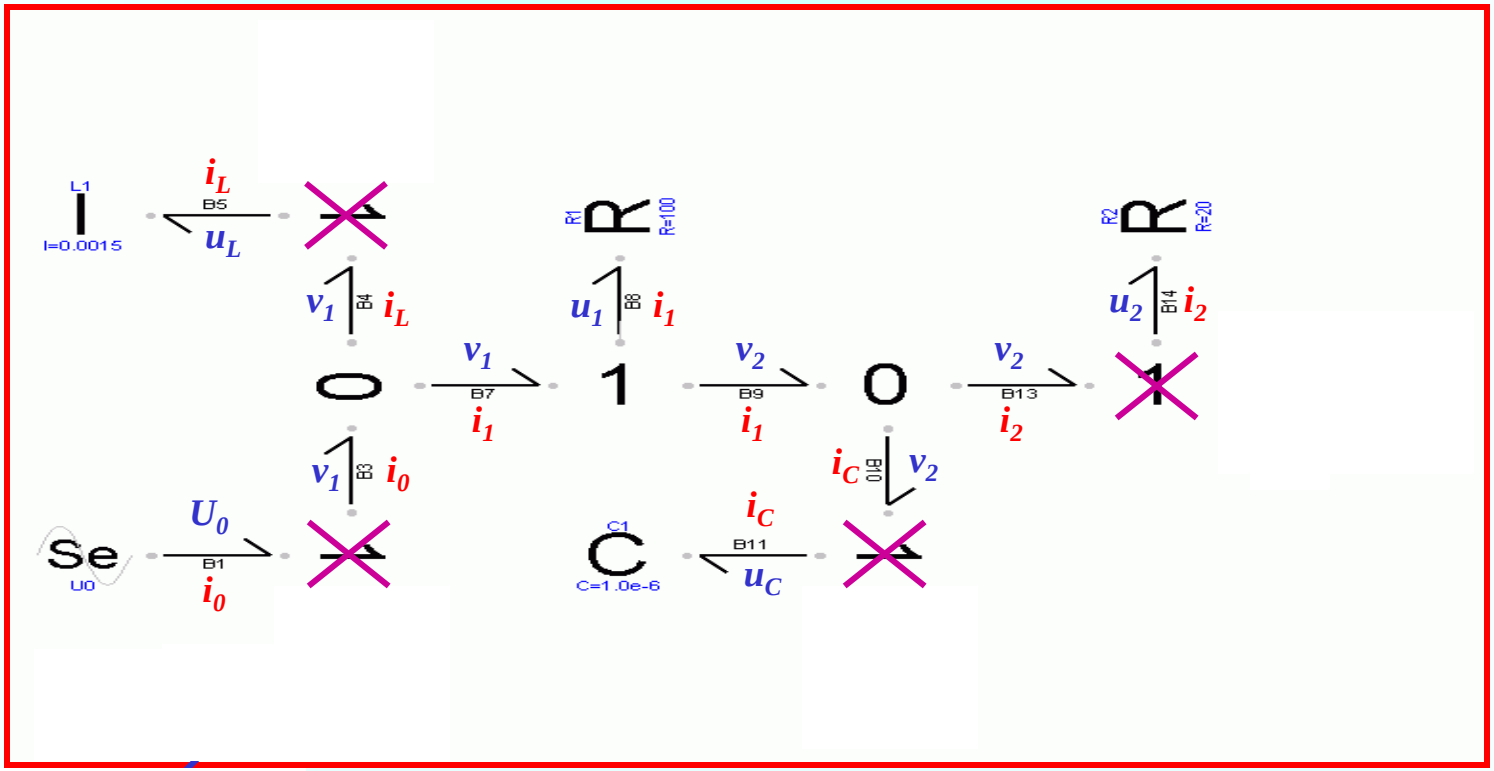
# Example: Electrical Circuit



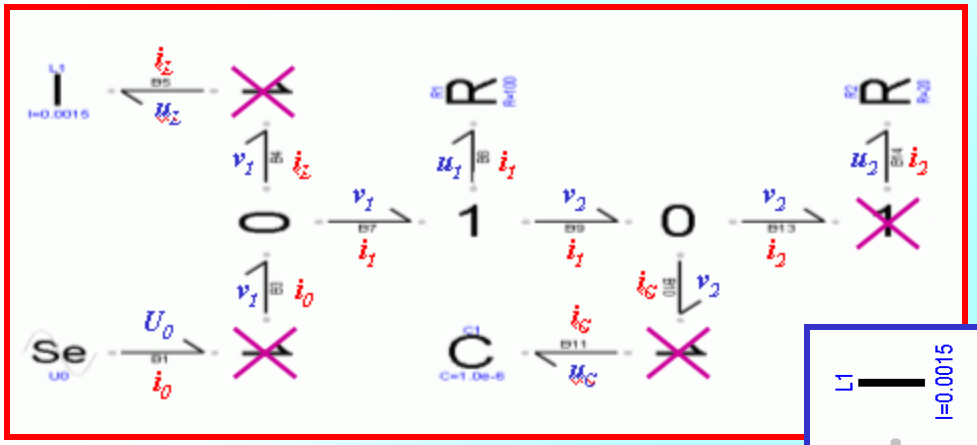



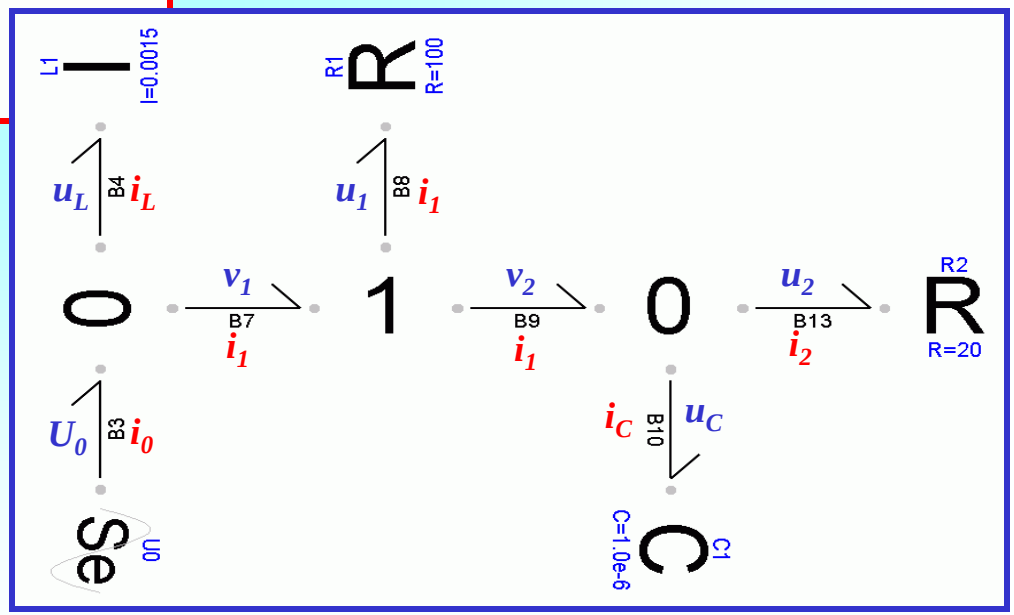



# An Example II



# An Example III



# Bond Graphs II

- Bond graphs offer the most primitive graphical user interface that is still fully object-oriented.
- Bond graph-based libraries enable the model designer to continue using graphical modeling as far down as possible.
- The underlying lower-most equation-based model layer is so simple and so generic that it can be coded once and for all.
- The model designer thus rarely needs to code any equations at all.
- The basic (leaf) models are simple and easily maintainable.
- However, bond graphs rarely represent a suitable end-user interface.
- Model abstraction is being used to wrap bond graph models into higher abstraction layers that are more comfortable to use.

# General Bond Graph Libraries from ETH

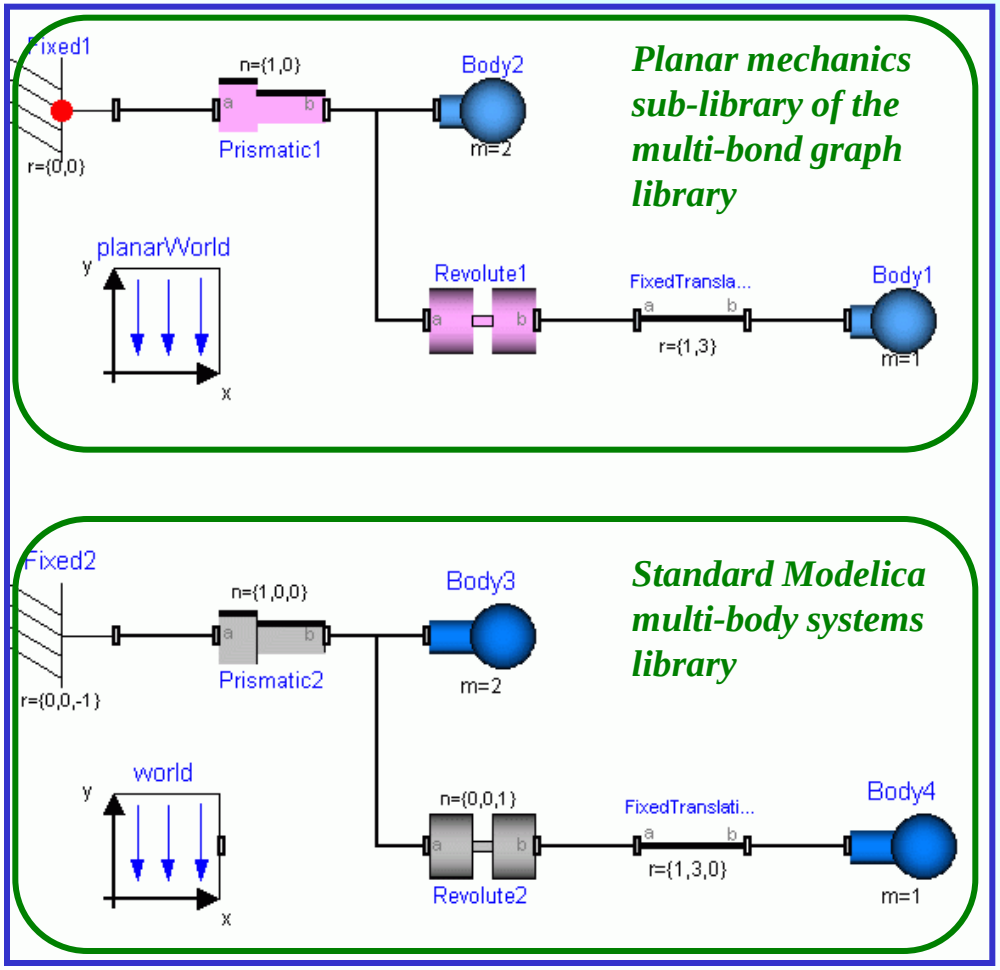
- ETH Zurich maintains three general bond graph libraries that are all in the public domain.
- They represent an alternative to the MSL in many respects.
- A disadvantage might be that these libraries are used by a smaller number of customers, and consequently, the models contained in them may be less thoroughly tested.
- An advantage is that the average bond graph models are smaller in size and therefore more easily understandable and maintainable.

# ETH General Bond Graph Libraries II

- ***BondLib*** is a library consisting of models making use of scalar (black) bonds. Sub-libraries exist for electrical circuits (including an implementation of SPICE), for mechanical 1D models, for hydraulics (not in the MSL) and pneumatics (not in the MSL), and for irreversible thermodynamics.
- ***MultiBondLib*** is a library consisting of models making use of (blue) vector bonds, called multi-bonds. Sub-libraries exist for multi-body system dynamics, especially 2D mechanical systems (not in the MSL), 3D mechanical systems, 3D mechanical systems with ideal impacts (not in the MSL), and 3D mechanical systems in a gravitational pool (not in the MSL).
- ***ThermoBondLib*** is a library consisting of models making use of (red) vector bonds representing simultaneous flows of mass, volume, and heat. This library is primarily useful for the description of convective flows (computational fluid dynamics) and chemical reaction dynamics.

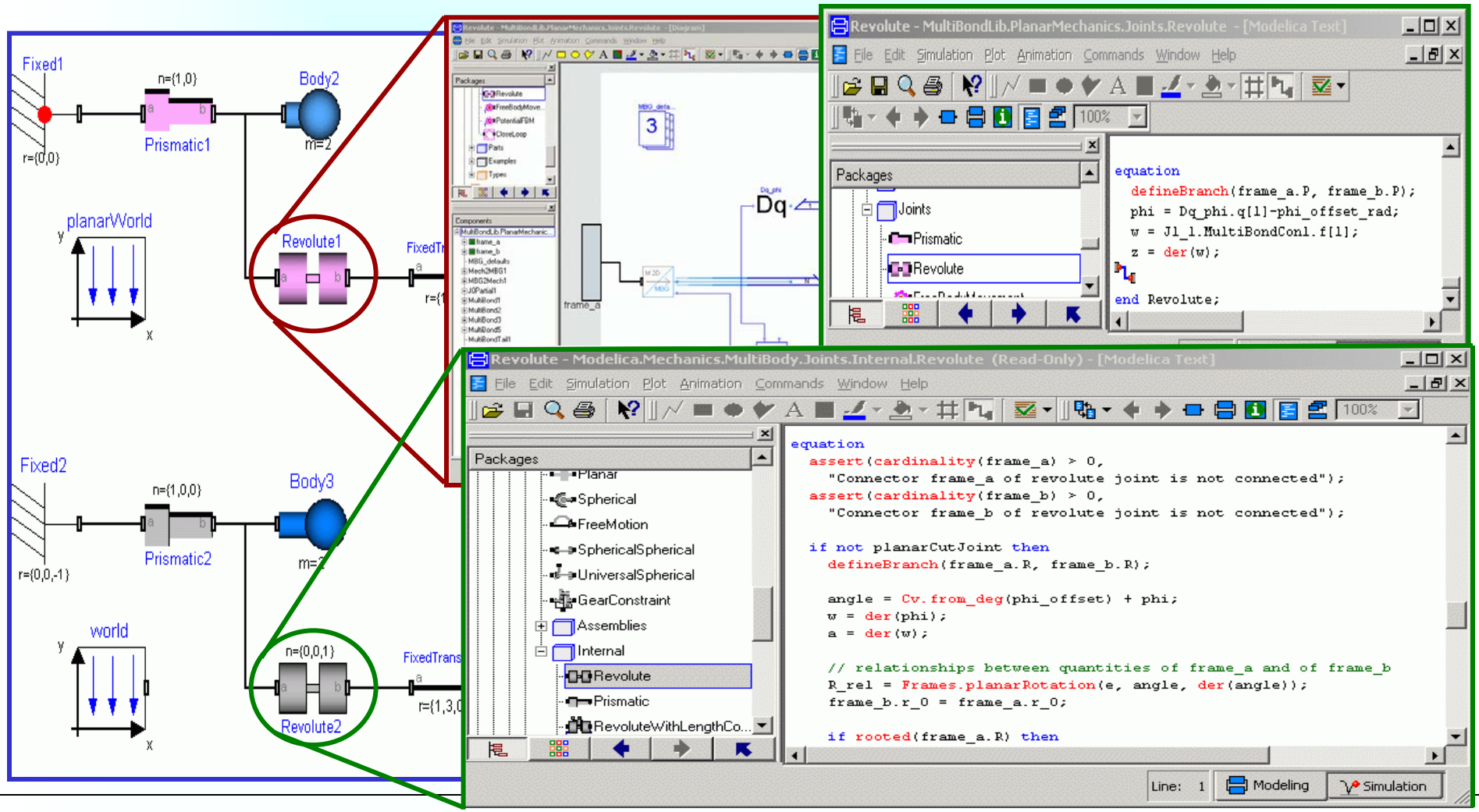



# A Crane Crab

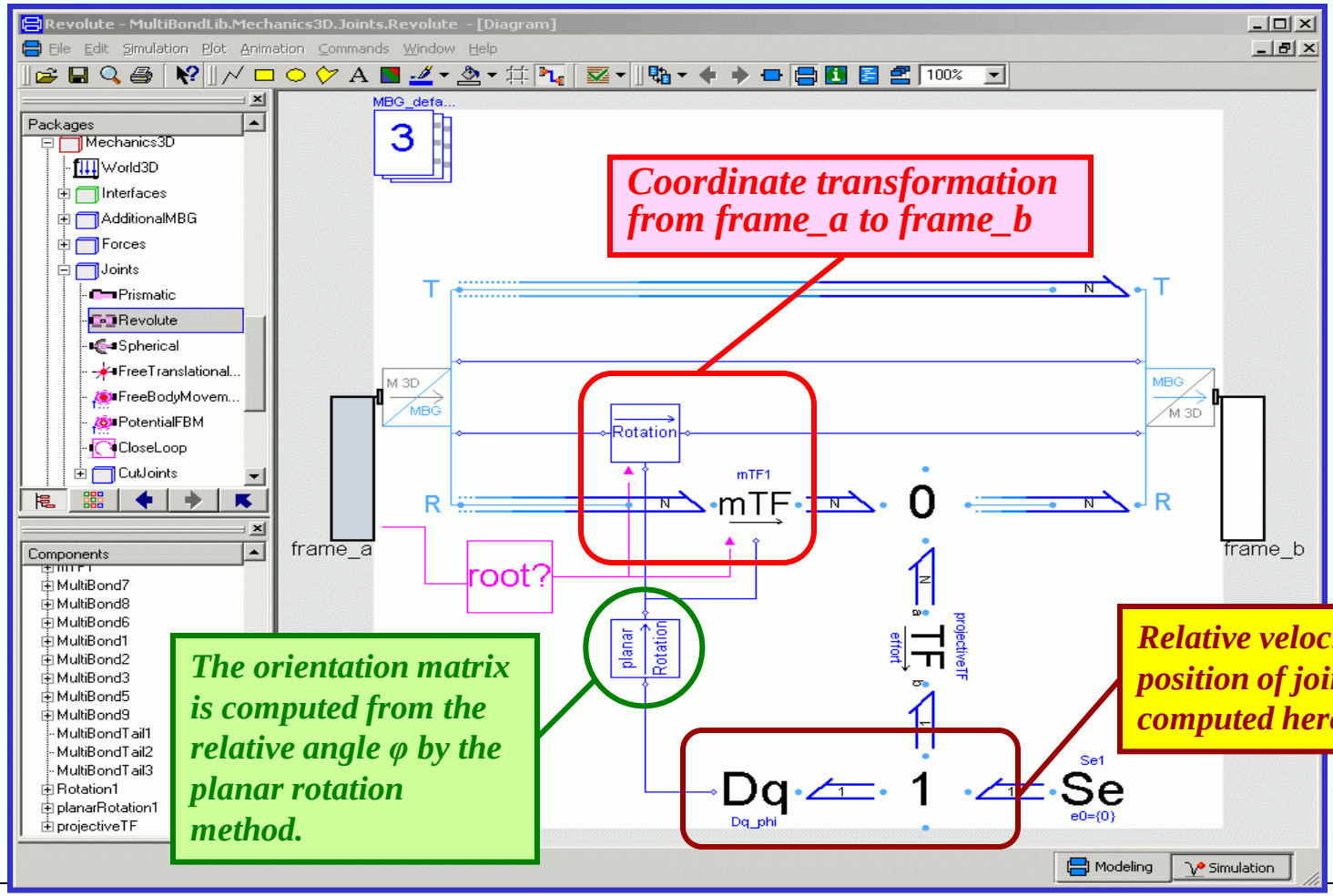



- The standard Modelica multi-body systems library is a general-purpose 3D mechanics library. No separate support for planar mechanics is currently being offered.
- The multi-bond graph library contains separate sub-libraries for planar mechanics and 3D mechanics, as well as for modeling hard collisions between mechanical bodies and for modeling gravitational pools (celestial mechanics).

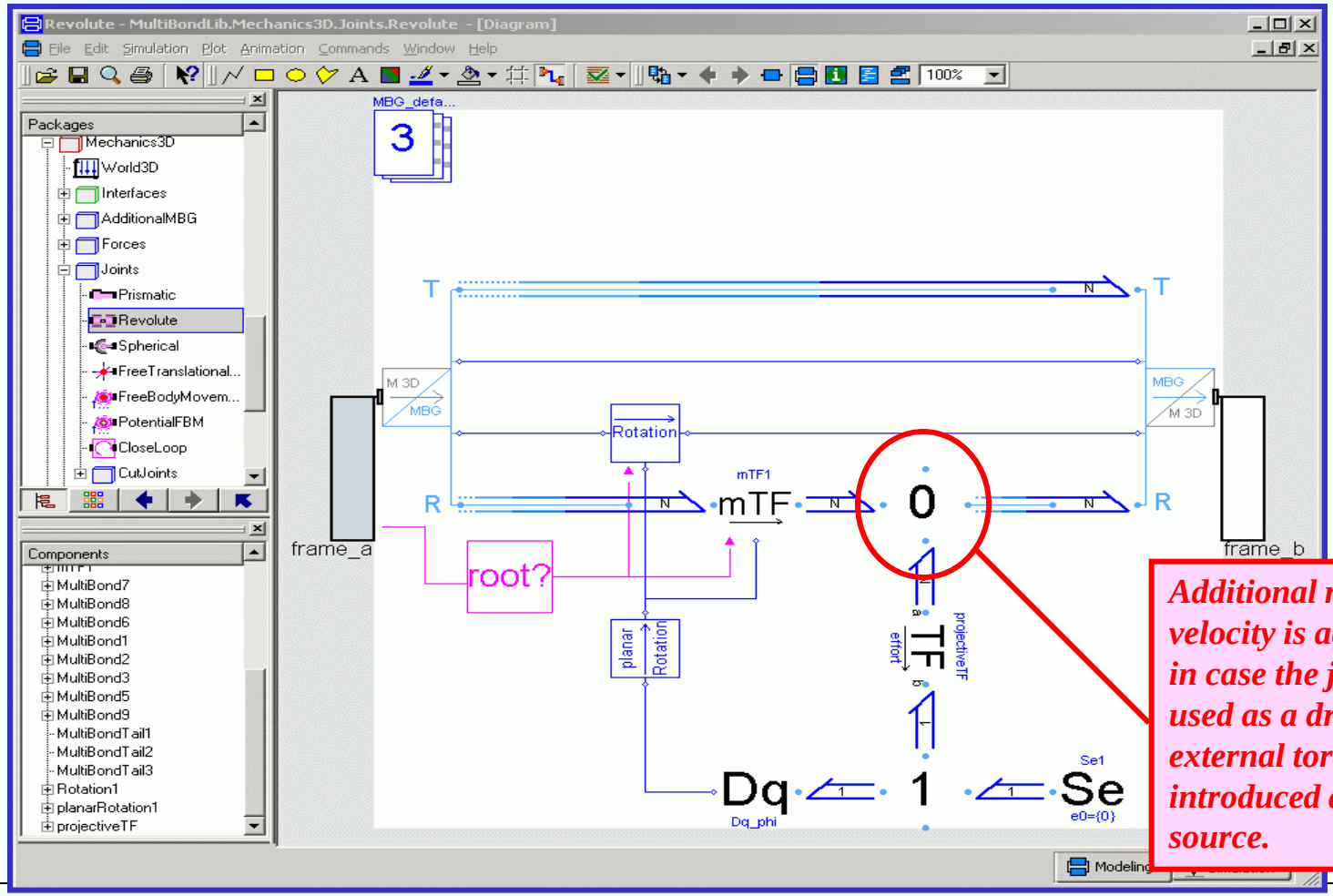
# Revolute Joints



# The 3D Revolute Joint



# The 3D Revolute Joint II



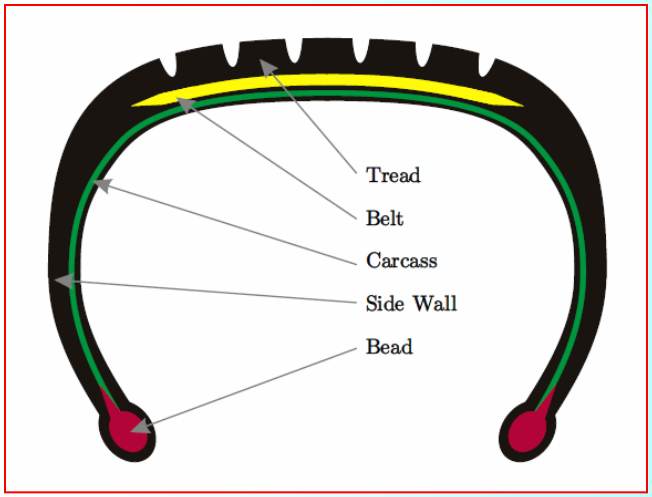

***Additional rotational velocity is added here, in case the joint is being used as a drive, i.e., if external torque is being introduced at the effort source.***

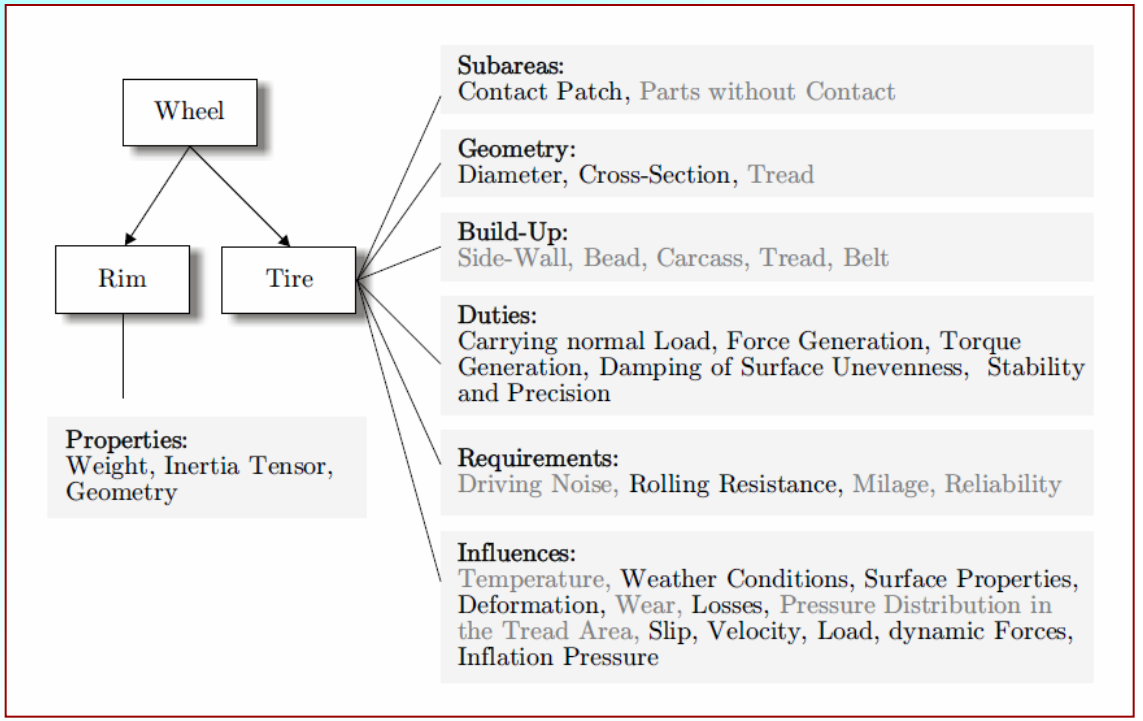
# Modularization of Model Design

- A car manufacturer chooses what his new car model should look like and what gadgets should be implemented. Many vehicle components are however being built by other companies. The new model may feature an engine from Japan and a transmission from Brazil. Car manufacturers would never dream of manufacturing the tires of their vehicles by themselves. Those are being designed and built by tire specialists, such as Michelin or Goodyear.
- The same applies to model-based design. Car manufacturers should not have to worry about creating models of their tires. They should be able to acquire tire models that are reliable and that were created by tire specialists.
- A good tire model is paramount to being able to test the driving characteristics of the new vehicle already in the simulation model. Such a model is quite complex, and designing it is no easy task.

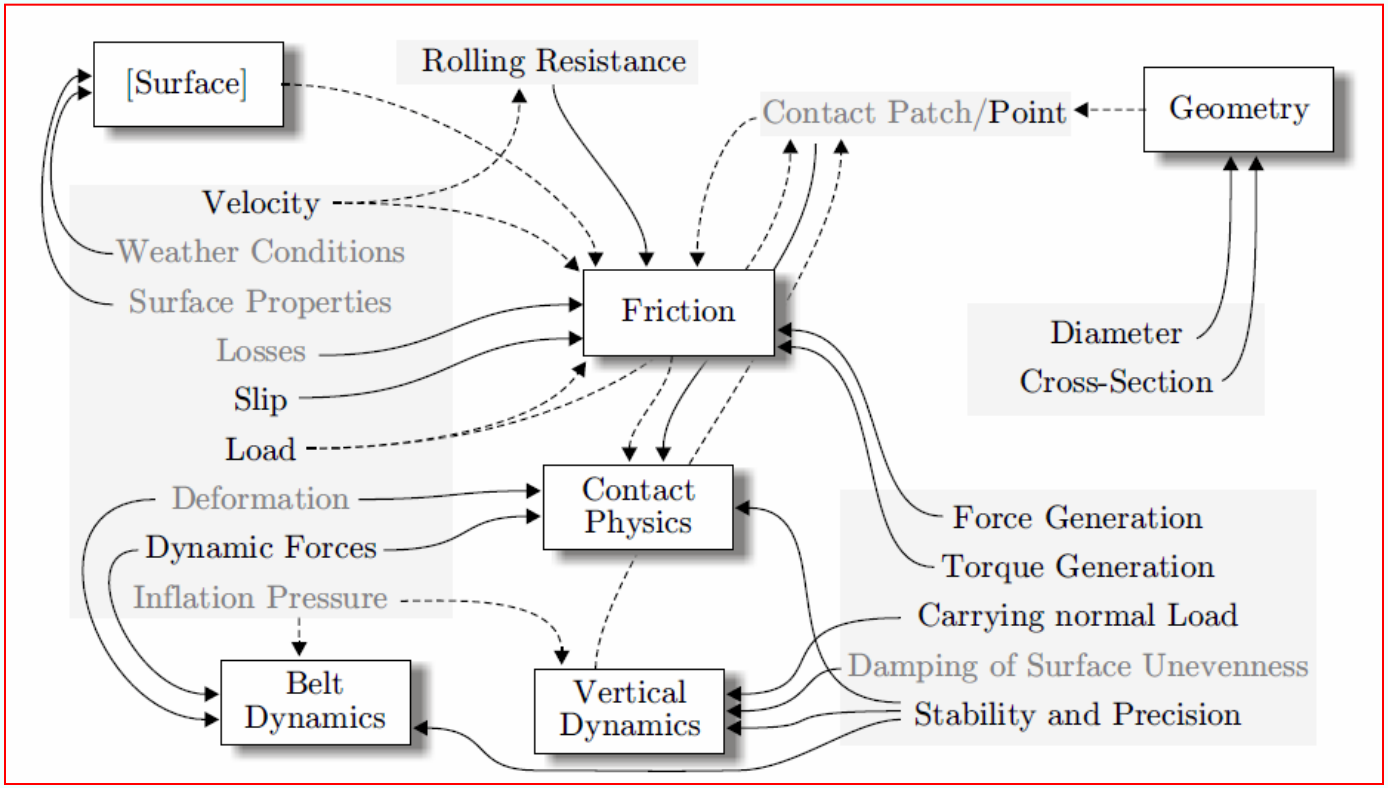



# Wheels and Tires

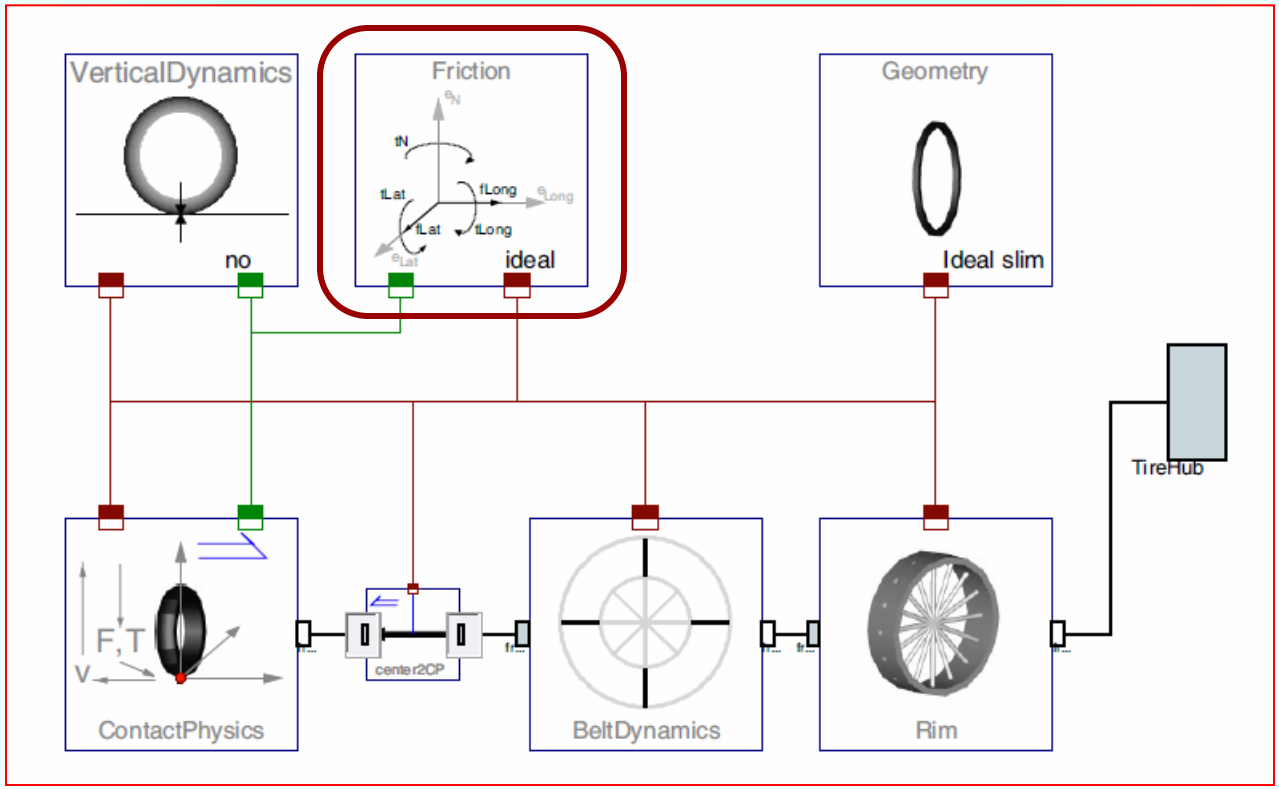


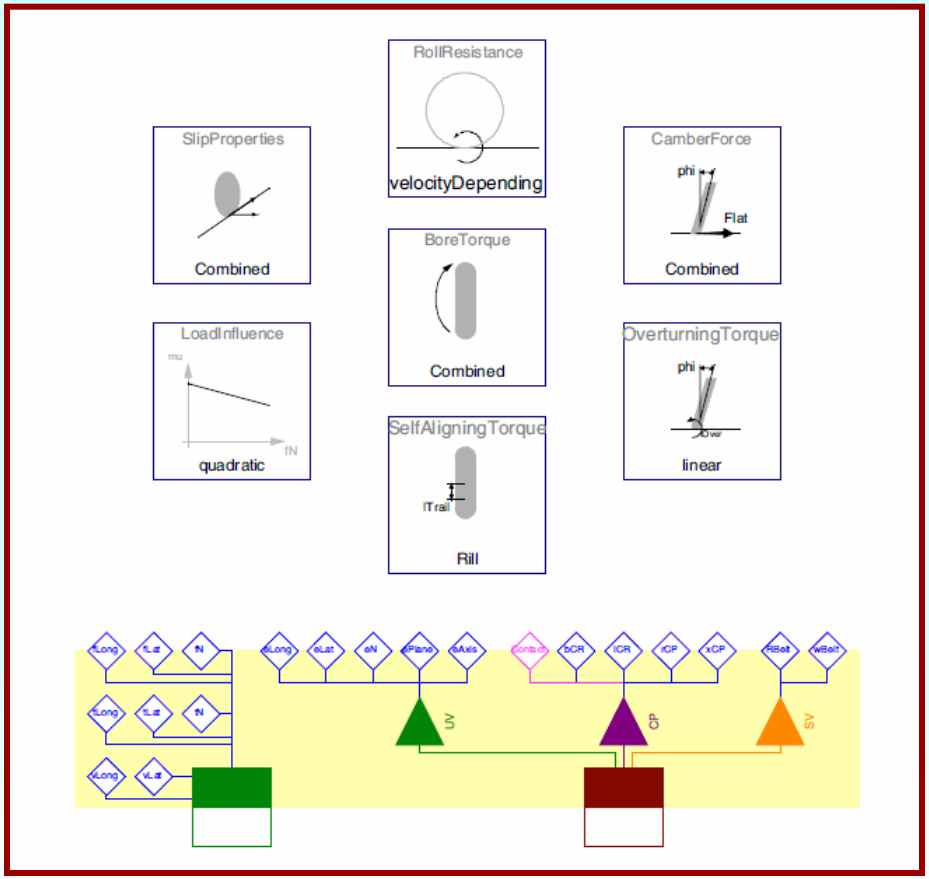
# Wheels and Tires: Influences



# Wheels and Tires: Modular Design



# Wheels and Tires: Friction Model



# Wheels and Tires: Summary

The freely available Library *Wheels and Tires* provides:

- A framework for the implementation of tire models or parts of these.
- A well-defined and expandable structure that allows a convenient customization.
- A number of predefined ready-to-use tire models.
- *Test Bench* models for testing of basic tire properties.
- Examples showing the application of the tire models.

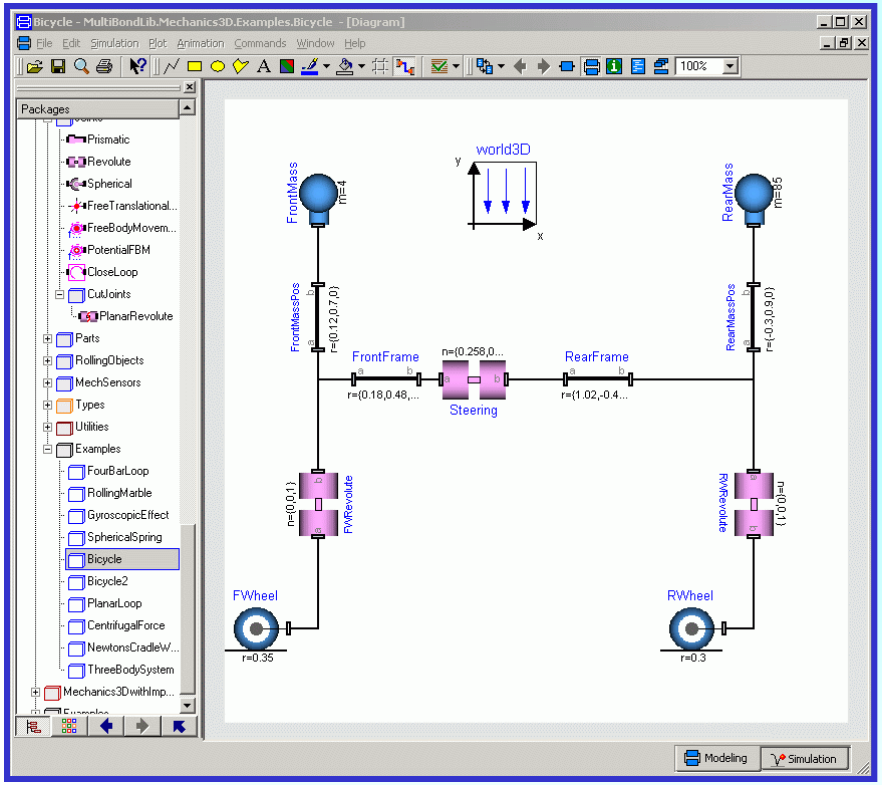
Our free ***Wheels and Tires*** library does not contain parameter sets for any commercial tires currently on the market. To provide those is typically the realm of commercial libraries.

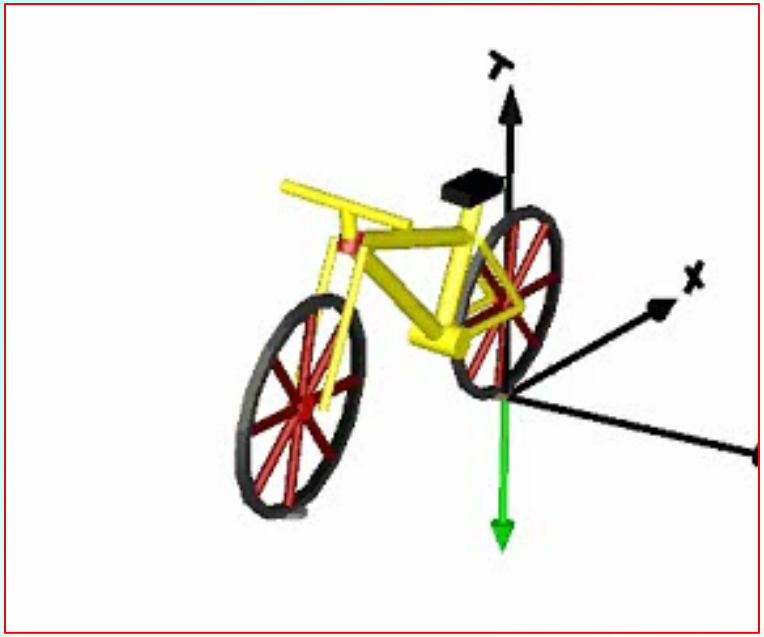




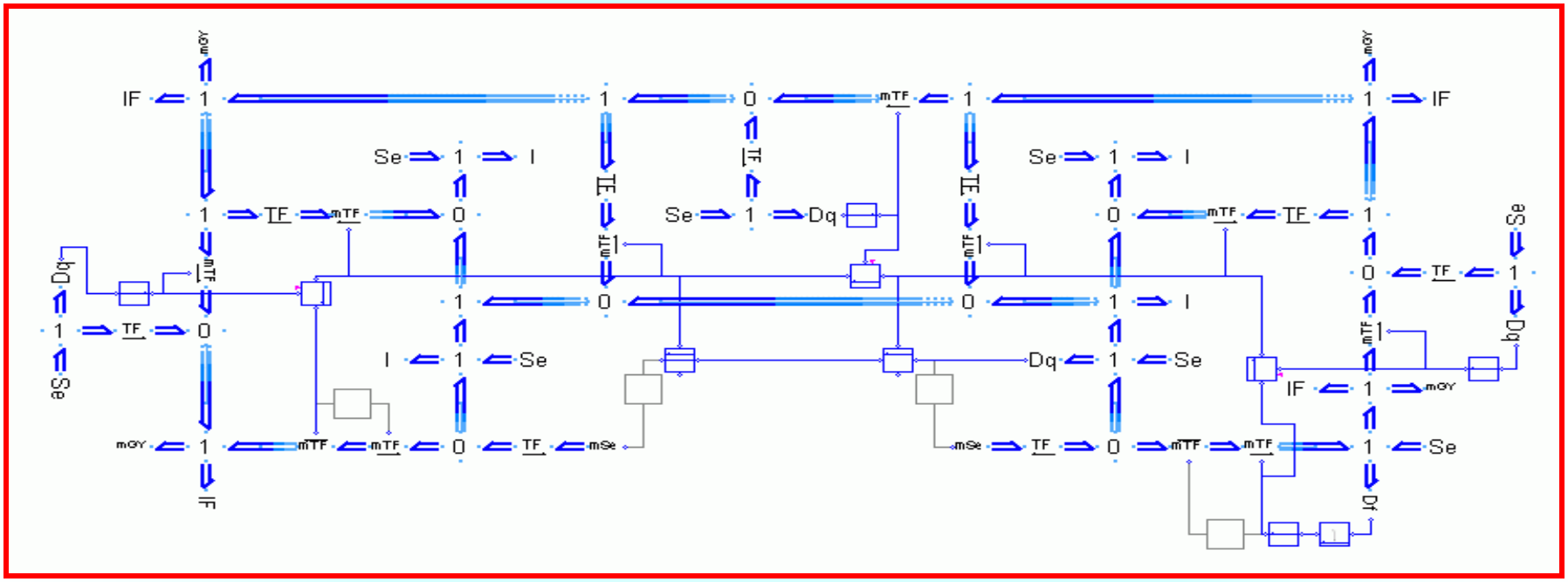


# A Bicycle Model



# A Bicycle Model II




- A multi-bond graph represents rarely the most suitable user interface. However, this is precisely the model that gets simulated. The multi-bond graph sits underneath the multi-body system description shown previously.

# Efficiency of Simulation Runs

- The following table compares the efficiency of the simulation code obtained using the multi-body library contained as part of the ***MSL*** with that obtained using the 3D mechanics sub-library of the ***MultiBondGraph*** library.

| | MSL | | | MultiBondGraph | | |
|---|---|---|---|---|---|---|
| experiment | linear equ. | non-lin. equ. | steps | linear equ. | non-lin. equ. | steps |
| Pendulum | 0 | 0 | 207 | 0 | 0 | 207 |
| Double pendulum | 2 | 0 | 549 | 2 | 0 | 549 |
| Crane crab. | 2 | 0 | 205 | 4 | 0 | 205 |
| Gyroscopic exp. with Cardans | 2,2 | 0 | 294 | 3,2 | 0 | 294 |
| Gyroscopic exp. with Quaternions | 4,3 | 4 | 24438 | 4,2 | 4 | 25574 |
| Planar Loop | 8,2 | 2 | 372 | 6,2,2 | 2 | 372 |
| Centrifugal exp. | 10,2,2 | 2,2 | 70 | 16,2,2 | 2,2 | 70 |
| Four bar loop* | 10,5,2 | 5 | 446 | 9,5,2 | 5 | 625 |
| Bicycle* | 15,5,3,2 | 1 | 97 | 15,3 | 1 | 84 |

# Special Bond Graph Libraries

- ***Wheels and Tires*** is a state-of-the-art free bond graph-based library offering complex models of wheels and tires. Tire models of different complexities can be assembled in an object-oriented fashion. The library is built on top of the *MultiBondLib* library.
- ***MotorVehicle*** is a free bond graph-based library offering models of components of motor vehicles including models of motor vehicle riders. It is built on top of the *MultiBondLib* and *Wheels and Tires* Modelica libraries.
- ***SpiceTestLib*** is a free bond graph-based library of electronic circuit models for testing the Spice implementation contained in the *BondLib* library. The Spice implementation inside *BondLib* is considerably more complete than the corresponding library offered as part of the MSL. It features both bipolar and Mosfet transistor models at many different levels of complexity.

# Summary

- ✓ ***Model-based design*** is the way of the future.
- ✓ The systems to be designed by engineers are getting ever more complex. The increasing system complexity can only be managed within reasonable time by ***virtualization of the design***.
- ✓ Your product will only sell if it is better than what the competition has to offer. Model-based design doesn't necessarily lead to *cheaper systems*, but it will invariably lead to ***better designed systems***.
- ✓ An object-oriented modeling and simulation environment, such as ***Modelica***, in combination with a ***rich library of well-designed base models*** is a powerful ally in product design virtualization.




# Summary II

- ✓ Typical models today contain a few dozen state variables and several hundred if not a few thousand (significant) algebraic variables.
- ✓ These (multi-energy domain) models are almost invariably stiff.
- ✓ Modelica implementations offer therefore a stiff system solver (usually DASSL) as their default simulation tool.
- ✓ Modelica compilers offer algorithms for symbolic index reduction (elimination of structural singularities) and for the treatment of algebraic loops including selection of small numbers of tearing (iteration) variables.
- ✓ They also offer sophisticated algorithms for robust discontinuity handling in complex models including methods for finding consistent sets of initial conditions after event handling (possibly involving algebraic loops in discrete state variables).
- ✓ They finally offer algorithms for dynamic state selection (for the elimination of dynamic singularities).




# Modeling and Simulation

# The Future

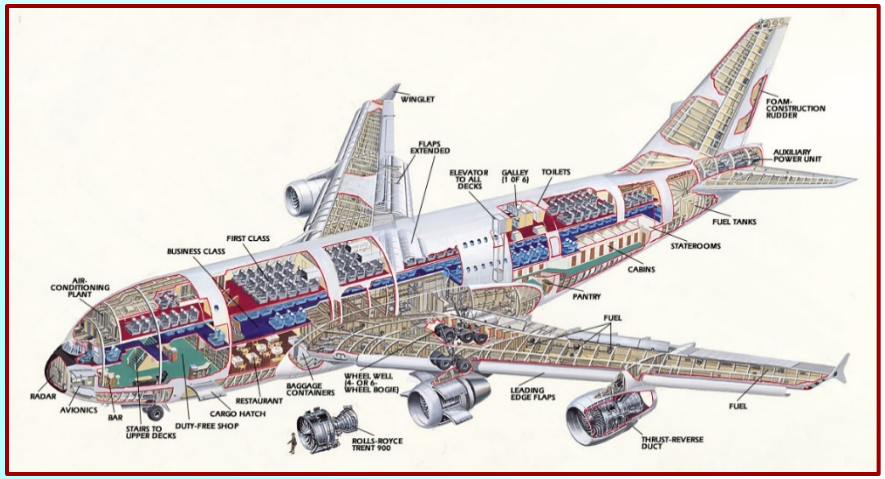






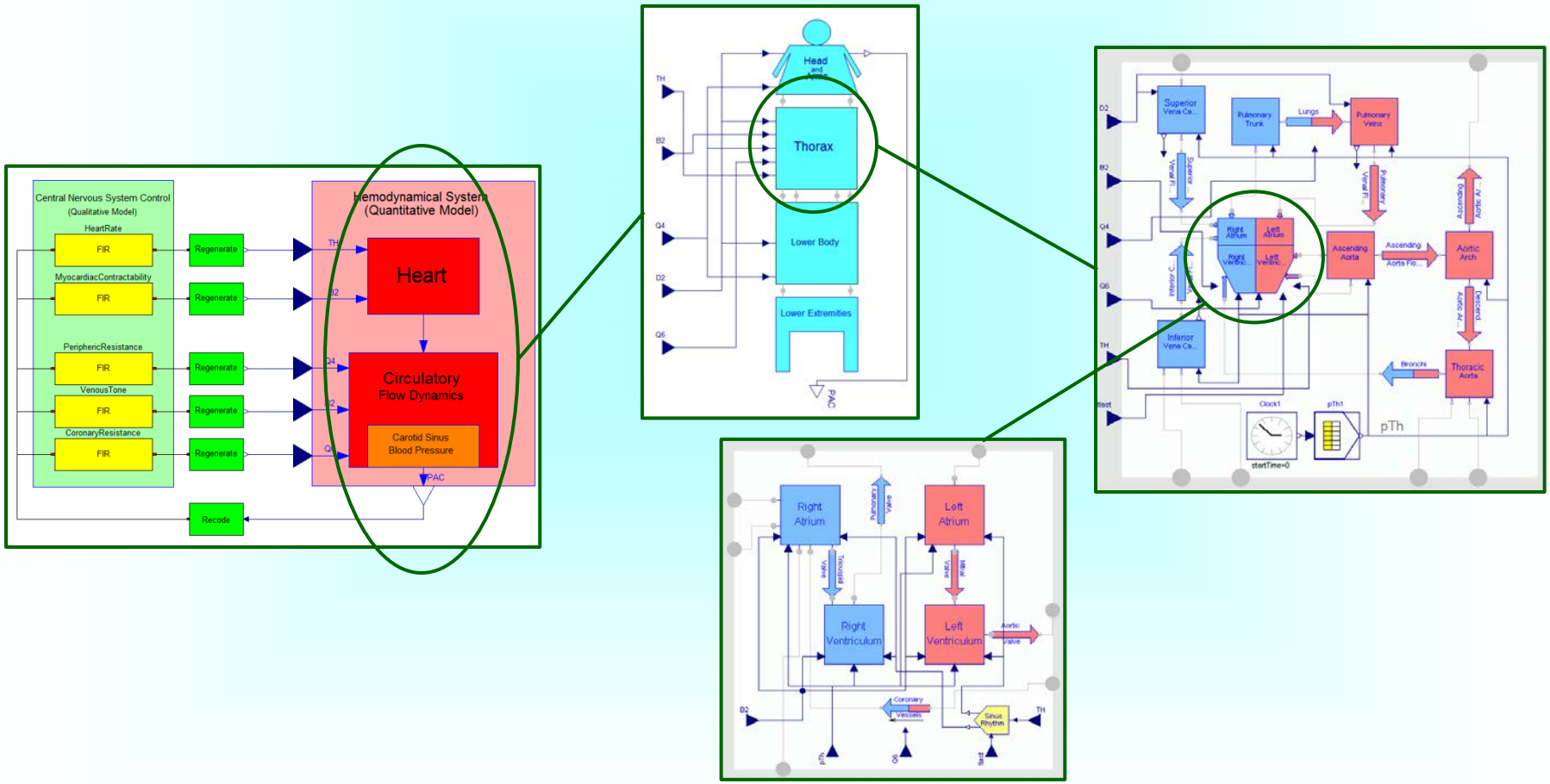
# Full System Design

- The first **Airbus A380** aircraft was delivered to the end customer with a delay of roughly one year.
- The system has become so complex that it is no longer possible to test such an aircraft in all of its potential modes of operation before delivering it to the customer.




- The electric cable tree alone is roughly 500 km long and has tens of thousands of connections.
- A full system model of the aircraft features thousands of state variables and tens of thousands of algebraic variables.
- This opens up an entirely new ballgame for modeling and simulation environments.

# Full System Design II



# Full System Design III

- In the ***cardiovascular system***, each major blood vessel needs to be represented as a container of blood, i.e., a compressible fluid with inertia. Hence each blood vessel contains a (bond graph) capacitor to represent the compressibility of the fluid and a (bond graph) inductor to represent the inert mass. Thus, each blood vessel exhibits second-order dynamics.
- There is already now demand for ***full-body simulations***, including not only the (relatively simple) cardiovascular system, but every single organ of the human body.
- A full-body human model will once again be characterized by thousands of state variables and tens of thousands of algebraic variables.

# Full System Design IV

We shall distinguish between three facets of system design support:

- ***Modeling support*** discusses features needed in a future system design software that are not currently supported in Modelica.
- ***Simulation support*** discusses aspects of simulators (differential and algebraic equation solvers) that are not currently supported in Modelica.
- ***System design support*** discusses facets of complex system design that go beyond mere modeling and simulation support and that will need to be offered by system design software environments of the future.

# Modeling Support

The object-oriented modeling paradigm scales generally well. Thus, formulating a model that is ten or even hundred times as big as current models is not problematic *per se*.

Sometimes it may be useful to model digital electronic circuits using analog components, i.e., detailed transistor models.

A digital circuit can easily contain many thousands of digital computing elements. Representing such a circuit using analog components is straightforward from a modeling point of view.

Such a model will not simulate in any of today's Modelica implementations, but this is a simulation problem and not a modeling problem. I shall explain in due course, why such a model cannot be simulated using today's simulation technology.




# Modeling Support: Variable Structure Systems

- A major shortcoming of Modelica even today is its inability to deal with ***variable structure systems***.
- Every mechanical system with a clutch needs to be represented by a variable structure model. Two axles that are connected by a clutch exhibit second-order dynamics while the clutch is engaged, but they show fourth-order dynamics when the clutch is disengaged. Thus, the number of differential equations changes dynamically during the simulation depending on a parameter value.
- A system exhibiting this property is called a variable structure system. Modelica does not currently support the simulation of variable structure models.





# Modeling Support: Variable Structure Systems II

Two tools that offer partial solutions to this problem are ***Mosilab*** and ***Sol***.

- ***Mosilab*** slightly generalizes the Modelica language. It makes use of ***dynamic model switching*** (similar to Modelica's approach to dynamic state selection) to deal with variable structure models. Unfortunately, the approach doesn't scale well as 10 dynamic structure switches call for 1024 different models to be maintained with potentially 523,776 possible transitions between them (!)
- ***Sol***, developed by Dirk Zimmer, a former Ph.D. student of mine, in his Ph.D. dissertation, proposes ***dynamic causalization*** instead. When a structure change occurs, the model is reconfigured on the fly by incremental compilation. However, Sol is only available as a prototype until now and doesn't implement many of the other algorithms that are offered by full-fledged Modelica compilers.




# Modeling Support: Model Activation/Deactivation

Support of variable structure models will become more important in future system design environments.

- In a complex system, it must be possible to activate models as new subsystems enter the system and deactivate them when they leave the system. For example, in a traffic simulation of a busy intersection, it may be desirable to represent individual cars by physical models. When a car enters the intersection to be simulated, its model needs to be added to the overall model. When a car leaves the intersection, its model needs to be removed.
- Similarly, it should be possible to replace a complex model of a component by a simpler one when not much activity is occurring in that subsystem.
- All of the above features require support of variable structure systems modeling.




# Modeling Support: Model Pruning

- A related feature is model pruning. It should be possible to simulate a system with different levels of granularity, i.e., different degrees of model resolution.
- To this end, it must be possible to simplify models in an automated fashion. For example in a DC motor, it may be desirable to exclude the armature inductance. If this is done dynamically in a current Modelica implementation (by setting L=0 at event time), the simulation will die with a division by zero.
- Model pruning once again leads to variable structure models, but the demands on the modeling support software are more severe in this case, because the software should be capable of recognizing on its own, where and when simplifications are feasible and be able to enact those without human intervention.




# Simulation Support

While the object-oriented modeling paradigm scales well, the same unfortunately does not hold true for the current class of stiff system solvers.

- All stiff system solvers are implicit solvers. It can be shown that explicit solvers can never be stiffly stable. They all force the solver to use step sizes that are small in comparison to the smallest time constant in the system. Otherwise the numerical stability of the algorithm will be lost.
- Traditional implicit solvers require that the (usually non-linear) set of model equations be solved iteratively during each step. To this end, a Newton iteration is set up that spans over all model equations.
- During each iteration step, a large linear set of equations needs to be solved.
- **The computational effort therefore grows *cubically* with the number of state variables in the model.**




# Simulation Support II

- If a simulation with 50 state variables executes in a few seconds, a simulation with 500 state variables will take a few hours to simulate.
- Some Modelica implementations, such as Dymola, meanwhile make use of ***sparse linear system solvers*** to mitigate the scaling problem.
- While sparse linear system solvers make the simulations faster, they still don't solve the problem. Large models still simulate too slowly.
- In order to overcome these problems, we need to get away from centralized schemes for simulating large systems.
- ***Quantized state system (QSS) solvers*** offer a way out in may cases.




# Simulation Support: QSS Solvers

- Traditional ODE and DAE solvers make use of ***time slicing***. Given the current and past state and state derivative values, the value of the state one time step into the future, i.e., at time $t_{k+1}$, is being estimated. At that time, the new state can assume any real-valued number.
- QSS solvers operate differently. Rather than discretizing the time axis, they discretize the state axis, i.e., they operate on ***quantized states***. They evaluate the next state value at the first time instant in the future at which the state variable differs from its current value by one quantum level, i.e., when $x_{k+1} = x_k \pm \Delta x$.
- QSS solvers are naturally asynchronous. Each state variable carries its own simulation clock.
- As QSS solvers offer ***dense output***, each solver knows the current values of all other neighboring states whenever it undergoes its own internal transition.



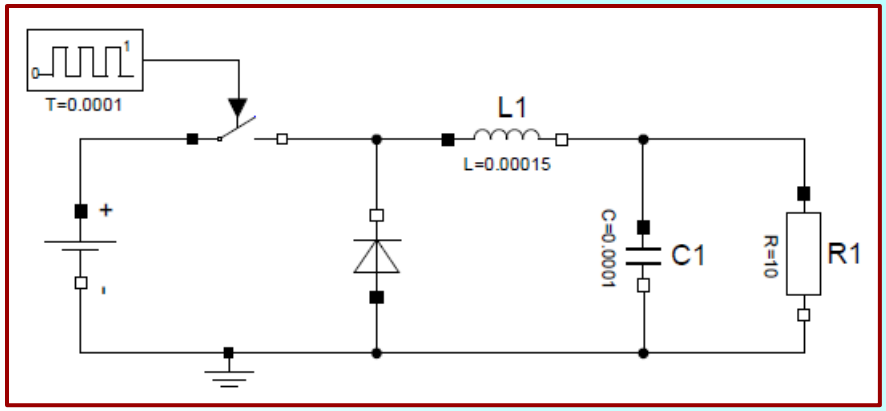

# Simulation Support: QSS Solvers II

- Convergence and stability properties for QSS solvers have meanwhile been established.
- QSS solvers even offer an advantage in this respect. Whereas traditional solvers only estimate the ***local integration error*** within one step, QSS solvers offer an upper bound on the ***global simulation error*** across the entire simulation at least for linear systems.
- When simulating a small-scale non-stiff model without discontinuities, QSS solvers are slightly less efficient than explicit Runge-Kutta solvers (there is a slightly bigger overhead). These models don't offer anything that the QSS solvers can exploit. However, such models are not of much interest today, because any solver can deal with them efficiently, and the QSS solvers will not fare much worth than any of the traditional solvers.

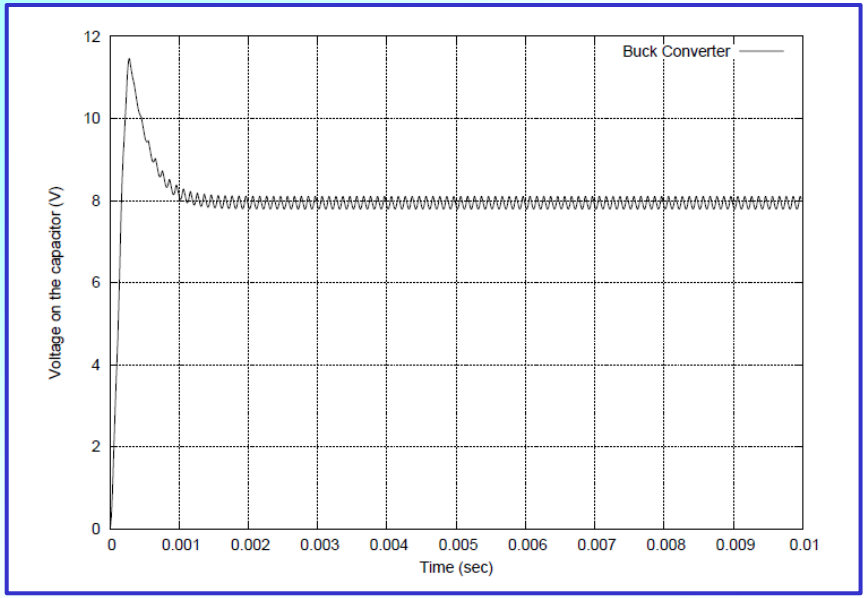



# Simulation Support: QSS Solvers III

QSS solvers are more efficient than traditional solvers when facing ***discontinuities***, as there is no need for iterating on (state) event times.




***Buck converter circuit***



# Simulation Support: QSS Solvers IV

| | | | 500 output points | | | 10000 output points | | |
|---|---|---|---|---|---|---|---|---|
| | | | CPU time (msec) | Steps | Simulation Error | CPU time (msec) | Steps | Simulation Error |
| QSS | LIQSS3 | $10^{-2}$ | 4 | 3351 | 5.84E-03 | 4 | 3351 | 5.83E-03 |
| | LIQSS3 | $10^{-3}$ | 8 | 4163 | 7.31E-04 | 8 | 4163 | 7.32E-04 |
| | LIQSS3 | $10^{-4}$ | 12 | 6804 | 4.60E-05 | 12 | 6804 | 4.61E-05 |
| | LIQSS3 | $10^{-5}$ | 20 | 11314 | 1.07E-06 | 20 | 11314 | 1.08E-06 |
| | LIQSS2 | $10^{-2}$ | 4 | 3863 | 7.83E-03 | 4 | 3863 | 7.84E-03 |
| | LIQSS2 | $10^{-3}$ | 8 | 6715 | 1.32E-03 | 8 | 6715 | 1.32E-03 |
| | LIQSS2 | $10^{-4}$ | 12 | 18519 | 1.15E-04 | 12 | 18519 | 1.15E-04 |
| | LIQSS2 | $10^{-5}$ | 32 | 53391 | 6.42E-06 | 32 | 53391 | 6.42E-06 |
| OpenModelica | DASSL | $10^{-3}$ | 22 | 4273 | 3.56E-03 | 70 | 5249 | 2.66E-04 |
| | DASSL | $10^{-4}$ | 28 | 5636 | 3.17E-03 | 72 | 5955 | 1.75E-04 |
| | DASSL | $10^{-5}$ | 32 | 7781 | 3.28E-03 | 74 | 7623 | 2.40E-05 |

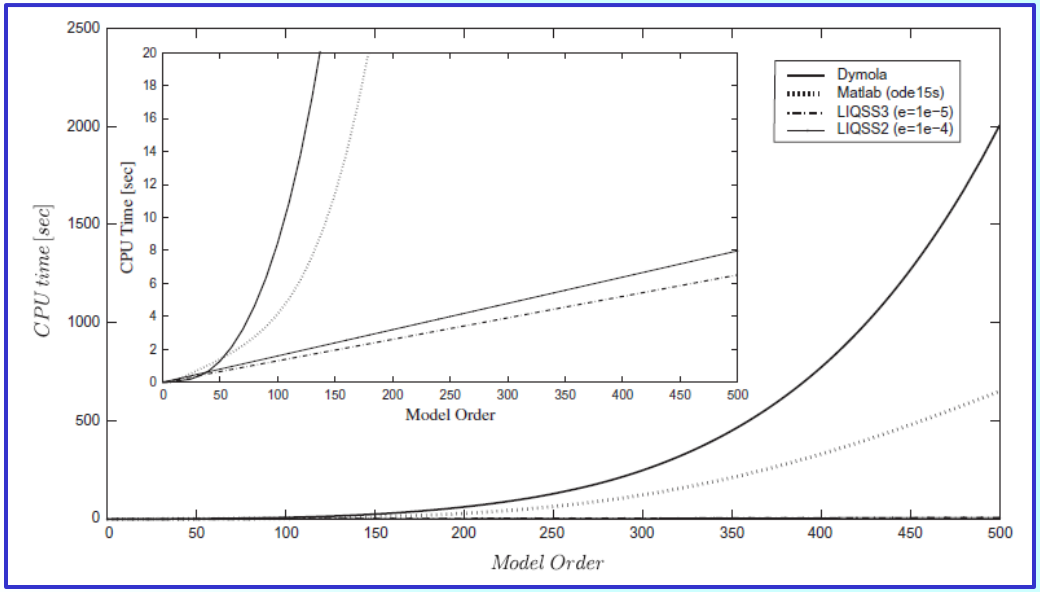
QSS solvers are more efficient than DASSL on this problem. The heavier and more frequent the discontinuities occur, the better will be the relative performance of QSS. On one example, QSS was about 20 times faster than DASSL.

# Simulation Support: QSS Solvers V

- A considerably more important advantage can be obtained when dealing with ***large-scale stiff systems***.
- A set of linearly implicit stiff quantized state system (LIQSS) solvers has been developed. Although these are implicit solvers, ***they do not call for Newton iteration***. The reason is that in each step, there are only two possible outcomes: the state either increases or decreases by one quantum level.
- To demonstrate the advantage, an inverter chain (a simplified model of a transmission line) with a fast load at the end was simulated. Each logical inverter is represented by a second-order system. Thus 500 inverters lead to 1000 differential equations. As the load is fast, the overall system is stiff, and consequently, Dymola has no choice but to employ a stiff system solver. Of the different stiff system solvers available in Dymola, ***Esdirk23a*** turned out most efficient for this simulation. Hence this is what we compared ***LIQSS*** against.




# Simulation Support: QSS Solvers VI




- The ***cubic growth of the computational load*** as a function of the system size for classical (BDF-type) stiff system solvers is clearly visible.
- In contrast, the computational load ***grows only linearly*** in the number of differential equations for LIQSS solvers.

# Simulation Support: QSS Solvers VII

| Integration method | Error tolerance | No of steps/events | Scalar $f_i$ eval. | Error | CPU time (s) |
|---|---|---|---|---|---|
| PowerDEVS | $\Delta Q_{rel} = 10^{-2}$ | 178,595 eV. | 714,380 | 0.61 | 2.72 |
| LIQSS2 | $\Delta Q_{rel} = 10^{-3}$ | 259,591 eV. | 1,038,364 | 0.022 | 3.81 |
| ($\Delta Q_{min} = \Delta Q_{rel}$) | $\Delta Q_{rel} = 10^{-4}$ | 533,200 eV. | 2,132,800 | 0.00023 | 8.04 |
| PowerDEVS | $\Delta Q_{rel} = 10^{-3}$ | 165,931 eV. | 995,586 | 0.443 | 2.99 |
| LIQSS3 | $\Delta Q_{rel} = 10^{-4}$ | 286,806 eV. | 1,720,836 | 0.119 | 4.95 |
| ($\Delta Q_{min} = \Delta Q_{rel}$) | $\Delta Q_{rel} = 10^{-5}$ | 368,857 eV. | 2,213,142 | 0.0133 | 6.49 |
| | $\Delta Q_{rel} = 10^{-2}$ | 81,377 eV. | 488,262 | 1.343 | 1.51 |
| | $\Delta Q_{rel} = 10^{-6}$ | 626,693 eV. | 3,760,158 | 0.00205 | 11.1 |
| Dymola | $e_{rel} = 10^{-1}$ | 7875 steps | >37,748,000 | 0.062 | 1786.59 |
| esdirk23a | $e_{rel} = 10^{-2}$ | 8664 steps | >43,937,000 | 0.046 | 2005.64 |
| | $e_{rel} = 10^{-3}$ | 10,005 steps | >51,935,000 | 0.037 | 2147.1 |
| Matlab Ode15s | $e_{rel} = 10^{-2}$ | 13,220 steps | >33,050,000 | 0.041 | 651.37 |
| Multirate II | $e_{rel} = 10^{-4}$ | – | 4,795,878 | – | 6.36* |
| Multirate II | $e_{rel} = 10^{-5}$ | – | 17,358,472 | – | 21.65* |

- LIQSS simulated this problem almost 1000 times faster than Dymola (using Esdirk23a), and 300 times faster than Matlab (using Ode15s).
- LIQSS was even almost twice as fast as a multi-rate solver that had been specifically designed for this particular problem (the problem specification was taken from a paper introducing the multi-rate algorithm).







# Simulation Support: Event Density

- Let us assume that, in a particular model, there occur $x$ discontinuities per unit of simulation time.
- Discontinuities can be assumed to occur statistically independent of each other.
- Thus, if the system is 10 times as big, we can expect roughly 10 times as many discontinuities per time unit.
- Therefore, the ***event density grows linearly with the size of the system*** to be simulated.
- In a classical (stiff or non-stiff) solver, the ***execution time grows quadratically*** in the number of events.
- In contrast, the execution time ***grows only linearly*** in the number of events when using QSS solvers.
- Once again, QSS solvers turn out to offer an advantage over classical solvers in this respect.




# Simulation Support: Event Density II

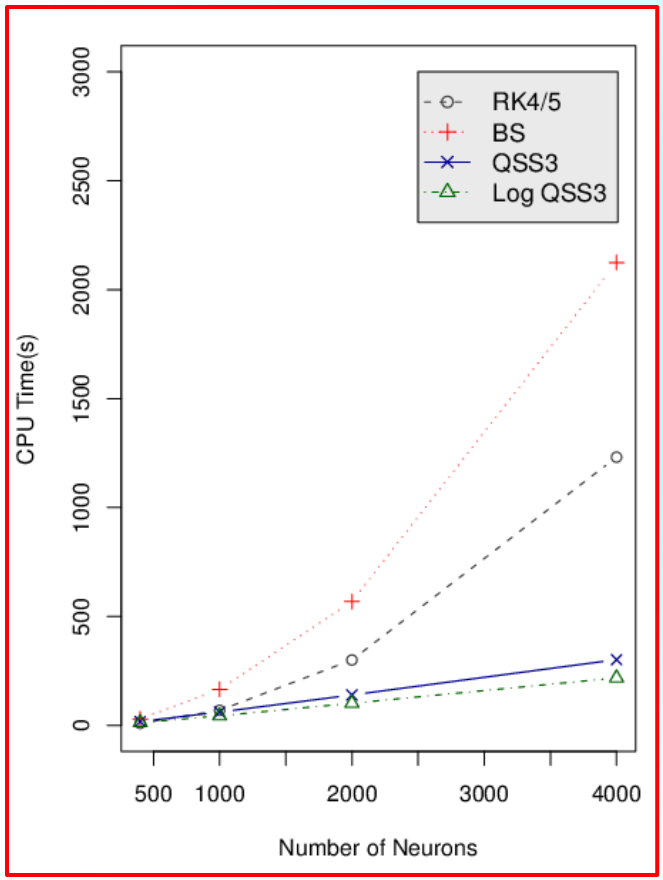
We simulated a ***spiking neural network*** with a varying number of neurons.

- Each neuron is represented by a set of differential equations with discontinuities.
- The larger the number of neurons, the larger the event density, i.e., the more discontinuities will occur per unit of simulation time.
- The model is non-stiff. Therefore we compared two versions of a non-stiff third-order accurate QSS solver (QSS3 with constant and logarithmic quantization) with two non-stiff (explicit) classical solvers: a Runge-Kutta-Fehlberg (RK4/5) and a Bulirsch-Stoer (BS) solver.





# Simulation Support: Event Density III




- The execution time ***grows linearly*** with the number of neurons when using non-stiff QSS solvers.
- It ***grows quadratically*** with the number of neurons when using classical non-stiff solvers such as a Runge-Kutta-Fehlberg (RK4/5) or a Bulirsch-Stoer (BS) solver.
- For small numbers of neurons the execution speed is approximately the same in all cases. RK4/5 and BS execute roughly 20% faster than QSS3 when simulating a single neuron, but this is irrelevant.

# Simulation Support: Parallelization

- Using a QSS solver, each state variable carries its own simulation clock.
- The simulations of different states execute in a ***naturally asynchronous fashion***.
- For this reason, QSS solvers lend themselves much more easily to parallelization (implementation on a ***multi-core architecture***) than classical solvers.
- This applies to both the stiff and the non-stiff versions, but the advantage becomes more pronounced in the stiff case.
- The parallelization scales very well, even with large numbers of cores. Classical schemes saturate much more quickly when parallelized.




# Simulation Support: Parallelization II

We studied the control of electric power consumption of a number of air conditioner units controlling the temperature values in different rooms of a big building.

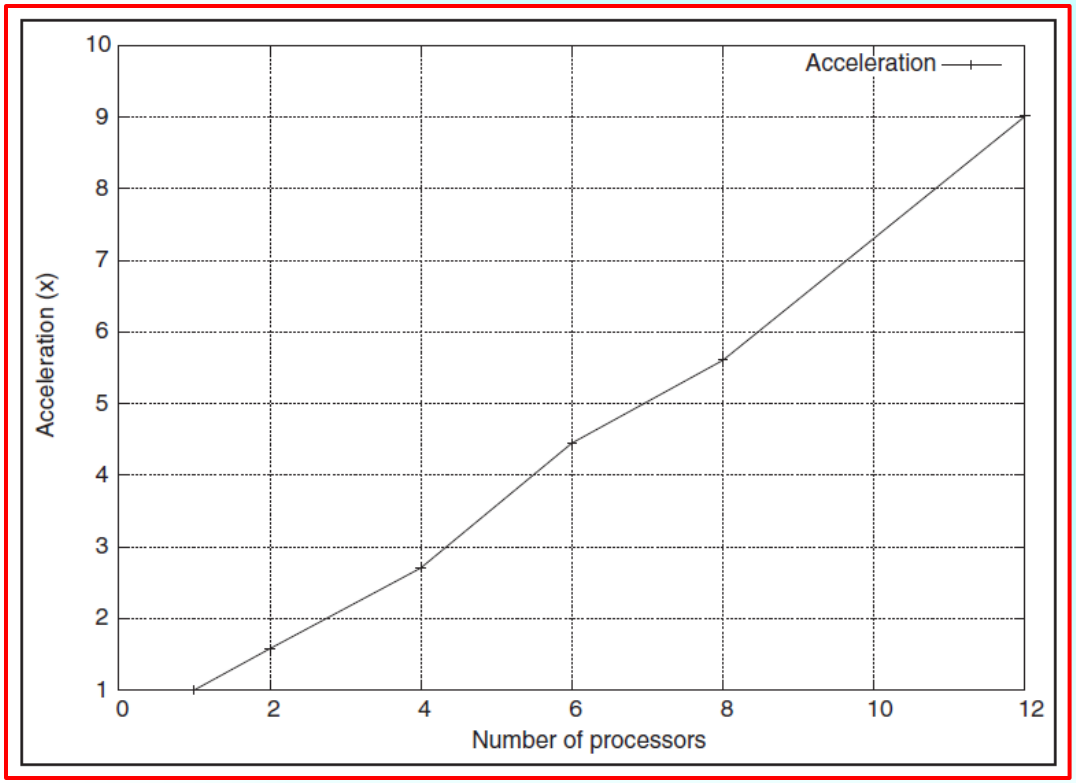
- We simulated the system using QSS solvers for all state variables in the system.
- The solvers were distributed over a multi-core architecture.
- Each solver synchronizes its own simulation clock with the wall clock, but no attempt was made to synchronize the local simulation clocks of the individual solvers between each other.
- We studied the performance of the simulation while varying the number of cores involved in the simulation.







# Simulation Support: Parallelization III




- The execution speed grows even a bit faster than linearly for up to 12 parallel processes (6 cores interleaved) tested.
- The faster-than-linear speedup is caused by reduced overhead associated with the shorter event queue.
- No saturation is noticeable.
- Classical schemes tend to already begin to saturate when more than 4 parallel processes are involved.

# System Design Support

- System design support entails much more than just modeling and simulation support. Series of simulation experiments need to be designed around a model.
- Most Modelica environments offer some support for experimental design in the form of a ***scripting language***.
- Unfortunately, the scripting language has not (yet) been standardized by the ***Modelica Consortium***, and it may be difficult to do so due to diverging interests among the Consortium members.
- For this reason, scripting support varies a lot from one environment to another.
- With respect to modeling and simulation support, ***Dymola*** is the strongest competitor on the market by a good margin. With respect to scripting support it is almost the weakest. Its scripting language is poor by design and even more poorly documented.




# System Design Support II

- Scripting support is important and will gain importance as we proceed to modeling and simulating ever larger systems.
- We will need a stable scripting platform before system design efforts can begin in earnest.
- For this reason, it is important that the ***Modelica Consortium*** tackles this problem, even if the negotiations should turn out to be difficult.
- Once a stable scripting platform is available, many players will be available to develop system design tools on the shoulders of that platform.
- This is comparable to the efforts that went into the development of the Modelica Standard Library (MSL). These efforts could not begin until the Modelica language had been standardized. In the meantime, hundreds if not thousands of man years of design and development work have gone into making the MSL what it has become.




# Conclusions

- ***Size matters!***
- ***Small-scale systems*** (containing a few state variables) can be modeled and simulated using any tool/algorithm available.
- ***Medium-scale systems*** (characterized by a few dozen state variables) require heavy modeling support for organizing the models. On the simulation side, they require a robust stiff system solver, such as DASSL.
- ***Large-scale models*** (defined by several hundreds or thousands of state variables) require decentralized simulation tools, such as QSS solvers.
- Support for ***variable structure system modeling*** as well as ***decent scripting support*** would already have been useful at the level of medium-scale system modeling and simulation, but are still lacking. At the level of large-scale system modeling, these will become even more important.