SYMPLER: SYMbolic ParticLE simulatoR with grid-computing interface

David Kauzlarić[a,b,*], Marek Dynowski[c,d], Lars Pastewka[b,e], Andreas Greiner[b], Jan G. Korvink[b,a]

[a] *School of Soft Matter Research, Freiburg Institute for Advanced Studies, University of Freiburg, Albertstr. 19, 79104 Freiburg, Germany*
[b] *Laboratory for Simulation, Department of Microsystems Engineering (IMTEK), University of Freiburg, Georges-Köhler-Allee 103, 79110 Freiburg, Germany*
[c] *Rechenzentrum, University of Freiburg, Hermann-Herder-Str. 10, 79104 Freiburg, Germany*
[d] *ZDV, University of Tübingen, Wächterstr. 76, 72074 Tübingen, Germany*
[e] *Fraunhofer Institute for Mechanics of Materials, Wöhlerstr. 11, 79108 Freiburg, Germany*

Abstract

We present the main design concepts of the object-oriented particle dynamics code SYMPLER. With this freely available software, simulations can be performed ranging from microscopic classical molecular dynamics up to the Lagrangian particle-based discretisation of macroscopic continuum mechanics equations. We show how the runtime definition of arbitrary degrees of freedom and of arbitrary equations of motion allows for modular and symbolic computation with high flexibility. Arbitrary symbolic expressions for inter-particle forces can be defined as well as fluxes of arbitrarily many additional scalar, vectorial or tensorial degrees of freedom. The integration in a high performance grid computing environment makes huge geographically distributed computational ressources accessible to the software by an easy-to-use interface.

Keywords: Molecular dynamics; Dissipative particle dynamics; Smoothed particle hydrodynamics; Symbolic computation; Runtime compilation; Object oriented programming

Program Summary

Manuscript Title: SYMPLER: SYMbolic ParticLE simulatoR with grid-computing interface
Authors: David Kauzlarić, Marek Dynowski, Lars Pastewka, Andreas Greiner, Jan G. Korvink; all authors of the SYMPLER code are listed in the `AUTHORS` file distributed with the code.
Program Title: SYMPLER: SYMbolic ParticLE simulatoR, version 2013.10.1.0
Journal Reference:
Catalogue identifier:
Licensing provisions: GNU General Public License version 3
Programming language: C++
Computer: Any system operatable with Linux.
Operating system: Linux, MacOS
RAM: tens of MB to several GB, depending on problem.
Number of processors used: usually 1, experimental OpenMP parallelisation for usually up to 8 cores, grid-version can use hundreds of cores.
Keywords: Molecular dynamics, Dissipative particle dynamics, Smoothed particle hydrodynamics, Symbolic computation, Runtime compilation, Object oriented programming
Classification: 7.7, 12, 16.1, 16.3, 16.13, 23
External routines/libraries: GSL, libxml2; *optional*: libsdl, OpenMP, libjama, libtnt, libsuperlu
Subprograms used: GNU autotools and GNU libtool for compilation.
Nature of problem: A unified flexible and modular simulation tool allowing for the investigation of structural, thermodynamic, and dynamical properties of fluids and solids from microscopic over mesoscopic up to macroscopic time and length scales with suitable particle based simulation methods such as molecular dynamics, dissipative particle dynamics or smoothed particle hydrodynamics. The user should be enabled to freely define own physical models without the need for recoding or code extensions.
Solution method: SYMPLER provides flexibility to the user by i) a modular object oriented structure that is passed to the user level and allows to easily switch among different integration algorithms, particle interaction forces, boundary conditions, etc. ii) an arbitrary number of particle-species for the simulation of complex multi-component systems iii) an arbitrary number of additional user-defined degrees of freedom per particle-species iv) symbolic definition of runtime-compiled mathematical expressions for particle interactions v) import of CAD-geometries vi) a flexible choice of available computational cores through a grid-computing interface, amongst others.
Restrictions: Classical deterministic and stochastic Newtonian dynamics.
Unusual features: Symbolic runtime-compiled user-defined expressions.
Additional comments: The current version and all future updates to the code are also found at `http://sympler.org`.
Running time: Some benchmarks are given in the paper. The running time is problem dependent and ranges from seconds to days.

1. Introduction

Particle based simulation based on Newton's equations of motion is a highly versatile modelling approach that can be used in a variety of fields such as biophysics [1, 2], solid-state physics [3], fluid-dynamics [4, 5], or astrophysics [6]. Based on the time and length scales of a given problem, the applied methods are molecular dynamics (MD) [7], dissipative particle dynamics (DPD) [8, 9] or smoothed particle hydrodynam-

*Corresponding author.
E-mail address: david.kauzlaric@frias.uni-freiburg.de

ics (SPH) [10]. A considerable number of particle dynamics software is freely available [1, 2, 11–14], with different foci and partially complementing each other. Most particle-dynamics codes offer a predefined set of functionalities to the user, in terms of time-integration schemes, types of particle-interactions or physical models with corresponding equations of motion. This hard-coded functionality, depending on the size of the code, and the already invested amount of person years (see, e.g., the GROMACS-web-page for an impressive number [12]), is more or less limited. We can always think of a new application which requires recoding, the implementation of an additional package, or moving from one software to another. We may therefore ask the question whether it is possible to introduce more flexibility into particle dynamics codes, avoiding recoding as much as possible. The particle dynamics software SYMPLER [15] is the proof that it is possible to create a versatile yet small multi-purpose particle dynamics simulation tool for multiple length and time scales without sacrificing computational efficiency. Typically hard-coded monolithic algorithms can be expressed in terms of expressions compiled at runtime and replaceable modules. No other particle dynamics tool currently provides that much freedom per computational module. This strategy reflects itself in the following main features of SYMPLER: i) An arbitrary number of degrees of freedom per particle of either scalar, vectorial or tensorial type (cf. section 4). ii) An arbitrary number of species, i.e., groups of particles with the same properties, for example the same combination of degrees of freedom or the same inter-particle interactions. iii) User-defined symbolic expressions that are runtime-compiled (cf. sections 5, 6). iv) With the SYMPLER-portlet, SYMPLER can be run on a grid for high performance computing (cf. section 7). v) CAD-geometries in the STL-format ("Surface Tesselation Language", also: "Stereolithography") [16] can be imported. As shown in section 8, the latter is especially useful for the simulation of injection moulding or casting into complex geometries, but also geometries of the aorta can be imported for example.

The focus of this paper is therefore the presentation of special design-features that allow flexibility in and rapid-prototyping of particle-simulations, similarly to a platform for computational tasks such as *Mathematica* [17], *Maple* [18] or *SciPy* [19]. These features are indeed presented for "yet-another" particle dynamics software, but the concepts are generally applicable. Especially the concept of symbolic expressions transforms SYMPLER from a particle-dynamics tool for special well-defined purposes into a general platform where the flexibility arises from the freedom in the user-input that can be provided. This also makes SYMPLER very compact in size and easy to install, only depending on standard freely available packages.

Before going into the details of the software design, we briefly summarise the relevant premises of particle methods in section 2. Then, section 3 gives a general overview of the current state of the art of SYMPLER. Due to the code-design SYMPLER can be applied on all scales. The user provides the mathematical expressions to the software which determine whether MD, DPD or SPH is performed. Section 8 reviews a few example problems from very different fields such as process-engineering for injection moulding with SPH or molecular physics with MD and DPD, where SYMPLER has already been successfully applied.

For the description of the object oriented code-design, diagrams based on the Unified Modelling Language (UML) are used [20]. A summary of the few concepts of the UML used in this paper may be found in the appendix. Reference to special elements of a given UML-diagram or of an excerpt of an input file is made in the text by adapting the font of the mentioned element.

2. Summary of particle dynamics methods

Any particle dynamics method requires the solution of the equation of motion

$$\dot{\mathbf{r}}_i = \frac{\mathbf{p}_i}{m_i} = \mathbf{v}_i \tag{1}$$

The index i denotes a discrete quantity located on and moving with a particle i, here, particle position $\mathbf{r}_i$, momentum $\mathbf{p}_i$, mass m_i, and velocity $\mathbf{v}_i$. The particle velocities or momenta are obtained from Newton's equation of motion, which for the momentum-equation of Dissipative particle dynamics (DPD) can be written in the form [9]

$$\dot{\mathbf{p}}_i = \sum_{j \neq i} \left(\mathbf{F}^C_{ij} + \mathbf{F}^D_{ij} + \mathbf{F}^R_{ij} \right) \tag{2}$$

where we write the RHS in a generic way as the sum of conservative, dissipative, and stochastic forces $\mathbf{F}^C_{ij}$, $\mathbf{F}^D_{ij}$, $\mathbf{F}^R_{ij}$. In Molecular dynamics (MD) [7] the RHS of equation (2) reduces to conservative forces only. Smoothed particle hydrodynamics (SPH) [10, 21] is a particle based discretisation scheme for continuum mechanical equations. While the design of SYMPLER allows to solve arbitrary equations of motion not limited to SPH, we here present the equations of SPH as an illustrative example. A common way to discretise the momentum equation

$$\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = \frac{1}{\rho} \nabla \cdot \boldsymbol{\sigma} \tag{3}$$

with stress tensor $\boldsymbol{\sigma}$ with SPH is [22]

$$\dot{\mathbf{v}}_i = \sum_j \frac{m_j}{\rho_i \rho_j} \left(\boldsymbol{\sigma}_i - \boldsymbol{\sigma}_j \right) \cdot \nabla W_{ij} \tag{4}$$

where ρ_i is the particle's local mass-density which is often computed by

$$\rho_i = \sum_j m_j W_{ij} \tag{5}$$

The calculation of the stress often requires the rate of strain tensor

$$\dot{\boldsymbol{\gamma}} = \left(\nabla \mathbf{v} + (\nabla \mathbf{v})^T \right) \tag{6}$$

which can be discretised as

$$\dot{\boldsymbol{\gamma}}_i = \sum_j \frac{m_j}{\rho_j} w_{ij} \left(\mathbf{r}_{ij} \mathbf{v}^T_{ij} + \mathbf{v}_{ij} \mathbf{r}^T_{ij} \right). \tag{7}$$

The kernel $W_{ij} \equiv W\left(r_{ij}\right)$ is an interpolation function of width r_c, i.e., it has a finite support and vanishes for $r_{ij} > r_c$ with $r_{ij} \equiv |\mathbf{r}_{ij}|$ and $\mathbf{r}_{ij} \equiv \mathbf{r}_i - \mathbf{r}_j$. For a correct interpolation, the kernel's volume integral is normalised to 1. We have also introduced the expression $w_{ij} \equiv w\left(r_{ij}\right)$ which is defined by

$$\nabla W_{ij} = -\mathbf{r}_{ij} w_{ij} \tag{8}$$

Alternatively to equation (5), the density can be computed by time integration by discretising the continuity equation for the mass density. Then one obtains

$$\dot{\rho}_i = \sum_j m_j \mathbf{v}_{ij} \cdot \nabla W_{ij} = -\sum_j m_j \mathbf{v}_{ij} \cdot \mathbf{r}_{ij} w_{ij} \tag{9}$$

where $\mathbf{v}_{ij} \equiv \mathbf{v}_i - \mathbf{v}_j$. Besides the usual degrees of freedom (DOFs) $\mathbf{r}_i$ and $\mathbf{v}_i$ (or alternatively $\mathbf{p}_i$), additional DOFs may be assigned to the particles. The discretised version of Phillips' model [23] for shear-induced migration of a solids fraction ϕ in a concentrated suspension is for example [24]

$$\begin{aligned} \dot{V}_{\phi,i} = &- D_c a^2 \sum_j \frac{w_{ij}}{\rho_i \rho_j} (\phi_i + \phi_j)(\phi_i \dot{\gamma}_i - \phi_j \dot{\gamma}_j) \\ &- D_\eta a^2 \sum_j \frac{w_{ij}}{\rho_i \rho_j} \left(\dot{\gamma}_i \phi_i^2 \left(\frac{\mathrm{d}\eta}{\eta \mathrm{d}\phi} \right)_i + \dot{\gamma}_j \phi_j^2 \left(\frac{\mathrm{d}\eta}{\eta \mathrm{d}\phi} \right)_j \right) \\ &\times (\phi_i - \phi_j) \end{aligned} \tag{10}$$

where the second invariant of the rate of strain tensor, i.e.,

$$\dot{\gamma} = \sqrt{\dot{\boldsymbol{\gamma}} : \dot{\boldsymbol{\gamma}}/2} \tag{11}$$

was used, and where the volume $V_{\phi,i} \equiv \phi_i/\rho_i$ occupied by the solid fraction was taken as degree of freedom instead of ϕ_i, due to better conservation properties of the resulting discrete equations [24].

3. General design

3.1. Design philosophy

The goal during the design of SYMPLER was to simultaneously allow for flexibility and computational efficiency. This means that all freedom to customise simulation parameters should lie in the user's hands, without significantly sacrificing computational speed. Having this in mind, the software has been designed top-down using an object-oriented approach. The simulation control file maps to the same object structure the code has. Thus, during the initialisation of the simulation, the XML [25] input file is read and the appropriate objects are constructed on the fly with user-defined parameters. The main elements that the user can define in the input file follow directly from the equations of section 2 and are, e.g., Forces, Integrators, and Boundaries. The XML input-file is fully transparent. On this level of control the philosophy is that the user can and should understand all the details of the simulation setup. On the other hand, the choice of an XML-interface also made it straightforward to construct a graphical user interface [26] together with template inputs allowing for easier use by non-experts. In this paper we only present direct XML-based input.

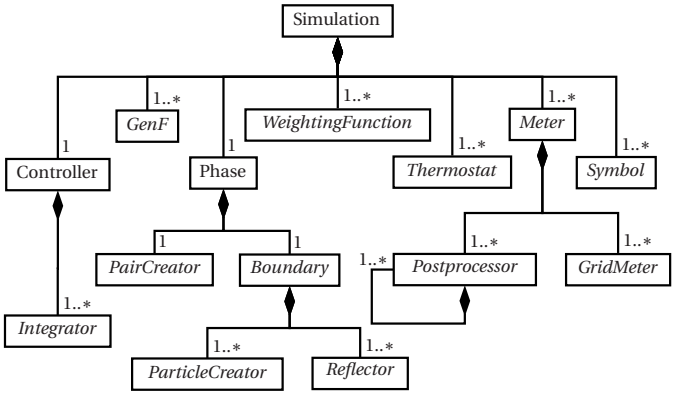

Figure 1: Class diagram of SYMPLER showing the associations between the user-visible classes. The usage of the corresponding modules in an XML user input is shown, e.g., in figure 2. Note that all *italic* class names indicate abstract classes. Therefore, in XML-input those are replaced by instances of derived classes. The `Thermostats` belong in fact to a more general hierarchy of `Callables` comprising various modules for manipulation of particle data [15].

3.2. General structure and usage

An overview over the object structure omitting helper classes of minor importance is given in figure 1. Just the classes the user is interacting with are shown. Figs. 2 and 7 show the corresponding structure of the XML input file with selected modules and attributes which we refer to later. All *italic* class names in figure 1 indicate abstract classes. Therefore, in XML input those are replaced by instances of derived classes which are called modules in the following. The design of the class structure was inspired by Kofke's molecular simulation API [27].

Going top-down through the class diagram (cf. figure 1), it is the class `Simulation` which ties the simulation process together. It reads the input file and hands over to the `Controller` object once this is done. `Simulation` keeps a list of `GenF` objects representing the actual physical forces on or fluxes to the particles, `Thermostats` assuring relaxation to a thermal equilibrium state by coupling the particles to a heat bath, and `Meters` which measure and output quantities of interest. The `Simulation` holds exactly one `Phase` which bookkeeps all the particles, and delegates the management of their neighbour lists to a `PairCreator`, and the description of the simulation domain to a `Boundary` object. Additionally, the `Simulation` contains two lists of objects for mathematical evaluations, namely `WeightingFunctions` used for the interpolation of particle quantities, and `Symbols` for all kind of additional mathematical operations required during the simulation process. The latter is discussed in detail in section 6.

The intermediate level of the `Controller` allows for the modularisation of different simulation protocols represented by different `Controllers`, e.g., different equilibration strategies, Monte Carlo based protocols, etc. The class `Controller` handles the bookkeeping of instances of `Integrators`. Each `Integrator` introduces the degree of freedom it is in charge of (cf. section 4.3) so that following modules can operate on these variables.

A `Boundary` may hold `Reflectors` representing one of many possible hard-wall boundary conditions, for exam-

ple a thermalising wall. `ParticleCreators` belong to the `Boundary` as well, since they must know about the domain where the particles should be created.

4. Arbitrary degrees of freedom

In the following, the term "degree of freedom" (DOF) follows the definition: A DOF χ_i is a parameter of a discrete particle i which evolves in time according to an equation of motion of the form

$$\dot{\chi}_i(t) = f_{\chi,i}(t), \tag{12}$$

where $f_{\chi,i}(t)$ may be called a generalised force or flux of the DOF χ_i. This concept is directly transfered to SYMPLER by the implementation of an abstract class `GenF`.

From the standard DPD equations (1) and (2) or from the SPH-discretisation (4) of the momentum equation, we see that usually required DOFs are the position $\mathbf{r}_i$ and the velocity $\mathbf{v}_i$ (or the momentum $\mathbf{p}_i$), where we have grouped the 6 DOFs in 3D into two convenient vectors. The density *is not* a DOF but a deduced quantity if we use a summation over neighbouring particles such as (5). But the density becomes an additional DOF if we replace the summation by the discretisation of the continuity equation (9). In equation (10) we have introduced a concentration ϕ_i as additional DOF. Other frequently encountered additional DOFs are for example the internal energy ϵ_i for non-isothermal simulations [28–32] or an orientation vector for magnetisations, polarisations or polymeric orientation [33]. The simulation of elastic materials with SPH requires to use in addition to the momentum equation (3), a discretised equation of motion for the deviatoric stress tensor $\boldsymbol{\sigma}_i$, which becomes a DOF according to our definition [22]. Any combination of the mentioned DOFs and arbitrarily many others is possible.

As a consequence a decision has to be taken on how to enable the user to perform particle dynamics with arbitrary DOFs. We should avoid the creation of new codes for each new application requiring a different combination of DOFs. Most state of the art codes define data structures at compile time. We here strive for a maximum of flexibility and define the data structure that stores per-particle DOFs and fluxes at runtime. This enables the user to dynamically extend the code with functionality that has not been precompiled. The realisation is shown in sections 4.2 and 4.3 dealing with the data structure for arbitrary DOFs and the hierarchy of available `Integrators`, respectively. In section 4.4 we show how data access for runtime defined DOFs is implemented. But first, an example input is presented which defines our goal.

4.1. User input

Figure 2 shows the input file of a fictitious simulation. Per `Integrator`, the user defines one additional DOF and assigns an identifier to it by the attribute symbol. Memory is reserved just for those DOFs which have been defined.

While IntegratorVelocityVerlet is responsible for the integration of $\mathbf{r}_i$ and $\mathbf{v}_i$ by applying the Velocity-Verlet algorithm [7], IntegratorEnergy introduces a scalar DOF to which the identifier `e` is assigned. This assignment is hard-coded and therefore

```
<Simulation simName="sim" ... >
  <Controller dt="0.02" ... >
    <IntegratorVelocityVerlet  ...  />
    <IntegratorEnergy ds_de="100/e" ... />
    <IntegratorScalar symbol="a" ... />
    <IntegratorTensor symbol="b" ... />
  </Controller>
  <ThermostatPetersEConserving dissipation="10" ... />
  <IEHeatConduction kappa="0.1" ... />
  <FPairScalar symbol="a" ... />
  <FPairTensor symbol="b" ... />
  <Phase>
    <BoundaryCuboid ... >
     <PCRandom e="1" a="10" b_xx="20" b_xy="-5" ... />
    </BoundaryCuboid>
  </Phase>
  <GridAveragerStructured ... >
    <InternalEnergy/>
    <Scalar symbol="a"/> <Tensor symbol="b"/>
    <OutputVTK fileName="grid/grid.vtk"/>
  </GridAveragerStructured>
</Simulation>
```

Figure 2: This input file for a fictitious simulation defines the following additional DOFs: an internal energy e, an arbitrary scalar a, and an arbitrary 3×3 tensor b.

not visible to the user. The attribute ds_de denotes $1/T = \partial s/\partial \epsilon$ and defines the relationship between the temperature T and the internal energy ϵ. In section 5 we discuss how mathematical expressions of this kind are processed. The `Thermostat` ThermostatPetersEConserving [30], the flux IEHeatConduction and the `GridMeter` InternalEnergy all check for the presence of an IntegratorEnergy and the corresponding scalar with its identifier `e`. Besides the historical fact that hard-coded modules such as IntegratorEnergy or IEHeatConduction had been available before the availability of expressions compiled at runtime in SYMPLER, the hard-coding is also performed for convenience, by providing an automatic calculation of the temperature. All of the hard-coded computations could equally well be performed with expressions compiled at runtime.

In contrast to IntegratorEnergy using a hard-coded identifier, the modules IntegratorScalar and IntegratorTensor allow the user to assign an arbitrary identifier to an additional scalar and 3×3 tensor, respectively. The fluxes FPairScalar and FPairTensor and the `GridMeters` Scalar and Tensor refer to these identifiers `a` and `b`. The former compute the fluxes $\dot{\chi}_i$ required by the `Integrators` while the latter record the respective additional DOFs χ for post-processing. DOFs require initial values to be set in a `ParticleCreator`. The attributes {e, a, b_xx, b_xy, b_xz, ...} are known to PCRandom because the corresponding DOFs have been defined by the `Integrators`.

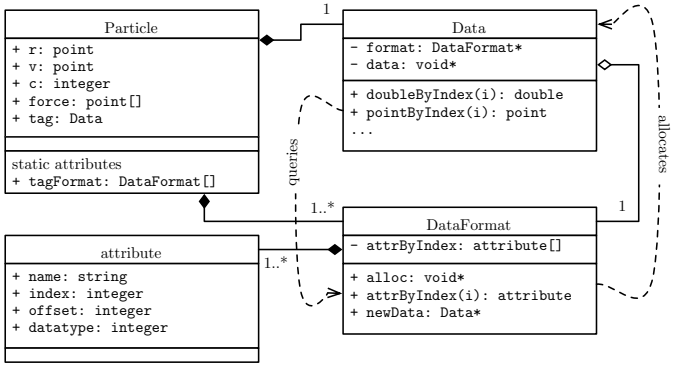

Figure 3: Interplay of the `Particle`, `Data` and `DataFormat` classes.

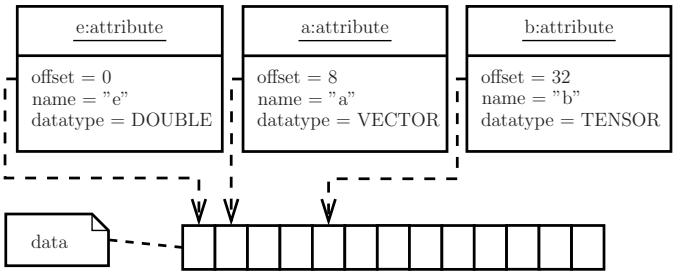

Figure 4: Access by memory offset for the three `Symbols` `e`, `a`, `b`, defined in the XML-input of figure 2, which are represented by three `attribute`-objects.

4.2. Data structure

The main building block around which additional DOFs can be implemented is a set of classes providing a data structure that can be defined during runtime. Hence each particle can be given a set of internal states. Internal states can be additional DOFs or properties that are computed once by `Symbols` and used many times (e.g., the local density or the local shear rate). In the current section we focus on the application of the presented data structure to additional DOFs. The `Symbol` classes are discussed in section 6.

The framework consists of the two classes `DataFormat` and `Data`. While the `DataFormat` holds the information concerning the structure of the internal states, `Data` contains the actual values. The class diagram in figure 3 shows the interplay of the classes with the `Particle` class.

The class `DataFormat` holds an array of `attributes`. Each `attribute` describes an entry of the data structure to be managed. Each `attribute` has a `name`, a `type` and an `index` unique within this `DataFormat` instance. Additionally, it stores the byte `offset` where the data can be found relative to the beginning of any memory block allocated by the `DataFormat`. This allows fast access for reading and writing. Among other data types there are the standard ones used for arbitrary DOFs, i.e., a scalar which is represented by a double precision number (`double`), a vector which is represented by three `doubles`, and a 3×3 tensor which is represented by 9 `doubles`. The data storage is not limited to `doubles` or arrays of `doubles`.

As shown in figure 3, each instance of `Data` holds a pointer to a corresponding `DataFormat` which contains the definition of the actually stored data. Additionally, `Data` holds a pointer to the actual data saved in an allocated contiguous block of memory. The `Data` class has methods allowing access to members by name, index or memory offset. In time-critical applications only access by offset should be used, which is illustrated in figure 4. When clients access specific data in the memory block, this is performed by functions such as `pointByIndex` or `pointByOffset`, which typecast the data in the correct form as denoted by the corresponding `attribute` in `DataFormat`. For example, the return value of the method `pointByOffset` is of type `point`, which is the representation of a three dimensional vector.

Each particle is represented by an instance of the `Particle` class (cf. figure 3). The position `r` and velocity `v` are hard coded DOFs, as well as the particle's "colour" `c`. A colour is the internal integer representation for the species of a particle, i.e., for a group of particles with identical properties. The Newtonian force is hard coded as well, i.e., the momentum flux of a particle in the current time step and a number of previous time steps (the flux history). This is because those properties are usually needed for all particles and most kinds of problems. Depending on the `Integrator` a flux-history of different length is stored. For example, in the case of the Velocity-Verlet algorithm [7], the force of the current time step and the force of the previous time step are needed for one update of the velocity.

All other DOFs, such as for example the internal energy ϵ_i, and associated information, such as the flux of the internal energy $\dot{\epsilon}_i$, are stored in the field `tag` which is of type `Data`. The corresponding `Integrator` is responsible for reserving memory for both the DOF itself and for its associated flux. An `Integrator` can define as many `tag`-entries as desired. This can for example be used to define a force history for an additional DOF.

The format description associated with `tag` is given by the `DataFormat` array `tag_format`. This is a *static* property belonging to the `Particle` class and not to individual instances. Each entry of `tag_format` gives the data-structure definition for a certain colour. This means that the information stored in `tag` is equal for all particles of the same species, but can vary from species to species.

4.3. The `Integrator` hierarchy

For the definition of additional DOFs and for their time evolution, suitable `Integrator` classes have to be introduced. Figure 5 shows the complete class-hierarchy of currently available `Integrators`. The only abstract classes (class-names in figure 5 written in *italic*) are the base class `Integrator` and its child-class `IntegratorPosition`.

The coloured classes are responsible for arbitrary DOFs, namely `IntegratorScalar`, `IntegratorVector`, `IntegratorTensor`, and most of their child-classes. The difference between the parent classes and their children is the implemented integration algorithm. While the parents implement Euler integration [7], the children use a predictor-corrector algorithm that generalises the Velocity-Verlet algorithm [9, 21].

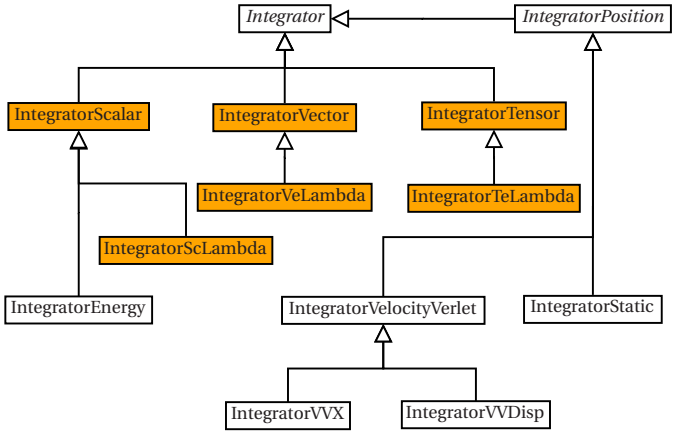

Figure 5: Class-hierarchy of `Integrators`. All classes written in *italics* are abstract, while all others represent `Integrator` modules which may be defined by user input. The coloured classes are those responsible for arbitrary DOFs. See the text for short descriptions.

Another class derived from `IntegratorScalar` is `IntegratorEnergy` which was already discussed in section 4.1. Note that the field `tag` of the class `Particle` is also used by `IntegratorEnergy` to store the DOF and its flux. Besides `IntegratorEnergy`, all other `Integrator` classes providing hard-coded functionality inherit from `IntegratorPosition`. As the name indicates, they are all responsible for the time-integration of the particle-position and, with the exception of `IntegratorStatic`, of the velocity. `IntegratorStatic` integrates the particle positions according to an imposed user-defined velocity, which may be time and space dependent as defined in a runtime-compiled expression, and ignores any forces acting on the particles.

The class `IntegratorVelocityVerlet` was already discussed. With its child `IntegratorVVX`, we can implement (for example) the so-called X-SPH algorithm [10], where the time evolution $\dot{\mathbf{r}}_i = \mathbf{v}_i$ of the position is replaced by

$$\dot{\mathbf{r}}_i = \bar{\mathbf{v}}_i = \mathbf{v}_i + \varepsilon \sum_j \frac{m_j}{\bar{\rho}_{ij}} \mathbf{v}_{ij} W_{ij} \qquad (13)$$

and where $\bar{\rho}_{ij} = (\rho_i + \rho_j)/2$ and $0 < \varepsilon < 1$. The smoothed velocity $\bar{\mathbf{v}}_i$ on the right hand side of equation (13) is stored in the `Particle`'s `tag` and the memory is reserved by `IntegratorVVX`. The `Integrator` could alternatively compute $\bar{\mathbf{v}}_i$ on the fly, but then it must be recomputed if needed by `Meters` or other clients. Currently, $\bar{\mathbf{v}}_i$ is not computed at all by `IntegratorVVX`, which simply expects this quantity to be available and does not care about its computation. This is performed by user-input in an external module PairParticleVector (cf. section 6), which makes it very easy to replace the form of $\bar{\mathbf{v}}_i$ given in equation (13) by other expressions as required. `IntegratorVVDisp` reserves memory in the `tag` for the integrated displacement vector $\mathbf{d}_i$ of each particle. If this quantity is computed from the beginning of the simulation until the time t, then the result is trivially $\mathbf{d}_i(t) = \mathbf{r}_i(t) - \mathbf{r}_i(0)$. Note that there could be clients which reset the displacement or make other modifications.

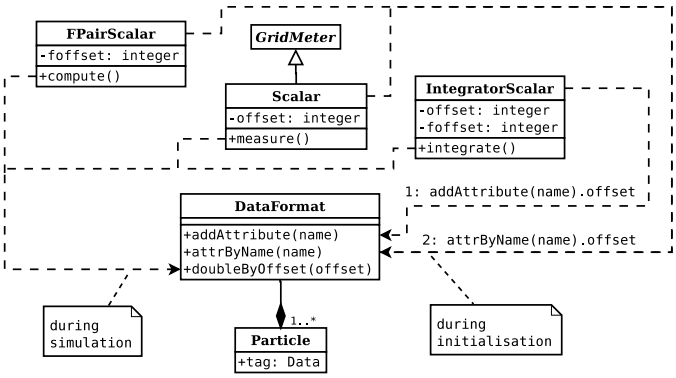

Figure 6: The class diagram shows the slow and fast access to a scalar stored in the `Particle`'s `tag`. Initially, slow access must be used during initialisation to set up the fast access during simulation. The classes `IntegratorScalar`, `FPairScalar`, and `Scalar` are used as examples.

4.4. Data access

To ensure speed of computation, we must distinguish between data access, and operations during initialisation and, on the other hand, in each time step of the simulation. The former is not time-critical but the latter is. During initialisation, data defned at runtime may be searched for by comparing string-identifiers or using access to `Data`'s storage by index (cf. section 4.2). Access by index is slow as well because `Data` stores different data types. Therefore, the memory-address corresponding to an index must be computed by summation of the sizes of all data types of the items stored before. Hence, storing the memory-address itself, or the offset to the beginning of the storage, is required during initialisation for fast access later in each time step. SYMPLER saves the offsets. Fortunately, it is sufficient that each client stores just those memory-offsets that it requires, and due to division of work among the classes, this is just a few. Figure 6 shows the communication during initialisation and during time-sensitive simulation.

First, during initialisation, the `IntegratorScalar` that serves as an example, needs to register the new scalar DOF in the `Particle`'s `tag` by calling the function `addAttribute()` of `DataFormat`. This function requires a few arguments which are not all shown, for example, a string-identifier (`name`) and a data type. The function returns the `attribute`, i.e., the description of the registered scalar (cf. figure 3). This enables the `Integrator` to get the memory offset for later use by applying `.offset` to the obtained `attribute`. The software notices if the user tries to introduce the same DOF twice (by two different `Integrators`), and reports an error.

Other clients may then obtain the `offset` as well during their initialisation. They request the attribute by calling the function `attrByName(name)` of `DataFormat`. The argument is the correct identifier of the scalar. The identifier may be hard-coded. For instances of classes such as `FPairScalar` or `Scalar`, the identifier is given by the user as shown in figure 2. An identifier for the flux is derived in a predefined way from the identifier of the DOF. Hence, the identifier for the flux is automatically known as well. As can be seen in figure 6, all clients just request and save those offsets they really need.

Then, all classes are able to access the required data by calling the function `doubleByOffset(offset)` of `DataFormat` for the case of a scalar. The argument is the memory offset which the clients have requested during their initialisation. This data access is fast. Hence, it may be used in each time step during simulation for example for evaluating fluxes (`FPairScalar`, other fluxes) for time integration (`IntegratorScalar`, other `Integrators`) and for recording data (`Scalar`, other `Meters`).

5. Arbitrary equations of motion

For now, we have achieved modularity which allows to apply methods such as MD, DPD and SPH in one single software tool. It is easy to exchange one available module, for example a force, against another available force. But no matter how large the database of forces or fluxes is, applications remain for which terms are not yet available. The straightforward approach is to implement the missing components.

This section presents an alternative solution which does not require any code extensions for new applications. In addition to modularity and arbitrary DOFs, we also require the possibility to define arbitrary algebraic expressions, in order to go beyond hard-coded modules that implement specific inter-particle interactions. This feature is useful for situations such as: i) A material-parameter (for example the viscosity) should depend on state-variables or DOFs. ii) Different SPH-discretisations should be compared. iii) Flexibility in the material dynamics is required: e.g., viscous fluids, elastic solids or visco-elasticity. iv) A new DOF with a new equation of motion is introduced and should be coupled to an existing model. One example is the concentration ϕ in equation (10), another may be the stress tensor in case iii).

These cases have in common that force or flux expressions are extended or replaced by others, but the main algorithm remains untouched. In SYMPLER these cases are dealt with by means of symbolic expressions that are compiled during runtime. This is described in the following. On the other hand, complex force-fields as they are common in MD, are not in the scope of this approach, even though they could in principle be handled. Generally it is more efficient to implement such a force-field for a very special application (for example the Brenner-potential for hydrocarbons [34]) as one or more derived classes of `GenF` (cf. section 3.2).

The MD-simulators LAMMPS [11] or GROMACS [12] allow users to choose among a large portfolio of predefined implemented pair-interactions. LAMMPS [11] also allows to use mathematical expressions and temporary variables in its input script. In the expressions, particle attributes known to LAMMPS can be used. With SYMPLER's Integrator hierarchy we can additionally define new attributes at runtime, which represent new DOFs. Then, SYMPLER integrates their equation of motion in time. Besides the freedom to define the LHS of equation (12), the user is also free in the definition of the RHS, by using runtime-compiled symbolic expressions.

The integration of the standard equations of motion of SPH for the mass density, momentum and heat flow, can be performed in LAMMPS by means of an additional external package. There are many of these packages whose source code was implemented by users [35]. The discretisation of further equations of motion with additional variables, such as a concentration ϕ would require an extension of the package's source code. From the point of view of SPH, i.e., of a dicretisation method for partial differential equations (PDEs), the approach presented here is a compromise between approaches of finite element (FE) based software such as COMSOL® [36] and DIFFPACK® [37]. COMSOL® offers a set of pre-discretised PDE-operators to the user which may be combined to the desired PDE. The user does not directly deal with the discretisation. DIFFPACK® requires the user to implement the discretised equations and to state explicitly the integrand of a weighted residual approach. By means of the expressions compiled at runtime described in the following, SYMPLER offers similar features for SPH and DPD. Similarly to DIFFPACK®, the user is responsible for the discretised form of the equations. In contrast to DIFFPACK®, the user just provides the expressions but does not have to care about C-programming. As in COMSOL®, also a limited amount of pre-defined operators are available, i.e., the hard-coded fluxes and `Thermostats` (cf. section 3.2).

In section 5.1, an input file is used again to demonstrate what is meant and achieved by the implementation of expressions compiled at runtime. Then, in section 5.2, the class hierarchy of `Functions` is presented, which implements the different use-cases of expressions compiled at runtime.

5.1. User input

Figure 7 shows an excerpt of an input file implementing the shear induced solids migration model of equation (10). In the module InputWF, a user-defined and runtime-compiled SPH-interpolation kernel $W(r_{ij})$ is introduced, which is required, e.g., in equation (4). To save space, only a part of the piecewise defined spline function is shown. This kind of runtime-compiled expression represents one of the use-cases, say "A", which has to be considered and which is discussed in section 5.2.

After the `Integrators`, and before the first flux FPairScalar, different kinds of `Symbols` are defined. ValCalculatorRho and VCNegDKernelDivr are hard-coded and compute the density summation from equation (5), and $w_{ij} = -W'(r_{ij})/r_{ij}$ from equation (8). Both modules require a loop over all interacting particle neighbours. But while ValCalculatorRho computes and stores one quantity per particle, VCNegDKernelDivr computes and stores one quantity per interacting pair of particles. The registration and storage of these quantities defined at runtime is done as described in section 4. Pair-quantities are currently stored analogously to the class `Particle` in a class `Pairdist` representing a pair of particles.

PairParticleTensor behaves much as ValCalculatorRho, but the expression to be computed is a user-defined tensorial quantity. As can be seen, the symbols `n` and `w` computed by ValCalculatorRho and VCNegDKernelDivr are used now and appear as `ni`, `nj` and `wij`. PairParticleTensor has three attributes containing expressions compiled at runtime, namely pairFactor,

```
...
<InputWF cutoff="1" name="M5"
  weight="...-8*step(r-0.6)*(r-1)^3)" ... />
<Controller timesteps="100000" dt="0.01" ... >
  <IntegratorVelocityVerlet  ...  />
  <IntegratorScalarLambda species="A" symbol="V" ... />
</Controller>
<ValCalculatorRho symbol="n" weightingF="M5" ... />
<VCNegDKernelDivr symbol="w" weightingF="M5" ... />
<ParticleScalar symbol="g"
  expression="sqrt(({G}+T({G})):({G}+T({G}))/2)" ... />
<PairParticleTensor symbol="G" cutoff="1"
  pairFactor="[rij]@([vi]-[vj])*wij/2"
  particleFactor_i ="unitMat(1/ni)" ... />
<ParticleScalar expression="n*V" symbol="phi" ... />
<FPairScalar scalar="V" pairFactor="-wij*(phii+phij)
␣␣*(phii*gi-phij*gj)/(ni*nj)" ... />
<Phase ... >
...
```

Figure 7: Input file with expressions compiled at runtime. This is an excerpt of the input required for the simulation of the shear induced solids migration model of equation (10). To save space, some attribute names are abbreviated as compared to the true names in SYMPLER. `T(G)` returns the transpose of the matrix `G`.

particleFactor_i, and particleFactor_j (not shown). The three expressions are used to implement the computation of the SPH velocity gradient tensor as defined in equation (7). The reason for three expressions is discussed in section 5.2. For these expressions, the number and name of the variables is not known before runtime. For each pair of particles *i* and *j*, all hard-coded or runtime defined variables of either one of the `Particles`, or of the place where pair-specific-information is stored (currently a class `Pairdist`), are allowed. This we call use-case "B".

Use-case "C" is encountered in the two `Symbols` with name ParticleScalar. They do not require a loop over pairs of particles but simply compute a new particle-property from already computed properties of the same particle. The first module computes the second invariant (11) of the strain rate tensor (7) where ":" is interpreted as a double contraction. Since the equation of motion (10) for the occupied volume V_ϕ instead of the solids concentration ϕ is integrated, for convenience, the last ParticleScalar computes $\phi = nV_\phi$ from the symbol `V` which was introduced by IntegratorScalarLambda.

The following flux FPairScalar of the scalar `V` implements the first part of the equation of motion (10) for shear induced solids migration. The second part was omitted to save space but can be implemented in the same way. The behaviour is very similar to PairParticleTensor. The main difference is that an FPairScalar always computes the contribution to a flux of a scalar DOF.

All expressions are compiled into machine code by classes derived from the abstract class `Function` (cf. section 5.2). In the current implementation this is done by transforming the expression into C-code and compiling the generated C-file into a dynamically linked library, using a suitable compiler available on the system. Then, the machine code is dynamically linked to the running software SYMPLER that had invoked the compilation.

The parsing process consists of two major steps. First, the mathematical expression is transformed into a tree representation. Then, the classes building the tree recursively construct the C-expression. Alternatively, the tree may be kept and used for the evaluation of the mathematical expression. This is also supported but a comparatively slow operation and therefore not recommended and not used except during initialisation. Each class used for parsing represents a node of the tree, such as a unary function, a binary operator, a variable, or a constant.

5.2. Hierarchy of `Function`s

As already indicated in the previous section, expressions compiled at runtime appear in three different major use-cases (cf. figure 7): A) The allowed variables are known at compile-time. Their names and their meaning are hard-coded inside a module (cf. InputWF). B) The quantity to be computed may depend on any hard-coded or runtime defined quantity which is related to a pair of particles. Hence, it must be distinguished between data belonging to particle *i*, data belonging to particle *j* and data being a property of the pair of particles (cf. PairParticleTensor). An example for the latter is the distance vector $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ which can be accessed by the expression "`[rij]`". On the other hand, $\mathbf{r}_i$ and $\mathbf{r}_j$ are particle data accessible by the expressions "`[ri]`" and "`[rj]`", respectively. C) A new particle property is to be computed from other particle properties: In B) and C) the number and name of the variables is not known before runtime. Every hard-coded quantity, and every quantity defined at runtime and stored in the `Particle`'s `tag` may be used (cf. ParticleScalar). Based on this case distinction, a class hierarchy is constructed. This is shown in figure 8.

The abstract parent class `Function` has two purposes. First, it generalises the fact that each derived `Function` requires a `parser` and a `compiler` for the user-defined `expression` it holds. Second, it serves as a common interface for the class (currently the `Controller`) responsible for requesting the compilation of all the `Functions` collected during parsing of the input file. On the other hand, those modules where a runtime-compiled expression can be defined always hold instances of specific classes derived from the abstract class `Function`, since they rely on their special additional abilities derived from one of the use cases A–C.

`FunctionFixed` implements use-case A. Each instance contains a hard-coded list of allowed `variables`. This list is filled by the module (e.g., InputWF in figure 7) holding this instance. The evaluation of the compiled expression is implemented by overloading the `()`-operator of the class. For `FunctionFixed` this requires to create as many combinations of arguments as required by classes holding an instance of `FunctionFixed`. In the current version of SYMPLER six versions with one to six scalar arguments are sufficient. The note on the `()`-operator shows that the request is forwarded to the `compiler` which holds the actual link to the compiled function in memory.

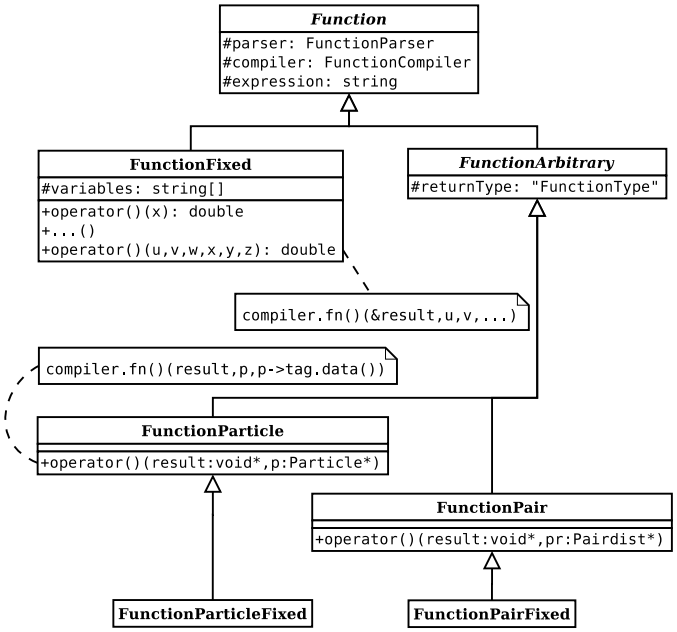

Figure 8: Class-hierarchy of `Functions` implementing expressions compiled at runtime. `FunctionFixed`, `FunctionParticle` and `FunctionPair` represent the three main use-cases. The two classes at the bottom represent combinations for special purposes.

The derived classes representing the remaining two use-cases B) and C) share many features which are combined in the common parent `FunctionArbitrary`. For example, the `returnType` is adjustable by the class holding a specific instance of a `FunctionArbitrary`.

`FunctionParticle` is the implementation of use-case C. The main difference in its `()`-operator is the arbitrary (`void*`) type of the field used for storing the computed result and the fact that just the pointer to a particle is required as second argument. Then this call is forwarded to the compiled function by passing to it the pointer to the particle and the memory-address of the storage place in the `Particle`'s `tag` for the result to be computed (cf. Figs. 3 and 4). In this way, primitive pointer arithmetic is sufficient for data-access in the C-code which is created from the user input. All data can be either accessed by a memory offset to the `Particle` pointer, or to the pointer to the `Particle`'s `tag` data. No special type information of the original SYMPLER source code must be included.

For `FunctionPair` implementing use-case B this mechanism is similar. The difference is just that the possible sources of data are now *two* `Particles` and one pair (currently represented by `Pairdist`). On the level of the `()`-operator of `FunctionPair` it is still enough to pass just the pair as an argument, since it allows access to all the rest. But on the level of primitive C-code, all of the 3×2 memory-addresses must be passed to the function, i.e., the address of the object itself and of its `tag` for the two `Particles` and for the `Pairdist` object, respectively. During parsing the translation of the indices "`i`", "`j`", and "`ij`" (cf. figure 7) to the corresponding particle or to the pair is made.

It is straightforward to combine the concept of fixed and arbitrary variables. This is useful if, for example, an external force should be modelled, which depends both on particle properties and on absolute time such as a time-dependent magnetic field. This is realised in the classes `FunctionParticleFixed` and `FunctionPairFixed`.

Concerning `FunctionPair`, the question remains as to why a module such as PairParticleTensor requires three different expressions compiled at runtime? When exchanging i for j in the SPH-discretisation of the rate of strain tensor (7), we see that most of the right hand side remains invariant because it is symmetric under this exchange. But there is also the asymmetric part m_j/ρ_j which turns into m_i/ρ_i. We may assume $m_i = m = \text{const.}$ for all i. But the density ρ_i may be different from one particle to the other leading to numerically unsymmetric contributions to the particles i and j, respectively. For completely symmetric (or anti-symmetric) expressions, computing the contribution of a pair once is sufficient. For the given example, in addition, the non-symmetric part is computed separately for both particles. This is the reason why the module PairParticleTensor (cf. figure 7) and every flux as well hs three expressions compiled at runtime with attribute names pairFactor, particleFactor_i, and particleFactor_j. If an attribute is not defined, the expression behaves by default as an identity element.

This solution is suitable for an algorithm looping over a list of pairs. For parallel distributed memory applications it can be necessary to separate the contribution of particle i to the flux of particle j completely from the contribution in the opposite direction. Then the present system is still usable by expressing the flux-expressions completely by particleFactor_i and particleFactor_j. The concept of expressions compiled at runtime can also easily be extended to 3- or 4-particle interactions.

6. Hierarchy of `Symbols`

This section summarises the available tools for data manipulation. All presented classes are packed into a hierarchy with the parent class `Symbol` (cf. figure 1). All modules make use of the data structure for adding additional information to particles (cf. section 4.2) or pairs. Some modules make use of expressions compiled at runtime and some have hard-coded functionality. Figure 9 shows the class-hierarchy. Those classes using expressions compiled at runtime have been coloured. It follows a short summary. Table 1 lists some of the non-abstract classes implementing modules. The second column states whether the input data comes from a loop over particles or a loop over pairs of particles. The third column gives the datatype of the computed output and the last column lists whether the output is stored per particle or per pair of particles.

The first specialisation of `Symbols` corresponds to the split in use-cases B and C from section 5.2. This is realised by the two inheriting abstract subclasses `ValCalculator` and `ParticleCache` (cf. figure 9). It can be seen from the member `species` that a `ParticleCache` is defined for a specific species of particles, i.e., of particles with the same properties. On the other hand, every `ValCalculator` is defined for a *pair* of species. The computation of the function, i.e., the method

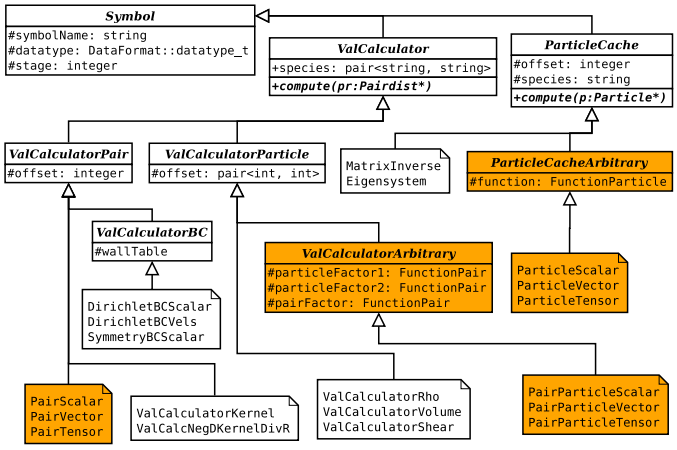

Figure 9: Class-hierarchy of `Symbols` for data manipulation. To save space the non-abstract classes implementing usable modules are given as lists. Their input and output behaviour is summarised in table 1. The classes using expressions compiled at runtime are coloured. To save space, some classes have been left and are not discussed.

`compute`, may also be already defined on this abstract level. Here, the difference lies in the argument the function takes, a pair representation for `ValCalculator` and a particle for `ParticleCache`. Table 1 shows this distinction for the respective derived classes.

`ValCalculators` require a further distinction into pair and particle concerning the produced output. The table shows this distinction. A `ValCalculatorPair` stores one result per pair of particles of possibly different species. Therefore, it requires *one* memory-offset to the location (currently in the `tag` of a `Pairdist`) where the result should be stored. On the other hand, a `ValCalculatorParticle` requires a pair of memory offsets to `tags` in the `Particles` (cf. section 4.2) because the two species of the pair interaction might be different. One might for example think of the definition of an interaction between fluid particles representing the species oil and water, respectively. `Particles` of different species may store different kinds and different amounts of data in their `tag`. As a consequence, the memory offsets for the output of a `ValCalculatorParticle` may differ from species to species and have to be saved separately. As `ValCalculatorPair`, the class `ParticleCache` needs one memory offset as well, but the offset is pointing into a `Particle`'s `tag`.

On the next level of the class hierarchy, the splitting between hard-coded and user-defined functionality is performed. From `ParticleCache` and `ValCalculatorParticle` the abstract classes `ParticleCacheArbitrary` and `PairParticleArbitrary` are derived, which are used to create the classes for expressions compiled at runtime that have already partially been discussed as modules in figure 7. A corresponding abstract class inheriting from `ValCalculatorPair` was not yet implemented. As discussed in section 5.2 the class `ParticleCacheArbitrary` requires *one* `FunctionParticle` object, while the class `PairParticleArbitrary` requires *three* `FunctionPair` objects in order to realise arbitrary expressions compiled at runtime. Most of the hard-coded modules are listed in figure 9, and the I/O-behaviour of some of them is given in table 1. A complete overview of all available modules (which is always up-to date) can be found in the built-in manual of SYMPLER [15].

A final word about the ordering of the computations performed by the `Symbols` is necessary. Each module being an instance of `Symbol` may be interpreted as a computational box with well defined input and output terminals. The terminals of different modules have to be connected correctly, i.e., if result *B* depends on result *A*, then *A* has to be computed first. As can be seen from figure 7, an ordering of the `Symbol` modules in the input file is not required. The dependencies and the correct ordering is determined iteratively during initialisation. Unresolvable loops in the dependencies are reported. If *B* depends on *A* and *A* on *B*, then the user may specify for example that old values of *B* should be used for the computation of *A*.

Table 1: Input and output properties of exemplary `Symbol` modules. The corresponding class hierarchy is given in figure 9.

Class/Module	Input	Output	Stored per
`ValCalcNegDKernelDivR`	pair	scalar	pair
`DirichletBCVels`	pair	vector	pair
`PairParticleScalar`	pair	scalar	particle
`PairParticleVector`	pair	vector	particle
`PairParticleTensor`	pair	tensor	particle
`ParticleScalar`	particle	scalar	particle
`ParticleVector`	particle	vector	particle
`ParticleTensor`	particle	tensor	particle
`Eigensystem`	particle	vector & tensor	particle

7. SYMPLER on the bwGRiD

SYMPLER is available to scientists on the eight high performance computing (HPC) clusters of the bwGRiD project [38, 39] in the federal state of Baden-Württemberg (BW), Germany. The bwGRiD provides HPC-Resources especially for academic researchers in BW but also from elsewhere in Germany and even abroad. SYMPLER is accessible to users via a SYMPLER-portlet [40] that was developed for the bwGRiD-Webportal [41]. Figure 10 shows two snapshots of the SYMPLER-portlet. The portlet provides an easy-to-use web-interface and allows users to send compute jobs to available nodes on one of the state's eight HPC-clusters. Input files for SYMPLER can be conveniently uploaded, created and edited on the portlet. Besides regularly released and statewide accessible versions, a nightly built version with the newest updates was created locally at one cluster site. Through the SYMPLER-portlet, both versions can be easily run from any web-access-point anywhere in the world, only by a web browser with installed grid certificates. Since the SYMPLER-portlet is compatible with the Java portlet specification standard JSR 168 [42], it is easily portable on portals of other grid-computing environments, and the design concepts of the SYMPLER-portlet are hence generally applicable.

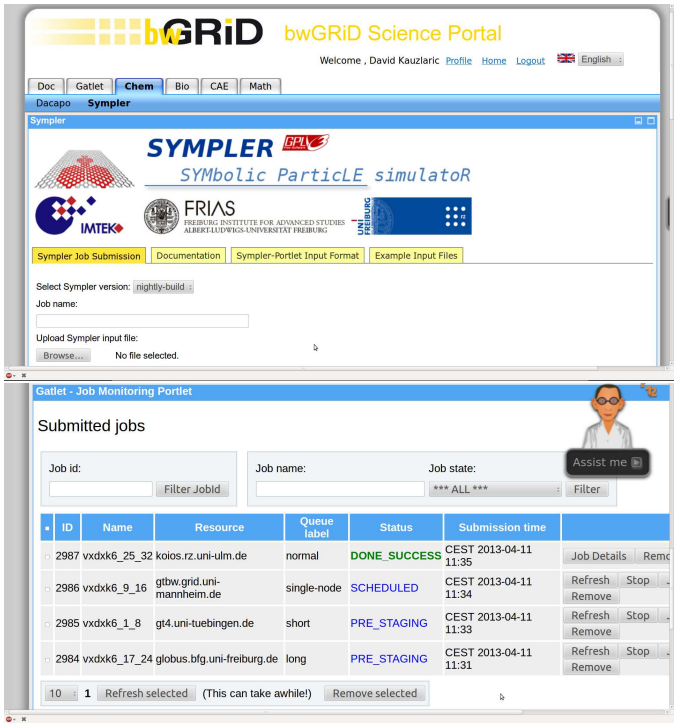

Figure 10: bwGRiD-portlet for SYMPLER with job-submission window (top) and job-monitor (bottom) showing 4 job-packages of 8 jobs each. Each package was sent to a different cluster in the federal state of Baden-Württemberg.

SYMPLER possesses some shared memory parallelism via OpenMP-parallelisation of most of the force-calculation and time-integration routines and of other loops over particles or particle interactions. The bwGRiD supports this parallelism and allows to distribute one compute job over several cores. The main focus during the development of the SYMPLER-portlet was put on so-called primitive or pseudo-parallelisation, still the best scaling parallelisation strategy. The reason for this focus was the observation of an increasing need for parameter screening and ensemble averaging in engineering and computational physics applications, such as the process engineering of micro powder injection moulding or the statistical mechanical coarse-graining of graphene and carbon nanotubes (cf. section 8). In these cases, many independent simulations have to be performed, which is a task ideally suited for a grid environment providing an immense number of computational nodes and cores. The SYMPLER-portlet was designed in such a way that several similar jobs can be automatically combined and calculated in parallel. The single threads are distributed on different nodes via the Message Passing Interface (MPI). The runtime-compiling capabilities of SYMPLER (cf. section 5) are explicitly accounted for by the SYMPLER-portlet by first managing and bookkeeping the runtime generated program code and then the runtime-compiled libraries that have to be linked to the respective instances of SYMPLER running concurrently. Required resources used for parallel execution such as nodes and cores are automatically calculated by an algorithm based on the number of threads and the hardware that is used for the calculation. The user must only provide a jobname, and estimates for the maximal wallclocktime and the maximal memory consumption. A directory structure for the SYMPLER output is created automatically by the portlet based on the number of XML input files and their contents. The webportal provides tools for monitoring the job and for downloading and organizing the data after jobs are finished. In addition to the html SYMPLER documentation, the portal offers an Avatar that provides help functionality, a step-by-step guide, and example input files to train new users and answer questions of the more experienced ones.

8. Some application examples

The design of SYMPLER enables users to apply the software to computational problems from various very different fields. A collection of problems where SYMPLER has already been applied is summarised in figure 11. One major field is the process simulation of micro powder injection moulding (micro-PIM) of powder filled feedstocks, and micro-casting of metal alloys for ceramic and metalic micro-parts (cf. left part of figure 11). Special aspects that have been investigated in detail for micro-PIM are yield-stress effects [43, 44] and powder segregation in the feedstock [24, 45]. Also, heat extraction by the mould was simulated both for micro-PIM and also micro-casting. In particular, the cast alloy can be coupled thermally to the cavity represented by a full or reduced finite element based heat transport model [45]. In all these cases particle dynamics was used for an SPH-simulation of the fluid dynamics of the moulded material. Further fluid dynamic applications of SYMPLER have been the simulation and design of electrowetting based microfluidic devices for the transportation of pico- to nanoliter droplets [46, 47], constraint SPH-simulations of haemodynamics for the estimation of blood flow measurements with magnetic resonance imaging [48], and hydrodynamics with convective transport and dynamics of local magnetisation [49].

Another major field of application is coarse-graining and the related mesoscopic simulation method dissipative particle dynamics (DPD). In particular, coarse grained models of carbon nanotubes [50, 51] and graphene [52–54] have been developed (cf. left part of figure 11). Here, SYMPLER was used both for DPD simulations and also MD-simulations. Based on the latter, coarse grained DPD-models have been constructed. For the simulation of the macromolecules, SYMPLER also provides bonded interactions, some of them using expressions compiled at runtime. In another approach to DPD, a new energy-conserving DPD-scheme was developed and successfully implemented and tested with SYMPLER [30].

The given application examples show that multi-physics and multi-scale problems can be handled in various ways: Hydrodynamics can be coupled, for example, with heat flow or elastodynamics, either fully within the particle based framework (with MD, DPD, or SPH) or even with a concurrently running solver for a linear system of equations representing, e.g., a finite element based model. Multiple length or time scales can be readily introduced by providing different interaction forces compiled at runtime for different combinations of particle species. Since interaction forces compiled at runtime may be arbitrary functions

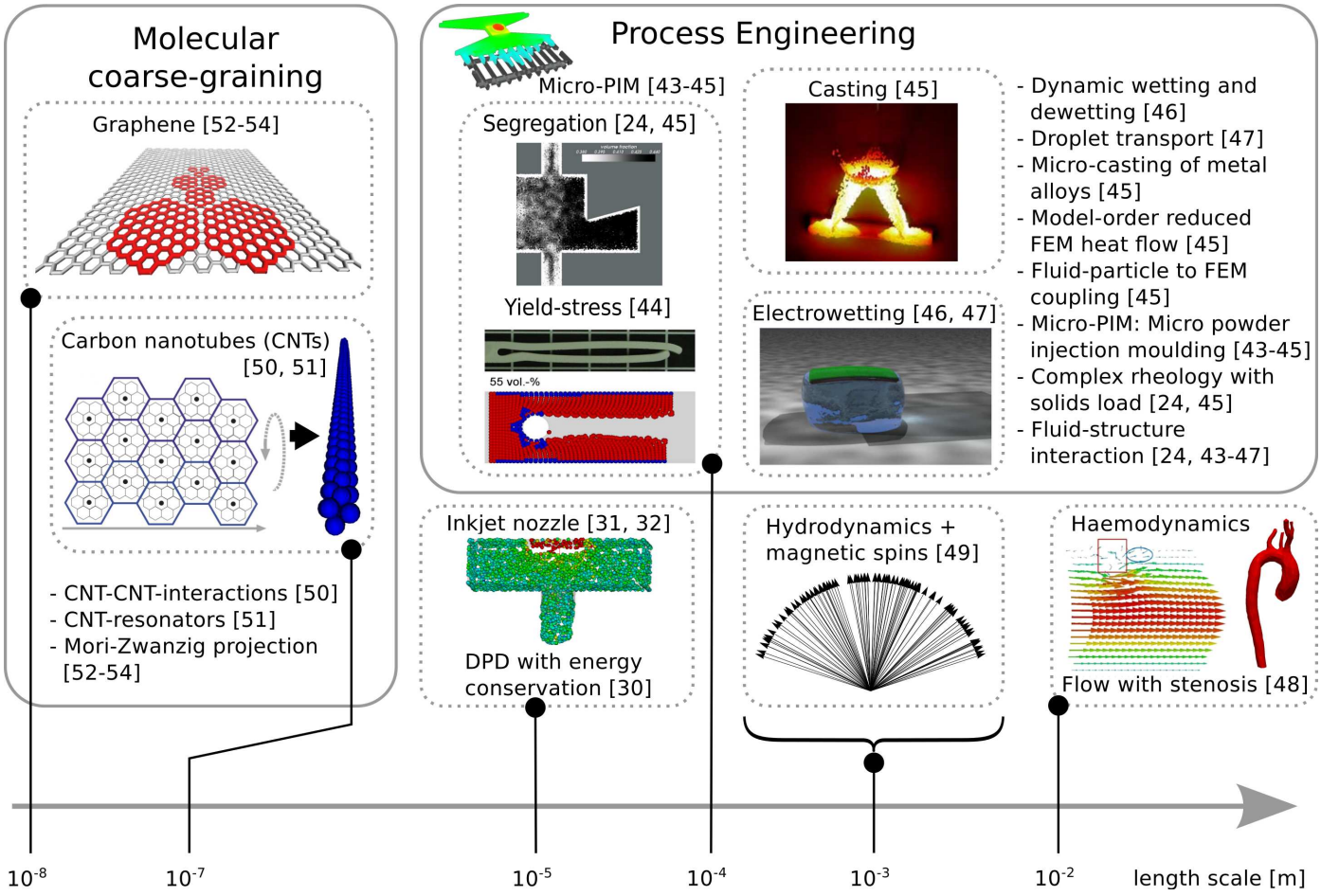

Figure 11: Collection of problems where SYMPLER has already been applied successfully. They are sorted from left (small) to right (large) according to the typically occurring maximal length scales, as indicated on the axis at the bottom.

of the particles' state, the user may also implement a seamless transition from MD to DPD or from DPD to SPH.

9. Summary and discussion

From an application point of view, SYMPLER could be summarised by its three outstanding features: i) An arbitrary number of particle species with arbitrary DOFs, ii) arbitrary symbolic expressions and iii) integration into the bwGRiD environment. Although not exhaustively, the main concepts i) – iii) have been discussed. The software uses object-orientation to guarantee modularity and flexibility, both for the user and the programmer.

For a versatile simulation tool expressions compiled at runtime are used in order to specify new physical problems described by additional partial or stochastic differential equations with additional DOFs. The solution consisted of adding a field for arbitrary additional data to each `Particle`. The `Integrators` were made responsible for introducing new DOFs to this data field and for integrating their respective equations of motion. This extension can be used for example for adding an internal energy or a concentration variable ϕ as additional DOF.

Runtime compilation on the other hand, provides us with flexibility in the *form* of the equations. Different SPH-discretisations can be tested, and new state-dependencies, for example of the transport parameters, are easily added. All this flexibility is gained without significant loss in performance because the expressions compiled at runtime provide a means of fast computation.

Both concepts are combined in the `Symbol` hierarchy. This hierarchy provides a collection of modules which are subdivided into three main groups: The first group computes new particle properties from already computed properties of the same particle. One hard-coded example is the computation of the eigensystem of an arbitrary 3×3 tensor. Other modules compute particle properties as well, but as the result of a summation over all its neighbours. The SPH-discretisation of the shear tensor is one example and might be the input for the computation of the eigensystem. The final group of `Symbols` stores computed properties in pairs, for example the derivative of an interpolation function.

It is often claimed that object-orientation is slow compared to procedural codes. The SYMPLER design splits operations into slow initialisation and uses strategies to achieve efficient and fast code for the computationally intensive particle operations.

Table 2: Comparison of runtimes for hard-coded and runtime-compiled simulations of the same problem. In addition, runtimes of comparable simulations with LAMMPS are shown as well. The results have been obtained from single-core simulations of 1000 particles over 5000 time steps on an ordinary Laptop computer. LJ: Lennard-Jones fluid at $T = 1$ with particle density $n = 0.8442$ and $r_c = 2.5$. DPD: Simulation of a DPD-gas at $T = 1$, $n = 3$, $r_c = 1$. The input for SYMPLER is distributed as example input together with the source-code [15].

	hard-coded	runtime	LAMMPS
LJ [s]	22.0	30.7	7.5
DPD [s]	13.5	17.1	7.6

Often the slow operations are used once to activate the fast operations for future use. For example, an arbitrary variable added and stored in the particle is accessed once by the slow operation of searching for its string identifier in order to connect the client with the variable by a memory address, allowing fast access therafter. A slow object-tree representation is built first from a user defined mathematical expression, in order to cast it into a compilable C-expression afterwards. The final C-expression is primitive procedural code. The only overhead during computation coming from object-orientation is the dereferencing when objects responsible for different tasks pass messages to each other. In SYMPLER this overhead amounts to a few tens of percent. This can be seen in table 2 where the required absolute simulation time is compared when using hard-coded modules versus expressions compiled at runtime within SYMPLER for two cases: an MD-simulation of a Lennard-Jones (LJ) fluid [7] and a DPD simulation of a dissipative gas (with $\mathbf{F}_{ij}^C = \mathbf{0}$ in equation (2)). These are two cases where SYMPLER still provides old hard-coded modules used before runtime compilation was introduced. These results show that the expressions compiled at runtime themselves produce only a negligible overhead. The comparison to LAMMPS shows that there is still room for improvement. Interestingly, the difference to LAMMPS is larger for the LJ-fluid. Since there are more interactions per particle in the LJ-case this indicates room for improvement especially in the interaction computation. And indeed, among the measures for performance improvement that are implemented in LAMMPS but not yet in SYMPLER are, for example, bin sizes of less than a cutoff for the cell subdivision [55], and particle sorting and similar measures to allow for cache-coherence [11]. A cache coherent storage mechanism for the particles' degrees of freedom is definitely possible within the framework of the `Particle`, `Data`, and `DataFormat` classes presented in figure 3. The general concepts presented here can therefore be kept untouched, and only some internal data structures would have to be modified. By presenting the code-design in this paper and by publishing the code under the GNU public license [15], we hope to create a community around this code which can help to improve and extend it by still missing algorithms. Especially the accessibility of the source code on the hosting platform GitHub [15] is an excellent prerequisite.

The design as a whole enables us to create and manage a single software project to grow richer in its capabilities. The emergence of an endlessly growing portfolio of codes and executables for special applications is avoided and a unified general framework is constructed instead for a broad range of applications. In particular, it was possible to apply SYMPLER to all application examples mentioned in section 8 because molecular dynamics, dissipative particle dynamics, and smoothed particle hydrodynamics methods are supported. Last but not least, SYMPLER was extensively used in lectures and practical labs, which enormously enriched the teaching by providing practical experience to the students.

Acknowledgements

The authors acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG) in the framework of the focused research area *Mikrourformen* (SFB499) and by the University of Freiburg through the German excellence initiative. DK acknowledges funding by the DFG via the project KA 3482/2 *Simulation graphenbasierter Nanoresonatoren: Systematische Reduktion von Freiheitsgraden*. Additionally, we gratefully thank the bwGRiD project [38] for the computational resources. We also acknowledge support in the maintenance and GUI-construction by I. Cenova, P. Dasheva, and P. Videm and fruitful discussions with E. Rudnyi and O. Rübenkönig. Finally, we acknowledge further contributions to SYMPLER not discussed in this paper by all the authors listed in the `AUTHORS` file distributed with the code.

[1] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, The amber biomolecular simulation programs, Journal of Computational Chemistry 26 (16) (2005) 1668–1688. doi:10.1002/jcc.20290.

[2] B. R. Brooks et al., Charmm: The biomolecular simulation program, Journal of Computational Chemistry 30 (10) (2009) 1545–1614. doi:10.1002/jcc.21287.

[3] H. Rafii-Tabar, Modelling the nano-scale phenomena in condensed matter physics via computer-based numerical simulations, Physics Reports 325 (6) (2000) 239–310. doi:10.1016/S0370-1573(99)00087-3.

[4] J. J. Monaghan, Simulating free surface flows with SPH, J. Comp. Phys. 110 (1994) 399–406.

[5] J. P. Morris, P. J. Fox, Y. Zhu, Modeling low reynolds number incompressible flows using SPH, J. Comp. Phys. 136 (1997) 214–226.

[6] S. Vanaverbeke, R. Keppens, S. Poedts, H. Boffin, GRADSPH: A parallel smoothed particle hydrodynamics code for self-gravitating astrophysical fluid dynamics, Comp. Phys. Comm. 180 (2009) 1164–1182. doi:10.1016/j.cpc.2008.12.041.

[7] D. Frenkel, B. Smit, Understanding Molecular Simulation, Academic Press, London, 1996.

[8] P. J. Hoogerbrugge, J. M. V. A. Koelman, Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics, Europhys. Lett. 19 (3) (1992) 155–160.

[9] R. D. Groot, P. B. Warren, Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation, J. Chem. Phys. 107 (11) (1997) 4423–4435.

[10] J. J. Monaghan, Smoothed particle hydrodynamics, Annu. Rev. Astron. Astrophys. 30 (1992) 543–74.

[11] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1) (1995) 1 – 19. doi:10.1006/jcph.1995.1039.
URL `http://lammps.sandia.gov/`

[12] H. Berendsen, D. van der Spoel, R. van Drunen, Gromacs: A message-passing parallel molecular dynamics implementation, Computer Physics Communications 91 (1995) 43 – 56. doi:10.1016/0010-4655(95)00042-E.
URL `http://www.gromacs.org`

[13] M. Gomez-Gesteira, A. Crespo, B. Rogers, R. Dalrymple, J. Dominguez, A. Barreiro, Sphysics – development of a free-surface fluid solver – part

2: Efficiency and test cases, Computers & Geosciences 48 (0) (2012) 300 – 307. doi:10.1016/j.cageo.2012.02.028.
URL http://www.sphysics.org
[14] [link].
URL http://isph.sourceforge.net/
[15] SYMPLER is published under the GPL. [link].
URL http://www.sympler.org
[16] Ennex Corp., The StL Format, http://www.ennex.com/~fabbers/StL.asp.
[17] [link].
URL http://www.wolfram.com/mathematica/
[18] [link].
URL http://www.maplesoft.com/products/maple
[19] [link].
URL http://www.scipy.org
[20] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Addison-Wesley, Reading, 1999.
[21] J. P. Gray, J. J. Monaghan, R. P. Swift, SPH elastic dynamics, Comput. Methods Appl. Mech. Engrg. 190 (2001) 6641–6662.
[22] P. W. Randles, L. D. Libersky, Smoothed particle hydrodynamics: Some recent improvements and applications, Comput. Methods Appl. Mech. Engrg. 139 (1-4) (1996) 375–408.
[23] R. J. Phillips, R. C. Armstrong, R. A. Brown, A. L. Graham, J. R. Abbott, A constitutive equation for concentrated suspensions that accounts for shear-induced particle migration, Phys. Fluids A 4 (1) (1992) 30–40.
[24] D. Kauzlarić, L. Pastewka, H. Meyer, R. Heldele, M. Schulz, O. Weber, V. Piotter, J. Hausselt, A. Greiner, J. G. Korvink, SPH-simulation of shear induced powder migration in injection moulding, Phil. Trans. Royal Soc. A 369 (2011) 2320–2328. doi:10.1098/rsta.2011.0043.
[25] W. W. W. Consortium, Extensible Markup Language (XML), http://www.w3.org/XML/.
[26] The SYMPLER-GUI will soon be available on GitHub. You may also directly contact the authors. [link].
URL https://github.com/kauzlari/sympler
[27] [link].
URL http://rheneas.eng.buffalo.edu/wiki/API
[28] J. B. Avalos, A. D. Mackie, Dissipative particle dynamics with energy conservation, Europhys. Lett. 40 (2) (1997) 141–146.
[29] P. Español, Dissipative particle dynamics with energy conservation, Europhys. Lett. 40 (6) (1997) 631–636.
[30] L. Pastewka, D. Kauzlarić, A. Greiner, J. G. Korvink, Thermostat with a local heat-bath coupling for exact energy conservation in dissipative particle dynamics, Phys. Rev. E 73 (2006) 037701. doi:10.1103/PhysRevE.73.037701.
[31] L. Pastewka, Additional degrees of freedom in modeling microfluidics with dissipative particle dynamics, Master's thesis, Albert-Ludwig University of Freiburg (2005).
[32] D. Kauzlarić, Particle simulation of MEMS/NEMS components and processes – Theory, software design and applications, Microsystem Simulation, Design and Manufacture, IMTEK Freiburg Vol. 1, Der Andere Verlag, Tönning, 2009.
[33] M. Ellero, P. Español, E. G. Flekkøy, Thermodynamically consistent fluid particle model for viscoelastic flows, Phys. Rev. E 68 (2003) 041504.
[34] D. W. Brenner, O. A. Shenderova, J. A. Harrison, S. J. Stuart, B. Ni, S. B. Sinnott, A second-generation reactive empirical bond order (REBO) potential energy expression for hydrocarbons, Journal of Physics: Condensed Matter 14 (4) (2002) 783–802.
[35] [link].
URL http://lammps.sandia.gov/doc/Section_packages.html
[36] Comsol multiphysics, http://www.comsol.com/.
[37] H. P. Langtangen, Computational Partial Differential Equations – Numerical Methods and Diffpack Programming, 2nd Edition, Texts in Computational Science and Engineering, Springer, Berlin, Heidelberg, 1999.
[38] bwGRiD (http://www.bw-grid.de), member of the German D-Grid initiative, funded by the Ministry for Education and Research (Bundesministerium fuer Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Wuerttemberg (Ministerium fuer Wissenschaft, Forschung und Kunst Baden-Wuerttemberg).
[39] M. Dynowski, M. Janczyk, J. Schulz, D. v. Suchodoletz, S. Hermann, Das bwgrid-High Performance Compute Cluster als flexible, verteilte Wissenschaftsinfrastruktur, 5. DFN-Forum Kommunikationstechnologien 5 (2012) 95–105.
[40] accessible in the "Chem" section of the bwGRiD-portal.
[41] [link].
URL http://portal.bw-grid.de
[42] [link].
URL http://www.jcp.org/en/jsr/detail?id=168
[43] R. Heldele, M. Schulz, D. Kauzlarić, J. G. Korvink, J. Hausselt, Micro powder injection molding: process characterization and modeling, Microsys. Technol. 12 (2006) 941–946. doi:10.1007/s00542-006-0117-z.
[44] A. Greiner, D. Kauzlarić, J. G. Korvink, R. Heldele, M. Schulz, V. Piotter, T. Hanemann, O. Weber, J. Haußelt, Simulation of micro powder injection moulding: Powder segregation and yield stress effects during form filling, J. Europ. Ceram. Soc. 31 (14) (2011) 2525–2534. doi:10.1016/j.jeurceramsoc.2011.02.008.
[45] D. Kauzlarić, J. Lienemann, L. Pastewka, A. Greiner, J. G. Korvink, Integrated process simulation of primary shaping: multi scale approaches, Microsys. Technol. 14 (2008) 1789–1796. doi:10.1007/s00542-008-0612-5.
[46] D. Weiß, J. Lienemann, A. Greiner, D. Kauzlarić, J. G. Korvink, SPH based numerical investigation on sessile, oscillating droplets, Phil. Trans. Royal Soc. A 369 (2011) 2565–2573. doi:10.1098/rsta.2011.0077.
[47] J. Lienemann, D. Weiß, A. Greiner, D. Kauzlarić, O. Grünert, J. G. Korvink, Insight into the micro scale dynamics of a micro fluidic wetting-based conveying system by particle based simulation, Microsystem Technologies 18 (2012) 523–530.
[48] I. Cenova, D. Kauzlarić, A. Greiner, J. G. Korvink, Constrained simulations of flow in haemodynamic devices: towards a computational assistance of magnetic resonance imaging measurements, Phil. Trans. Royal Soc. A 369 (1945) (2011) 2494–2501. doi:10.1098/rsta.2011.0028.
[49] M. Azhar, D. Kauzlarić, A. Greiner, J. G. Korvink, Simulation of spin hydrodynamics using dissipative particle dynamics, in: Proceedings of the 6th ECCOMAS Conference on Smart Structures and Materials, 2013.
[50] O. Liba, D. Kauzlarić, Z. R. Abrams, Y. Hanein, A. Greiner, J. G. Korvink, A dissipative particle dynamics model of carbon nanotubes, Molecular Simulation 34 (2008) 737–748. doi:10.1080/08927020802209909.
[51] O. Liba, Y. Hanein, D. Kauzlarić, A. Greiner, J. G. Korvink, Investigation of the mechanical properties of bridged nanotube resonators by dissipative particle dynamics simulation, Int. J. Multiscale Comp. Eng. 6 (2008) 549–562. doi:10.1615/IntJMultCompEng.v6.i6.40.
[52] D. Kauzlarić, J. T. Meier, P. Español, S. Succi, A. Greiner, J. G. Korvink, Bottom-up coarse-graining of a simple graphene model: the blob picture, J. Chem. Phys. 134 (2011) 064106. doi:10.1063/1.3554395.
[53] D. Kauzlarić, P. Español, A. Greiner, S. Succi, Three routes to the friction matrix and their application to the coarse-graining of atomic lattices, Macromol. Theory Simul. 20 (7) (2011) 526–540. doi:10.1002/mats.201100014.
[54] D. Kauzlarić, P. Español, A. Greiner, S. Succi, Markovian dissipative coarse grained molecular dynamics for a simple 2d graphene model, J. Chem. Phys. 137 (23) (2012) 234103. doi:10.1063/1.4771656.
[55] W. Mattson, B. M. Rice, Near-neighbor calculations using a modified cell-linked list method, Computer Physics Communications 119 (1999) 135 – 148. doi:10.1016/S0010-4655(98)00203-3.

Appendix A. Used UML-concepts

In the presented work, the Unified Modelling Language (UML) is used to describe design aspects of the object-oriented software SYMPLER. Here, we briefly explain the concepts used in this paper. A general introduction can be found in [20].

A common strategy of the UML is to show just the relevant information. Therefore, for example, the description of a class is seldomly complete, but rather focussed on certain aspects. The left part of figure A.12 shows how classes are represented in the UML. "Class" is a class name. In this paper, (and in the C++-code of SYMPLER) the rule is applied that a class name starts with a capital letter.

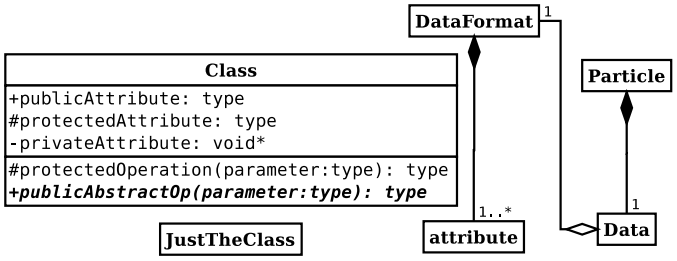

Figure A.12: Left: The box entitled "`Class`" shows the full description of a class. Below, the most compact form of a class representation in the UML is shown. Right: Connections with filled diamonds show a composition, while empty diamonds denote an aggregation. For further explanations see the text.

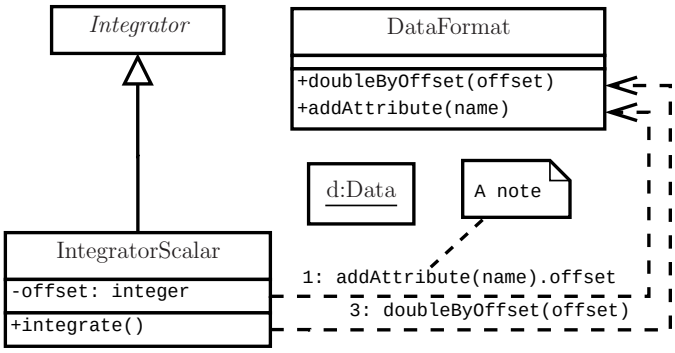

Figure A.13: This class diagram demonstrates how the inheritance of a class `IntegratorScalar` from an abstract class `Integrator` is denoted. Additionally, the diagram shows two dependencies of `IntegratorScalar` on `DataFormat` and the UML-representation of an object, in this case an instance of the class `Data` with name `d`. See the text for further explanations.

In the next section below the class name, class attributes may be listed. Public attributes are marked by a "+", protected attributes by a "#", and private attributes by a "−" in front of the name. Behind the class name, separated by a colon, the type of the attribute may be given as an optional information. The special type `void*` denotes a pointer to an object of arbitrary type. This concept is used for the introduction of arbitrary DOFs in section 4.

Below the attributes, the class operations may be listed in a third section. The distinction among public, protected, and private operations is performed exactly as for attributes. For operations, the type after the colon denotes the return type. In brackets behind the operation's name, one or more arguments may be given, again optionally together with their type. The second operation is written in *italic* style, which denotes that it is a polymorphic function.

The class `JustTheClass` demonstrates the most compact form of a class representation. This compact form is often sufficient if the focus is just on the relation among different classes.

One family of relations is shown on the right of figure A.12. An empty diamond denotes an aggregation, while a full diamond represents a composition. Both relations describe an is-part-of-relationship. The difference is that a composition is stronger, i.e., it is expected that if the "owner" situated on the side of the diamond dies, its "part" on the other side dies as well. In figure A.12 this is the case for the `attribute` being part of the `DataFormat` or the `Data` belonging to the `Particle`. On the other hand, if one `Data` object dies, there might be other `Data` objects left which are described by the same `DataFormat`. In programming practice, a composition is usually realised by letting the owner hold the object itself, while for an aggregation, the owner just holds a pointer to the object (cf. figure 3). The numbers indicate multiplicities. Since `DataFormat` may hold one to many `attributes`, this is denoted by "`1..*`". On the other hand, the fact that each `Particle` holds exactly one `Data` object, and that each `Data` object is related to exactly one description of its `DataFormat` is denoted by the multiplicity "`1`". The class `attribute` is written with small letters, because it is in fact a C-struct, i.e., a class with public members and operations only. The left part of figure A.13 denotes that the class `IntegratorScalar` inherits the properties described in the class `Integrator`. This is indicated by a connection with an empty triangle on the side of the parent class. In the class diagram, `Integrator` is written in *italic* style because it is an abstract class, i.e., it serves as a generalisation and as the description of an interface, but no instances can be directly created from this class.

Generally, dashed arrows indicate dependencies. In this work they are, e.g., used to indicate message passing which is a special type of dependency. By attaching both ends of the dependency at the right positions, one may provide additional information. For example the message `addAttribute` is an operation of `DataFormat` which is called by an instance of `IntegratorScalar` in order to obtain information about the memory offset of the scalar it should integrate (cf. section 4.4). The output of the call is stored in the corresponding member `offset` of the `IntegratorScalar`. Afterwards, it is the operation `integrate`, which uses this information to access the scalar by means of another call to `DataFormat`. The numbering of the messages indicates the order of their execution.

Sometimes, a further explanation or comments are required in a class diagram. For this purpose a "note" may be used such as in figure 6. In figure 9 from section 6 the note is misused to display a list of derived classes in a more compact way.

In the centre, figure A.13 shows the UML-representation of an object, in this case an instance of the class `Data` with name `d`.