ESPResSo Summer School 2013

October 7 to October 11, University of Stuttgart

cecam
Centre Européen de Calcul Atomique et Moléculaire

Simulating Soft Matter with ESPResSo, ESPResSo++ and VOTCA

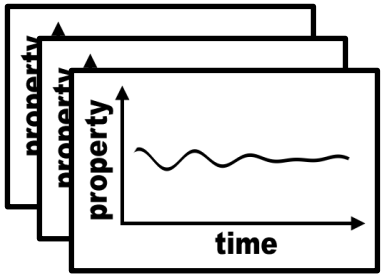
Max Planck Institute for Polymer Research

http://www.espresso-pp.de

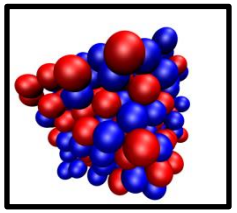
ESPResSo++

Key Features

- **ESPResSo++** is an open-source simulation package designed to perform *molecular dynamics* and *monte carlo* simulations of condensed matter systems using traditional and state-of-the-art techniques.
- Flexibility due to Python/C++ integration
- Extensibility due to object oriented design
- Easy script-guided creation of complex topologies
- Quick incorporation of new interactions and analysis methods
- Use of standard short- and long range interactions: e.G. Lennard-Jones, Stillinger-Weber, FENE, OPLS, Dihedrals, Ewald Coulomb, etc.
- Efficient implementation of advanced algorithms: e.G. (H-)AdResS, Parallel Tempering, dynamic bonds, etc.
- On-the-fly analysis and visualization
- Applicable to large systems and large numbers of CPUs due to efficient parallelisation



Analysis



Visualization

NumPy SciPy

PyMol VMD

Python scripting language

Advanced C++ Kernel

- Parallelization
- Time-critical algorithms
- Data storage
- Extensible modular design

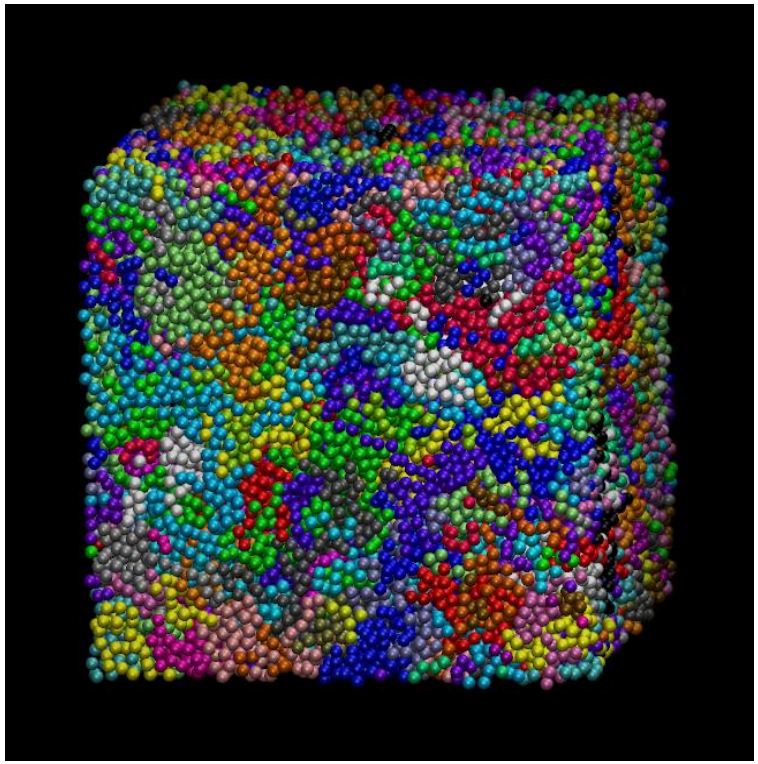
Outline of the talk

- Simulating a dense polymer melt (general workflow)
- Some things to know about python
- Typical ESPResSo++ python script and ESPResSo++ modules hierarchy
- System setup and basic classes
  - Some information about PMI in between
- Inside ESPReSo++ an overview of the C++ class structure
- Integrator, Interactions, Storage

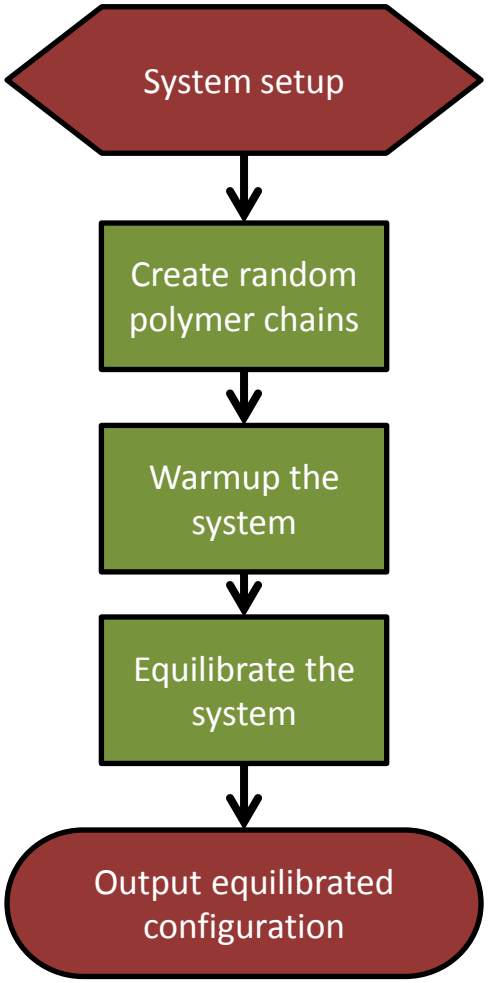
Outline of the afternoon tutorial session

- Installation of ESPResSo++
- Basic System Setup
- Simple Lennard Jones System
- Advanced Lennard Jones System
  with graphical output
- Polymer Melt

Example: simulating a dense polymer melt

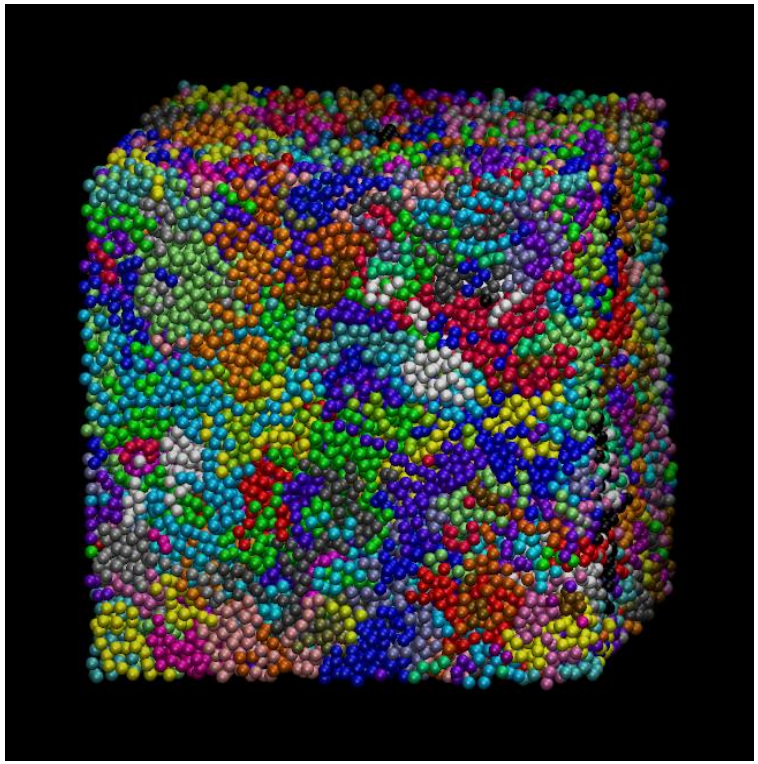


Snapshot of an equilibrated dense ring polymer melt, number of chains N_c=200, chain length l_c=200 particles, density ρ=0.85 [N/V^3]

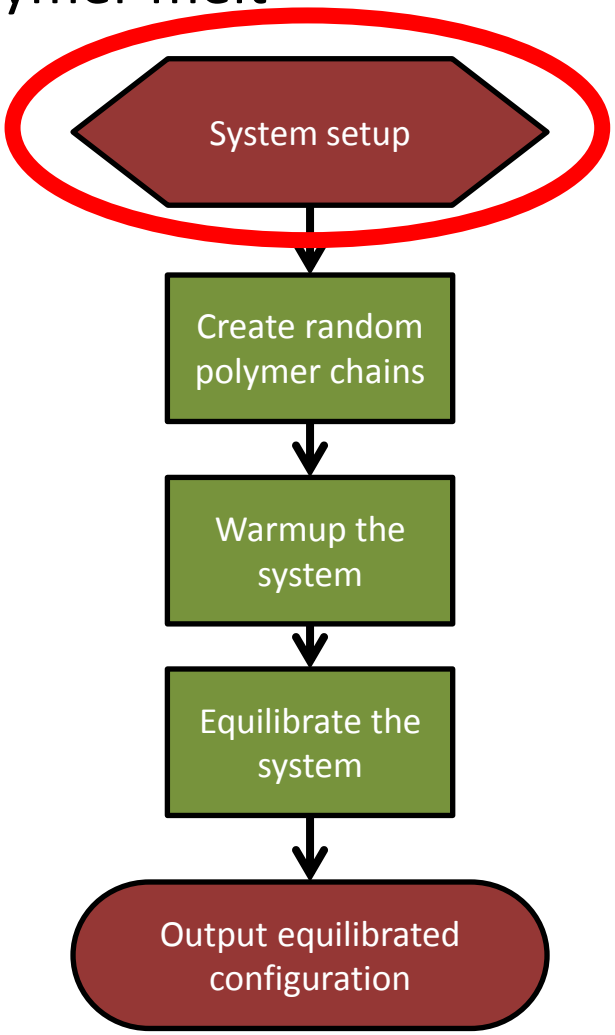


You will have time in the afternoon tutorial session to work with this example in more detail.

Example: simulating a dense polymer melt



Snapshot of an equilibrated dense ring polymer melt, number of chains N_c=200, chain length l_c=200 particles, density ρ=0.85 [N/V^3]



The following slides will show how basic modules of ESPResSo++ can be used to setup a system for the simulation of a polymer melt.

Some (very few) things to know about Python:

- Python is an interpreted language and does not have to be compiled
- Python has advanced object oriented structures
- Python can be used interactively

There is one syntactic specialty in python (and I don't know any other language that has this) : **begin and end of a block is not marked by any keyword or brackets but only by indentation**.
(Therefore it is very important to be disciplined when indenting the lines !)
examples:

```
for i in range(100):
  s += i
  print s, i
```

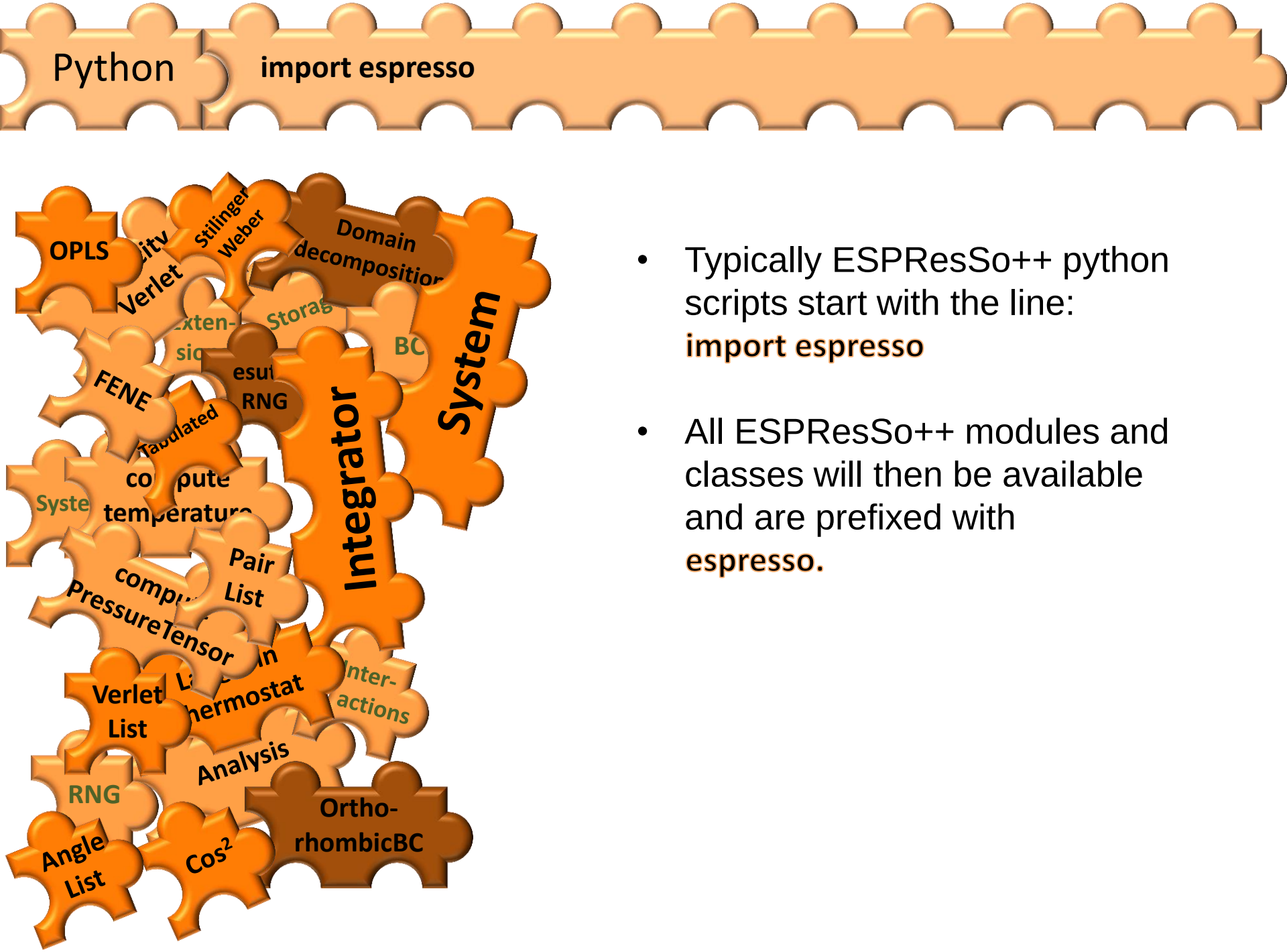
or

```
if (sum > 100):
  print 'sum = ', sum
  sum = 0
```

A typical ESPResSo++ python script consists of the following elements:

- System setup (define box, interactions, parameters, ...)
- Read in particle and topology information from a file (e.g. pdb, xyz, GROMACS or LAMMPS) or setup a new random configuration
- Integrate Newtons equations of motion
- Analysis, trajectory files, visualization and many other things can be done during integration
- Print final results

There is only one python script needed to reflect the workflow of a complete project !



- Typically ESPResSo++ python scripts start with the line:
  import espresso
- All ESPResSo++ modules and classes will then be available and are prefixed with
  espresso.

The main ESPResSo++ modules are

- **espresso** *(basic classes, e.g. Particle, data types like Real3D or Tensor and the System class, ...)*
- **espresso.analysis** *(e.g. Temperature, PressureTensor, g(r), ...)*
- **espresso.bc** *(boundary condition classes, e.g. OrthorhombicBC)*
- **espresso.esutil** *(e.g. random number generators, ...)*
- **espresso.integrator** *(e.g. VelocityVerlet, extensions like thermostats and barostats, ...)*
- **espresso.interaction** *(e.g. LennardJones, FENE, OPLS, Tabulated, ...)*
- **espresso.io** *(new file I/O classes, e.g. DumpXYZ, ...)*
- **espresso.storage** *(e.g. domain decomposition)*
- **espresso.tools** *(old file I/O classes, e.g. writepdb, interface to other programs like VMD, GROMACS, LAMMPS, ...)*

- **This hierarchy is also reflected by the directory structure of the src (espresso) directory-tree**

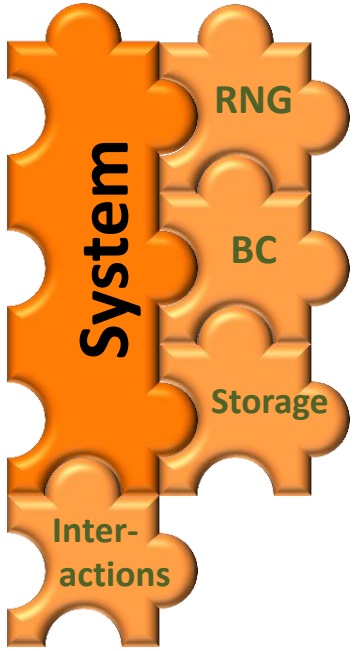
System setup
and
basic classes

Python

system = espresso.System()

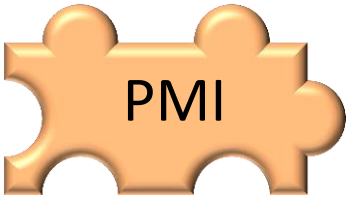
The System class holds links to other classes that are necessary for most simulations:

- Random number generators
- Boundary conditions
- Storage
- Interactions



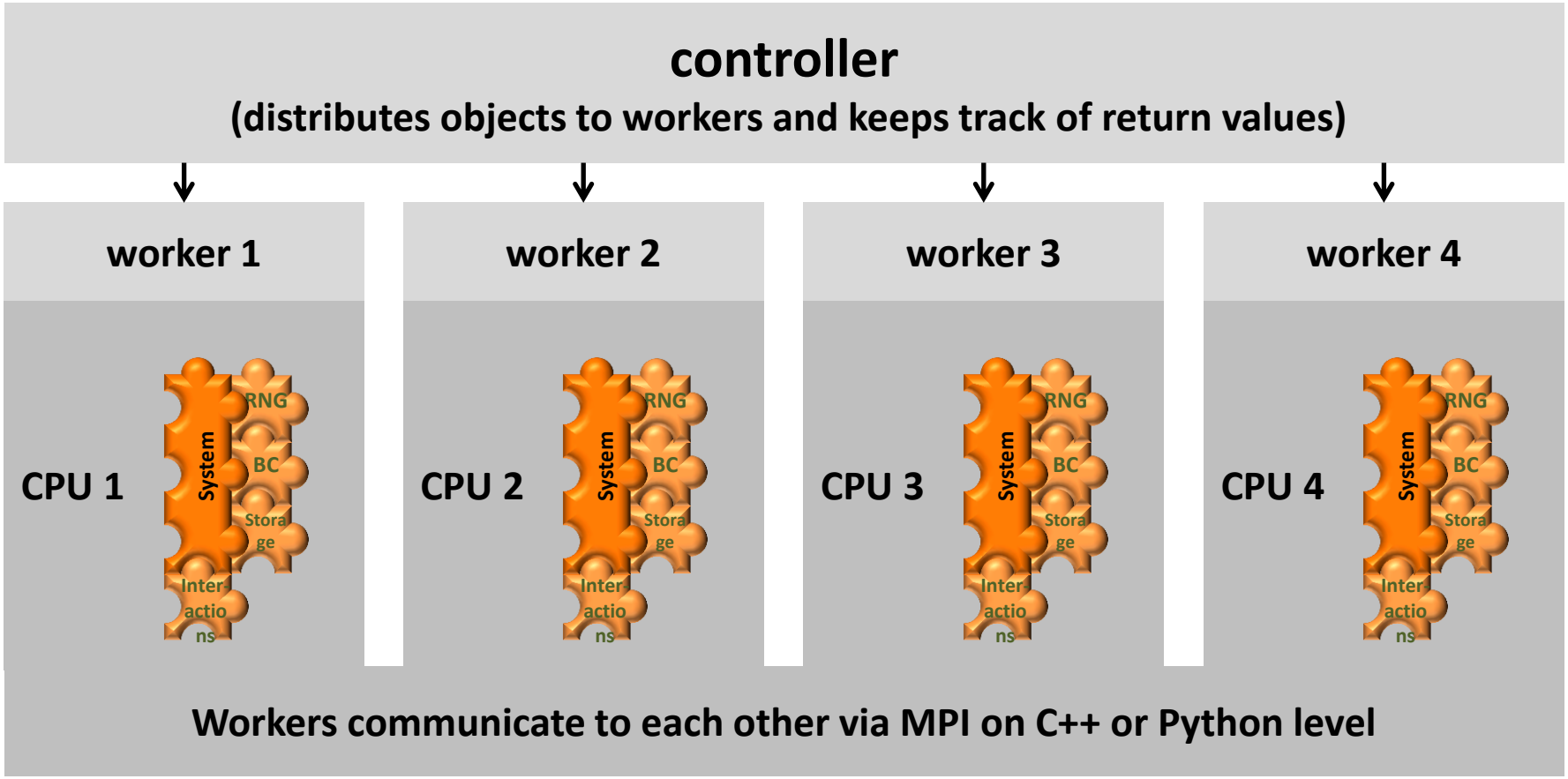
- PMI takes care of the parallelisation on the python level
- This will be explained in more detail on the next slide

Some words about parallelisation and PMI:

Python module 

distributes objects (transparently for the user) to parallel threads and invokes them

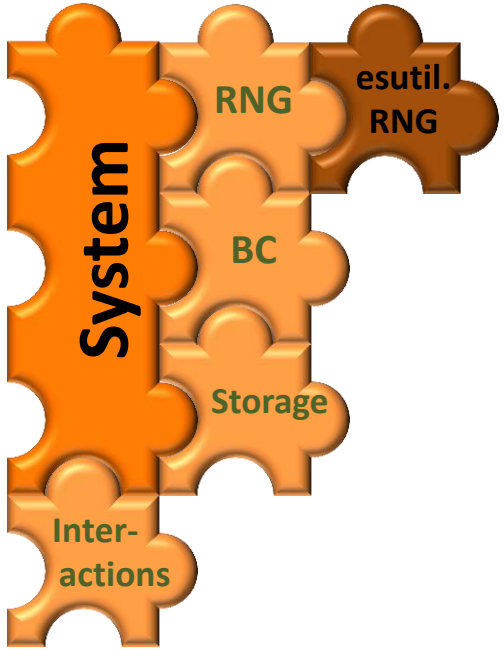
(Parallel **M**ethod **I**nvokation)



Python

system.rng = espresso.esutil.RNG()

- We create a random number generator object and connect it to the system.
- (Remember: PMI will take care of doing this on all the workers.)
- Other objects, e.g. the Langevin thermostat extension can use the RNG of the system

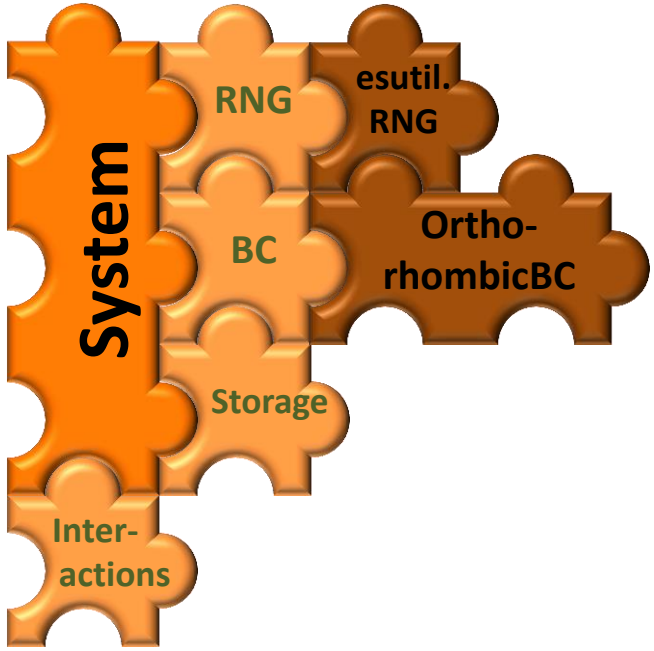


- Always remember: PMI takes care of distributing the objects to the works
- Usually this is totally transparent for the user

Python

system.storage = espresso.bc.OrthorhombicBC(box, ...)

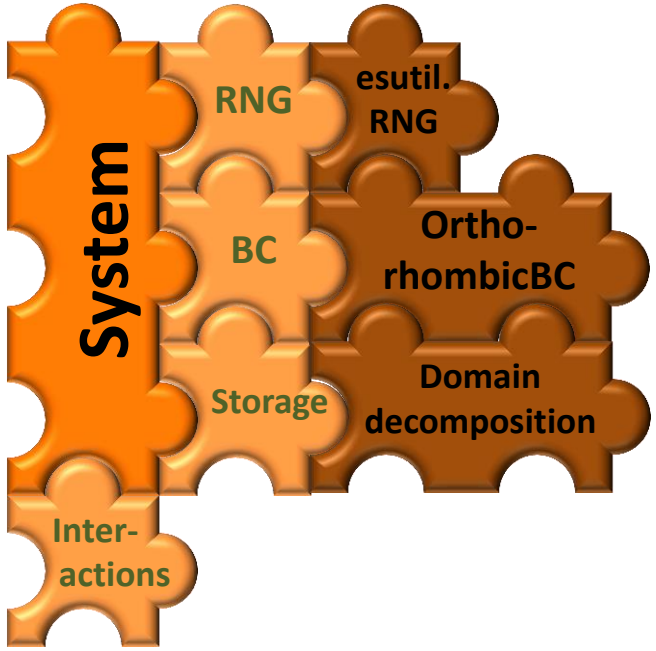
- Next comes the boundary condition object.
- This class takes care of measuring distances
- It has information about the size of the simulation box
- And it can create random coordinates within the simulation box
- Therefore it also needs a RNG



Python

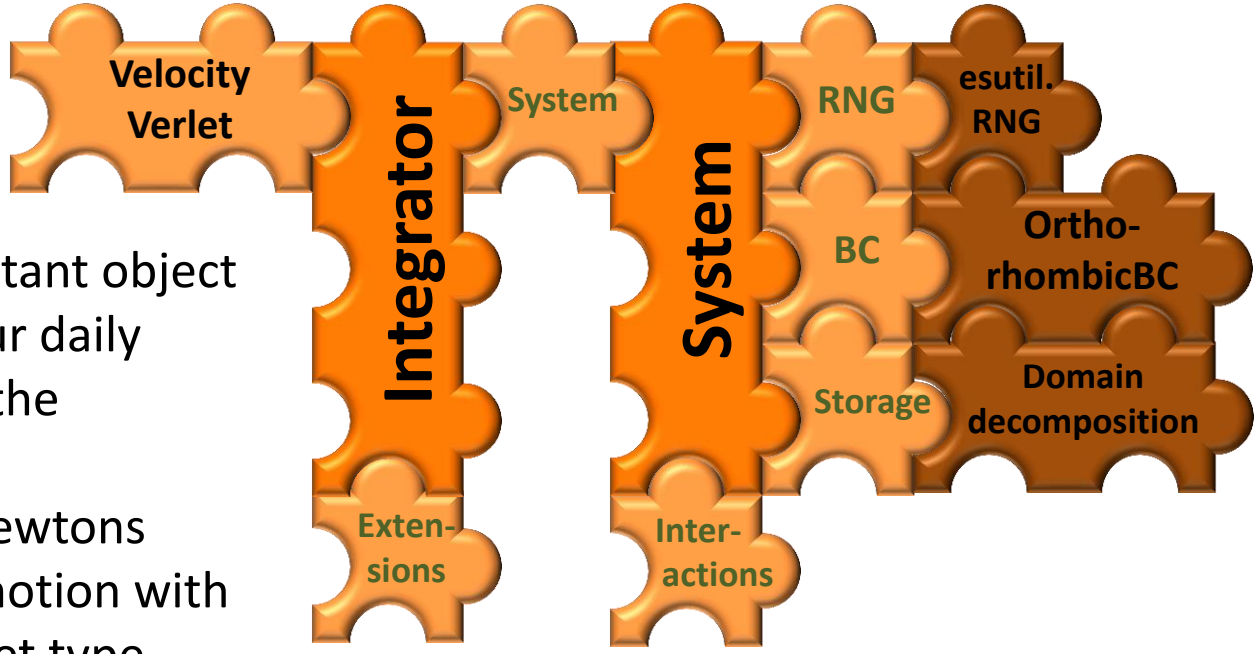
system.storage = espresso.storage.DomainDecomposition(*)

- The next important object that we need to create is the domain decomposition object
- It takes care of storing all the particles in a multi CPU environment
- It also handles sending around particles between CPUs via mpi_recv and mpi_send (we use boost::mpi for that)
- It informs other objects (like particle lists or bond lists) when this happens so that they can also update their data



Python

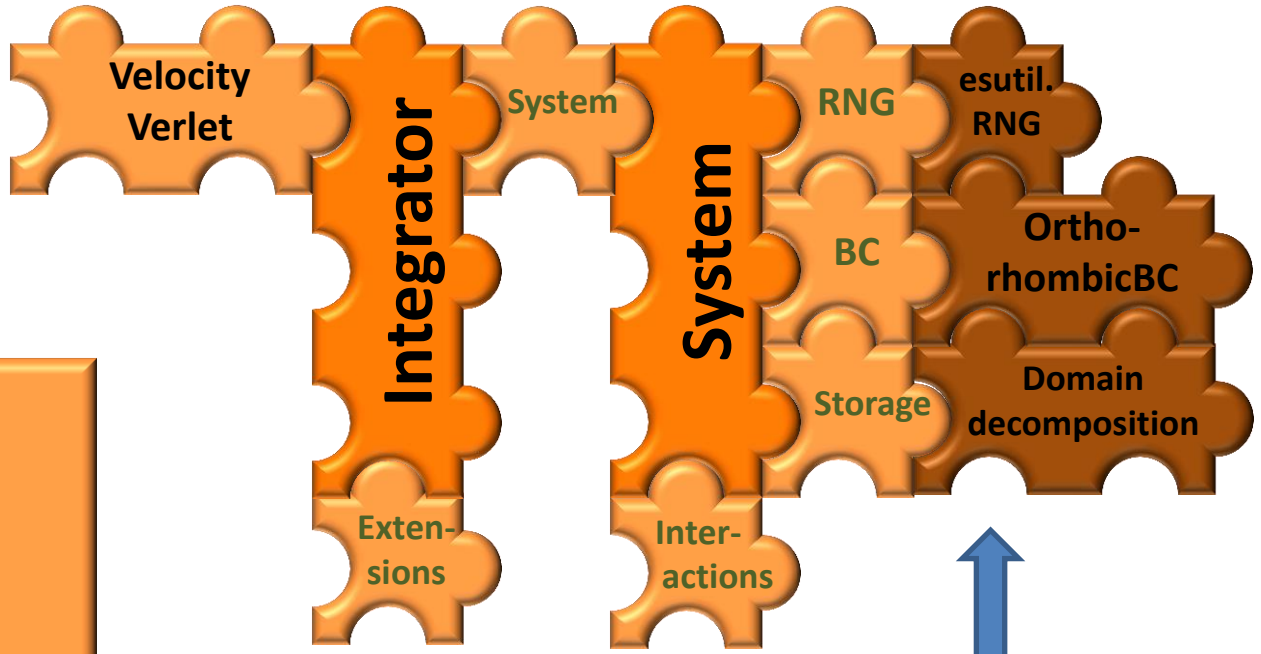
```
integrator = espresso.integrator.VelocityVerlet( system )
```



- Another important object we need for our daily simulations is the integrator
- It integrates Newtons equations of motion with a Velocity-Verlet type scheme
- It also send signals to the integrator extensions and allows them interfere in the scheme

Python

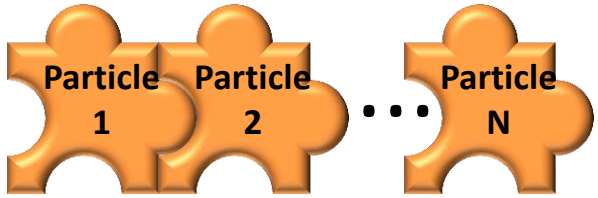
system.storage.addParticles(particleList, *particleProperties)

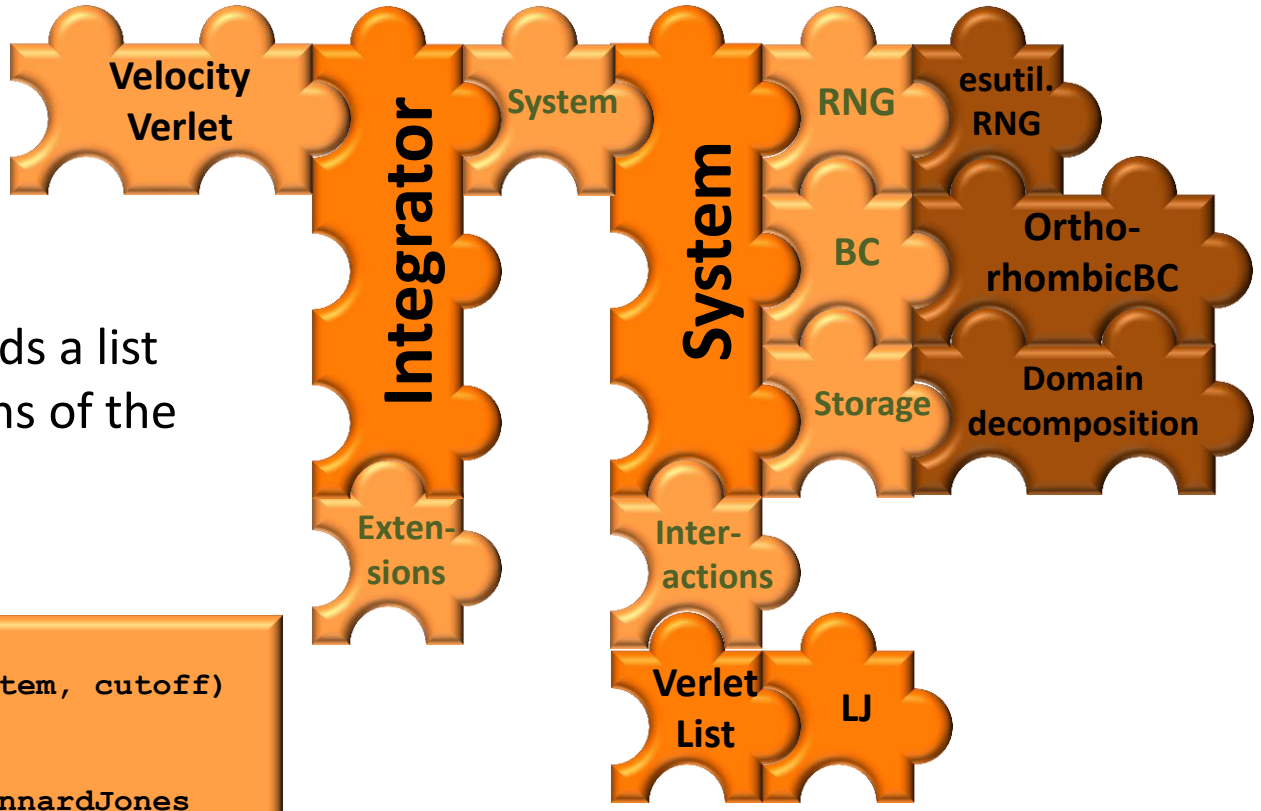



```
particleList =
      [ [1, 0, pos1, 1.0],
        [2, 0, pos2, 1.0],
              .
              .
              .
       [N, 0, posN, 1.0] ]
```

```
particleProperties =
 ['id', 'type', 'pos', 'mass']
```

- We can add some particles now
- Each particle is stored on one CPU only

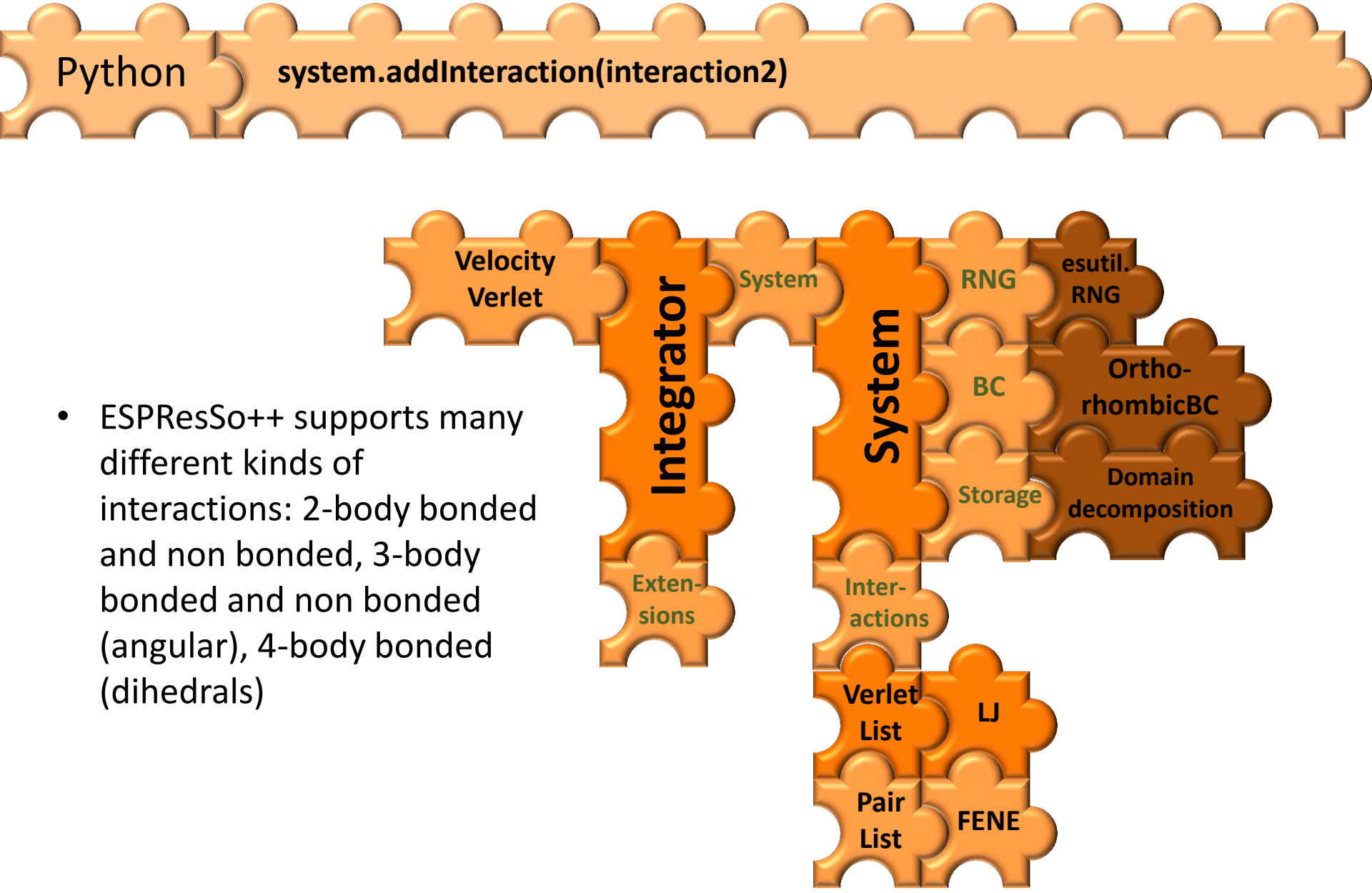




- The system also holds a list to all the interactions of the simulation

```
verletlist =
  espresso.VerletList(system, cutoff)

LJpot =
  espresso.interaction.LennardJones
  (epsilon, sigma, cutoff, shift)

interaction1 = espresso.interaction.
  VerletListLennardJones(verletlist)

interaction1.setPotential
  (type1=0, type2=0, LJpot)
```

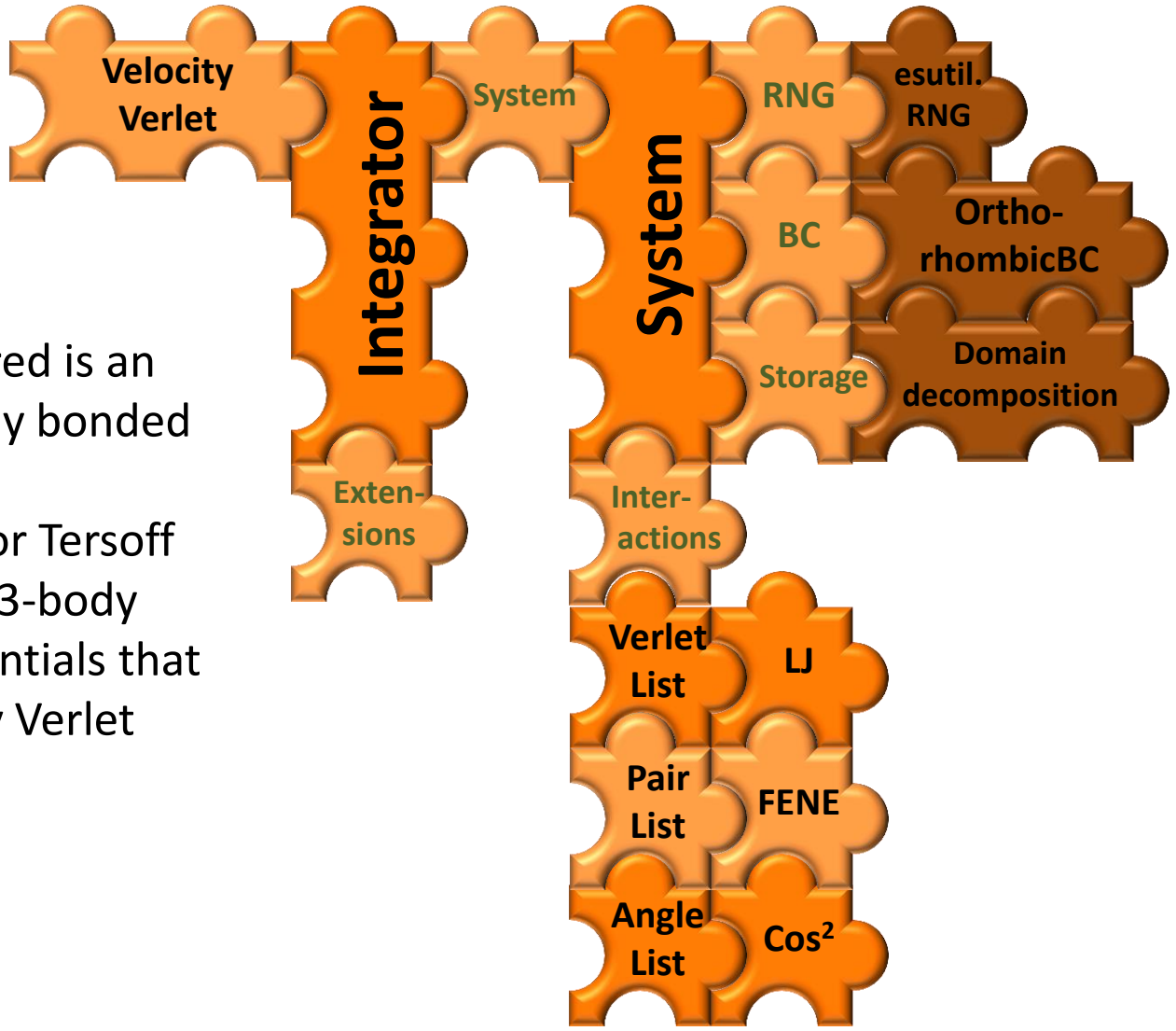
- Interactions consist of two parts: the actual potential (e.g. Lennard Jones) and the type (nonbonded=Verlet*, bonded=Fixed*Lists)



- ESPResSo++ supports many different kinds of interactions: 2-body bonded and non bonded, 3-body bonded and non bonded (angular), 4-body bonded (dihedrals)

Python

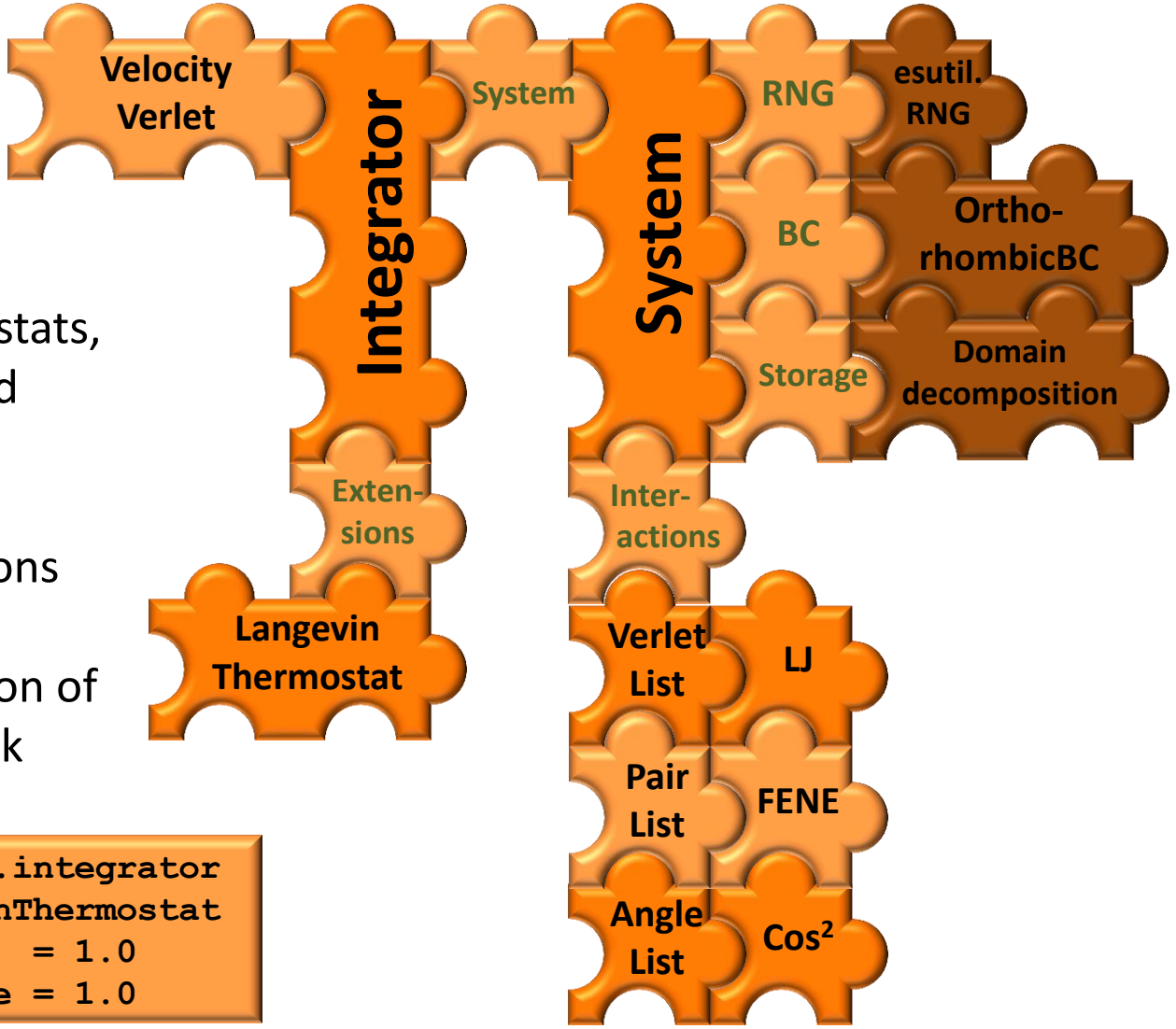
system.addInteraction(interaction3)



- The cosine_squared is an example of 3-body bonded potential
- Stillinger Weber or Tersoff are examples for 3-body non bonded potentials that work with 3-body Verlet lists

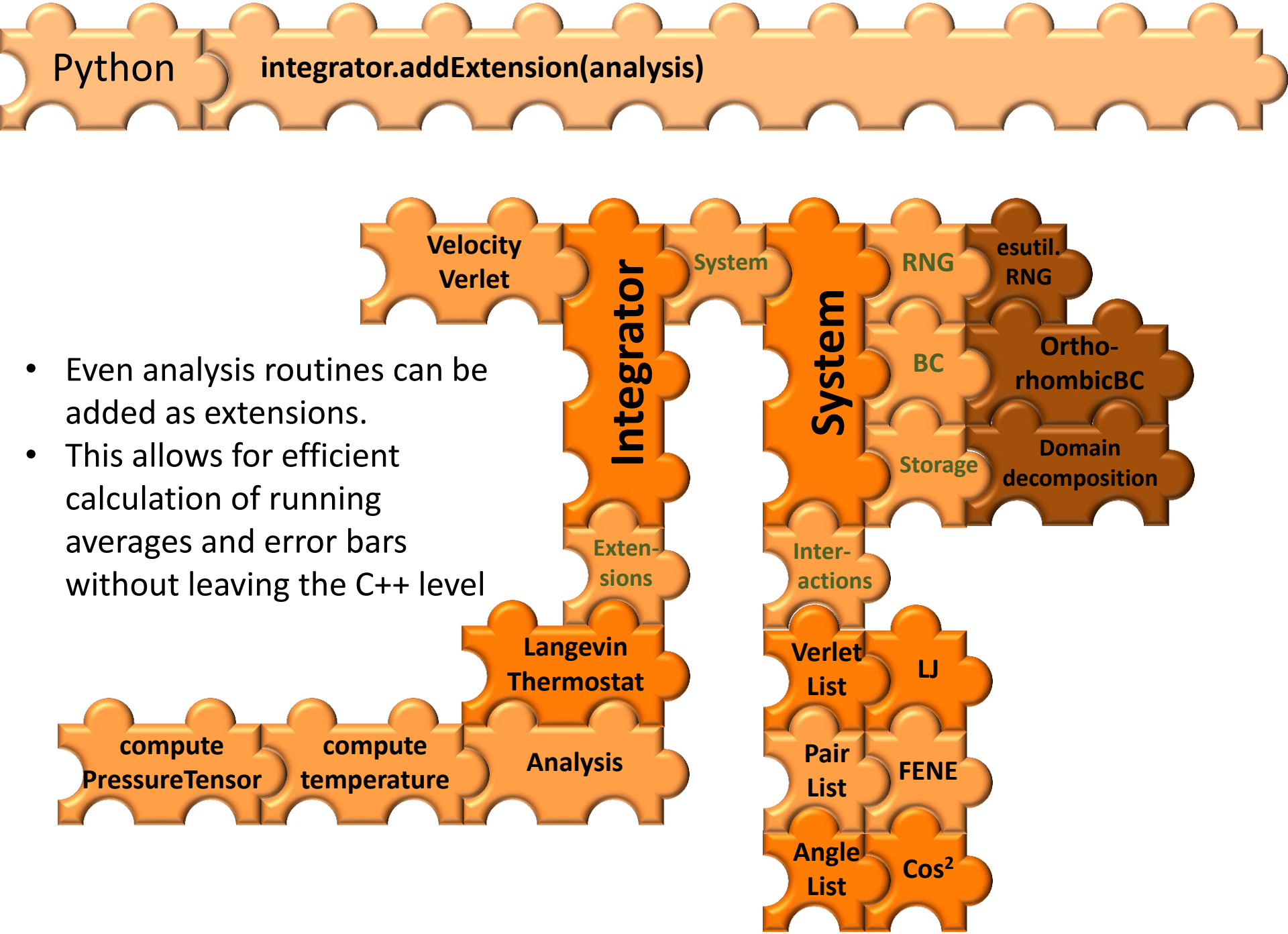
Python

integrator.addExtension(thermostat)



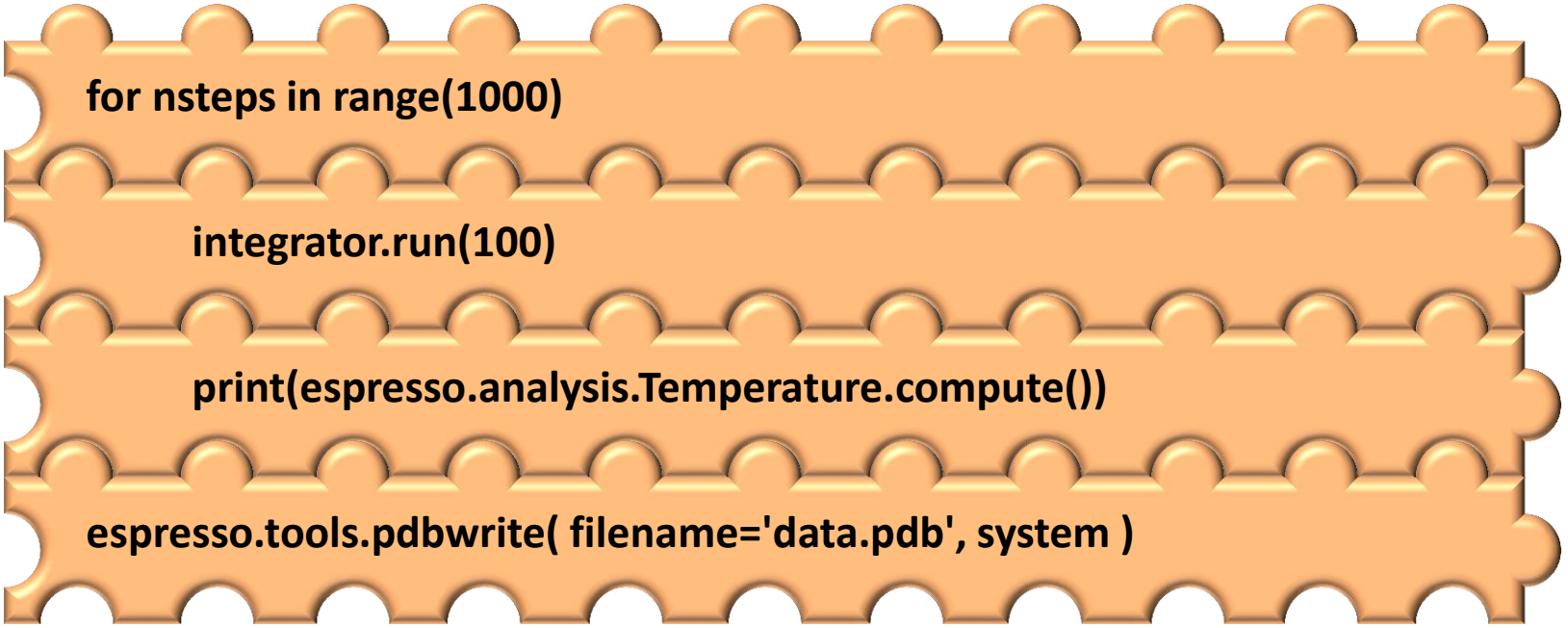
- Things like thermostats, barostats, external forces and many others are implemented as integrator extensions
- There will be a detailed explanation of this later in the talk

```
thermostat = espresso.integrator
             .LangevinThermostat
thermostat.gamma       = 1.0
thermostat.temperature = 1.0
```

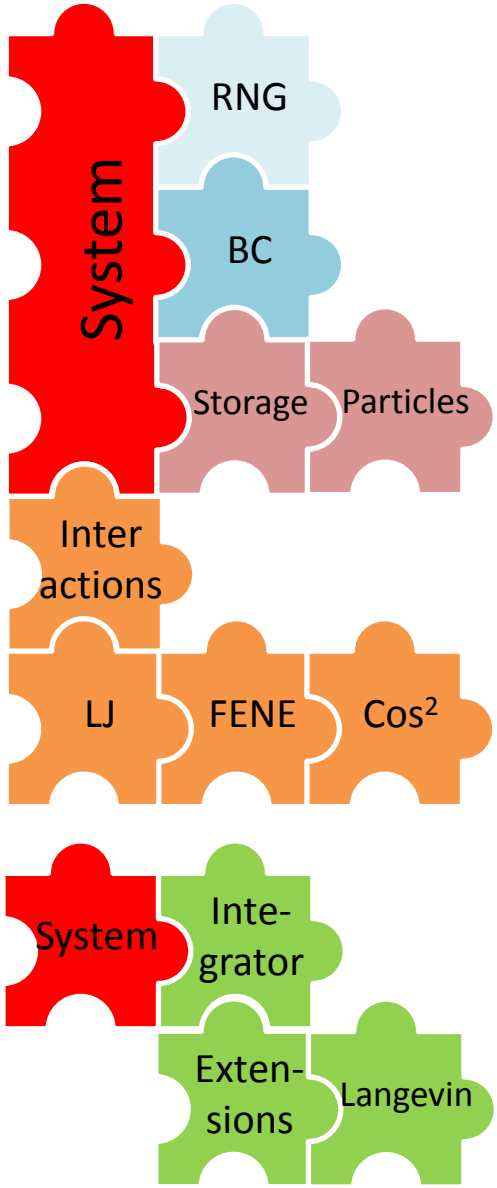


- Even analysis routines can be added as extensions.
- This allows for efficient calculation of running averages and error bars without leaving the C++ level

- Finally integrate Newtons equation of motion
  - print some analysis information during integration
- Write final configuration to PDB file



Simplyfied summary

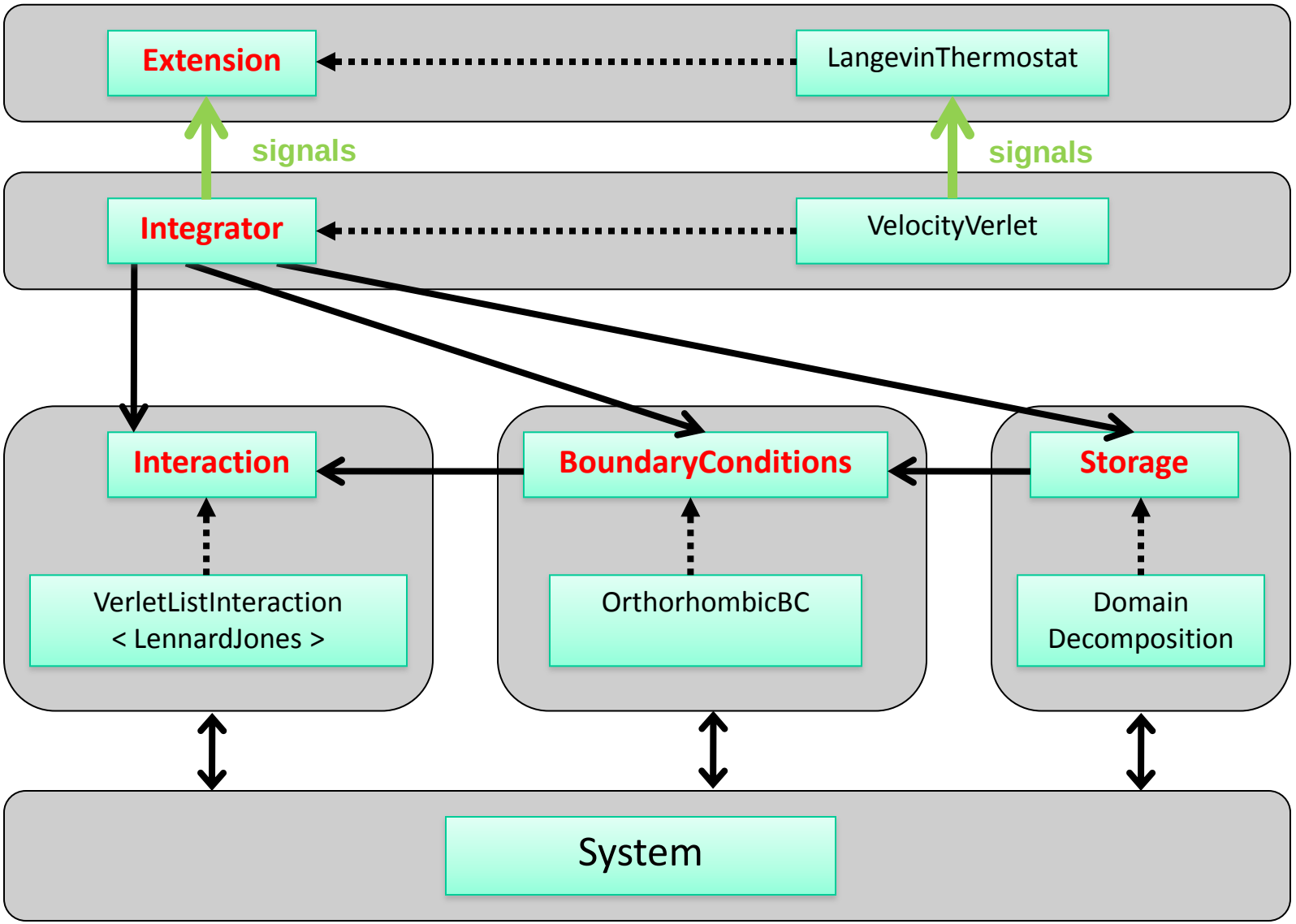



```
s1          = System()
s1.rng      = esutil.RNG()
s1.bc       = bc.OrthorhombicBC(<args>)
s1.storage =
   storage.DomainDecomposition(<args>)
      .
      . <create particlelist>
      .
s1.storage.addParticles(*property_list,
                        particle_list)
      .
      . <define interactions with parameters>
      .
s1.addInteraction(LJ)
s1.addInteraction(FENE)
s1.addInteraction(CosSq)

i1 = integrator.VelocityVerlet(s1)
      .
      . <create thermostat with parameters>
      .
i1.addExtension(langevin_thermostat)

i1.run(10000)
```

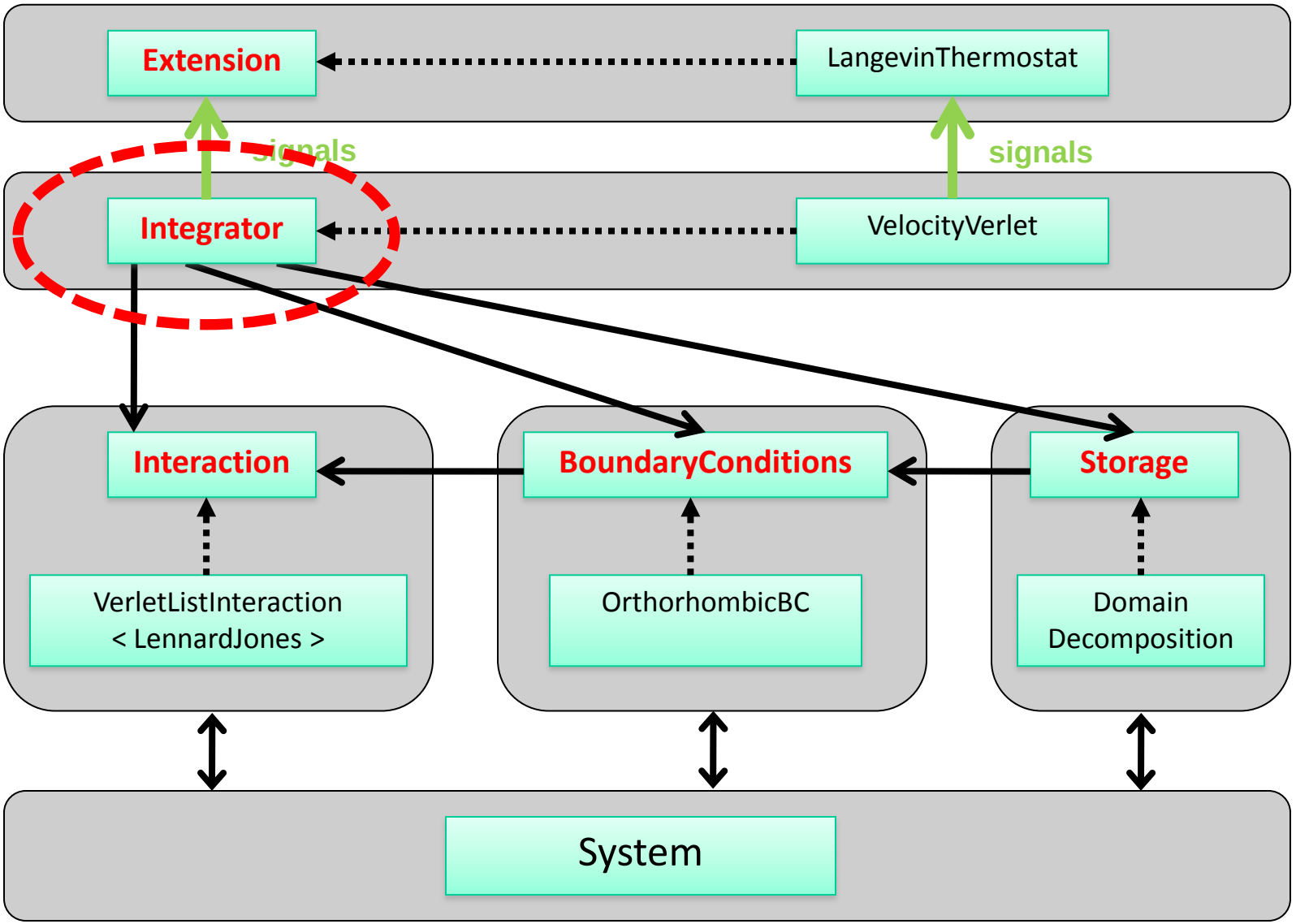
Inside ESPResSo++

- Abstract classes
- The integrator
- How integrator extensions work
- Interaction types
- Potentials
- Storage

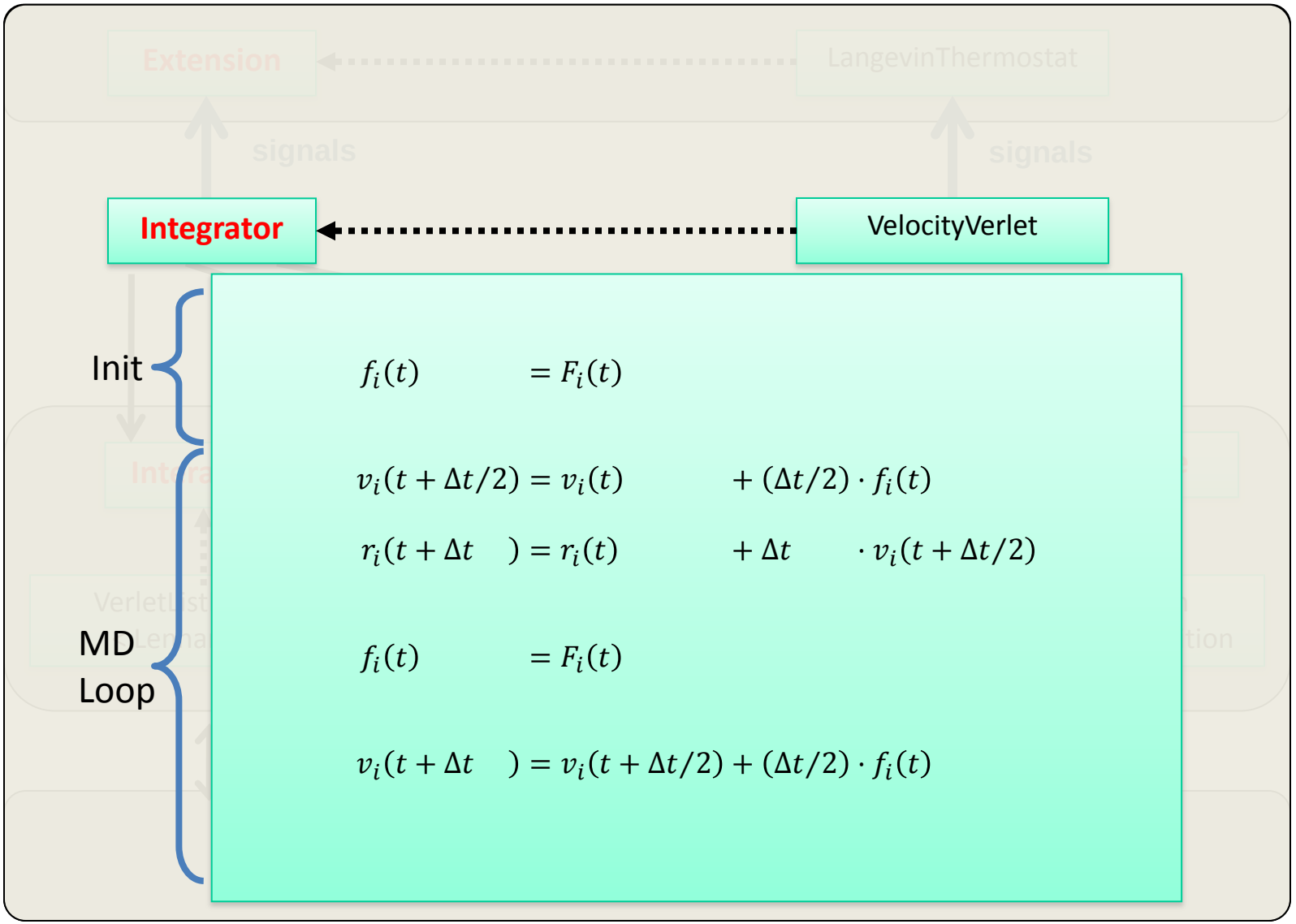
Abstract classes of ESPResSo++



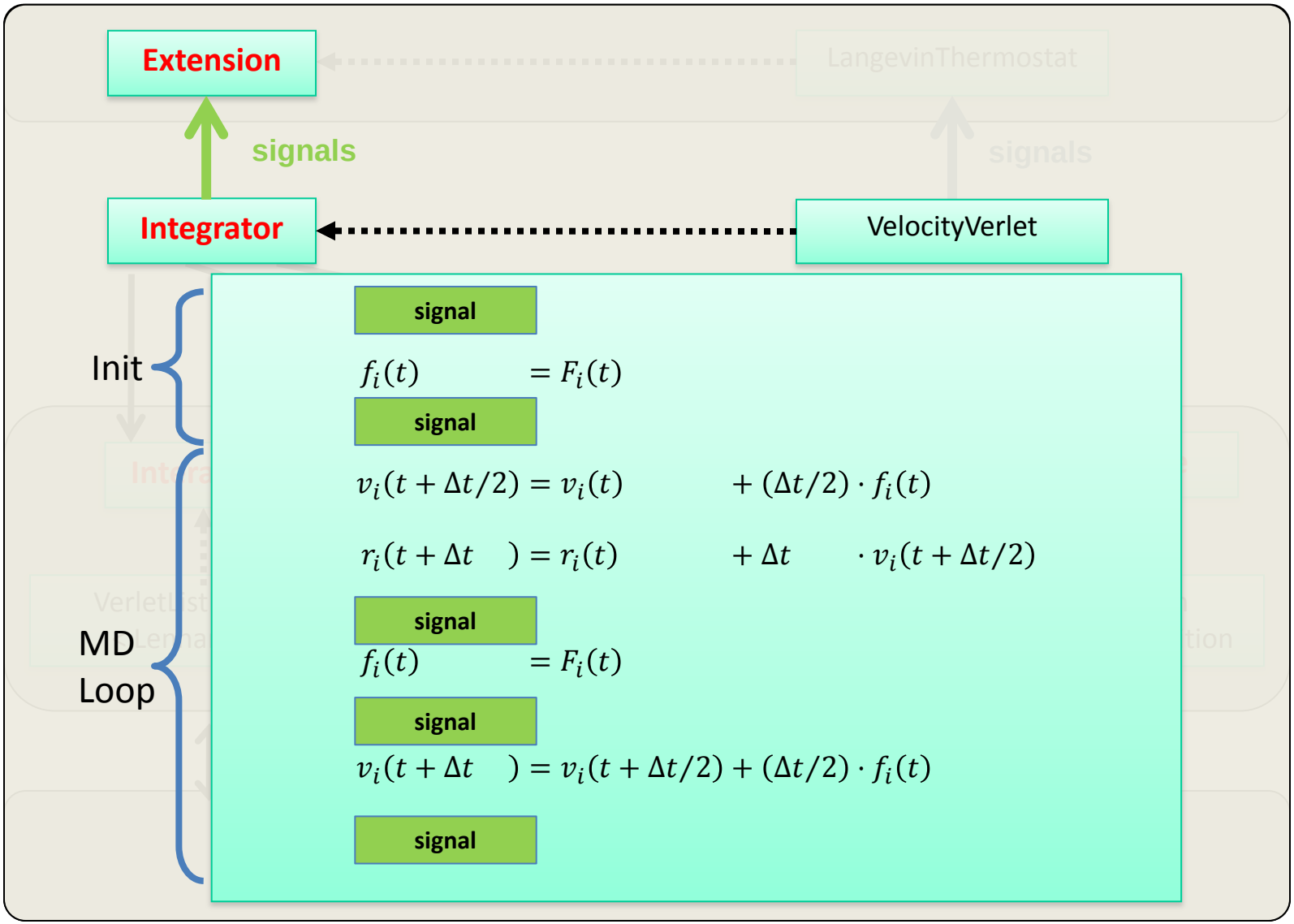
Abstract classes of ESPResSo++



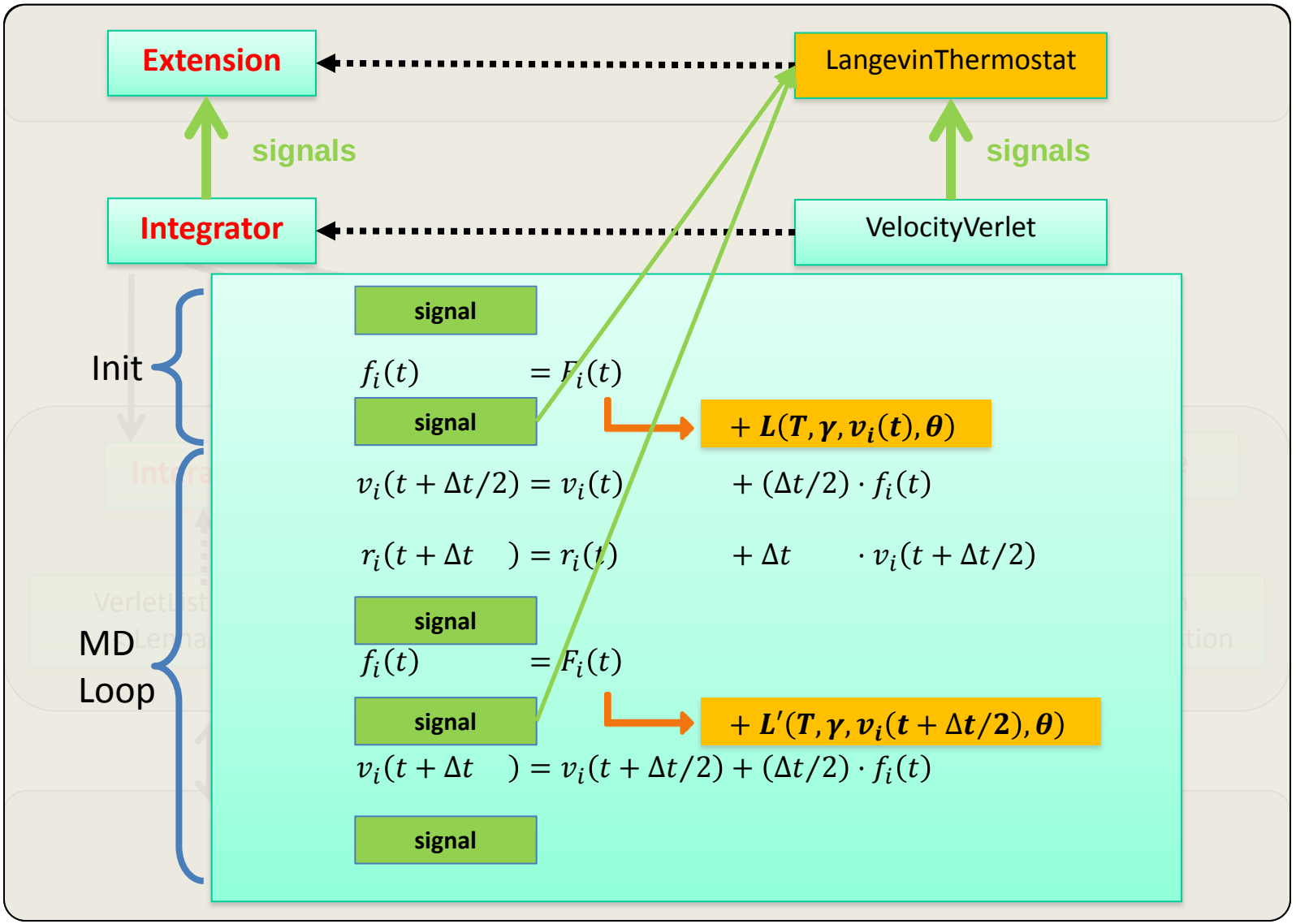
Integrator


Integrator
VelocityVerlet
Init
MD Loop
$f_i(t) = F_i(t)$
$v_i(t + \Delta t/2) = v_i(t) + (\Delta t/2) \cdot f_i(t)$
$r_i(t + \Delta t) = r_i(t) + \Delta t \cdot v_i(t + \Delta t/2)$
$f_i(t) = F_i(t)$
$v_i(t + \Delta t) = v_i(t + \Delta t/2) + (\Delta t/2) \cdot f_i(t)$

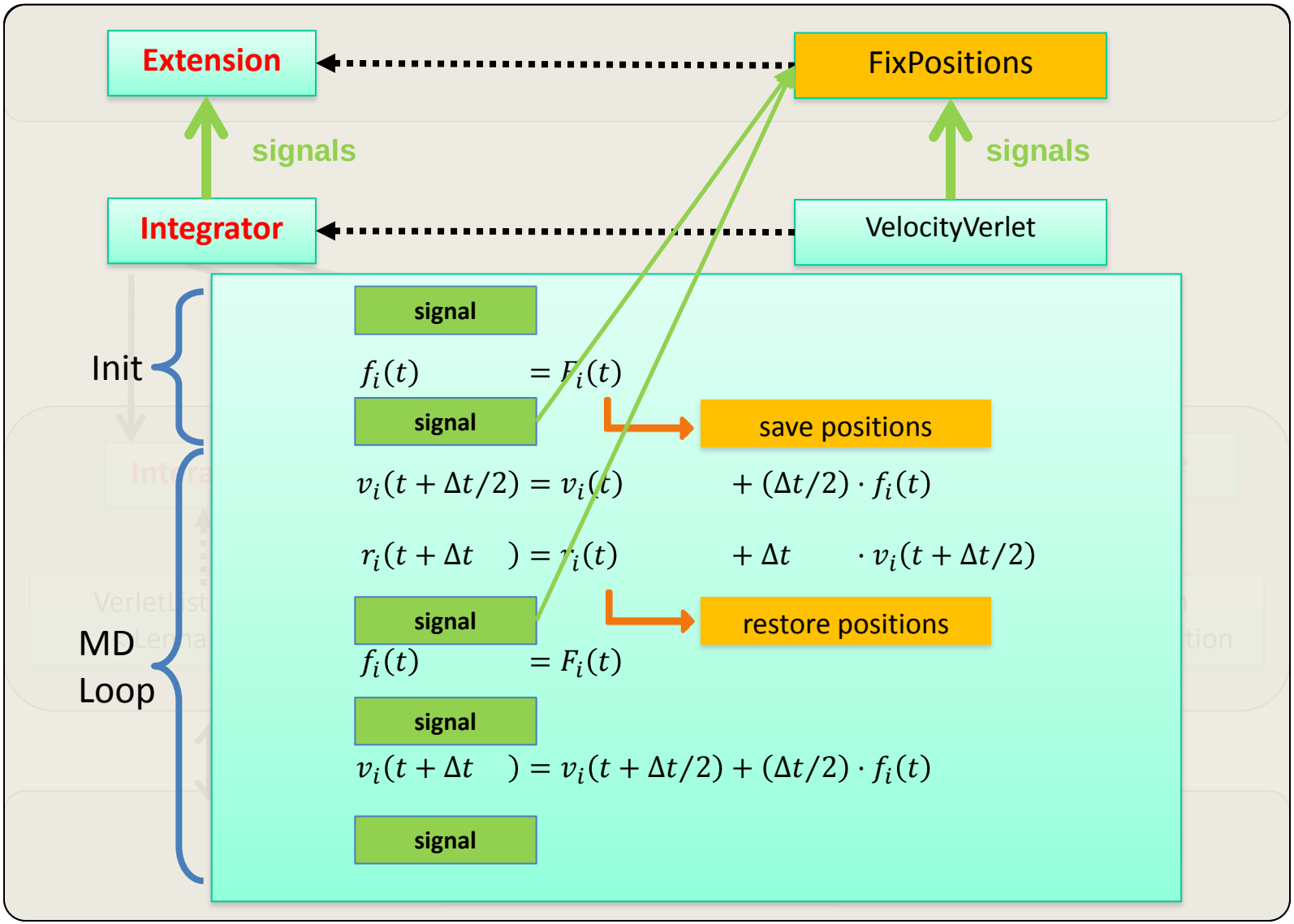
How integrator extensions work



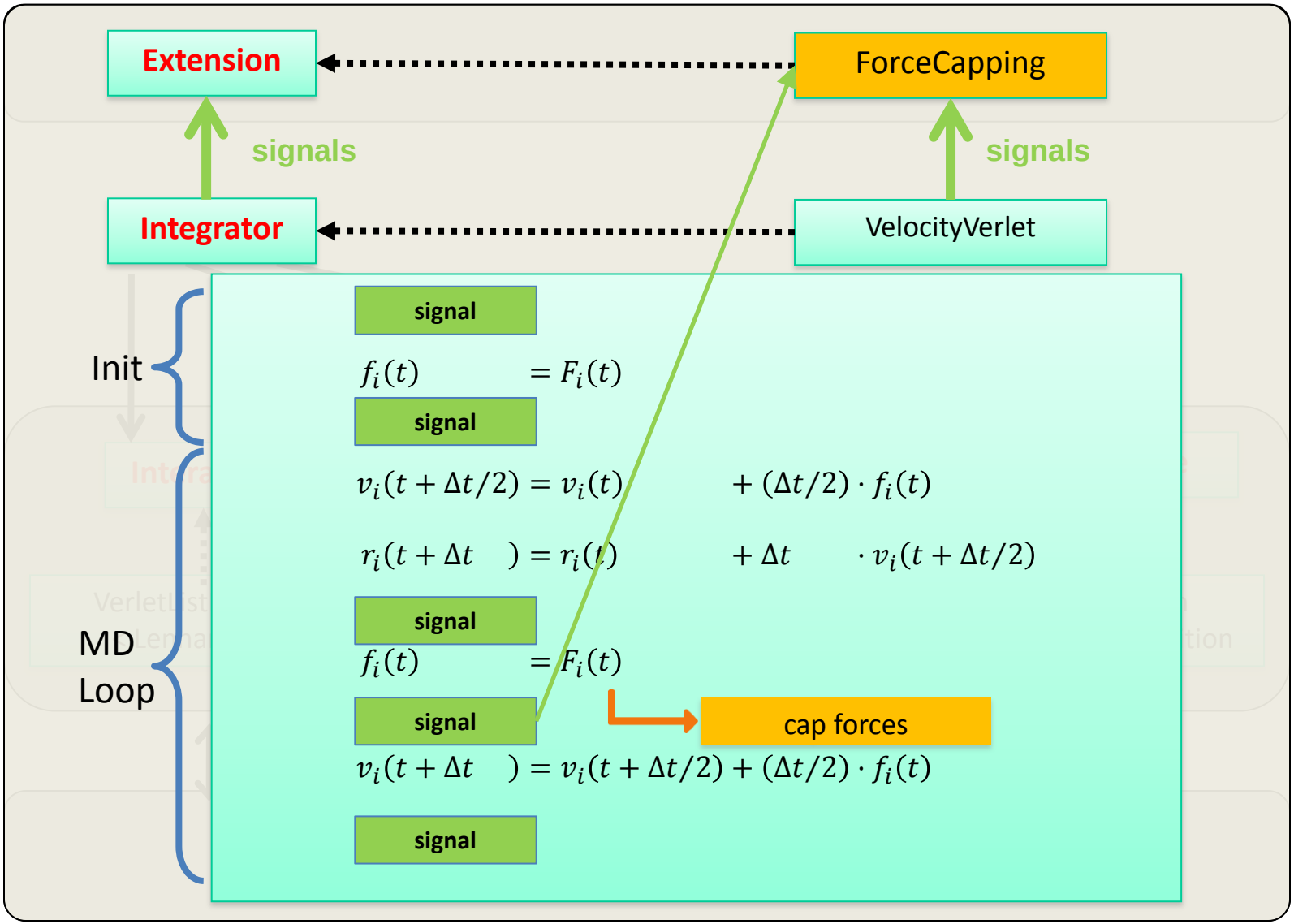
Extensions


Extension
LangevinThermostat
signals
signals
Integrator
VelocityVerlet
Init
MD Loop
signal
$f_i(t) = F_i(t)$
signal
$+ \boldsymbol{L}(\boldsymbol{T}, \boldsymbol{\gamma}, \boldsymbol{v_i}(\boldsymbol{t}), \boldsymbol{\theta})$
$v_i(t + \Delta t/2) = v_i(t) + (\Delta t/2) \cdot f_i(t)$
$r_i(t + \Delta t) = r_i(t) + \Delta t \cdot v_i(t + \Delta t/2)$
signal
$f_i(t) = F_i(t)$
signal
$+ \boldsymbol{L'}(\boldsymbol{T}, \boldsymbol{\gamma}, \boldsymbol{v_i}(\boldsymbol{t} + \boldsymbol{\Delta t/2}), \boldsymbol{\theta})$
$v_i(t + \Delta t) = v_i(t + \Delta t/2) + (\Delta t/2) \cdot f_i(t)$
signal

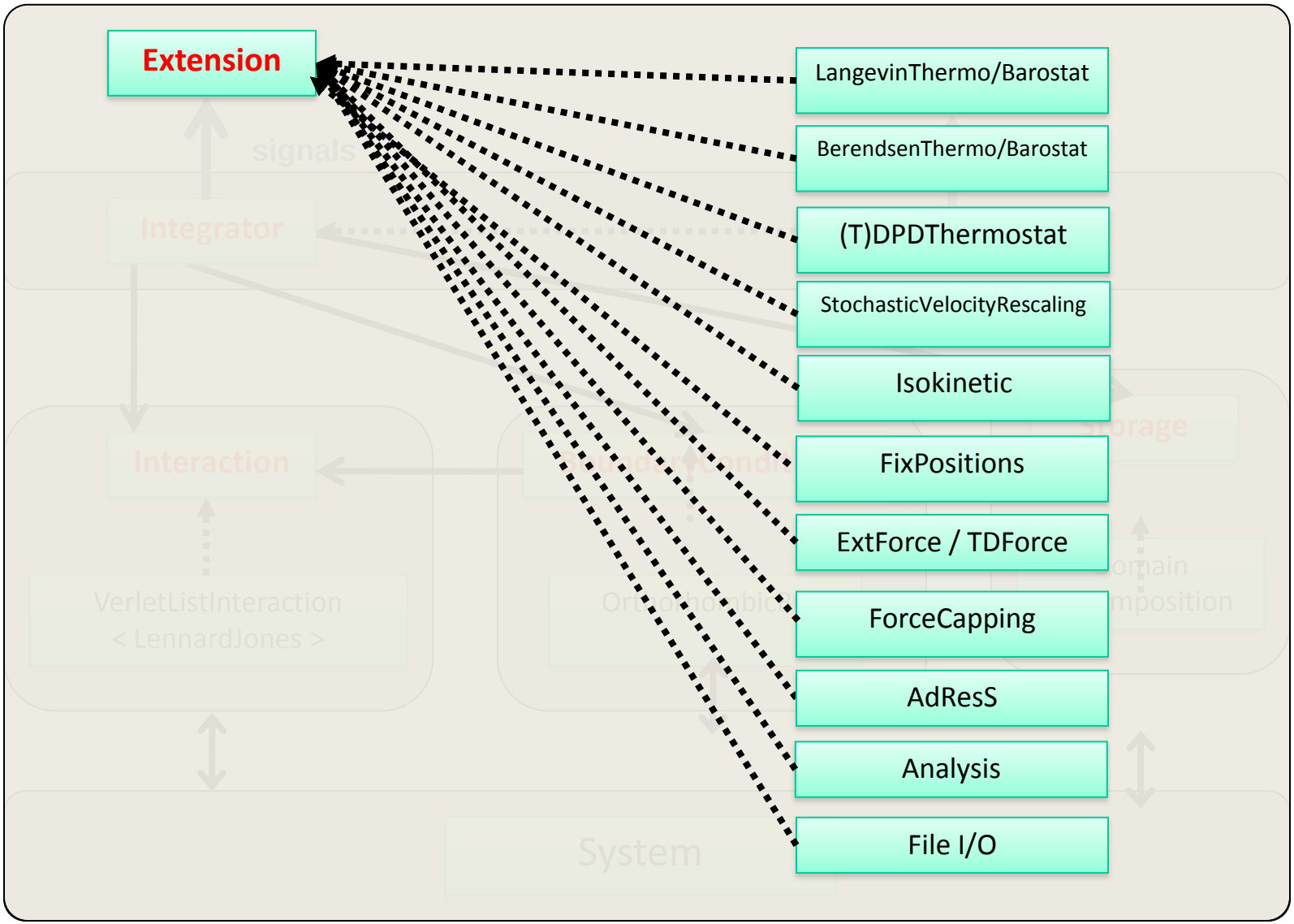
Extensions


Extension
FixPositions
signals
signals
Integrator
VelocityVerlet
Init
MD Loop
signal
$f_i(t) = F_i(t)$
signal
save positions
$v_i(t + \Delta t/2) = v_i(t) + (\Delta t/2) \cdot f_i(t)$
$r_i(t + \Delta t) = r_i(t) + \Delta t \cdot v_i(t + \Delta t/2)$
signal
restore positions
$f_i(t) = F_i(t)$
signal
$v_i(t + \Delta t) = v_i(t + \Delta t/2) + (\Delta t/2) \cdot f_i(t)$
signal

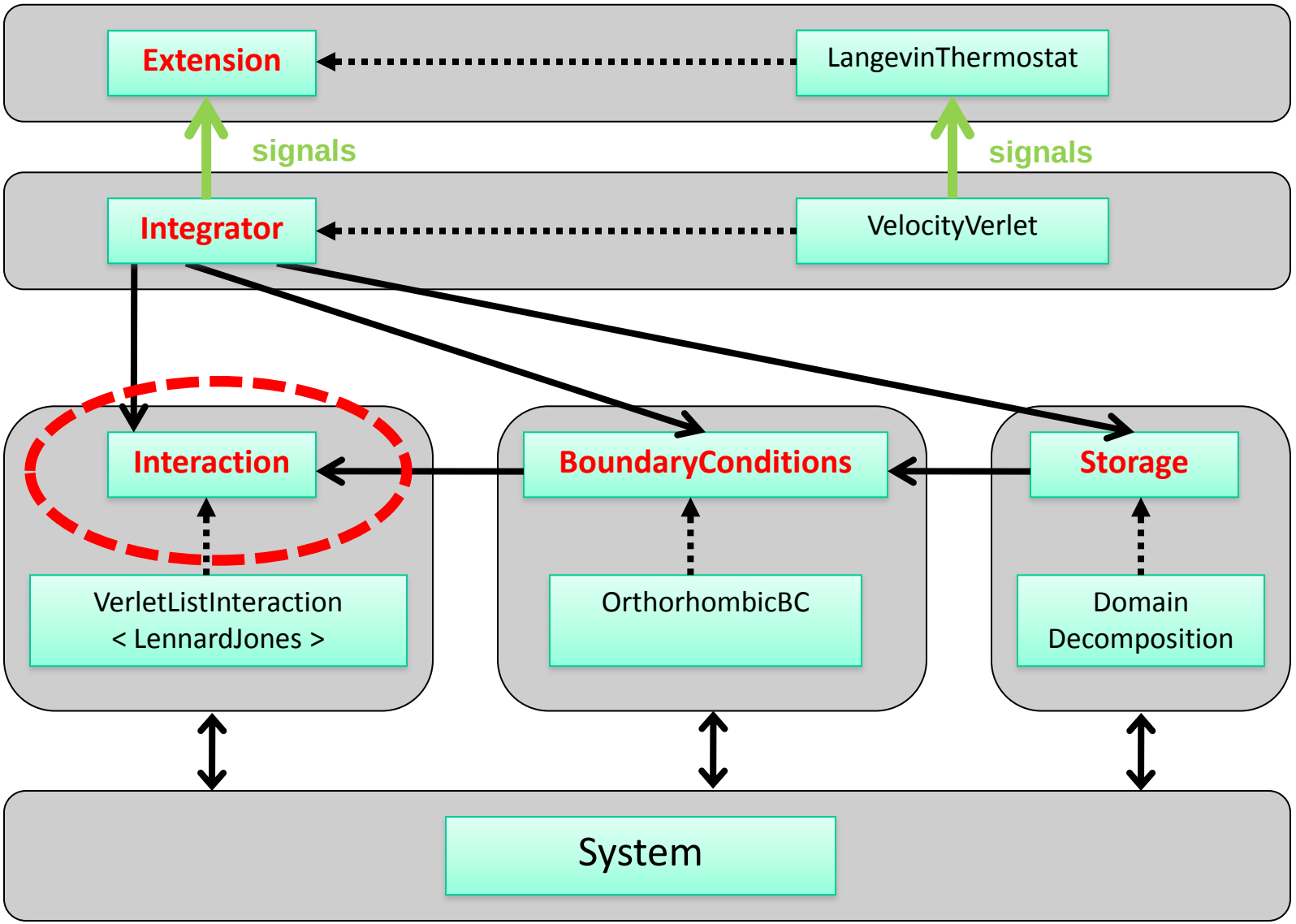
Extensions


Extension
ForceCapping
signals
signals
Integrator
VelocityVerlet
Init
MD Loop
signal
$f_i(t) = F_i(t)$
signal
$v_i(t + \Delta t/2) = v_i(t) + (\Delta t/2) \cdot f_i(t)$
$r_i(t + \Delta t) = r_i(t) + \Delta t \cdot v_i(t + \Delta t/2)$
signal
$f_i(t) = F_i(t)$
signal
cap forces
$v_i(t + \Delta t) = v_i(t + \Delta t/2) + (\Delta t/2) \cdot f_i(t)$
signal

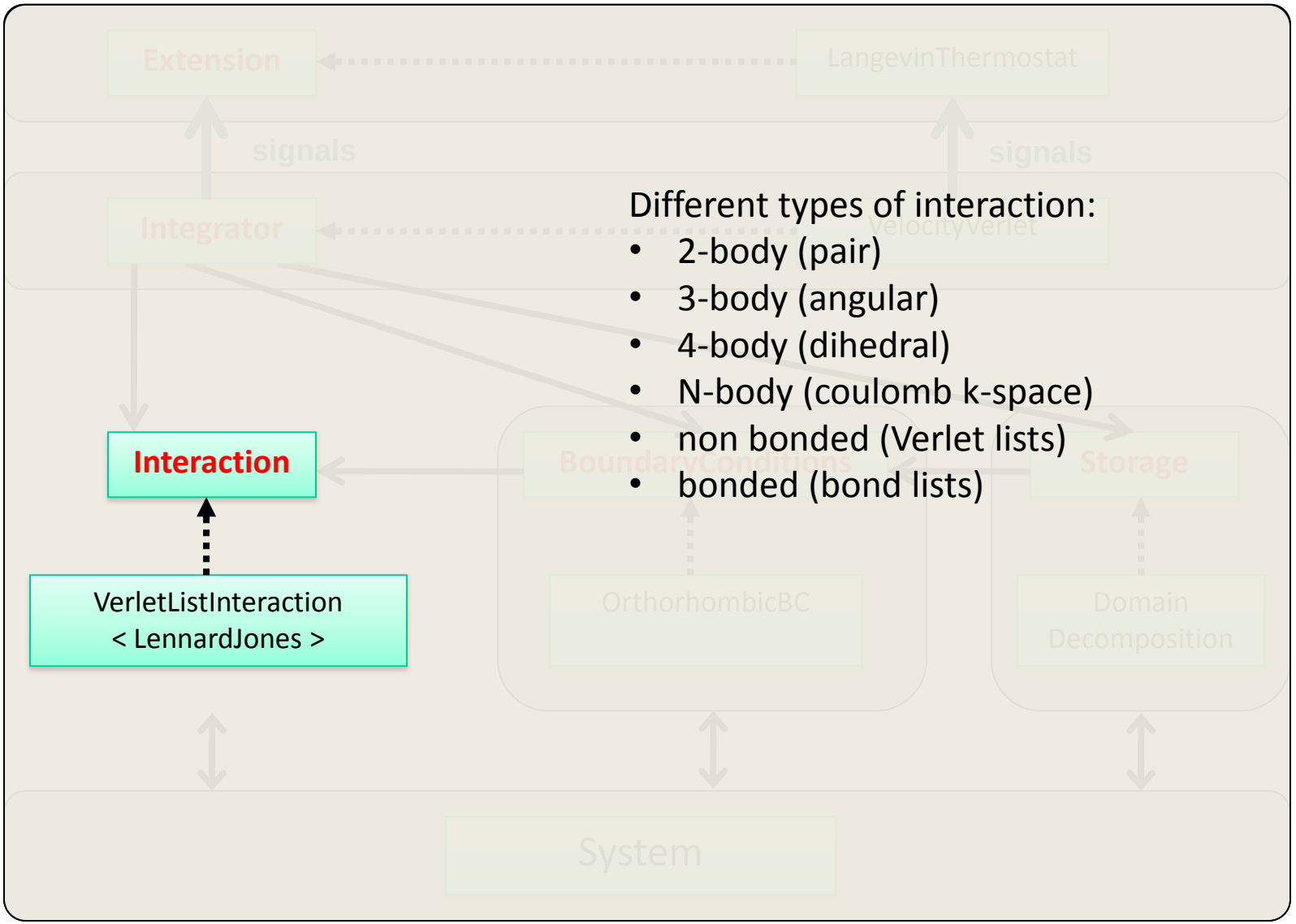
Extensions


Extension
LangevinThermo/Barostat
BerendsenThermo/Barostat
(T)DPDThermostat
StochasticVelocityRescaling
Isokinetic
FixPositions
ExtForce / TDForce
ForceCapping
AdResS
Analysis
File I/O
signals
Integrator
Interaction
VerletListInteraction
< LennardJones >
System

Abstract classes



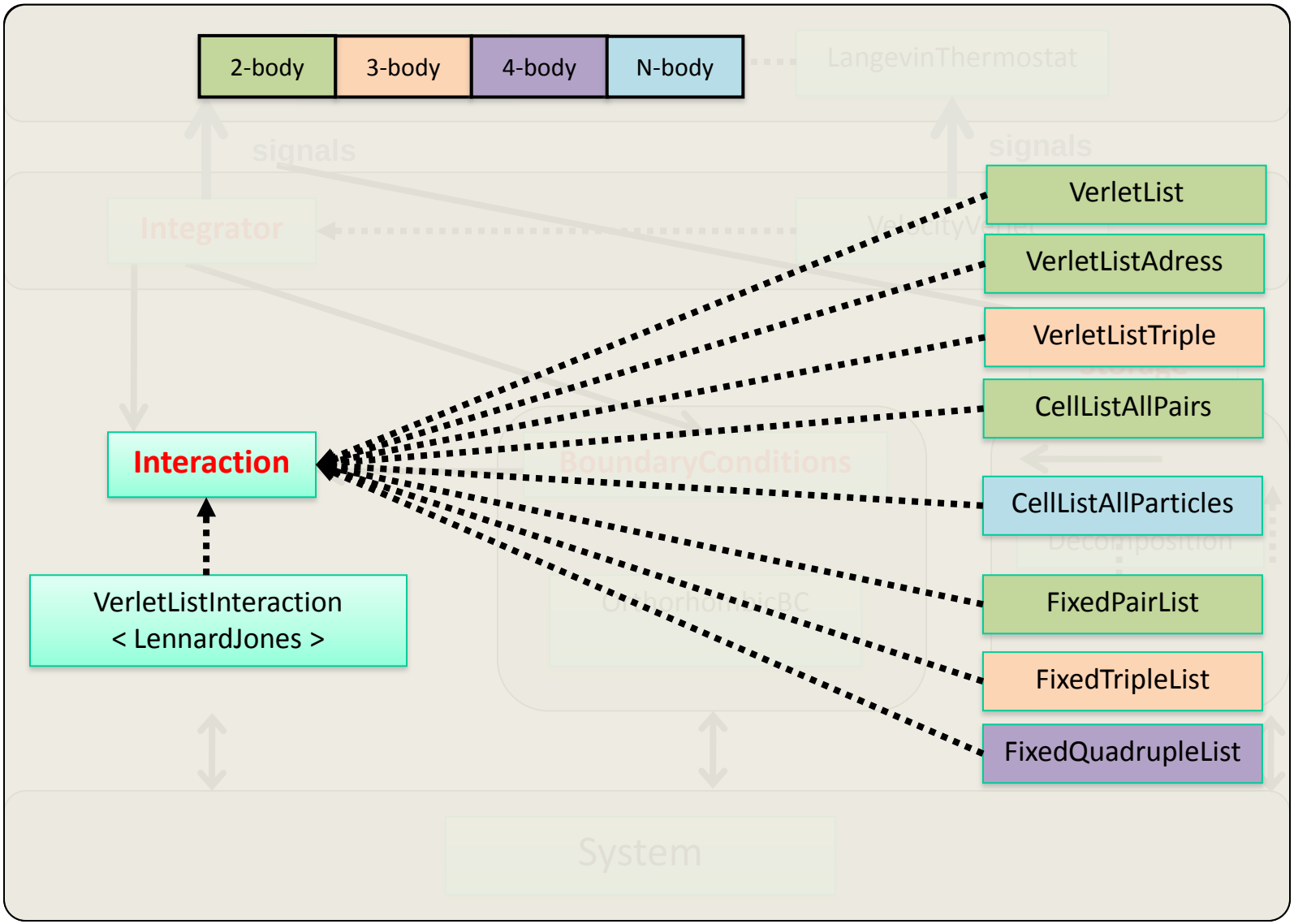
Interaction types



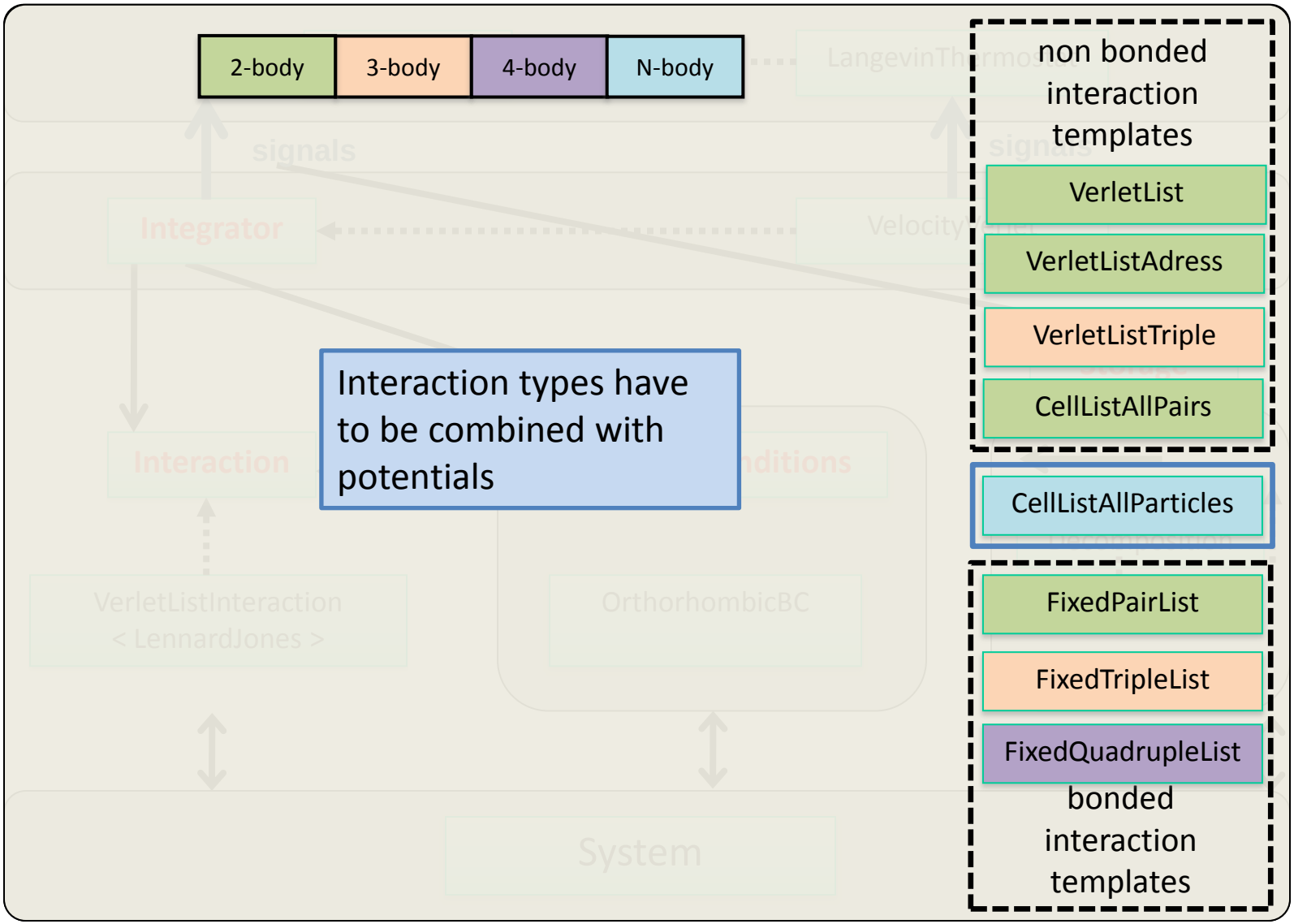
Different types of interaction:

- 2-body (pair)
- 3-body (angular)
- 4-body (dihedral)
- N-body (coulomb k-space)
- non bonded (Verlet lists)
- bonded (bond lists)

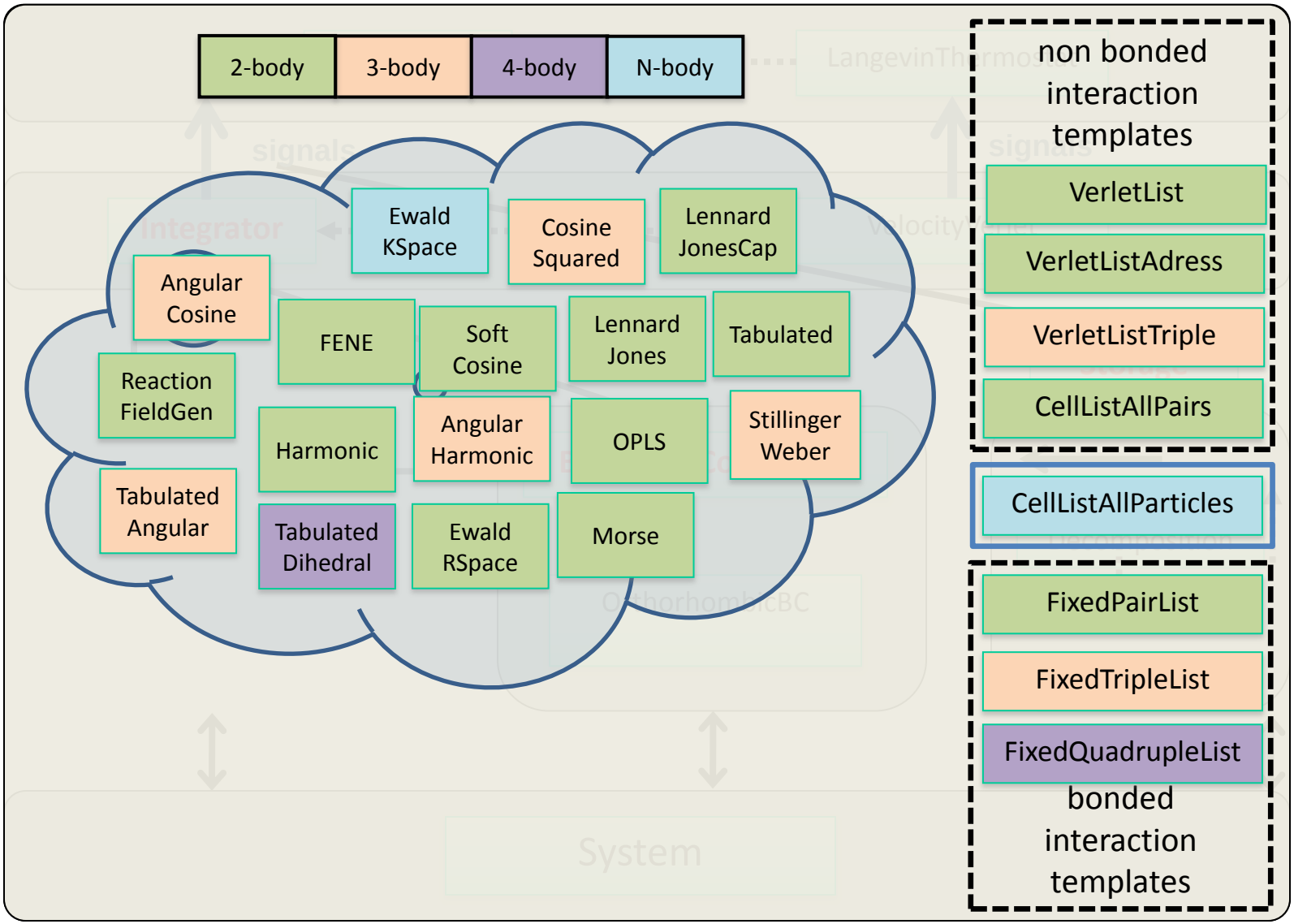
Interaction types



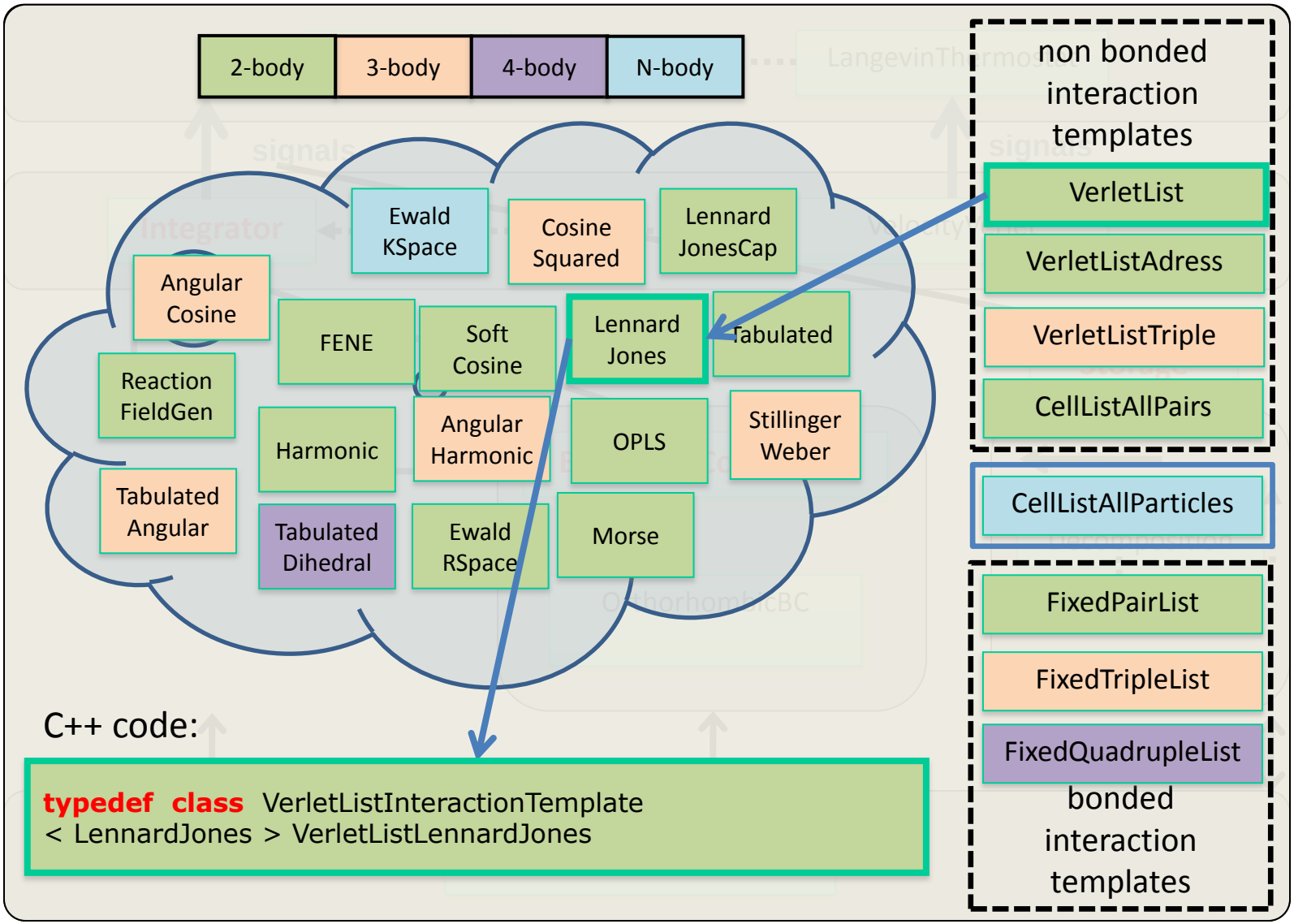
Interaction types



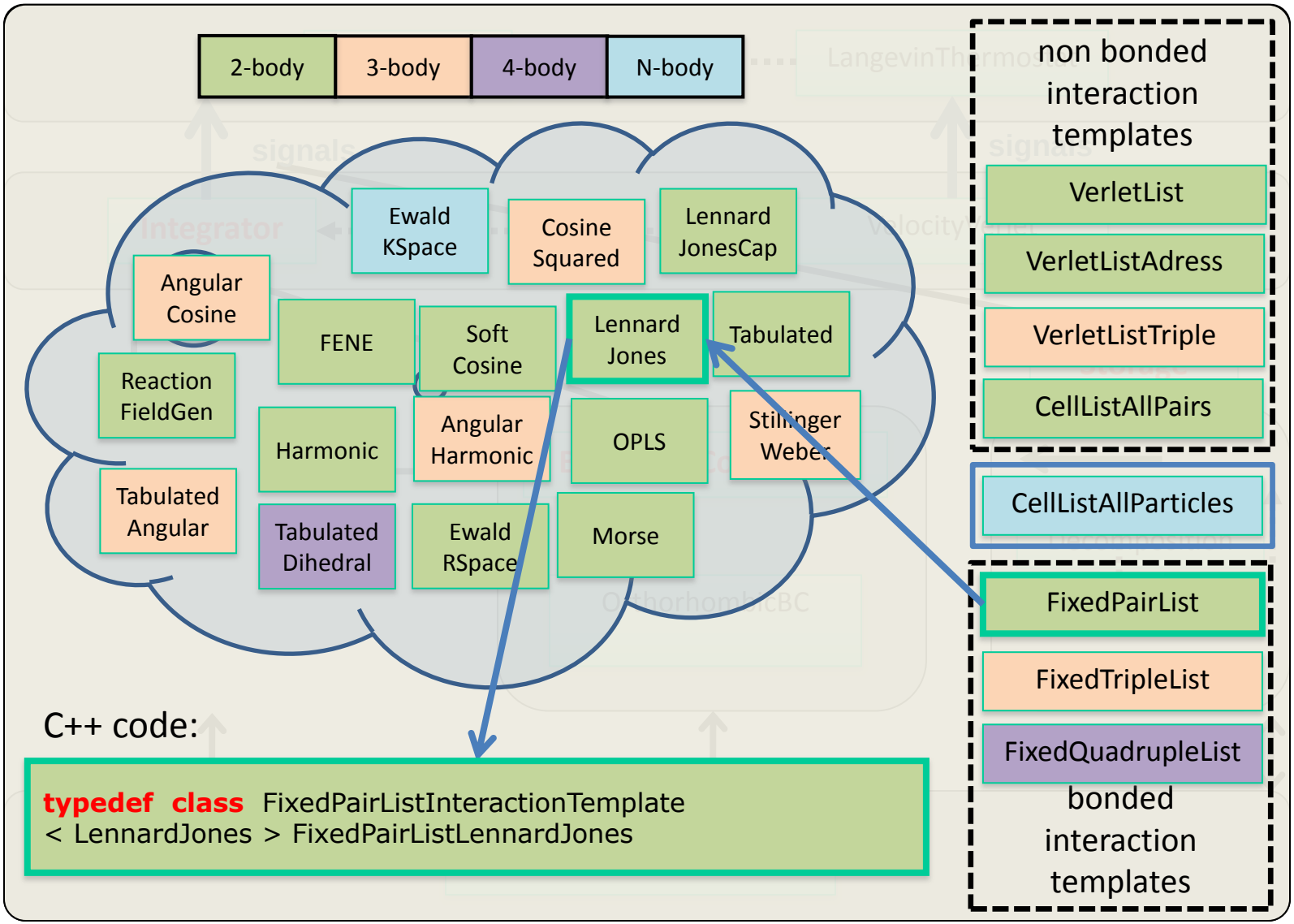
Potentials


2-body
3-body
4-body
N-body
Ewald KSpace
Cosine Squared
Lennard JonesCap
Angular Cosine
FENE
Soft Cosine
Lennard Jones
Tabulated
Reaction FieldGen
Harmonic
Angular Harmonic
OPLS
Stillinger Weber
Tabulated Angular
Tabulated Dihedral
Ewald RSpace
Morse
non bonded interaction templates
VerletList
VerletListAdress
VerletListTriple
CellListAllPairs
CellListAllParticles
FixedPairList
FixedTripleList
FixedQuadrupleList
bonded interaction templates

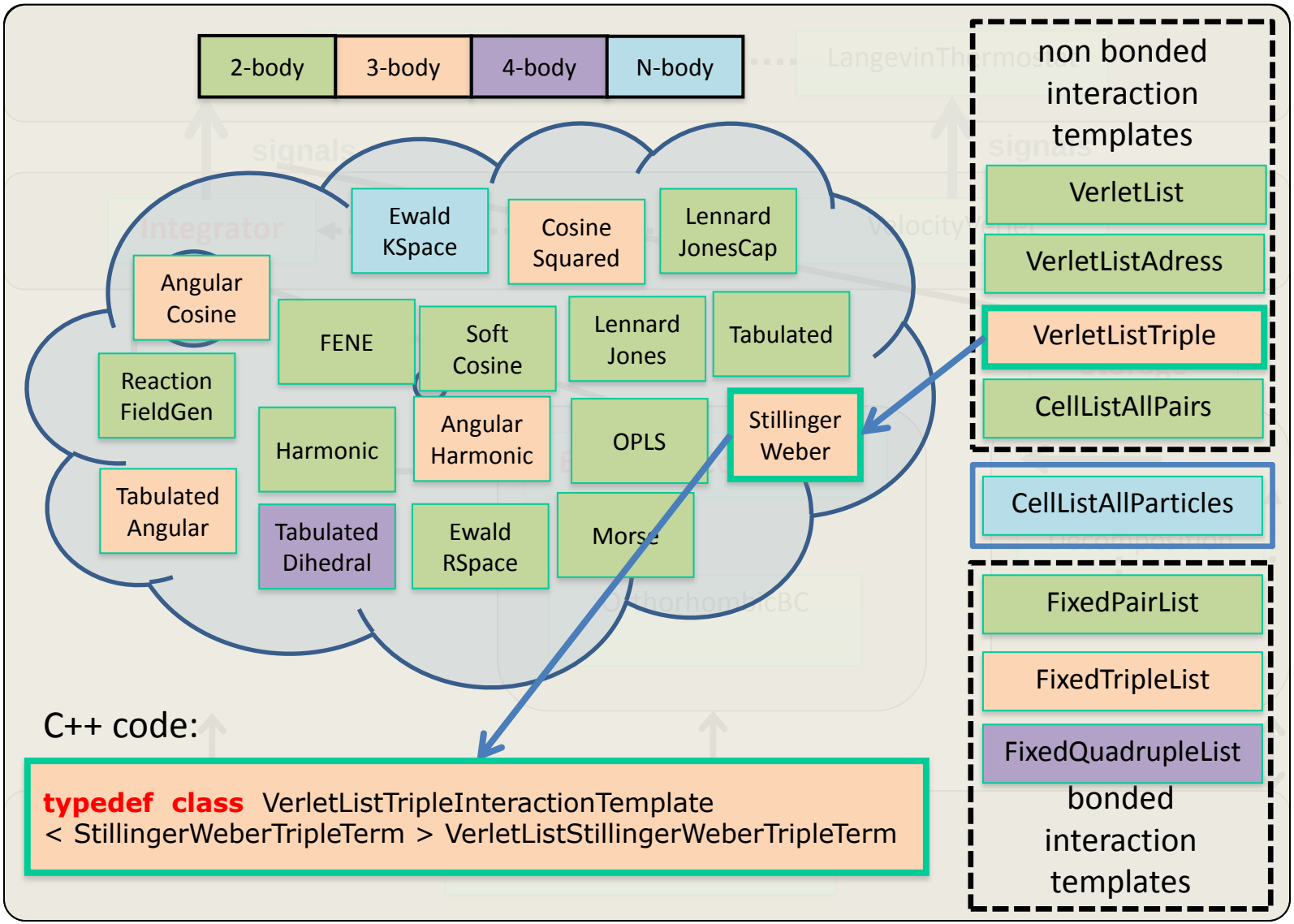
Potentials


2-body
3-body
4-body
N-body
Ewald KSpace
Cosine Squared
Lennard JonesCap
Angular Cosine
FENE
Soft Cosine
Lennard Jones
Tabulated
Reaction FieldGen
Harmonic
Angular Harmonic
OPLS
Stillinger Weber
Tabulated Angular
Tabulated Dihedral
Ewald RSpace
Morse
C++ code:
typedef class VerletListInteractionTemplate
< LennardJones > VerletListLennardJones
non bonded interaction templates
VerletList
VerletListAdress
VerletListTriple
CellListAllPairs
CellListAllParticles
FixedPairList
FixedTripleList
FixedQuadrupleList
bonded interaction templates

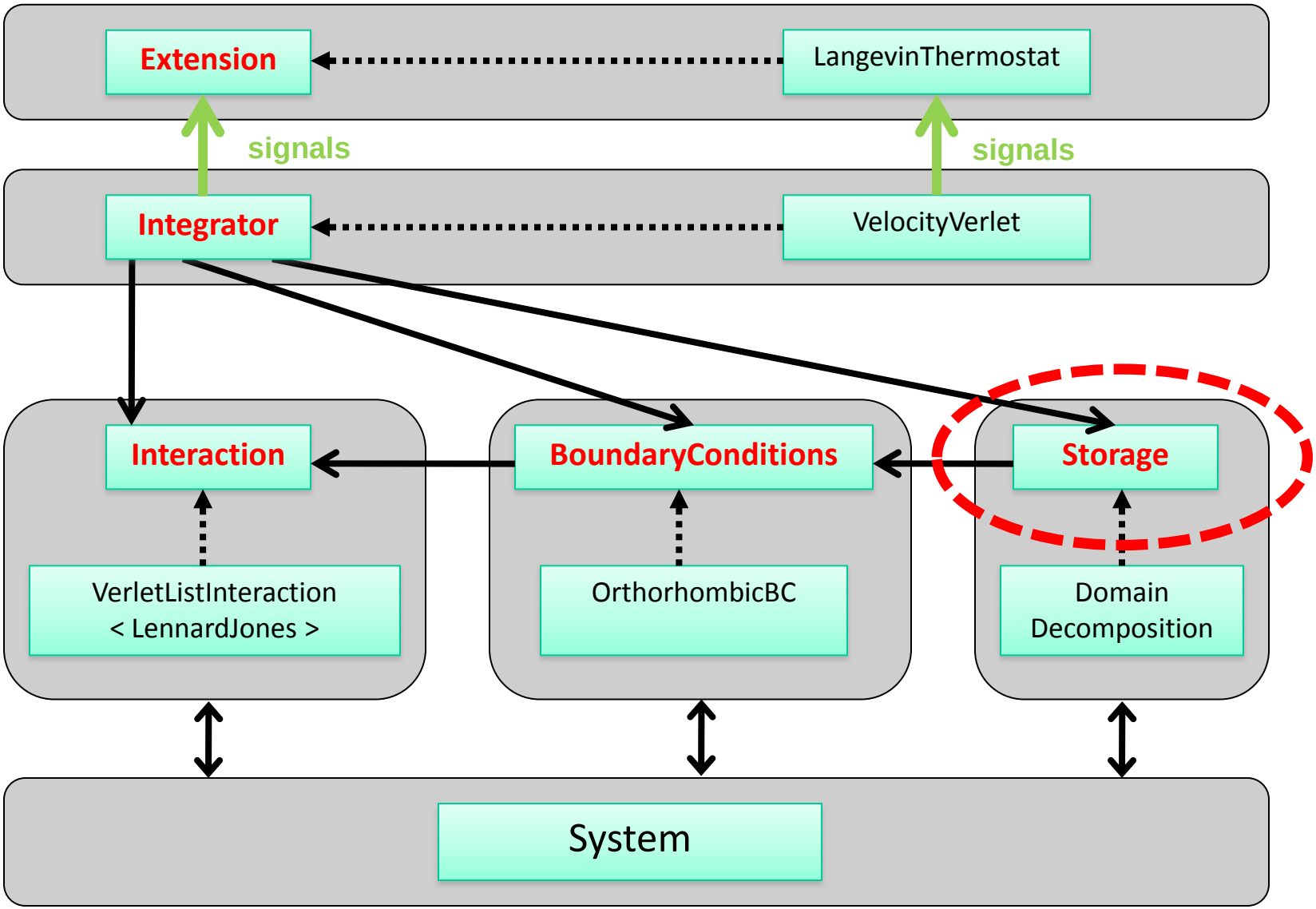
Potentials


2-body
3-body
4-body
N-body
Ewald KSpace
Cosine Squared
Lennard JonesCap
Angular Cosine
FENE
Soft Cosine
Lennard Jones
Tabulated
Reaction FieldGen
Harmonic
Angular Harmonic
OPLS
Stillinger Weber
Tabulated Angular
Tabulated Dihedral
Ewald RSpace
Morse
C++ code:
typedef class FixedPairListInteractionTemplate
< LennardJones > FixedPairListLennardJones
non bonded interaction templates
VerletList
VerletListAdress
VerletListTriple
CellListAllPairs
CellListAllParticles
FixedPairList
FixedTripleList
FixedQuadrupleList
bonded interaction templates

Potentials


2-body
3-body
4-body
N-body
Ewald KSpace
Cosine Squared
Lennard JonesCap
Angular Cosine
FENE
Soft Cosine
Lennard Jones
Tabulated
Reaction FieldGen
Harmonic
Angular Harmonic
OPLS
Stillinger Weber
Tabulated Angular
Tabulated Dihedral
Ewald RSpace
Morse
non bonded interaction templates
VerletList
VerletListAdress
VerletListTriple
CellListAllPairs
CellListAllParticles
FixedPairList
FixedTripleList
FixedQuadrupleList
bonded interaction templates
C++ code:
typedef class VerletListTripleInteractionTemplate
< StillingerWeberTripleTerm > VerletListStillingerWeberTripleTerm

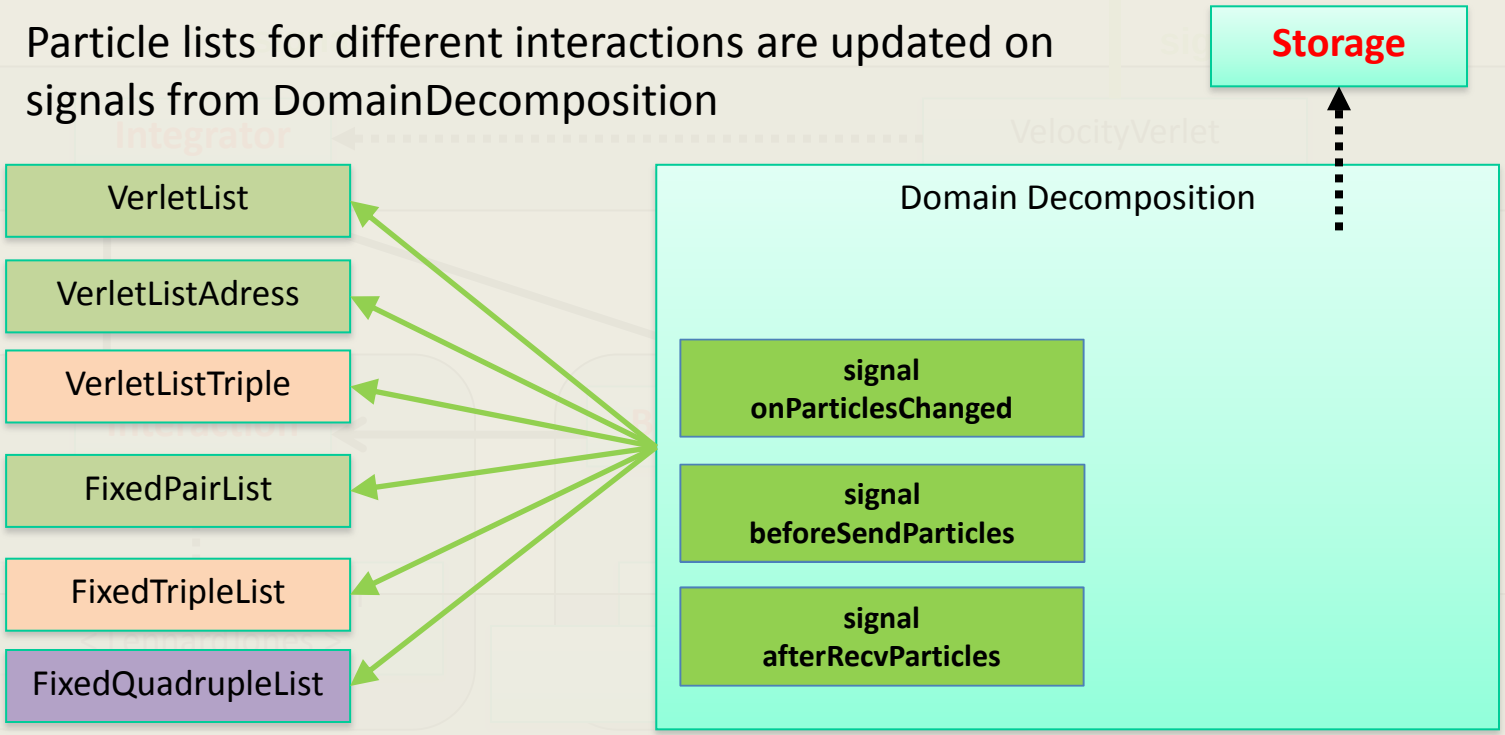
Abstract classes



Storage

- Storage takes care of parallelisation
- Distributes particles to the MPI tasks
- Updates topology information on all MPI tasks
- Sends and receives communication buffers

Particle lists for different interactions are updated on signals from DomainDecomposition



ESPResSo++ Team:

Current developers:

- Torsten Stuehn (MPIP)
- Vitalii Starchenko (MPIP)
- Sebastian Fritsch (MPIP)
- Konstantin Koschke (MPIP)
- Livia Moreira (MPIP)
- Raffaello Potestio (MPIP)
- Karsten Kreis (MPIP)
- Stas Bevc (NIC, Slovenia)

Former developers:

- Thomas Brandes (SCAI)
- Dirk Reith (SCAI)
- Axel Arnold (ICP)
- Olaf Lenz (ICP)
- Jonathan Halverson (BNL, USA)
- Victor Ruehle (Cambridge, UK)
- Christoph Junghans (LANL, USA)

Thank you for your attention !