Postprint of: Kozicki J., Donzé F.V., A new open-source software developed for numerical simulations using discrete modeling methods, Computer Methods in Applied Mechanics and Engineering, Vol. 197, Iss. 49–50 (2008), pp. 4429-4443, DOI: 10.1016/j.cma.2008.05.023

A new open–source software developed for numerical simulations using discrete modeling methods

J. Kozicki

Gdańsk University of Technology, Civil Engineering Department, Gdańsk-Wrzeszcz 80-952, Narutowicza 11/12, Poland

F.V. Donzé

Laboratoire Sols, Solides, Structures et Risques, Grenoble Universités Domaine Universitaire - B.P. 53 - 38041 Grenoble cedex 9 - France

Abstract

The purpose of this work is to present the development of an open–source software based on a discrete description of matter applied to study the behavior of geomaterials. This software uses Object Oriented Programming techniques, and its methodology design uses three different methods, which are the Discrete Element Method (DEM) [10, 11], the Finite Element Method (FEM) [27, 31] and the Lattice Geometrical Method (LGM) [17–20]. These methods are implemented within a single object–oriented framework in C++ using OOP design patterns. The bulk of the original work consisted mainly of finding common objects which will work for these different modeling methods without changing a single line of the C++ code. With this approach it is possible to add new numerical models by only plugging–in the corresponding formulas. The advantages of the resulting YADE framework are the following: (1) generic design provides great flexibility when adding new scientific simulation code, (2) numerous simulation methods can be coupled within the same framework like for example DEM/FEM and (3) with the open–source philosophy, the community of users collaborate and improve the software. The YADE framework is a new emerging software, which can be downloaded at the `http://yade.wikia.com` webpage.

Key words:
open–source software, generic programming, discrete element method, finite element method, lattice model, geomechanics modeling
PACS:

Email addresses: `jkozicki@pg.gda.pl` (J. Kozicki), `donze@geo.hmg.inpg.fr`

1 Introduction

Developing a simulation software often causes scientists to focus on marginal problems not related to their scientific work, such as: program interface, input/output of data, handling of geometries, mesh generation or visualization of results. One solution is to use existing scientific frameworks, and plug–in one's own calculation algorithms (eg. Abaqus, Dyna, Adina, PFC3D, and others). However these frameworks rarely give the possibility of combining different modeling methods such as FEM, DEM, SPH or other customized simulations like LGM. It is often a commercial software which limits one's ability to improve/modify the existing code–base. A common solution is to write one's own home–brewed software to perform a simulation. Time constraints often cause this software to be very specific for a particular problem, and even when released for the public it is difficult to reuse in another application. Currently existing open–source software (like OOFEM, Salome or SOFA) specializes either in FEM or DEM and none of them allows easy addition of new models or coupling between them. There again, the same drawbacks exist as for commercial software. However the ability to modify or reuse the open–source code makes this problem less important. The problem which we want to address is the lack of a truly versatile open–source software which would provide a stable base for scientists to operate on, regardless of the kind of model they want to work with: DEM, FEM, SPH or other new models, like LGM. With the experience gained from writing the SDEC code [10, 11], prior implementation of LGM and by examining other pieces of work in the field (e.g. OOFEM code or PFC capabilities) we have developed a new YADE framework. We believe its strongest point is that it uses an open–source approach, since it allows direct feedback from the research community. YADE can then grow with time, as more people are using it. What distinguishes YADE from other software is its capability to handle various different numerical models within a single package, which makes it a common research platform and allows coupling of models in between. Also, the visualization and input/output methods are already provided which means the researchers can focus strictly on the formulas. With proper software design the valuable work of others can be preserved and reused.

This paper is organized as follows. Section 2 presents the theoretical background of problems which will be implemented using OOP methods discussed in Sect. 3. Section 4 introduces YADE and Sect. 5 explains the software design used. Finally in Sect. 6 examples using YADE for selected theoretical models are shown as well as a validation case. The validation of these models for other cases can be found in different papers (DEM is validated in [4, 11, 29], FEM/DEM coupling in [14, 27] and LGM in [17–20]).

(F.V. Donzé).

2 Overview of simulation methods

In the overview of the simulation methods, the objective is to find a framework solution which is capable of handling different simulation models to study geomaterials and materials in general. To perform this task, the common parts of various models must be found by analyzing them. In this section we will consider the Discrete Element Method [7, 10, 11, 29], the Lattice Geometrical Model [17–20] and the Finite Element Method [27, 31]. The derivations presented here serve as a background from which the common objects are extracted in following sections and applications are presented in Sect. 6.

2.1 Discrete Element Method

DEM [7,10,11,29] was initiated by Cundall in 1979 [7]. It is a numerical model which represents the discontinous nature of granular materials by a set of discrete elements. Different kind of geometries for the discrete elements can be used in YADE: polyhedric, ellipsoid, spherical or clusters of such elements. However, currently only spheres, clusters of spheres and boxes (walls) were validated while other shapes are under development and will be presented in our future publications. Therefore, only the interaction laws for the simplest geometry, i.e. spherical, is presented below. The proper interactions between elements are defined to account for the mechanical properties of the medium. Thus, the macro–mechanical response of the physical material (deformability, strength, dilatancy, strain localization and other) is reproduced by determining the micro–properties of the material in the contact interaction forces (see Fig. 1), i.e. normal, tangential and rolling stiffnesses, local friction and non–dimensional plastic coefficient (these quantities are defined below). This method provides new insight into constitutive modeling because the physical processes which govern the constitutive behavior can be understood at the local scale. Discrete Elements can have different geometries, but to keep a low calculation cost, the spherical geometry is often chosen and it will be the case here. Let two spherical discrete elements A and B, be in contact. The radii of these elements are r_A and r_B. In the global set of axes, their positions are defined by two vectors $\vec{x}_A$ and $\vec{x}_B$. The interaction force vector $\vec{F}$ which represents the action of element A on element B may be decomposed into a normal and a shear vector $\vec{F}^n$ and $\vec{F}^s$ respectively, which may be classically linked to relative displacements, through normal and tangential stiffnesses, k^n and k^s.

$$\vec{F}_i^n = k^n u^n \vec{n}_i, \tag{1}$$

$$\Delta\vec{F}_i^s = -k^s \Delta\vec{u}^s, \tag{2}$$

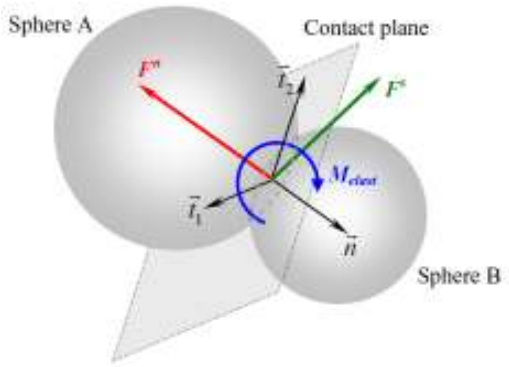


Fig. 1. Interaction between two spherical discrete elements with its normal $\vec{F}^n$, shear $\vec{F}^s$ and moment $\vec{M}_{elast}$ components [4].

where u^n is the relative normal displacement between two elements, $\vec{n}_i$ is the normal contact vector, $\Delta\vec{u}^s$ is the incremental tangential displacement. The shear force $\vec{F}^s$ is obtained by summing the $\Delta\vec{F}^s$ increments. The normal and tangential stiffnesses are given by:

$$k^n = r\frac{K_A^n K_B^n}{K_A^n + K_B^n}, \tag{3}$$

$$k^s = r\frac{K_A^s K_B^s}{K_A^s + K_B^s}, \tag{4}$$

where K_A^n, K_A^s, K_B^n, K_B^s define the input values of normal and tangential stiffnesses for both elements A and B of a contact, r corresponds to the mean value of the two radii.

To reproduce the behavior of non cohesive geomaterials, a Mohr-Coulomb rupture criterion is used:

$$\left|\vec{F}^s\right| \leq \left|\vec{F}^n\right| \tan\mu, \tag{5}$$

where μ is the local friction angle.

Note that, to overcome the simplification of the grain description when using a spherical geometry, an additional component, which is a rolling resistance, can be supplied at each point of contact. The major reason for introducing the rolling resistance is because the representation of the roughness of grains is missing in spherical DEM models. Thus, the results are in very good agreement with experimental results, even if the numerical medium is made of a small amount of particles [4].

YADE is a code based on the Discrete Element Method, using a force–displacement approach, Newton's second law of motion describes the motion of each element

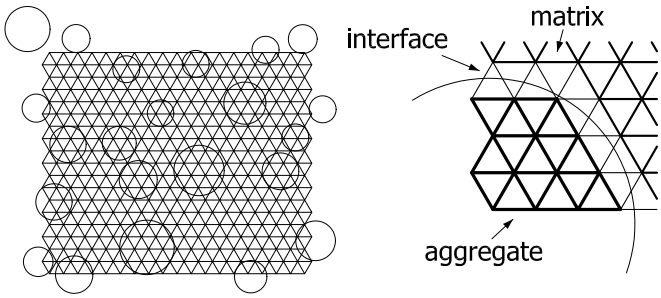


Fig. 2. Lattice of beams for concrete consisting of aggregate, cement matrix and interface [24]

as the sum of all forces applied on this element. The dynamic behavior of the system is solved numerically by a time algorithm in which the velocities and the accelerations are constant at each time step. The system evolves and an explicit finite difference algorithm is used to reproduce this evolution. It is the key feature of DEM, which makes it possible to follow a non–linear interaction of a large number of particles without excessive memory requirements or the need for an iterative procedure.

2.2 Lattice Geometrical Model

Two different types of lattice models exist. In the first type [21, 24, 28, 30] (called lattice beam model), the material is discretized as a network of classical two–noded Bernoulli beams transferring normal forces, shear forces and bending moments which are calculated using a conventional simple beam theory. The heterogeneity of the material is taken into account by assigning different strengths to beams (see Fig. 2 for the case of concrete). Fracture is simulated by performing a linear elastic analysis up to failure under loading and then removing a beam element that exceeds the tensile strength. The advantages of this approach are its simplicity and a direct insight into the fracture process at the micro–structure level. In the second type of models [8] (called particle lattice model), the lattice struts connecting adjacent particles transmit axial and shear forces. The struts are not removed. The shear response of struts exhibits friction and cohesion, and the tensile and shear behavior are sensitive to the confining pressure.

In YADE the Lattice Geometrical Model [17–20] is a discrete model which is close to the first type model, but it consists of rods with flexible nodes and longitudinal deformability, rotating in the form of a rigid body rotation. Thus, shearing, bending and torsion are represented by a change of the angle between rod elements connected by angular springs. This model is of a kinematic type. The calculations of element displacements are carried out on the basis of the consideration of successive geometrical changes of rods due to translation, ro-

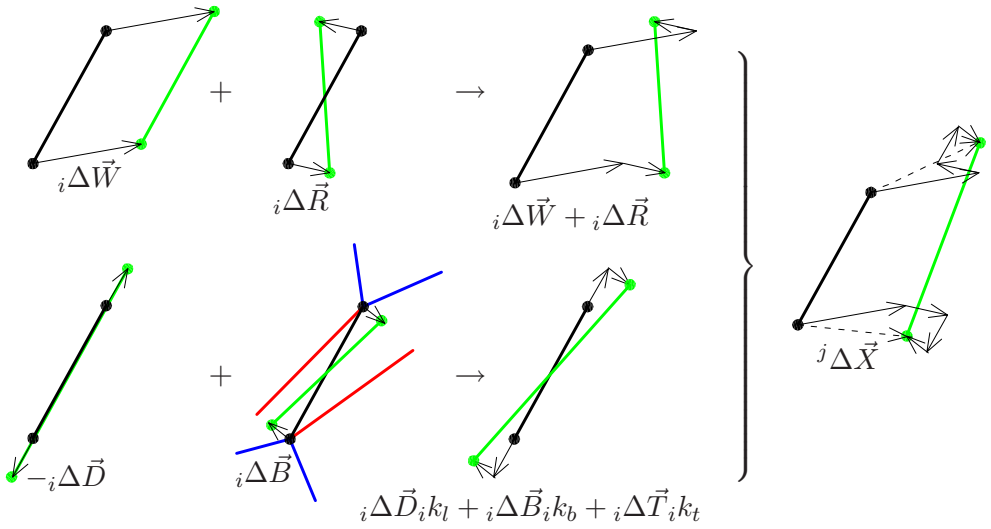


Fig. 3. General scheme to calculate displacements of elements in the Lattice Geometrical Model (torsional component $_i\Delta\vec{T}$ is not shown)

tation and normal and bending deformation. Thus, the global stiffness matrix does not need to be built and the calculation method has a purely explicit character. In contrast to the beam model [21], torsion in three–dimensional simulations is included as well. The material was discretized in the form of a 3D lattice mesh using a Delaunay's construction scheme (each Delaunay's edge is a lattice rod element). The elements possessed a longitudinal stiffness described by the parameter k_l (controls the changes of the element length), a bending stiffness described by the parameter k_b (controls the changes of the angle between elements) and a torsional stiffness described by the parameter k_t (controls the changes of the torsional angle between elements).

The displacement of the center of each rod element was calculated as the average displacement of its two end nodes from the previous iteration step:

$$_i\Delta\vec{X} = \frac{^A\Delta\vec{X} + {}^B\Delta\vec{X}}{2}, \tag{6}$$

wherein $^A\Delta\vec{X}$ and $^B\Delta\vec{X}$ – displacement of the end nodes A and B in the rod element i, respectively. The displacement vector of each element node was obtained by averaging the displacements of the end of elements belonging to this node caused by translation, rotation, normal, bending and torsional deformation (Fig. 3):

$$^j\Delta\vec{X} = \sum_i \frac{_i\Delta\vec{W} + {}_i\Delta\vec{R}}{^j n_{sum}} + \frac{\sum_i \frac{1}{_i d_{init}} \left({}_i\Delta\vec{D}_i k_l + {}_i\Delta\vec{B}_i k_b + {}_i\Delta\vec{T}_i k_t \right)}{\sum_i \frac{1}{_i d_{init}} \left({}_i k_l + {}_i k_b + {}_i k_t \right)}, \tag{7}$$

wherein: ${}^{j}\Delta\vec{X}$ – resultant node displacement, ${}_{i}\Delta\vec{W}$ – node displacement due to the rod translation, ${}_{i}\Delta\vec{R}$ – node displacement due to the rod rotation, ${}_{i}\Delta\vec{D}$ – node displacement due to a change of the rod length (induced by k_l), ${}_{i}\Delta\vec{B}$ – node displacement due to a change of the angle between rods (induced by k_b), ${}_{i}\Delta\vec{T}$ – node displacement due to torsion between two neighboring rods (induced by k_t), k_l – longitudinal stiffness, k_b – bending stiffness, k_t – torsional stiffness, i – successive rod number connected with node j, j – node number, ${}^{j}n_{sum}$ – number of rods belonging to node j and ${}_{i}d_{init}$ – initial length of rod i.

The node displacements were calculated successively during each calculation step beginning first with the elements along the boundaries which are subjected to prescribed displacements. By applying Eq. 7, a full strain equilibrium was obtained in each node (this required about 10 iterations). The resultant force F in a selected specimen's cross–section A was determined with the help of corresponding normal strain ε, shear strain γ, stiffness parameters k_l, k_b, modulus of elasticity E, shear modulus G and the specimen's cross–section A:

$$F = A \sum \left(\varepsilon k_l E + \gamma k_b G\right), \tag{8}$$

where the sum is made over all elements that intersect a selected specimen's cross–section and the strains ε and γ are projected on the section normal vector. For the bending stiffness parameter $k_b = 0$ in Eq. 7, the elements behaved as classical bars. Each element was removed from the lattice if the assumed local critical tensile strain $\varepsilon_{\min}$ was exceeded. Numerical calculations were strain controlled. For a complete description of the model see [17–20].

2.3 Coupling the Discrete Element Method with the Finite Element Method

FEM has been implemented in YADE because the analysis of large structures with the DEM proves difficult, as the computation time increases with the number of Discrete Elements. A number of authors have pointed out that such a method is limited to small structures due to the computation cost of DEM. Use of the FEM outside the area described by the DEM represents one way of minimizing this constraint since in most cases, severe deformation phenomena, such as fragmentation or particle flow, are localized. In addition, the FEM is widely used, and an efficient mesh generation software already exists that can dramatically reduce modeling duration, with the potential for faster calculations than when applying a full DEM approach because various discretization sizes can then be handled. These facts naturally lead to proposing a coupled FEM/DEM approach. Such a coupling is based on partitioning the structure into two sub–domains: an initial FEM domain where nonlinear phenomena may be neglected, and a second DEM domain where severe nonlinear degradation phenomena may occur

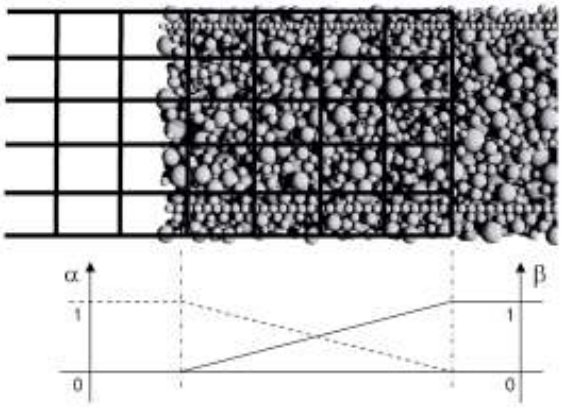


Fig. 4. Bridging domain and bridging parameter

A coupling method based on a bridging domain (Fig. 4) with energy weighting has been developed by Xiao and Belytschko [31]. This method proposes minimizing the Hamiltonian (H), which is the sum of the Hamiltonians of both the FE and DE.

$$H = \alpha H_{FE} + \beta H_{DE} \quad \text{with} \quad H = E_k + E_p \quad \text{and} \quad \alpha + \beta = 1, \tag{9}$$

where α and β are the weight parameter of the FEM Hamiltonian (H_{FE}) and the DEM Hamiltonian (H_{DE}), respectively. Parameters α and β are defined in Fig. 4, which presents an example of the bridging domain with four FEM layers, and E_k and E_p are kinetic and potential energy respectively. To guarantee kinematic continuity, the degrees of freedom of both domains within the interface zone must be linked. Several approaches could be considered. Xiao and Belytschko [31] proposed directly linking the Discrete Element degrees of freedom with Finite Element degrees of freedom using Lagrange multipliers. Eq. 10 presents the kinematic conditions on bridging domain r, with d DE displacements and u FE displacements:

$$\vec{d_r} = \bar{\bar{k}}\vec{u}_r. \tag{10}$$

The Arlequin method [12] uses displacement and strain projections over shape functions, in assuming field continuity. This method enables a spatial relaxation of the kinematic conditions. With Discrete Element models, the displacement and strain are not known at all points and such projections are not simple to use. This method may be adapted to DEM models, by approximating the displacement and deformation fields, although this would increase computation time. The preferred method therefore calls for using rigid kinematic conditions, like those in [31]. Complementary kinematic constraints must then be added to link the DE rotations. The following equation lists the kinematic conditions linking DE rotations ω and displacements d with FE displacements

u over bridging domain r:

$$\vec{d_r} = \bar{\bar{k}}\vec{u}_r \quad \text{and} \quad \vec{\omega}_r = \bar{\bar{h}}\vec{u}_r. \tag{11}$$

The solution has been derived by minimizing Eq. 12, with kinematic conditions being taken in account using Lagrange multipliers λ^d for displacements and λ^ω for rotations.

$$H_g = H + \vec{\lambda}^d \left(\vec{d_r} - \bar{\bar{k}}\vec{u}_r\right) + \vec{\lambda}^\omega \left(\vec{\omega}_r - \bar{\bar{h}}\vec{u}_r\right). \tag{12}$$

The time discretization relies upon an explicit scheme:

$$u(t + \Delta t) = 2u(t) - u(t - \Delta t) + \Delta t^2 \ddot{u}(t). \tag{13}$$

Each degree of freedom value is calculated without taking coupling into account. Then a correction is applied using the Lagrange Multipliers (Eq. 12). The difference in discretization size between FE and DE may induce wave reflection at the interface. This effect can be mitigated in different ways, e.g. by damping, yet the choice of a damping coefficient is not straightforward. We have proposed a method to attenuate the reflection by introducing a reduction parameter for the Lagrange multiplier influence. This method leads to a temporal relaxation of the kinematic constraints and is equivalent to a penalty method with an automatic process for optimizing the penalty parameter. All of these methods are discussed in [14].

3 Introducing the Object Oriented Architecture

The examples in the previous section show that some simulation elements (be it spheres, FEM elements or rods) interact using physical rules that involve calculating a force or displacement due to interaction. The interaction may need to be detected (also over a distance), or can be pre–determined. It can be of a different kind such as: a collision, a cohesive link or being governed by a stiffness matrix or a non–local dependency [3, 17]. To allow co–existence of such various methods in a single simulation framework their similarities must be discovered and transformed to an abstract generalization. However, differences between them must be allowed to be implemented as a particular concrete realization of this abstraction. To solve this problem correctly, the techniques offered by object oriented programming must be carefully used as seen in the following section.

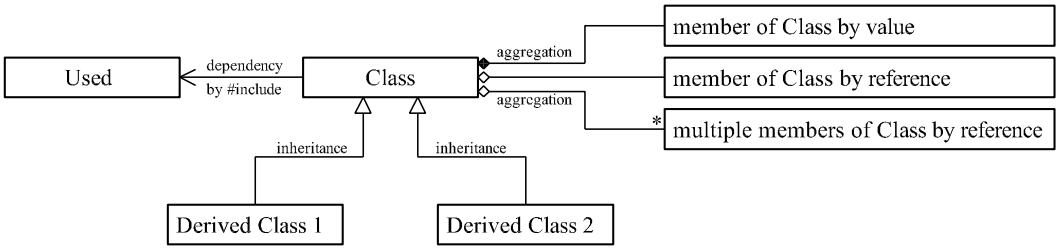


Fig. 5. Unified Modelling Language — representation of class relationships.

3.1 The UML notation

The Unified Modelling Language [5] is used to represent classes and associations between them. It is a language that unifies the industry's engineering practices for describing object oriented designs. A small subset of UML used in this article is shown in Fig 5, it represents classes and their relations. In this article *CapitalizedItalics* are used to denote a class.

3.2 Language choice

It is well known that Object Oriented languages are the most useful if developing highly generic and library reusable code. Two languages that provide OO programming paradigms were considered: Java and C++. Criterions used for choosing are the speed of code execution and platform interoperability. A small application was written which performs a Sweep and Prune [6] collision test for DEM applications. Only small changes were needed to port collision algorithm from one language to another. The codes were compiled with maximum optimization, then their speed was compared. Java was three times slower [1] than C++. Another widely used .NET platform was not considered because of poor platform interoperability (no support for unix, solaris, irix or linux that usually power the supercomputers), and vendor lock–in.

3.3 Generic programming approach

One of the reasons for a code to be highly specific is the tight coupling between the underlying data structures and algorithms operating on them. It is possible however, using the generic programming approach [2, 15], to separate data structures from algorithms. A good example of this is the C++ Standard Template Library (STL) [16, 23]. The STL library provides container classes modeling linear sequences like vector, map, list, set, etc. It also provides generic

[1] see `http://svn.berlios.de/viewcvs/yade/snippets/trunk/JavaVsC++`

versions of linear sequence algorithms such as counting, sorting, copying, etc. The containers are generic in the sense that they are independent on the type of items they hold, eg. a `float` or a specific class *Box* (content agnostic). Algorithms are generic in the sense that they are independent on the container type, eg. a map or a list (container agnostic). This approach effectively makes for example the sorting algorithm to be content agnostic.

Presented here YADE framework splits the data classes and algorithm classes (called *engines*) in a similar way. Data storage classes are accompanied by *engine* classes that calculate their state, their interactions or draw them on the screen using OpenGL. Because *engines* are separated from *data*, it is easy to mix them with another subset of *engines* written by different user with more specific requirements. Whereas *data* classes still remain stable with consistent interface.

Principles of Object Oriented Class Design

In [22] Robert C. Martin has described symptoms of rotten Object Oriented design, like rigidity (software is difficult to change, because one change requires many modifications), fragility (software easily breaks on smallest change), immobility (inability to reuse software from other projects), viscosity (easy to make small modification as hack, difficult to make it in proper way). Then the cause for those problems was identified as constantly changing requirements and poor dependency management. Solutions for these problems were presented, which are the base for OO design, and are briefly described in the following paragraphs.

The Open Closed Principle (OCP) is that *"A module should be open for extension but closed for modification"*. In other words it should be possible to change what modules do, without changing the source code of the modules. This sounds contradictory but is possible with techniques based on *abstraction*: dynamic polymorphism (virtual methods, inheritance) and static polymorphism (templates). This way, a code that already works is not changed, so it can't be broken.

The Liskov Substitution Principle (LSP) *"All subclasses should be substitutable for their base classes"*, means that a user of a base class should continue to work properly if a derivative of that base class is passed to it. This is possible when derived methods do not expect more conditions or parameters than their base classes and as a result give no less than its base class.

The Dependency Inversion Principle (DIP) *"Depend on Abstractions. Do not depend on concretions"* depicts the strategy of depending upon interfaces or abstract functions and classes. This principle is the enabling force behind

component design (COM, CORBA). No dependency should target concrete class.

The Interface Segregation Principle (ISP) *"Many client specific interfaces are better than one general purpose interface"* says that instead of adding numerous methods to single class it is better to dedicate one class to each client or task.

3.4 Object Oriented Design Patterns

When following the principles described above [22] to create OO architecture, one finds, that one repeats the same structures over and over again. Those repeating structures of design and architecture are known as design patterns. A design pattern is a well known good solution to a common problem. Since many design patterns exist [2,15,22], only those of great importance for YADE design will be briefly described.

Generalized Functors [2,15] present a powerful abstraction that allows decoupled inter–object communication. In brief they are any processing invocation that C++ allows, encapsulated as a typesafe first–class object. They allow to store processing requests as values, pass them as parameters, and invoke them apart from their creation. Generalized functions are a much modernized version of pointers to functions and are important part of Multimethod and Command patterns described below.

The Command Pattern [15, 22] encloses a request for a specific action inside an object and gives it a known public interface. Important purpose of the command pattern is to keep the program and user interface completely separated from the actions they initiate. In YADE abstract interface class named *Engine* provides two abstract virtual methods: *bool activated()* and *void action()*. Each plug–in that provides new algorithm inherits from it, or from its derived classes.

Typelists [2] are a C++ tool for manipulating collection of types. They offer for types all the fundamental operations that lists of values support. When using traditional coding techniques, manipulating types is possible only by sheer repetition (eg. by `switch`–ing `dynamic_cast`–s). Most people don't think it could get any better than that. Typelists bring power from functional programming paradigm to C++, enabling it to support new interesting idioms. Notable examples are Visitor and Multimethods.

The Visitor Pattern [15] is a powerful – if controversial – design pattern that changes the dependency trade–offs involved in class design. It gives surprising flexibility in certain area: adding virtual functions to classes without recom-

piling them or their clients. This flexibility comes at the expense of disabling features that designers take for granted: adding new leaf to the class hierarchy is not possible without recompiling the hierarchy and all its clients. Therefore Visitor's operational area is limited to very stable hierarchies (new data classes are seldom added), and heavy processing needs (virtual functions are often added). YADE is designed to have stable core and data hierarchy, while allowing easy adding of new scientific algorithms. In this implementation Visitor pattern appears as a one dimensional Multimethod (see Sect. 5.2 and Fig. 15). In YADE all *engine* classes are either commands (eg. *EgiConstitutiveLaw*, *EgiConditionApplier*) or multimethod dispatchers (*EngineDispatcher*).

The Multimethod Pattern [2] allows dispatching of a function call depending on types of **multiple** objects. The standard C++ virtual function mechanism allows dispatching of a call depending on the dynamic type of **single** object. A universally good implementation requires language support (eg. ML or Haskell). C++ lacks such support, so its emulation is left to library writers. When operation on several polymorphic objects (class *Shape*) should exhibit behavior varying with the dynamic type of more than one of those objects (class *Sphere* or *Box*), the need for multimethods arises. Collision is a typical category of problems best solved with multimethods. For example different code handles *Shape* collision between *Sphere* and *Sphere*, than between a *Sphere* and a *Box*. Alexandrescu [2] has provided several solutions with various trade–offs in speed, flexibility, and dependency management. In the YADE design there is a need for absolute speed and absolute scalability, which comes at a price. Each base class for multivirtual call must hold an index, which is done by inheriting from class *Indexable*. This index serves as a coordinate in multidimensional callback matrix (held by class *EngineDispatcher*), in which generalized functors (class *EngineFunctor*) are stored. The solution proposed by Alexandrescu was extended with recursive templates, and up to four dimensions are supported. In YADE multimethods have important role for providing abstract generalization, which is explained in Sect. 5.2 and in Fig. 16.

4 Constructing the framework

The purpose of the YADE framework is to provide stable and uniform environment for scientists to implement computational algorithms. It allows easy code reuse, exchange and extensibility, while also providing many common low–level operations, through plug–ins and libraries. Given that, scientists can now focus on their work instead of reinventing the wheel of input/output or display. The YADE framework is divided into several layers shown in Fig. 6. Each layer can depend on layers below it. Libraries in the lowest layer are not related to the simulation itself, and can be used by other software.

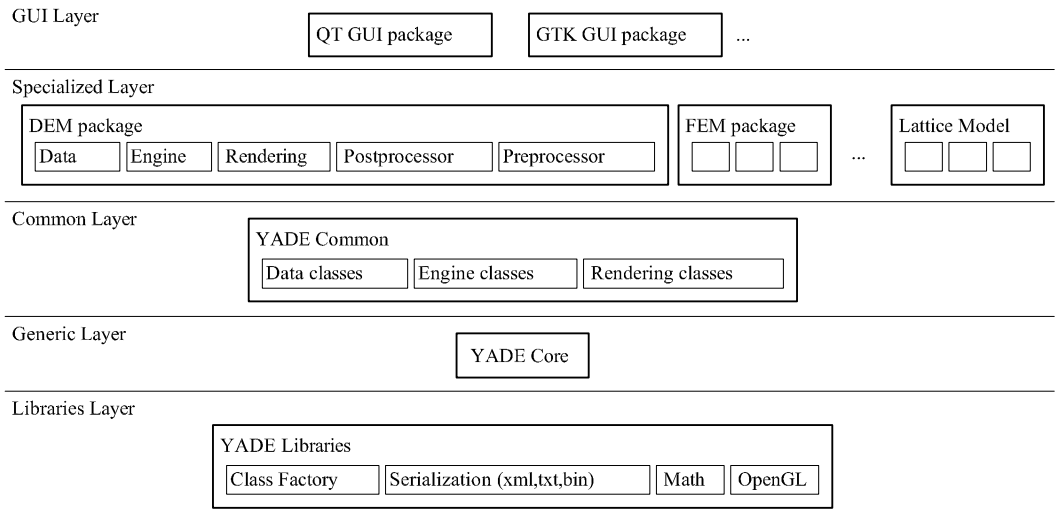


Fig. 6. Layered structure of YADE framework.

Class Factory (lowest layer in Fig. 6) is a C++ wrapper for dynamic linking loader (`dlopen()`, etc.). It handles loading and unloading plugins given their class name as a string, after which plug–in file on the hard drive is named. Since it works during runtime, it is easy to switch between different concrete implementations of currently tested class, such as: different plugins to solve or detect body interactions, different methods of drawing graphics with OpenGL, or saving results to xml or binary format – which comes handy when benchmarking and testing during development. Plug–ins are inheriting from the class *Factorable*.

The Serialization library supports de/serializaing data with random access to class components during the process, easily human readable xml format and support for creation of new formats (like txt, yaml or binary). With this library it is recommended to inherit from class *Serializable* to obtain an easy to use serialization interface. In the future the `boost::serialization` [1] library will be used.

The Math library provides quaternion, vector and small matrices calculus optimized for 3D operations.

OpenGL library provides a C++ wrapper for glut. Other libraries used in the framework are: STL [16, 23], Boost [1] and QGLViewer [9].

The generic layer in Fig. 6 represents the core of YADE and provides abstract interfaces to all concepts of scientific simulation: *engines*, *bodies* and *interactions* (see Sect. 5). Class *World* (see Sect. 5.3) stores the simulated world, increments iterations and synchronizes threads. The abstract interfaces for GUI and rendering are also here.

The YADE common layer in Fig. 6 contains components commonly used by

various simulation types (DEM, FEM, Lattice or SPH), like:

- Newton's law or Hooke's law,
- time integration algorithms (Leapfrog [13], Newmark, Runge–Kutta 4, etc.),
- damping methods (eg. Cundall non viscous damping [7]),
- collision detection algorithms (eg. Sweep and Prune [6] or a grid based collision detection),
- boundary conditions (imposing translation, applying gravity, etc.),
- *data* classes that store information about bodies or interactions.
- Common OpenGL methods for drawing popular geometries,
- a common Multimethod interface – the sensibly expected *EngineDispatcher*s that can call *EngineFunctor*s from specialized layers.

The specialized layer is based on the common layer. It contains code that cannot be shared between different methods (see Sect. 2). Many specialized packages can exist – implementation of three examples from Sect. 2 are explained in Sect. 6.

The top layer is a Graphical User Interface and one based on QT is currently provided, also GTK, ncurses or even winAPI are possible as plugins. Moreover, a command line interface with python scripting language can be used to perform computations remotely.

5 Common objects underlying scientific simulation

Consider that the simulation involves *bodies* between which *interactions* occur (Fig 7). These interactions can be detected and processed by certain computational algorithms and *physical rules* (which are *engines* in general). The result of these algorithms can be a moment, a force, a displacement, etc. (class *BodyExternalVariables*), which in general produce a *response* that affects *body state*. All *bodies*, *interactions* and the simulation loop that processes them (*engines*) are stored inside the *World* class.

In general three kinds of *data* are distinguished:

- *bodies*,
- *interactions*,
- *intermediate data*.

All algorithms are *engines*, but they have been divided to:

- Command Pattern: *Engine*,
- Multimethods Pattern: *EngineFunctor* stored inside *EngineDispatcher*.

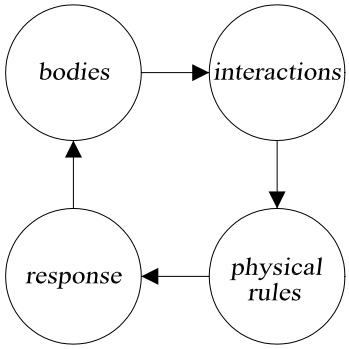


Fig. 7. Simplified schematics of simulation loop.

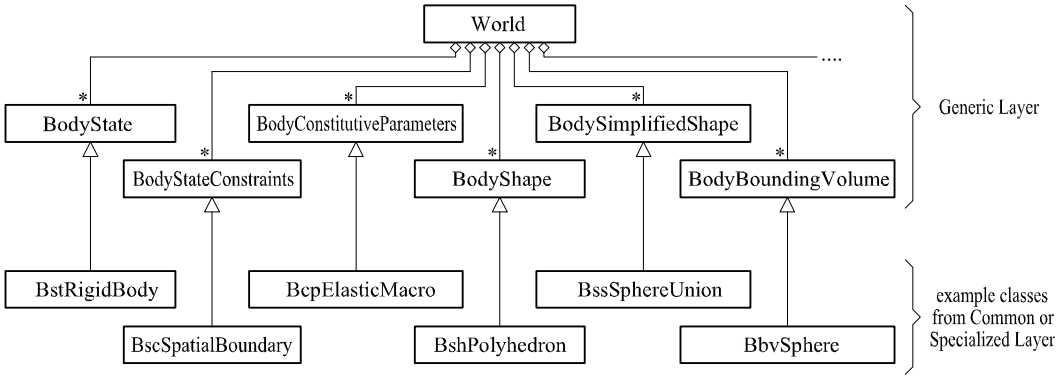


Fig. 8. Information about bodies stored in *World* class, with example derived classes.

5.1 Data classes

The objects of *data* classes cannot move or interact themselves, as they only contain data. Their movement and interaction are handled by the *engine* classes. The *body* is represented by six *data* classes: *BodyState*, *BodyStateConstraints*, *BodyConstitutiveParameters*, *BodyShape*, *BodySimplifiedShape* and *BodyBoundingVolume* (Fig. 8). They are held inside *World* using `boost::multi_index` container. The seemingly obvious notion to create a *Body* class that would hold all six of them proved to be wrong, since a single *body* can sometimes be described by multiple instances of *BodyShape* (eg. a DEM cluster) or thousands of *bodies* can share a single instance of *BodyConstitutiveParameters* (eg. some are the concrete, others are the reinforcement). The purpose of those six abstract data classes follows (see examples in Fig. 9):

BodyState *(Bst)* – information about a body that changes during the simulation process and is different for each instance of a body in the simulation, like position, velocity, acceleration and mass or inertia.

BodyStateConstraints *(Bsc)* – information about constraints imposed on a body state. A constrained value can be eg.: kept at limiting value, determine a body deletion etc. Example constraints include: maximum strain, crossing spatial boundary or a sliding support. Many bodies can use the

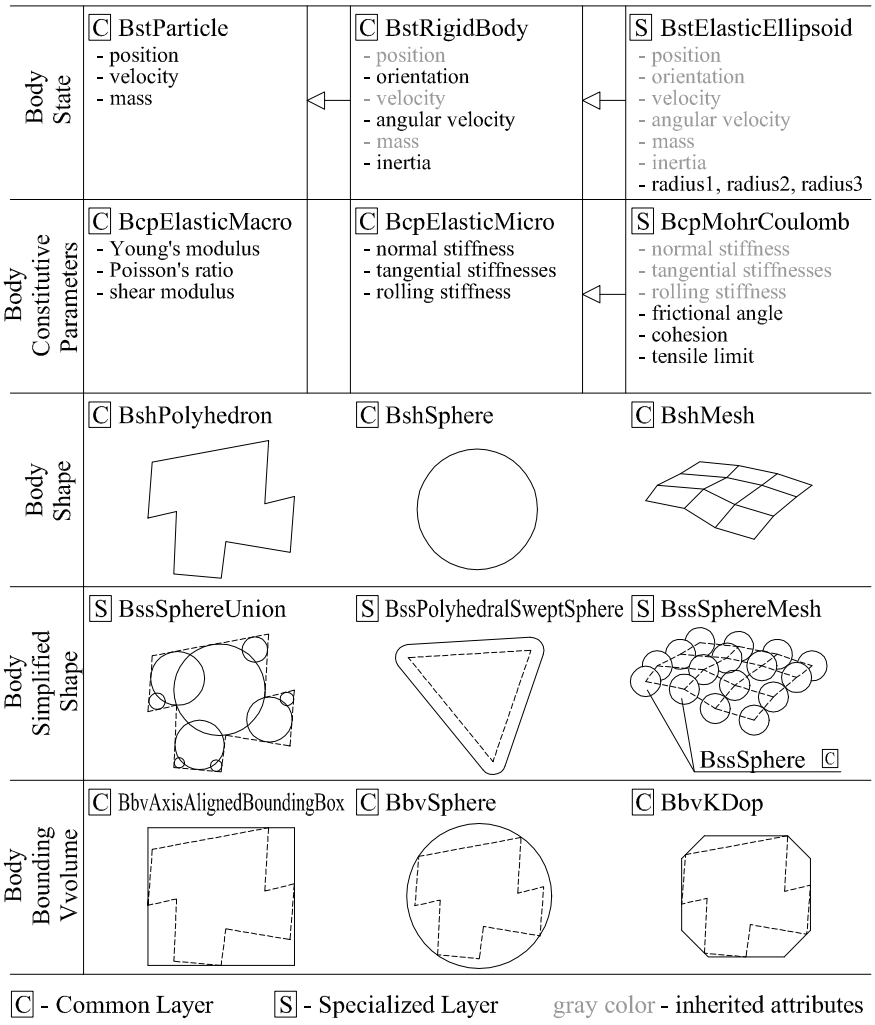


Fig. 9. Examples of concrete classes that describe a *body*

same constraints or not use constraints at all.

BodyConstitutiveParameters *(Bcp)* – information about a body that usually does not change during the simulation and is the same for many instances of bodies. It is intended to be an information used by constitutive laws, like stiffness or cohesion.

BodyShape *(Bsh)* – the idealized geometrical shape of a body that is simulated: it is used to create a simplified shape, and for display.

BodySimplifiedShape *(Bss)* – a shape of the body used for performing the actual simulation, may be different from idealized shape, because it is merely its representation used for the purpose of the simulation.

BodyBoundingVolume *(Bbv)* – a bounding volume is used to detect potential interaction between bodies, usually is built from information stored inside simplified shape.

The *interaction* is represented by two data classes: *InteractionState* and *InteractionConstitutiveParameters* (Fig. 10). To store them, `boost::multi_index`

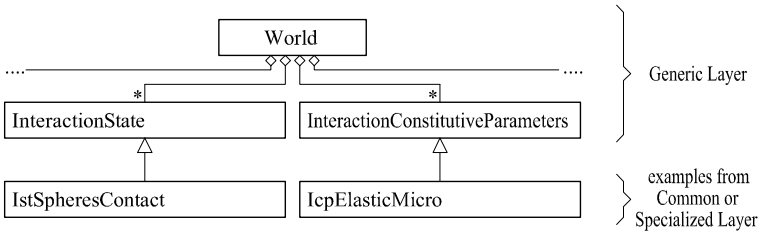


Fig. 10. Description of *interactions* stored in *World* class, with examples of concrete derived classes.

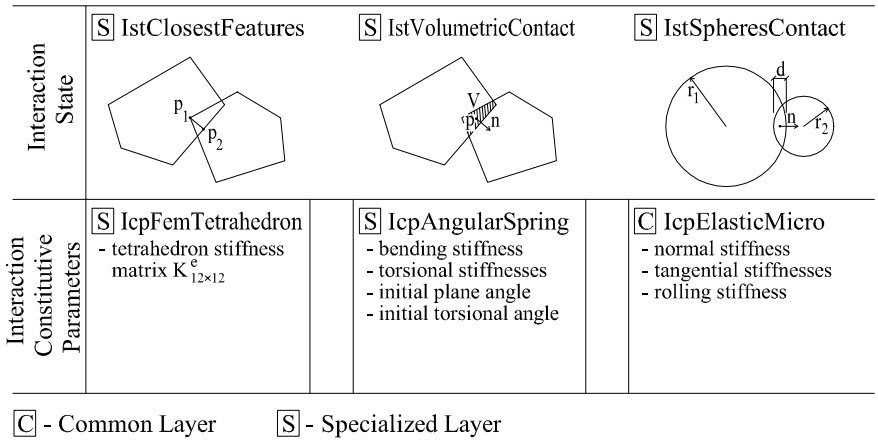


Fig. 11. Examples of concrete classes that describe an *interaction*

is used. They serve following purposes (see examples in Fig. 11):

InteractionState *(Ist)* – information about an interaction happening between bodies which changes while the interaction evolves during the simulation (eg.: penetration depth, shearing force, contact points or volume of contact V).

InteractionConstitutiveParameters *(Icp)* – information about an interaction happening between bodies which usually does not change during the simulation, even when bodies disconnect and reconnect again (eg.: contact stiffness).

Finally two data classes contain intermediate data, those are: *BodyExternalVariables* and *OutputData* (Figs. 12 and 13):

BodyExternalVariables *(Bex)* – this information is an intermediate stage to calculate future values of *BodyState* for the next execution of simulation loop. Usually it contains the sum of effects calculated by some *physical rules*. For example a sum of forces and moments acting on a sphere is used to change body's position and orientation. It is discussed separately from other *BodyXxxx* classes, because it does not describe *bodies* themselves — just changes to them.

OutputData *(Odt)* – this data is used to store results that cannot be directly obtained from other *data* classes. Usually some *Engine* will interpret

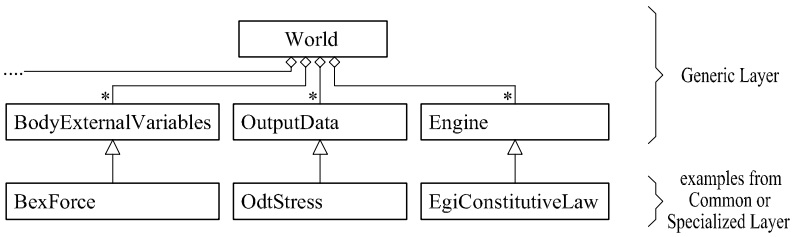


Fig. 12. The *intermediate data* classes stored in *World*, with examples of concrete derived classes. The *Engine* class is shown here only for completeness and is discussed further.

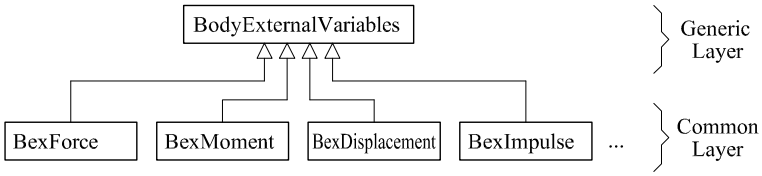


Fig. 13. Abstract class *BodyExternalVariables*, with examples of concrete derived classes.

necessary data and store it here, eg.: an averaged stress, a number of bodies that fulfill some criterion, etc.

To store all *data* classes, the `boost::multi_index` container is used. It allows to cross–reference class instances and to iterate over data elements with respect to different keys. Eg.: to iterate over all *bodies* involved in a selected *interaction* or alternatively to iterate over all *interactions* in which a given *body* takes part — a different view on the very same data.

5.2 Engine classes

Every operation concerning *data* is performed by a dedicated *Engine*. Creating, modifying, destroying, displaying, loading, saving, calculating, converting, interpreting — all those functions are performed by some specific *Engine* class. Figures 14–16 show some example classes of two kinds: commands and multimethods.

The Command Pattern classes (deriving from *Engine*) have some empty subcategories serving to help organize the *engines* derivation tree. Concrete implementations of algorithms are inheriting from them:

EgiConditionApplier *(Econ)* – performs tasks that depend on conditions from outside, like: applying force as a *boundary condition* of the simulation or imposing a kinematic translation according to data read from file on hard disk.

EgiBoundingVolumeCollider *(Ebvc)* – detects collisions using various al-

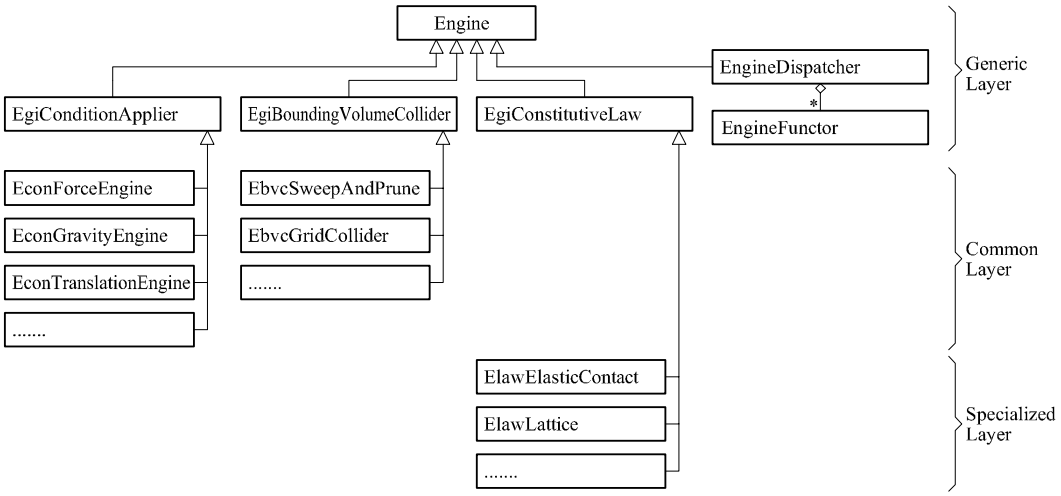


Fig. 14. Class *Engine* and example algorithms inheriting from it

gorithms, like Sweep and Prune [6],

EgiConstitutiveLaw *(Elaw)* – the constitutive law for any given calculation method (compare with Sect. 2), eg. *ElawElasticContact* used for DEM (Eq. 1–2) or *ElawLattice* for LGM (Eq. 7).

EgiTimeStepper *(Etim)* – methods for choosing the optimal time step if the simulation is dynamic, it can be based on maximum velocity of *bodies*, their mass and stiffnes or other criteria.

EgiDataProcessor *(Edat)* – methods for calculating any results which are to be stored in *OutputData*.

Adding more *Engine* subcategories is implied by design flexibility and will happen during the framework evolution.

The Multimethod Pattern dispatchers (class *EngineDispatcher*) perform tasks which are specific to some kind (or kinds) of *data*. Elements of *data* are distinguished by their *Indexable* class index. This provides an emulation of (multi)virtual functions for any given *data* class.

Addition of new plug–ins operating on data classes is possible by writing only two files: `.hpp` and `.cpp` with short code inside. A convention for naming those

Table 1
Adaptation of naming syntax of C++ member virtual functions for naming of Multimethod *EngineFunctor* classes

dimension	**C++ naming syntax**	***EngineFunctor***, *(F•)*
1	`Class::function()`	`Ef1_Class_function()`
2	`Class1::Class2::function()`*	`Ef2_Class1_Class2_function()`
⋮		

*this name is not allowed as a C++ language construct

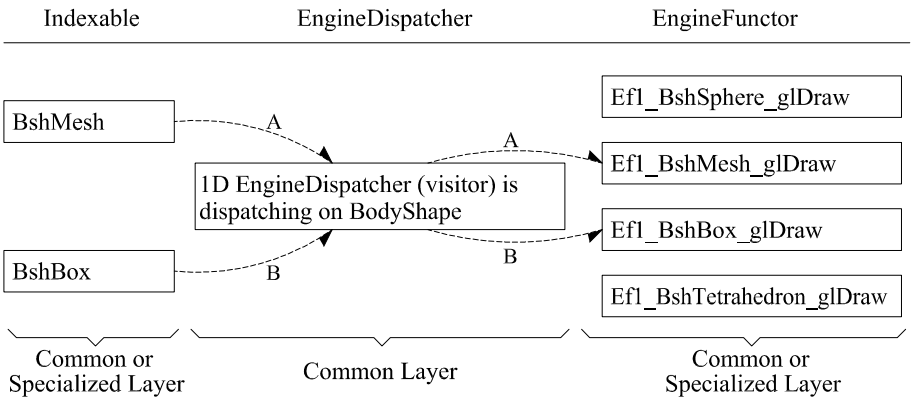


Fig. 15. One dimensional *EngineDispatcher* — two example calls **A** and **B** are correctly dispatched to perform OpenGL draw of a *BodyShape*.

plug–ins had to be assumed. Table 1 presents how the C++ member functions naming syntax was adapted for naming of multimethod constructs which are not supported by the C++ language itself.

Two classes play an important role here:

EngineDispatcher *(Ed•)* – the dispatcher is implemented in YADE common layer for all 1D and 2D variants currently used in specialized layers. The • indicates the number of dimensions, if necessary a higher number of dimensions is possible.

EngineFunctor *(Ef•)* – this is a parent class for concrete code used in multimethod pattern, the • indicates the number of dimensions.

With all *data* classes there are 100 two–dimensional *EngineFunctor* families possible to create. Categorizing them into few topics similarly as in command pattern was not successful yet. New *EngineFunctor*-s are added to the code assuming that in the future some categorization will become possible.

Figure 15 shows two example calls in one dimensional multimethod (a visitor) for drawing a *BodyShape*. Different code is executed for drawing a *BshMesh* and a *BshBox*. Few more examples are listed below:

Ef1_BbvAxisAlignedBoundingBox_glDraw,
Ef1_BbvSphere_glDraw – methods that draw OpenGL bounding volumes depending on the type of *BodyBoundingVolume*. An attempt to unify *BbvSphere* with *BshSphere* (since both are spheres) is a subject for further research, eg. by multiple inheritance from single class *Sphere*.

Ef1_BstParticle_rungeKutta4Integrator,
Ef1_BstRigidBody_rungeKutt4aIntegrator,
Ef1_BstParticle_leapFrogIntegrator,
Ef1_BstRigidBody_leapFrogIntegrator – example time integration methods: Leapfrog [13] and Runge Kutta 4 implemented for position/velocity

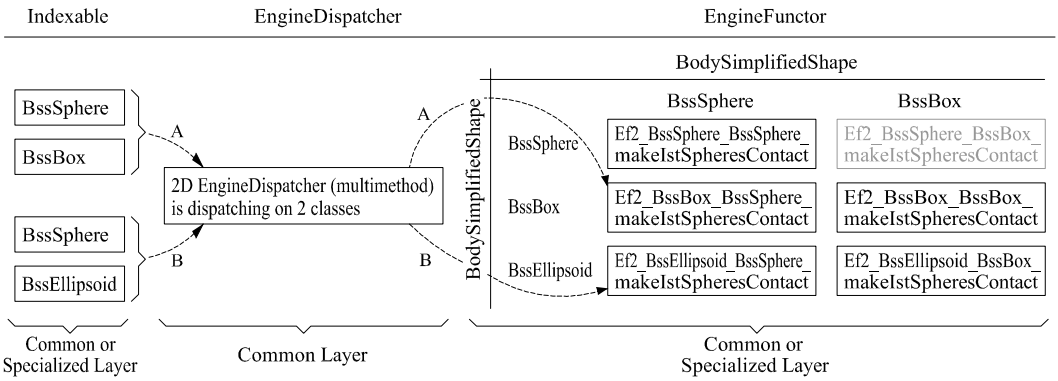


Fig. 16. Two dimensional *EngineDispatcher* — two example calls **A** and **B** are correctly dispatched to create collision *data* (in this example stored inside class *IstSpheresContact* – see Fig. 11) between two *BodySimplifiedShape*-s. Symmetry is automatically detected (gray color).

(class *BstParticle*) and orientation/angular velocity (class *BstRigidBody*).

Figure 16 shows two example calls in a two dimensional multimethod for processing collision between two *BodyShape*-s. An *InteractionState* data class (named *IstSpheresContact*, Fig. 11) is created. The code to create it differs depending on what bodies collide, a *BssSphere* with a *BssBox* or a *BssEllipsoid*. Few more examples follow:

Ef2_BcpElasticMicro_BcpElasticMicro_makeIcpElasticMicro
Ef2_BcpMohrCoulomb_BcpMohrColumb_makeIcpMohrCoulomb – create the *InteractionConstitutiveParameters* contact information class *IcpElasticMicro* (or *IcpMohrCoulomb*), from data stored in two colliding bodies that use *BcpElasticMicro* (or *BcpMohrCoulomb*) constitutive parameters (Eq. 3–4). In the case when colliding *bodies* use *BcpElasticMicro* and *BcpMohrCoulomb* the multimethod dispatcher behaves in the same way as the C++ virtual function mechanism – it falls back to the common base class, in this case it is *BcpElasticMicro* (see inheritance in Fig. 9).

Ef2_BexForce_BstParticle_cundallNonViscousDamping,
Ef2_BexMoment_BstRigidBody_cundallNonViscousDamping – implementation of Cundall non viscous damping [7] for a force and a moment inflicted on a particle or a rigid body.

Ef2_BssBox_BbvAxisAlignedBoundingBox_makeBbv,
Ef2_BssSphere_BbvKdop_makeBbv – functions to create a *BoundingVolume (Bbv)* from a given *BodySimplifiedShape (Bss)* depending on which kind of *Bbv* it is required, eg. a *BbvAxisAlignedBoundingBox* or *BbvKdop* (compare with Fig. 9).

Thanks to multimethods each algorithm resides in a separate plug–in class, which increases modularity. This solution allows easy modification, debugging and exchanging algorithms when needed. The drawback are long class names,

which seems to be unavoidable when a naming clarity is necessary.

5.3 Simulation overview

The class *World* is a top–level object representing the simulated world. It contains both *data* and the *engines* operating on it. The engines are executed by calling sequentially the *activated()* method for each *Engine*, and if the answer was positive, then calling *action()*. It is up to the user to specify what *engines* are inside the *simulation loop*. Usually this involves detecting interactions, solving them, applying solution results to bodies and saving some data (eg. position, velocity, forces) to disk.

If a graphical interface is used, then OpenGL display is performed by a separate thread, which is synchronized with the simulation loop.

6 Application of the YADE framework

YADE was designed so that new simulation models can be added easily and already defined algorithms reused. Examples in the subsections below describe what had to be implemented in specialized layers to perform simulation with DEM, FEM and LGM.

6.1 Discrete Element Method

To test the flexibility of the YADE framework, the SDEC [10, 11] algorithms were first implemented. In DEM the contact is described by the radii of two spheres: r_1, r_2, the penetration depth d and the normal vector to the contact plane $\vec{n}$. To allow interaction between the sphere and a non–spherical object, an imaginary mirror sphere of double radius is created (as proposed by Donzé [11]). Following this definition a new class *IstSpheresContact* was added (see Fig. 11). Then two different *EngineFunctor*-s with algorithms to build this contact description were added to *EngineDispatcher*: one to build the contact between two spheres and the other to build the contact between a sphere and a box. This contact description can be used only if at least one object in the contact is a sphere.

A class describing *InteractionConstitutiveParameters* was added with the name *IcpElasticMicro* which contains information about the contact: normal stiffness, tangential stiffness and rolling stiffness (see Fig. 11). When a contact occurs, this information is calculated by a dedicated *EngineFunctor* which

calculates macro–micro relationship according to the formula given by Donzé in [11].

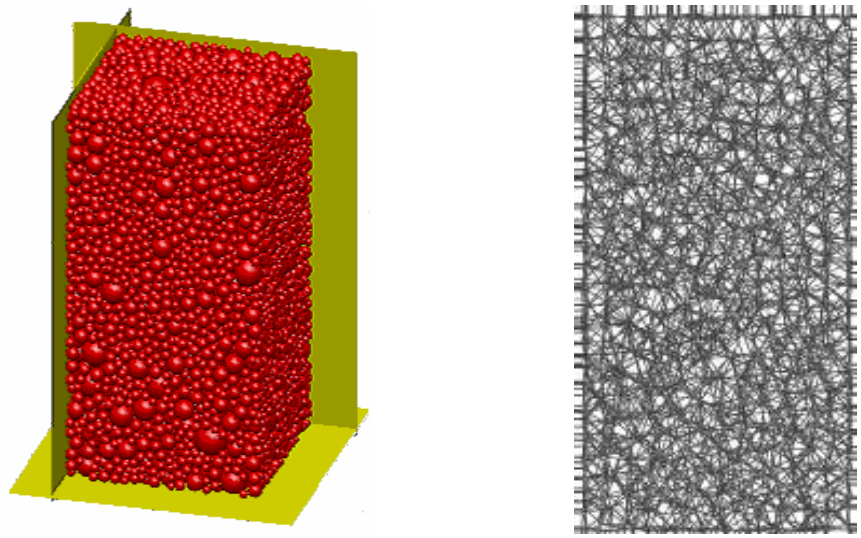
Finally a *simulation loop* for DEM calculation was built:

- Calculating time step with elastic criterion (the *EtimElasticCriterion* class),
- building a *BoundingVolume* (Fig. 8) using an *EngineDispatcher* with eg. *Ef2_BssSphere_BbvAxisAlignedBoundingBox_makeBbv*,
- performing collision detection with a Sweep and Prune collider [6] using the previously calculated bounding volumes (the *EbvcSweepAndPrune* is shown in Fig. 14),
- building *InteractionState* and *InteractionConstitutiveParameters* (in this case the classes *IstSpheresContact* and *IcpElasticMicro* shown in Fig. 11) using a 2D *EngineDispatcher* as shown in Fig. 16,
- solving interactions with DEM formulation with the class *ElawElasticContact* which contains Eq. 1–2,
- applying the calculated response (classes *BexForce* and *BexMoment*) to the bodies by calculating their new acceleration and angular acceleration,
- and performing the time integration of bodies according to their new acceleration (eg. using a leapfrog or Runge–Kutta 4 integration method).

This loop directly implements DEM and is repeated until the calculation is terminated.

As an example, the behavior of the "Labenne" sand during a triaxial test was modeled using the DEM formulation implemented in YADE [4]. The triaxial test was modeled by confining a dense discrete element medium within six smooth walls. The consolidation took place under gravity–free conditions, so that the 10 000 discrete elements arrangement was considered to be almost isotropic Fig. 17. The top and bottom boundaries moved vertically as loading platens, either under force–controlled condition or under strain–controlled conditions. Lateral boundaries simulate the confining pressure experienced by the sample sides. In the numerical simulation, the sample is loaded in a strain–controlled mode by specifying the velocities of the top and the bottom walls.

The five main input micro–parameters in the numerical model are: the normal contact stiffness, tangential contact stiffness, rolling contact stiffness, local friction and a dimensionless coefficient which controls the plastic limit of rolling. For a confining pressure of 100 kPa, these local parameters were chosen to fit the stress–strain and the volumetric curves. Laboratory tests are available under different confining pressures (100, 200 and 300 kPa). Once calibrated for 100 kPa, the numerical results obtained by the model for confining pressures of 200 kPa and 300 kPa, are shown on Fig. 18. The results indicate that the non linear stress–strain behavior of sand including dilatancy is covered by the numerical model. This shows that the model can be used as a predictive tool.



(a) (b)

Fig. 17. On the left, the numerical sample is presented and on the right, the normal force distribution between the discrete elements is plotted [4]

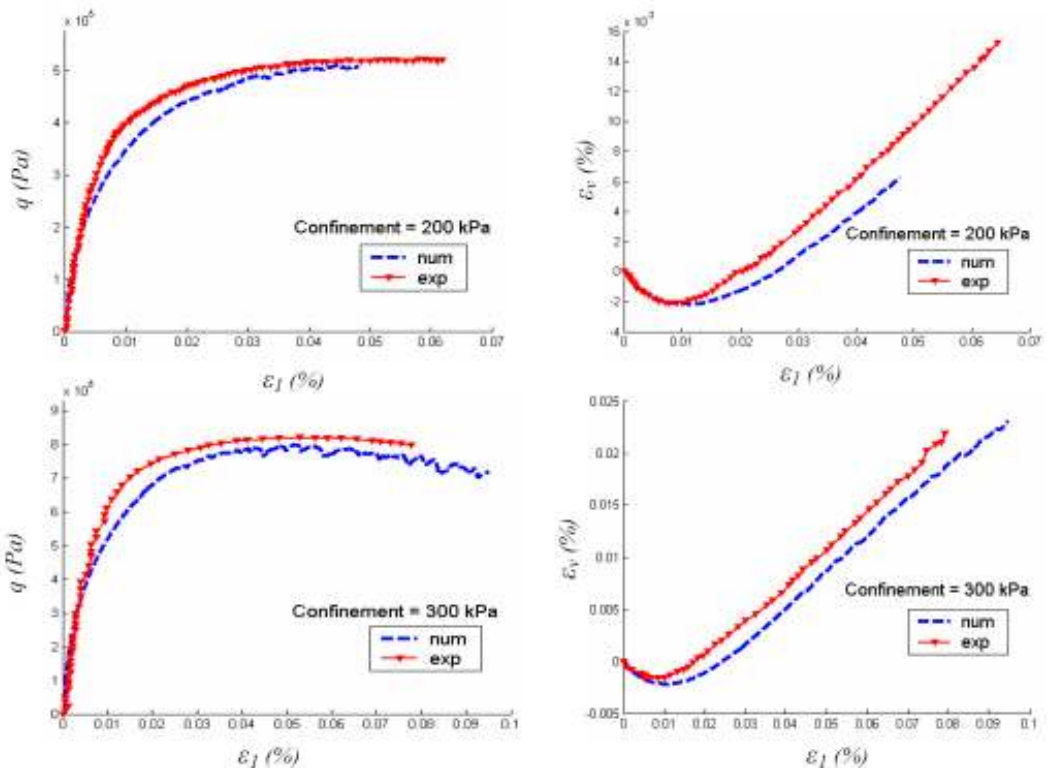


Fig. 18. Comparison between the DEM predicted stress q and volumetric strain ε_v plotted versus axial strain and the experimental data for confining pressures of 200 kPa and 300 kPa [4]

(a) (b) (c)

Fig. 19. A $10 \times 10 \times 10$ cm^3 concrete specimen subject to uniaxial extension with average element length of 3 mm and 8 000 000 elements [20]: (a) aggregate density - 25%; (b) aggregate density - 50%; (c) aggregate sieve curves with $d_{50} = 12$ mm (maximum and minimum curve for generated samples)

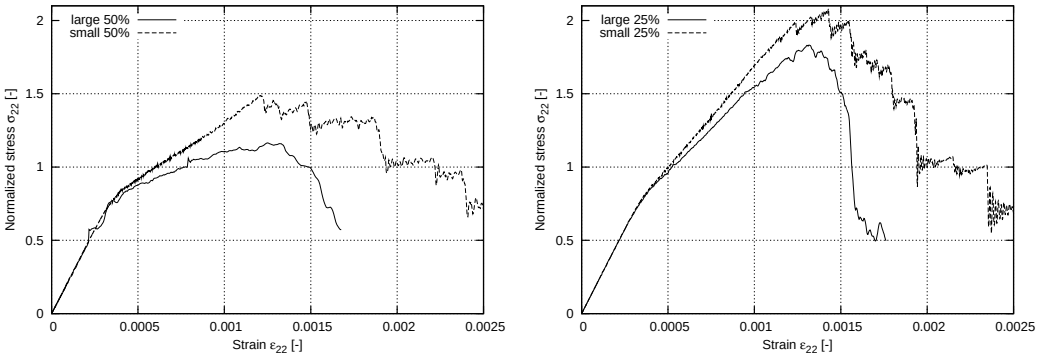
6.2 Lattice Geometrical Model

Another model implemented in YADE is the Lattice Geometrical Model [17–20]. The rod is a *body* in YADE which has two nodes (class *BstNode*), where each node is shared between multiple rods. The necessary data classes were added for a node, a rod and an angular spring between two rods to store their respective information. Then a new engine *ElawLattice* was written to perform the computations [17]. Finally a *simulation loop* was built:

- Building a bounding volume for the whole lattice specimen,
- applying some boundary conditions like displacement of selected nodes (using an *EgiConditionApplier* displacement engine),
- calculating interactions between rods, namely the current angle and torsion of angular springs that connect them,
- solving interactions of lattice rods (the *ElawLattice* class), by calculating a *BexDisplacement* for each node in the model,
- removing the rods according to a fracture criterion (using a common *BodyStateConstraints*),
- and finally applying the calculated response (stored in *BodyExternalVariables*) displacement to the nodes (notably skipping Newton's Law and time integration),

This loop shows similarities with the previous loop for DEM. The most important difference is that it does not use a time integration. This is because the lattice model is geometrical, has a quasi static nature and does not depend on time.

Figures 19 and 20 show the calculated size effect in uniaxial extension test of two 3D cubic specimens: $5 \times 5 \times 5$ cm^3 and $10 \times 10 \times 10$ cm^3 (with an average element length of 3 mm) [20]. The specimens had 1 000 000 (smaller) and



(b) (a)

Fig. 20. Size effect obtained for 3D specimens ($5\times5\times5$ cm^3 and $10\times10\times10$ cm^3 with average element length of 3 mm) [20]: (a) aggregate density - 25%; (b) aggregate - 50% (σ_{22} – vertical normal stress, ε_{22} – vertical normal strain)

8 000 000 elements (larger, shown in Fig. 19). The rods were non–uniformly distributed, and concrete was represented as a three–phase material with aggregate sieve curve presented in Fig 19c. The calculations of bigger specimen took 1 day using PC 3.6 GHz. Figure 20 compares the obtained stress–strain curves using 50% (Fig. 20a) and 25% (Fig. 20b) aggregate content. The larger the specimen, the smaller the strength and the larger the brittleness. The obtained outcome of the size effect for uniaxial tension is qualitatively in agreement with experiments performed by van Vliet [25].

6.3 Coupling the Discrete Element Method with the Finite Element Method

In the current FEM implementation each node is a *BstNode* and each FEM element is a multi–body interaction. One might argue that a FEM element is really an interaction, because it contains a K_e stiffness matrix just like a spring interaction or elastic contact interaction contain a stiffness property (compare with Fig. 11). An *EngineFunctor* was added to calculate the K_e stiffness matrix of element, which is done once at start of the simulation. An *ElawFem* engine was written (see Fig. 14) with classical explicit FEM formulation. Finally a *simulation loop* for FEM calculation was built:

- Building a bounding volume analogously as in Fig. 16 for the whole FEM entity,
- calculation of the response on individual nodes using an explicit FEM formulation by the *ElawFem* class, which produces a *BexForce* for each *body*, but does not apply the force yet,
- add gravity force to each *BexForce* using a gravity engine,
- applying boundary conditions such as translating selected nodes by a certain distance according to some velocity, or adding some other external force,
- using calculated forces to calculate acceleration, just like in the DEM sim-

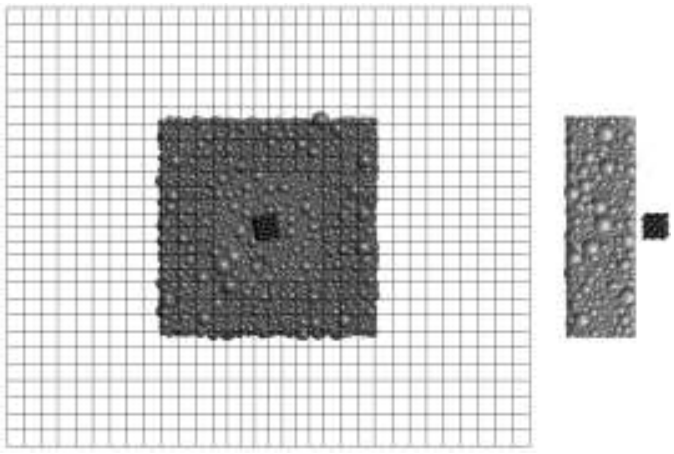


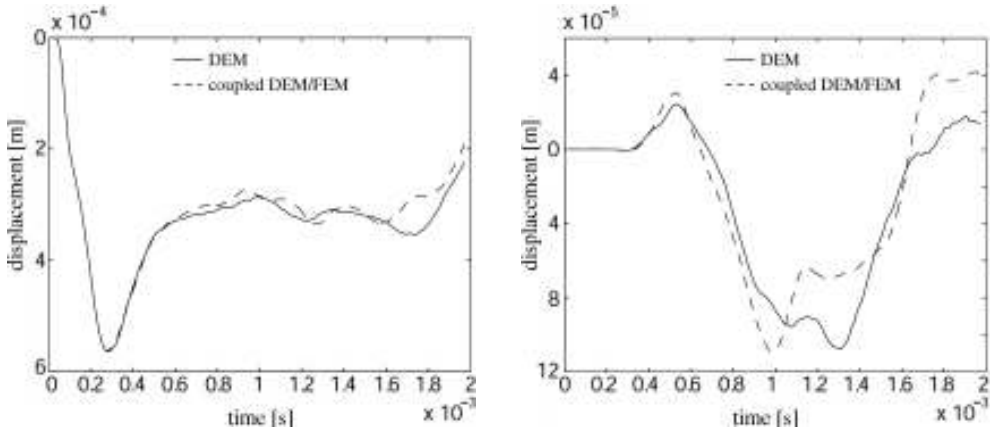
Fig. 21. Combined FEM/DEM model, and a side view of the DEM part (from [27])

ulation loop,
- and performing the time integration of bodies according to their new acceleration (eg. using a leapfrog or Runge–Kutta 4 integration method).

The loop repeats until terminated. It should be noted that with the modular architecture of YADE, the same numerical integration scheme can be used in both DEM and FEM or it can be replaced with another numerical scheme for example, substituting a Newmark method for the Runge–Kutta 4 integration.

The example application consists of a rock impact on a concrete slab. A cubic rock block with a 30 cm side length impacts a concrete slab 2.5 m long, 2.0 m wide and 0.28 m thick. The velocity of the impacting object was set at 40 m/s. The two opposite sides were locked in the direction perpendicular to the medium slab plane. The impact simulations were carried out using two different models: one using the DEM/FEM coupling and another using only the DEM formulation. In the DEM/FEM model, the slab was first divided into two parts: the center modeled by Discrete Elements, and the sides modeled by Finite Elements with three bridging domain layers (see Fig. 21).

Figure 22 presents the comparison of displacement between the model using only a DEM formulation (solid line) and the coupled DEM/FEM model (dashed line) at two different points. The maximum displacement predictions of both models are similar and, on the whole, the time response for the two models is the same. This combined FEM/DEM model is still under development [27] and it needs more work to be validated in a general framework. However, one major advantage of this coupled model is its efficiency; for this example, the coupled FEM/DEM model ran ten times faster than the full DEM model.



(a) (b)

Fig. 22. Displacement vs. time, (a) close to impacted area; (b) at greater distance from impacted area (from [27])

7 Conclusions

The task to find the underlying abstractions of various types of numerical simulations (DEM, FEM and LGM) has been completed and explained with examples. This allows us to implement a flexible software using Object Oriented techniques such as template metaprogramming, inheritance, encapsulation and OO design patterns. Implementation of such software performed by authors is open–source and can be downloaded from the `http://yade.wikia.com` webpage.

The implementation of DEM, FEM, LGM and possibly other models in a single framework makes the task of coupling them with each other relatively simple. Such a single framework with different models simplifies code exchanges. Since it is an open–source code, the improvements are as simple as sending a patch to the authors, then all the users can benefit from it, whereas commercial frameworks (like Abaqus, Dyna, Adina, Pfc3d) are closed source. The YADE framework is flexible, which gives more power to the user and minimizes obstacles when implementing a new kind of model.

The major idea behind DEM and LGM is to circumvent the complexity of a large assembly by considering instead many simple elements, the behavior of which can be simulated accurately. Because of this approach, DEM and LGM require careful calibration and validation with real experiments in order to produce trustworthy results. This research advances over the work done by Peters [26] by adding FEM and LGM to the same Object Oriented simulation framework.

References

[1] D. Abrahams, J. Garland, B. Dawes, C. Daniel, D. Gregor, J. Maurer, J. Maddock and others, The Boost Library, Boost Consulting, 2007, http://www.boost.org

[2] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, 2000, Addison-Wesley

[3] Z. P. Bazant & M. Jirasek, Nonlocal integral formulations of plasticity and damage: survey of progress, 2002, Journal for Engineering Mechanics, 1119–1149, 128, 11

[4] N. Belheine, J.-P. Plassiard, F.-V. Donzé, F. Darve and A. Seridi Numerical Simulation of Drained Triaxial Test Using 3D Discrete Element Modeling, 2007, Computers and Geotechnics, DOI 10.1016/j.compgeo.2008.02.003, 2008.

[5] R. Burkhardt, UML — Unified Modelling Language, 1997, Addison–Wesley

[6] J. D. Cohen, M. C. Lin, D. Manocha and M. K. Ponamgi, I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments, 1995, Symposium on Interactive 3D Graphics, 189–196, 218

[7] P.A. Cundall & O.D. Strack, A discrete numerical model for granular assemblies, 1979, Geotechnique, 47–65, 29

[8] G. Cusatis, Z.P. Bazant and L. Cedolin, Confinement–shear lattice CSL model for fracture propagation in concrete 2005, Comput. Methods Appl. Mech. Engrg, 192(52), 7172–7171

[9] G. Debunne, The QGLViewer Library, http://artis.imag.fr/Members/Gilles.Debunne/QGLViewer/

[10] F. Donzé & S.A. Magnier, Formulation of a three–dimensional numerical model of brittle behavior, 1995, Geophys. J. Int., 790–802, 122

[11] F. Donzé, S.A. Magnier, L. Daudeville and C. Mariotti, Numerical study of compressive behaviour of concrete at high strain rates, 1999, Journal for Engineering Mechanics , 1154–1163

[12] H. Ben Dhia, G. Rateau, 2005, The Arlequin method as a flexible engineering design tool, Int. J. Numer. Meth. Engng, 62, 1442–1462

[13] D. Fincham, Leapfrog rotational algorithm, 1992, Molecular Simulations, 1165–1170, 9

[14] E. Frangin, P. Marin, L. Daudeville, 2006, On the use of combined finite/discrete element method for impacted concrete structures, J. Phys. IV France 134, 461–466.

[15] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object–Oriented Software, 1995, Addison-Wesley, ISBN: 0201633612

[16] N. M. Josuttis, The C++ Standard Library: A Tutorial and Reference, 2000, Addison-Wesley

[17] J. Kozicki, Application of discrete models to describe the fracture process in brittle materials. 2007, PhD thesis, Gdańsk University of Technology

[18] J. Kozicki & J. Tejchman, 2D Lattice Model for Fracture in Brittle Materials, 2006, Archives of Hydro–Engineering and Environmental Mechanics, 71–88, 53, 2

[19] J. Kozicki & J. Tejchman, Effect of aggregate structure on fracture process in concrete using 2D lattice model, 2007, Archives of Mechanics, 365–384, 59, 4–5

[20] J. Kozicki & J. Tejchman, Modelling of fracture process in concrete using a novel lattice model, 2008, Granular Matter, DOI 10.1007/s10035-008-0104-4, (in print)

[21] G. Lilliu & J.G.M. van Mier, 3D lattice type fracture model for concrete, 2003, Engineering Fracture Mechanics, 70, 927–941

[22] R. C. Martin, Design Principles and Design Patterns, 2000, http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[23] S. Meyers, Effective STL, 2005, Addison-Wesley

[24] J.G.M. van Mier, E. Schlangen and A. Vervuurt, Lattice type fracture models for concrete, 1995, Continuum Models for Materials with Microstructure, H.B. Mühlhaus, ed., John Wiley & Sons, 341–377

[25] M.R.A. van Vliet, Size effect in tensile fracture of concrete and rock, 2000, PhD. Thesis

[26] B. Peters & A. Džiugys, Numerical simulation of the motion of granular material using object–oriented techniques, 2002, Computer methods in applied mechanics and engineering, 191, 1983–2007

[27] J. Rousseau, E. Frangin, P. Marin and L. Daudeville 2008, Discrete Element modelling of concrete structures and coupling with a Finite Element model, Computer and Concrete, (in print)

[28] E. Schlangen & E.J. Garboczi, Fracture simulations of concrete using lattice models: computational aspects, 1997, Engineering Fracture Mechanics, 57, 319–332

[29] C. Thornton & L. Zhang, A DEM comparison of different shear testing devices (invited lecture). 2001, Powders and Grains conference, Kishino (ed.)

[30] A. Vervuurt, J.G.M. van Mier and E. Schlangen, Lattice model for analyzing steel–concrete interactions 1994, Computer Methods and Advances in Geomechanics, Siriwardane and Zaman, eds., Balkema, Rotterdam, 713–718

[31] S.P. Xiao, T. Belytschko, 2004, A bridging domain method for coupling continua with molecular dynamics, Computer Methods Appl. Mech. Engrg. 193 1645–1669.