Universität Rostock
Traditio et Innovatio

The Role of Domain Specific Languages For Spatial, Multi-Level Modeling and Simulation

Adelinde Uhrmacher
University of Rostock
Institute of Computer Science

Domain specific languages

- A domain-specific language (DSL) is a programming language that is tailored specifically for an application domain.
- A DSL "offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."
  - *internal* DSLs – pro: minimal implementation effort, easily extendable, cons: similarity with the host language
  - *external* DSLs – pro: complete freedom of syntax, cons: own interpreter.

van Deursen A. et al. (2000): Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6): 26-36

Modeling and Simulation

- “A model for a system S and an Experiment E is anything to which E can be applied to answer questions about S”
- “A simulation is an experiment performed with a formal model and executed on a computer”

Our approach: tailored domain-specific languages for modeling AND executing the experiments with these models, i.e., simulation.

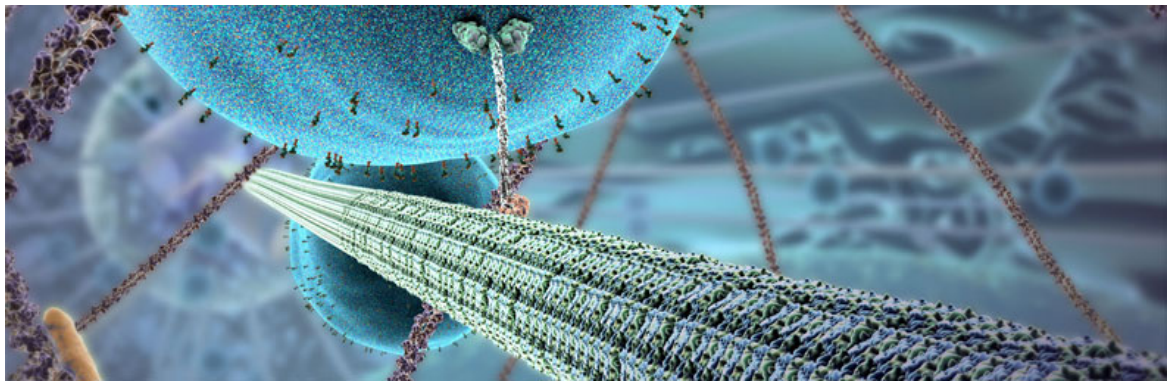
Cellier F. (1991): Continuous Systems Modeling, Springer.

DSLs for modeling - As with any language – first of all what would we like to describe?

Spatial dynamics of cells

only three things matter: location, location, location (Science, Vol 326, no. 5957, 2009).

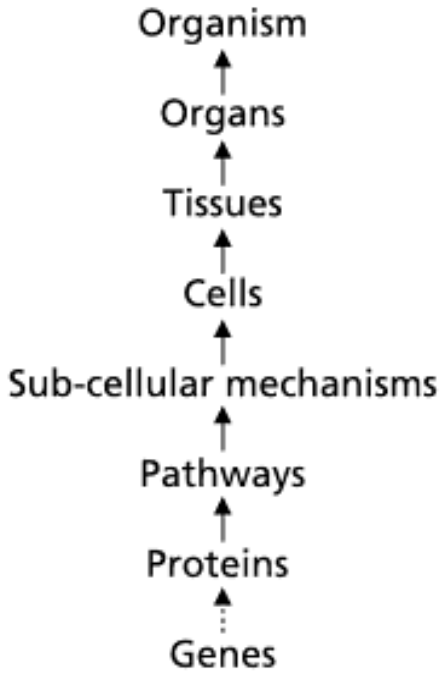
- no homogeneous distribution in the cell
- correlation of localization and function, e.g. at the membranes, in the nucleus
- excluded volume effects, e.g. molecular crowding
- diffusion, and active transport

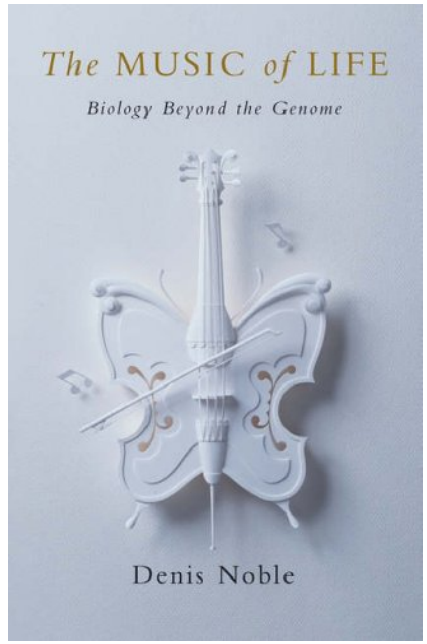


http://multimedia.mcb.harvard.edu/media.html

Multiple leves of organization

Reductionist thinking

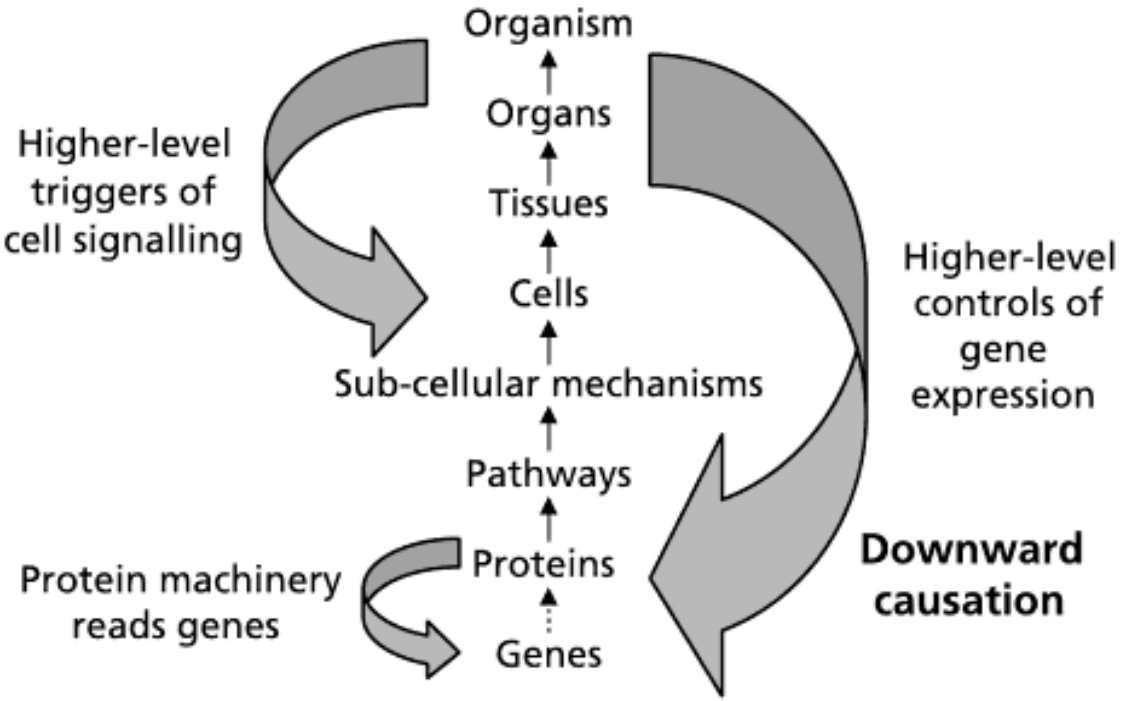


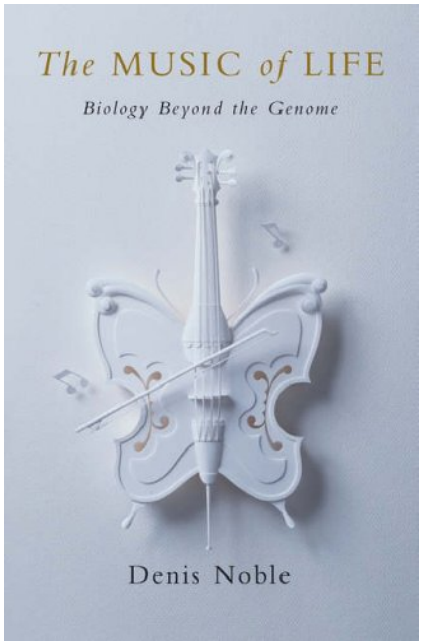


Denis Noble (2006). The Music of Life. Oxford University Press.

Multiple leves of organization

Complex systems involve upward AND downward causation



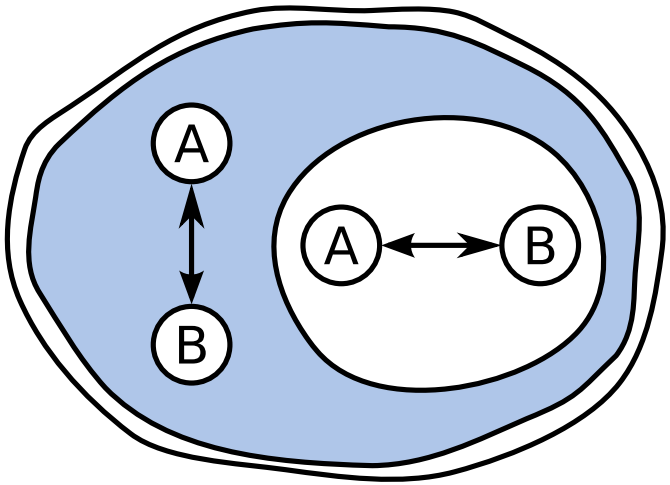


Denis Noble (2006). The Music of Life. Oxford University Press.

How would we like to describe these spatial, multi-level systems?

Modeling with classical approaches (e.g. ODEs)

- Structure (different levels) only implicitly
- Leads to many similar model parts (redundancy)
  => high model complexity



$$\frac{d[A_{cyt}]}{dt} = k_r[B_{cyt}] - k_f[A_{cyt}]$$

$$\frac{d[A_{nuc}]}{dt} = k_r[B_{nuc}] - k_f[A_{nuc}]$$

$$\frac{d[B_{cyt}]}{dt} = k_f[A_{cyt}] - k_r[B_{cyt}]$$

$$\frac{d[B_{nuc}]}{dt} = k_f[A_{nuc}] - k_r[B_{nuc}]$$

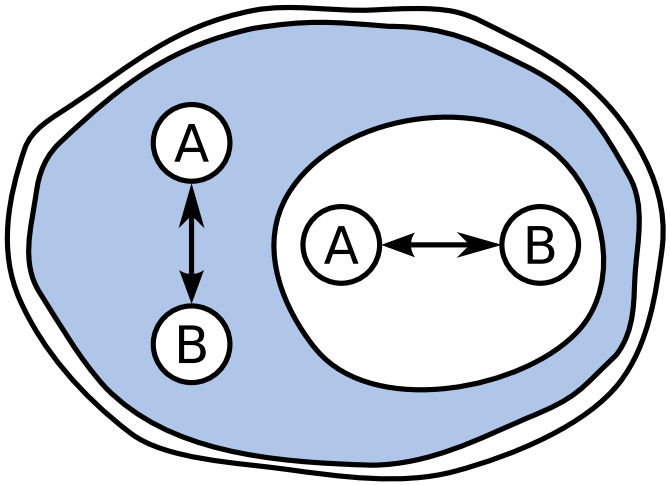
ML-Rules - An external DSL for Modeling

A rule-based language for multi-level modeling and simulation in cell biology[1]

- multi-level modeling
- species with attributes, and constraining reactions based on these attributes (allows e.g. to mimick the next subvolume method)
- dynamic nesting (variable structure models)
- stochastic semantics

[1] Carsten Maus et al. (2011): Rule-based multi-level modeling of cell biological systems. BMC Systems Biology 5: 166

Multi-compartment model revisited



$$\frac{d[A_{cyt}]}{dt} = k_r[B_{cyt}] - k_f[A_{cyt}]$$

$$\frac{d[A_{nuc}]}{dt} = k_r[B_{nuc}] - k_f[A_{nuc}]$$

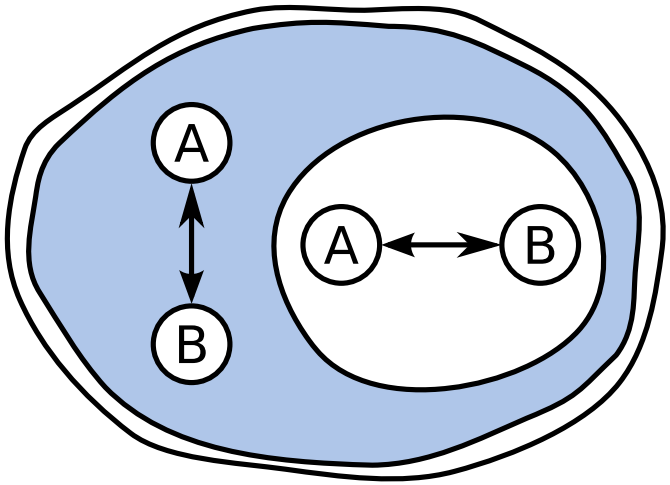
$$\frac{d[B_{cyt}]}{dt} = k_f[A_{cyt}] - k_r[B_{cyt}]$$

$$\frac{d[B_{nuc}]}{dt} = k_f[A_{nuc}] - k_r[B_{nuc}]$$

ML-Rules: reducing complexity by applying rules to different solutions

$$A^a \xrightarrow{a \cdot k_f} B$$
$$B^b \xrightarrow{b \cdot k_r} A$$

Multi-compartment model revisited



$$\frac{d[A_{cyt}]}{dt} = k_r[B_{cyt}] - k_f[A_{cyt}]$$

$$\frac{d[A_{nuc}]}{dt} = k_r[B_{nuc}] - k_f[A_{nuc}]$$

$$\frac{d[B_{cyt}]}{dt} = k_f[A_{cyt}] - k_r[B_{cyt}]$$

$$\frac{d[B_{nuc}]}{dt} = k_f[A_{nuc}] - k_r[B_{nuc}]$$

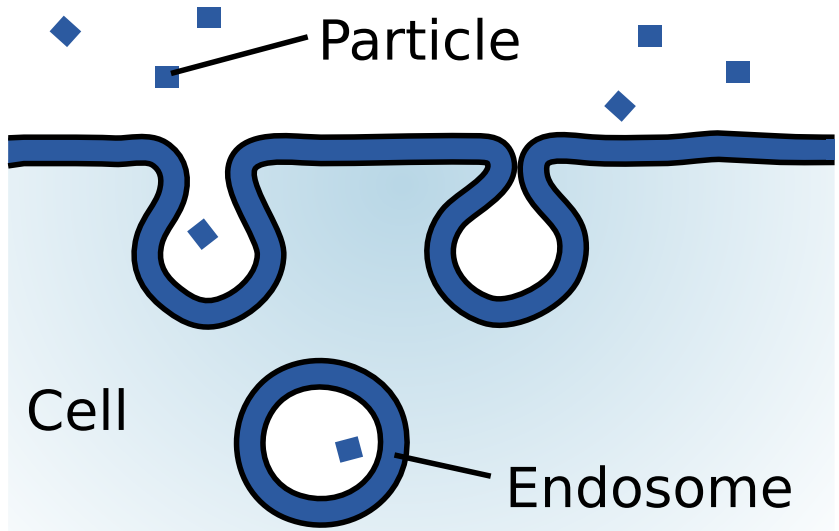
ML-Rules: reducing complexity by applying rules to different solutions

$$A^a \xrightarrow{a \cdot k_f} B$$
$$B^b \xrightarrow{b \cdot k_r} A$$

Dynamic manipulation of model hierarchies

Example: endo- and exocytosis

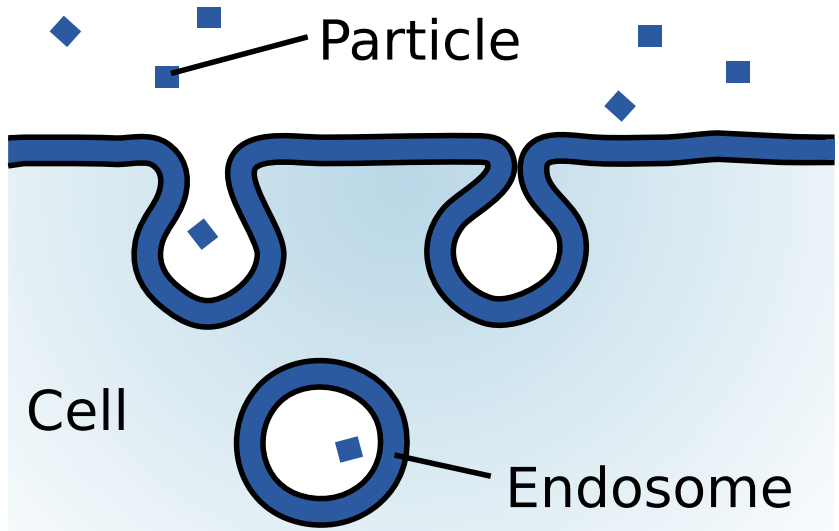
$$Cell[\,] + Particle \longleftrightarrow Cell[Endosome[Particle]]$$



Dynamic manipulation of model hierarchies

Example: endo- and exocytosis

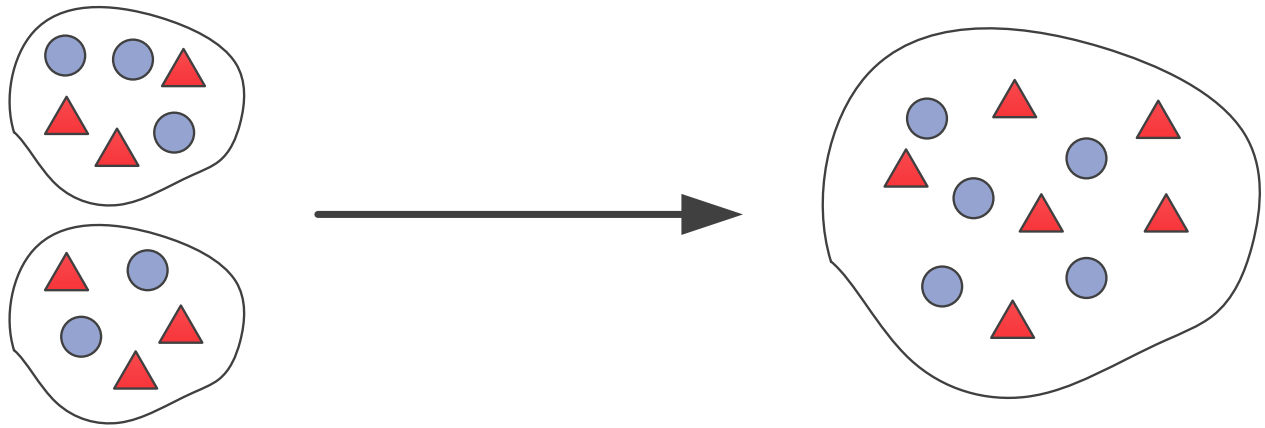
$$Cell[solution?] + Particle \longleftrightarrow Cell[Endosome[Particle] + solution?]$$



Dynamic manipulation of model hierarchies

Example: mitochondrion fusion

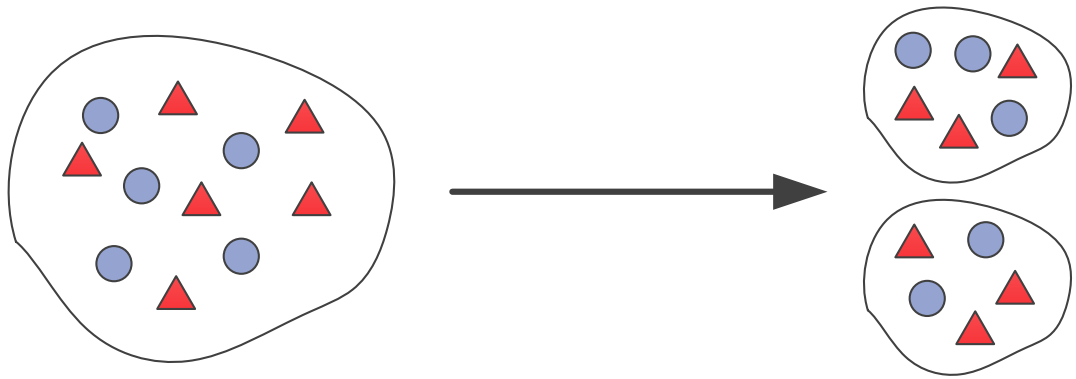
$$Mitochondrion[s1?] + Mitochondrion[s2?] \rightarrow Mitochondrion[s1? + s2?]$$



Dynamic manipulation of model hierarchies

Example: mitochondrion fission

$$Mitochondrion[s?] \rightarrow Mitochondrion[s1?] + Mitochondrion[s2?]$$
$$\text{where } (s1?, s2?) = split(s?, 0.5)$$



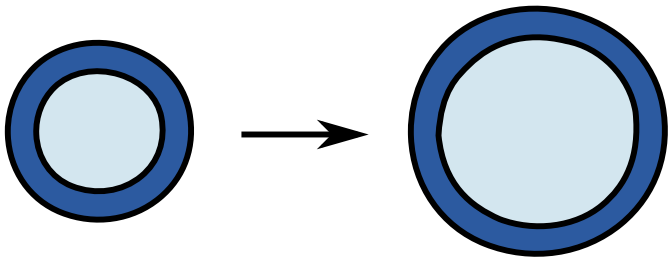
Species attributes: own state at each level

Attributes allow to equip each level with own states and dynamics that are constrained by their attributes.

E.g., the size of a cell may be described by an attribute of the *Cell* species.

Example: cell growth

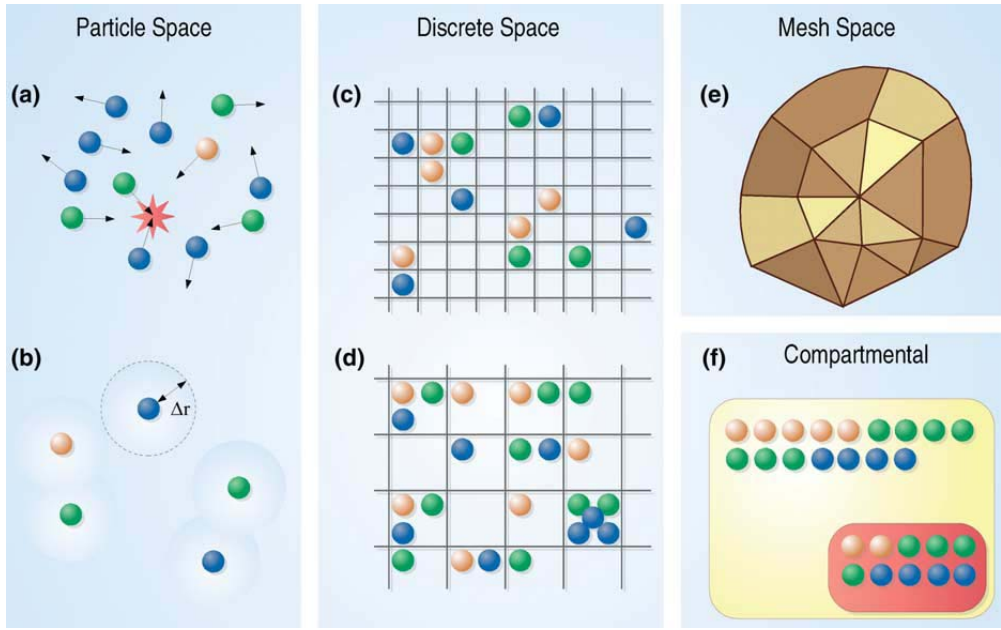
$$Cell(volume)[sol?] \rightarrow Cell(volume + \Delta V)[sol?]$$



Reaction Diffusion in the dictyostelium

```
66 // intra-cellular dynamics - computed continuously
67 CAR1:c -> ACA + CAR1 @ k1*#c;
68 ACA:a + PKA:p -> PKA @ k2*#a*#p;
69 CAMPi:a -> PKA + CAMPi @ k3*#a;
70 PKA:p -> @ k4*#p;
71 CAR1:c -> ERK2 + CAR1 @ k5*#c;
72 PKA:p + ERK2:e -> PKA @ k6*#p*#e;
73 CELL(x,y)[s?]:c -> CELL(x,y)[RegA + s?] @ k7*#c;
74 ERK2:e + RegA:r -> ERK2 @ k8*#e*#r;
75 ACA:a -> CAMPi + ACA @ k9*#a;
76 RegA:r + CAMPi:a -> RegA @ k10*#r*#a;
77 CELL(x,y)[ACA:a + s?] -> CAMPe(x,y) + CELL(x,y)[ACA + s?] @ k11*#a;
78 CAMPe(x,y):a -> @ k12*#a;
79 $$ROOT$$[CAMPe(x,y):a + CELL(x,y)[c?] + r?] -> $$ROOT$$[CELL(x,y)[CAR1 + c?] + CAMPe(x,y) + r?] @ k13*#a/(1 + countTw
80 CAR1:c -> @ k14*#c;
81
82 // movement of cell to adjacent position depending on external cAMP amount - computed stochastically
83 CELL(x1,y1)[s?] + CAMPe(x1,y1):a1 + CAMPe(x2,y2):a2 -> CELL(x2,y2)[s?] + CAMPe(x1,y1) + CAMPe(x2,y2)
84     @ if ((#a2>#a1) && ((x1!=x2) && (y1!=y2)) && ((x1-x2<=1)&&(x1-x2>=-1)) && ((y1-y2<=1)&&(y1-y2>=-1))) then kd_dic
85
86 // cAMP diffusion - computed continuously
87 CAMPe(x,y):a -> CAMPe(x,y+1)   @ if (y<ymax) then kd_camp*#a else 0;
88 CAMPe(x,y):a -> CAMPe(x+1,y+1) @ if (x<xmax) && (y<ymax) then kd_camp*#a else 0;
89 CAMPe(x,y):a -> CAMPe(x+1,y)   @ if (x<xmax) then kd_camp*#a else 0;
90 CAMPe(x,y):a -> CAMPe(x+1,y-1) @ if (x<xmax) && (y>1) then kd_camp*#a else 0;
91 CAMPe(x,y):a -> CAMPe(x,y-1)   @ if (y>1) then kd_camp*#a else 0;
92 CAMPe(x,y):a -> CAMPe(x-1,y-1) @ if (x>1) && (y>1) then kd_camp*#a else 0;
93 CAMPe(x,y):a -> CAMPe(x-1,y)   @ if (x>1) then kd_camp*#a else 0;
94 CAMPe(x,y):a -> CAMPe(x-1,y+1) @ if (x>1) && (y<ymax) then kd_camp*#a else 0;
```

So far compartment and reaction-diffusion dynamics



What about excluded volumes, mobility etc. $\rightsquigarrow$ particles in continuous space?

Takahashi K., Arjunan S., Tomita M. (2005): Space in systems biology of signaling pathways - towards intracellular molecular crowding in silico, FEBS Letters

The external DSL ML-Space

Populations in discretized space…

- only entities in same spatial unit (e.g. subvolume) can react with each other
- all diffusions follow the same rule pattern (only diffusion constant may differ)

  $\Rightarrow$ no change in reaction rule definition style needed,
  separate definition of diffusion constants and initial spatial distributions

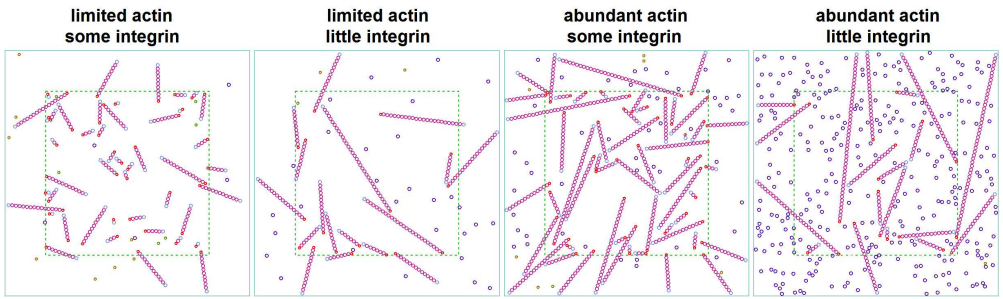
Individual particles in continuous space…

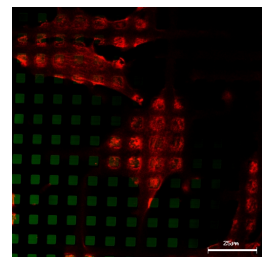
- 2nd order reactions are triggered by collisions (higher order not useful)
- spatial extensions (shape, volume) of entities needed
- movement can be implicit, too (Brownian motion given diffusion constant), explicit rules needed for movement across boundaries

ML-Space: Actin

```
Integrin() + SurfStruct() -> SurfStruct()[Integrin(focal:yes)] @ 1

Integrin(focal:yes)<bs:FREE> + Actin()<pointed:FREE> ->
     Integrin(diffusion:0)<bs:new>.Actin(diffusion:0)<pointed:new>
Actin()<pointed:OCC,barbed:FREE> + Actin()<pointed:FREE> ->
     Actin()<pointed:OCC,barbed:new>.Actin(diffusion:0)<pointed:new>
```





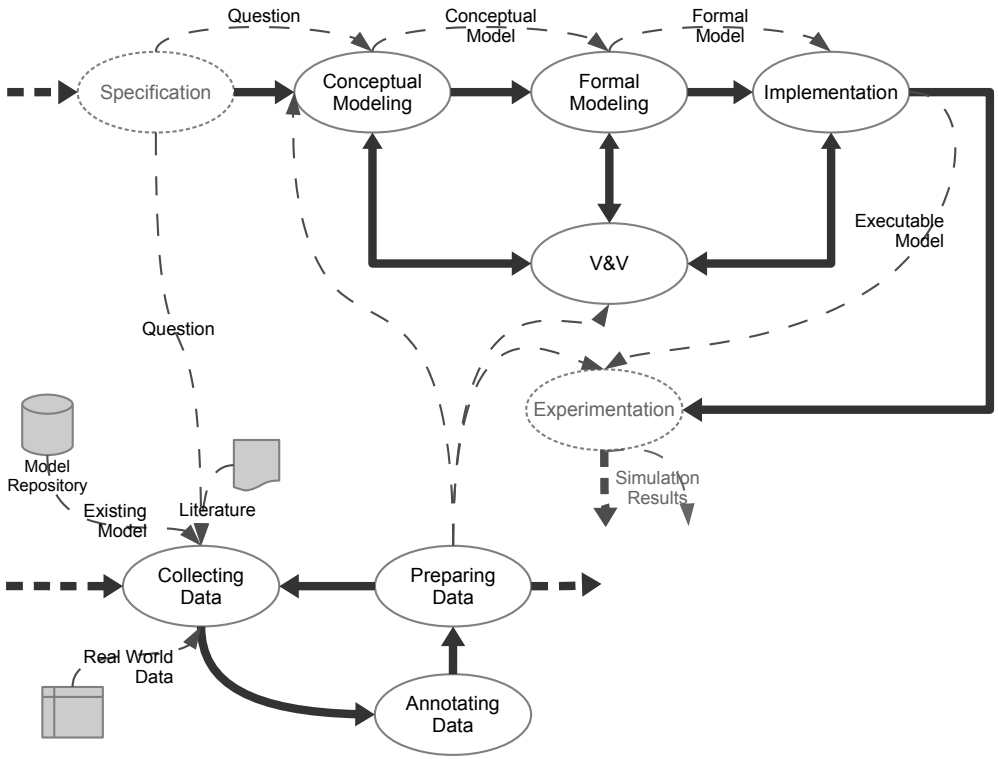
Bittig A.T., et al. Membrane related dynamics and the formation of actin in cells growing on micro-topographies: a spatial computational model. BMC Systems Biology, 2014

Summary DSLs for modeling

- offer the possibility to combine a compact, succinct description of models with a clear semantics, e.g.,
- ML-Rules (compartmental dynamics + reaction diffusion system on a grid) based on a CTMC semantics
- ML-Space (compartmental dynamics + particle dynamics + reaction diffusion system on a grid) based on a hybrid spatial semantics
- Many more do exist ...

DSLs for simulation: what do we want to describe?

In-Silico Experiments



Rybacki S. et al. (2014): Developing simulation models - from conceptual to executable model and back - an artifact-based workflow approach. Proc. of Simutools '14

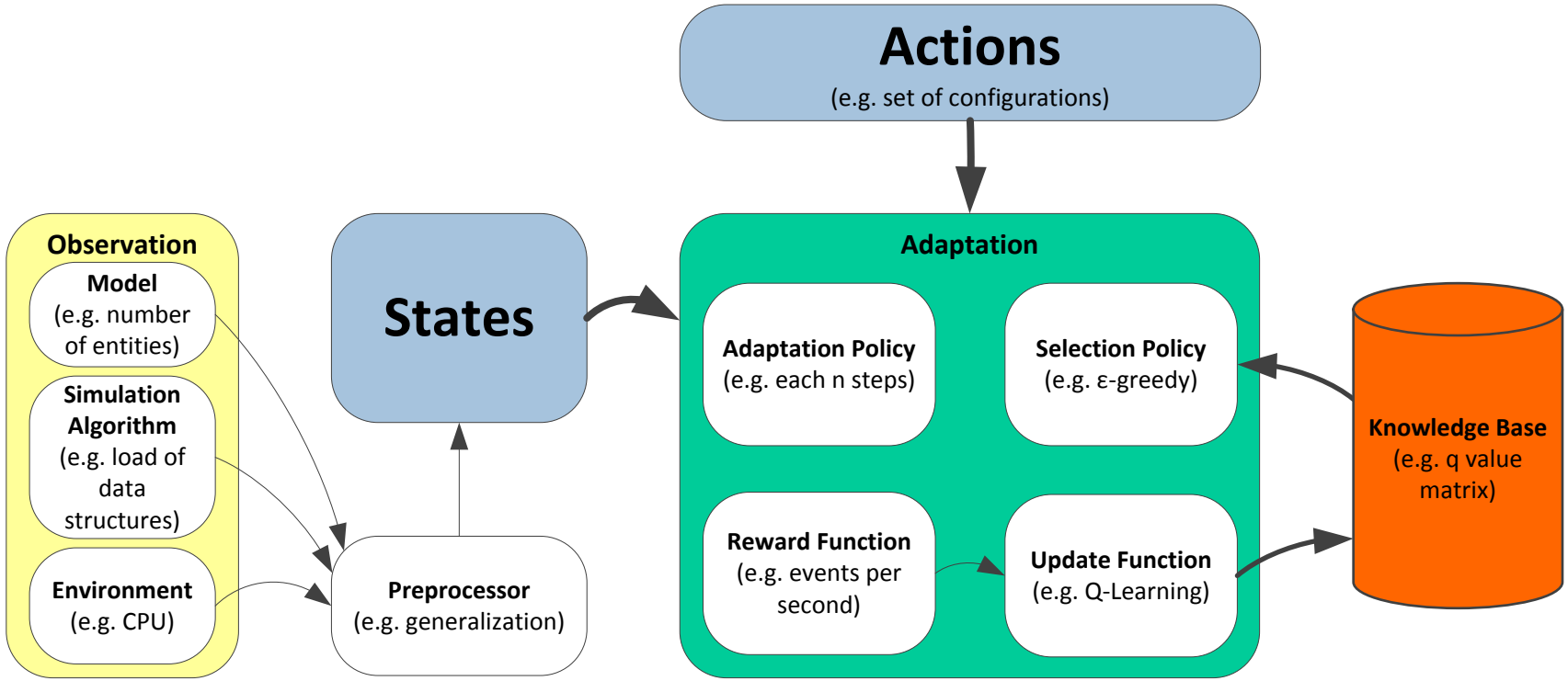
For any experiment, the model needs to be executed

- Parallelization: fine-grained (within one single simulation run) or/and coarse-grained (over multiple simulation runs)
- Exploiting GPUs
- Approximative methods: trading accuracy for speed
- Suitable configuration of simulation engines
- Hybrid methods
- Adaptive methods

horses for courses

https://en.wikipedia.org/wiki/Edgar_Degas

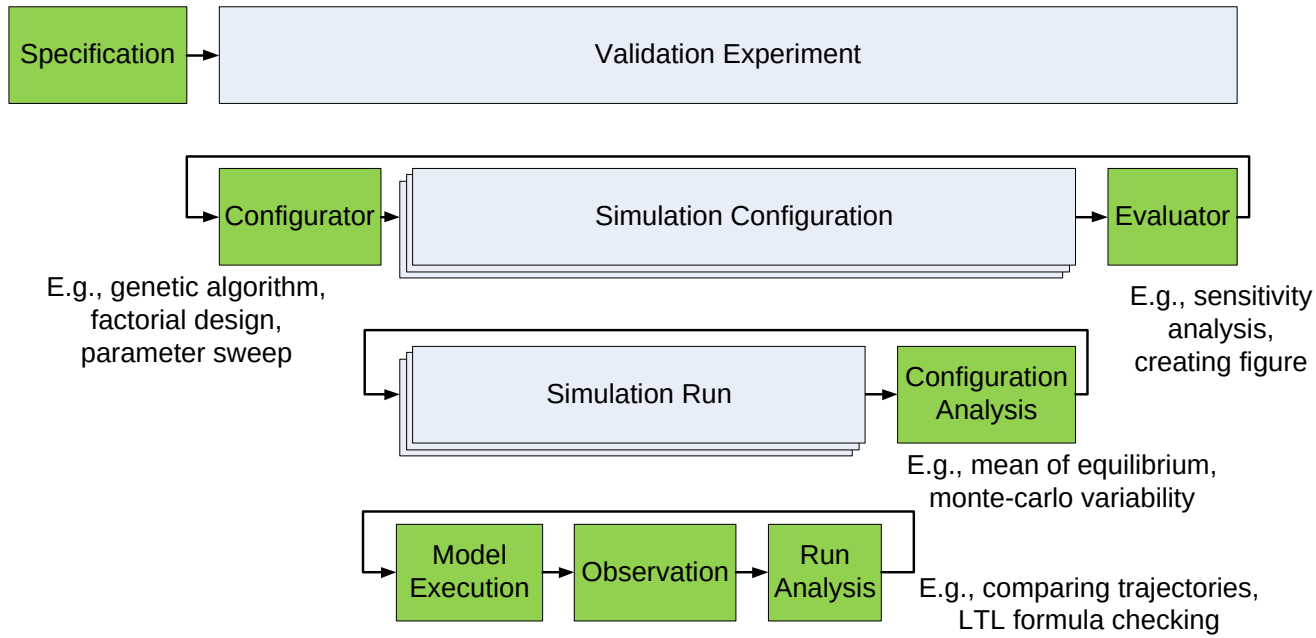
Adaptive Automated Selection and Configuration of Algorithms



Helms T. et al. (2013): Automatic Runtime Adaptation for Component-based Simulation Algorithms. TOMACS, 2015

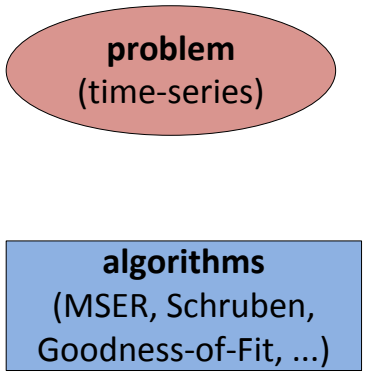
Good execution algorithms is a must, but more is needed!

Experiments - more than only one run



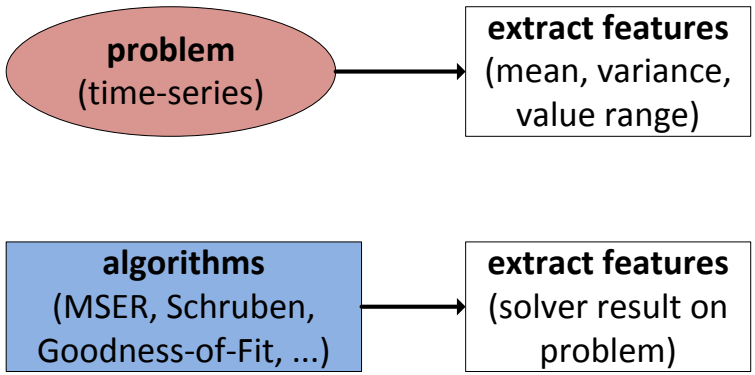
Leye S. et al. (2010): A flexible and extensible architecture for experimental model validation. Proc. of SimuTools 2010

Generating synthetic problem solvers by ensemble learning



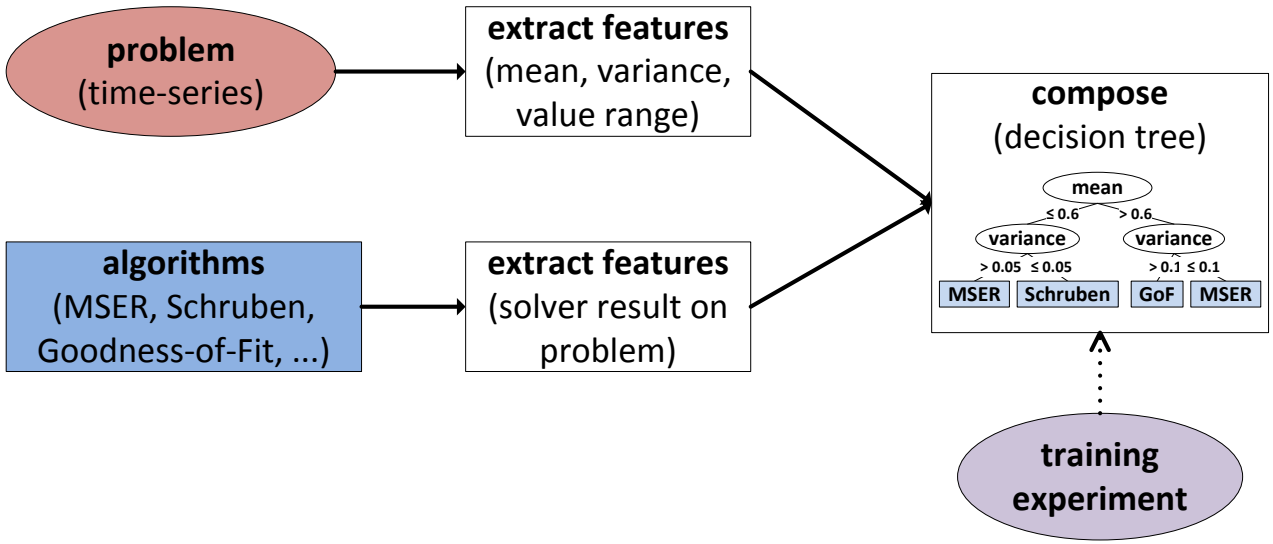
Leye S. et al. (2014): Composing Problem Solvers for Simulation Experimentation: A Case Study on Steady State Estimation. PLoS ONE 9(4)

Generating synthetic problem solvers by ensemble learning



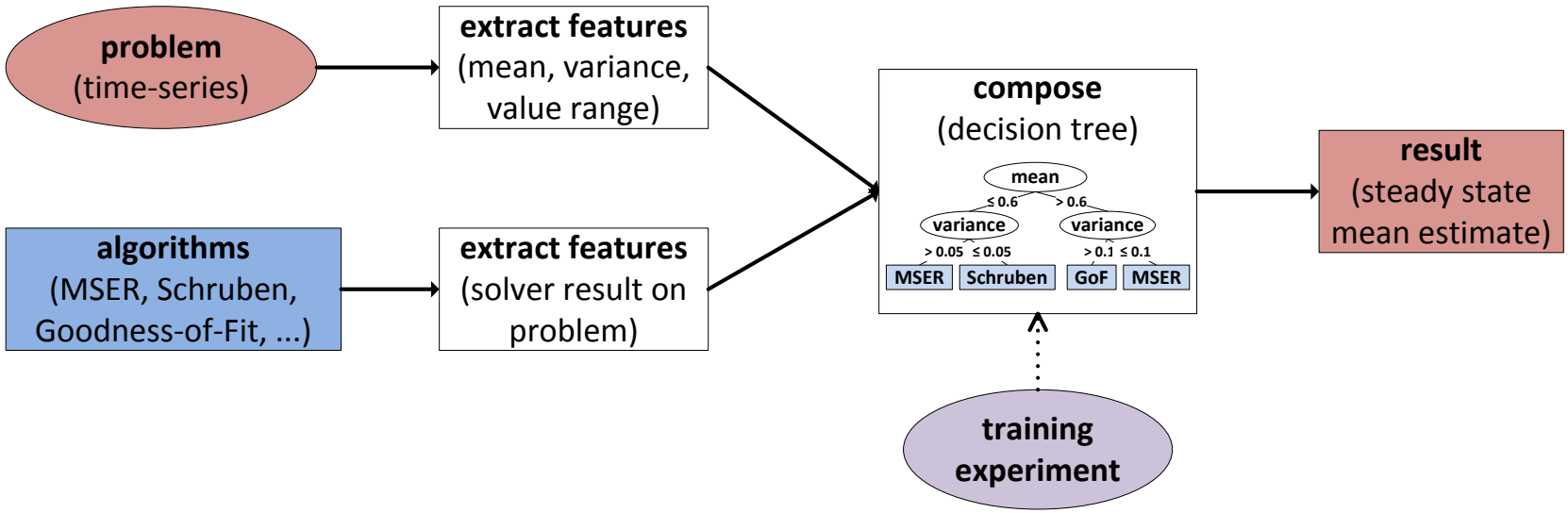
Leye S. et al. (2014): Composing Problem Solvers for Simulation Experimentation: A Case Study on Steady State Estimation. PLoS ONE 9(4)

Generating synthetic problem solvers by ensemble learning



Leye S. et al. (2014): Composing Problem Solvers for Simulation Experimentation: A Case Study on Steady State Estimation. PLoS ONE 9(4)

Generating synthetic problem solvers by ensemble learning



Leye S. et al. (2014): Composing Problem Solvers for Simulation Experimentation: A Case Study on Steady State Estimation. PLoS ONE 9(4)

How to describe these kind of experiments ?

An internal domain-specific language SESSL

SESSL (Simulation Experiment Specification via a scala layer) is an internal domain specific language for specifying experiments.

```
execute { // execute experiment
  new Experiment with ParallelExecution { // create experiment
    model = "sampleModel.file" // use model stored in this file
    // complex stopping and replication conditions are supported:
    stopCondition = AfterSimTime(0.6) and
        (AfterWallClockTime(seconds = 30) or AfterSimSteps(10000))
    replications = 100
    rng = MersenneTwister(1234) // use random number generator
    parallelThreads = -1 // exploit parallelism, leave one core idle
    // define factorial experiment:
    scan("x" <~ (1, 2), "y" <~ range(1, 1, 10)) } }
```

Ewald R. et al. (2014): SESSL: A Domain-Specific Language for Simulation Experiments, TOMACS

Optimization

```
val ref = Set(0, 7561, 8247, 7772, 7918, 7814, 7702)
minimize { (params, objective) =>
 execute {
  new Experiment with Observation with ParallelExecution {
    model = "file-mlrj:/." + dir + "/Wnt_apCrine.mlrj"
      // Set model parameters as defined by optimizer:
      set("kLphos" <~ params("p")) ....
      observe("Cell/Nuc/Bcat()")....
      withRunResult(results => {
        runResults += scala.math.sqrt(mse(numbers, ref))})
      withReplicationsResult(results => {
        // Store value of objective function:
        objective <~ runResults /count }) } }
} using (new Opt4JSetup {
  param("p", 0.1, 0.1, 10) // Optimization parameter bounds
  optimizer = sessl.opt4j.SimulatedAnnealing ... }})
```

Statistical model checking

```
val exp = new Experiment with Hypothesis {

  //model configuration
  model = "file-sr:/./LotkaVolterra.mlrj"
  set("nWolf" <~ 50,
      "nFox" <~ 500,
      "nFood" <~ 100)

  //simulation configuration
  simulator = MLRulesTauLeaping()
  replications = 10
  stopCondition = AfterSimTime(500)

  //property
  assume{(Probability >= 0.8)(
    P(Peak("wolf","wolfPeakH"), time < 250, "wolfNumPeaks"),
    Id("wolfPeakH") > 90 and Id("wolfPeakH") < 110,
    E(Increase("wolf"), length >= 100, "wolfNumIncreases"),
    Id("wolfNumIncreases") after Id("wolfNumPeaks")
    )}
  }
```

Summary DSLs for simulation

- offer the possibility to combine a compact, succinct description of experiments, e.g., simulation/optimization/ etc. system agnostic as SESSL, or even as an exchange standard like SED-ML
- DSLs help description, design and reuse of experiments
- DSLs can be specified for
  - complex experiments that can be run batch-like
  - data extraction
  - properties to be checked
  - ...
- a variety of DSLs for simulation are in use

DSLs for modeling and simulation

- two external DSLs for modeling: ML-Rules, and ML-Space (rather similar syntax but different semantics)
- one internal DSL for executing experiments: SESSL

Independently whether used for modeling or simulation, requirements are:

- compactness
- composibility
- ease of use
- sufficiently flexible (how much can be expressed?) and expressive (how easy can things be expressed?)
- clear semantics

To evaluate whether these requirements are met is not trivial but important to move the field ahead.

Contributions

- Arne Bittig: ML-Space, actin model
- Fiete Haack: Lipid raft models, Wnt-model, executing wet-lab studies
- Tobias Helms: ML-Rules simulation engine, ML-Rules τ leaping, automatic selection of execution algorithms
- Danhua Peng: reuse of SESSL experiments, automatic generation of experiments
- Johannes Schuetzel: Languages for observing and instrumenting models and streaming data
- Tom Warnke: Formal Semantics of ML-Rules, domain-specific language for demography, domain-specific language for statistical model checking, and statistical model checker
- Roland Ewald*: SESSL, evaluation and automatic selection of execution algorithms
- Jan Himmelspach*: Work on James II
- Stefan Leye*: Experimentation layer of James II, automatic generation of components
- Carsten Maus*, Mathias John*: Work on ML-Rules
- Stefan Rybacki*: Experimentation as workflow: workflows in M&S (WORMS) and artifact-based approach, CA-based modeling and simulation approaches, and co-work on ML-Rules

thank you for your attention