OOFEM — an Object-oriented Simulation Tool for Advanced Modeling of Materials and Structures

Bořek Patzák

Department of Mechanics, Faculty of Civil Engineering, Czech Technical University in Prague, Thákurova 7, 166 29 Prague, Czech Republic

Corresponding author: bp@cml.fsv.cvut.cz

Abstract

The aim of this paper is to describe the object-oriented design of the finite element based simulation code. The overall, object-oriented structure is described, and the role of the fundamental classes is discussed. The paper discusses the advanced parallel, adaptive, and multiphysics capabilities of the OOFEM code, and illustrates them on the basis of selected examples.

Keywords: object-oriented design, finite element analysis, parallel and distributed processing, multiphysics simulations.

1 Introduction

The aim of this paper is to describe the object-oriented design of the finite element based simulation code. The design follows several design patterns that have contributed to an extremely modular and extensible structure and have sustained nearly twenty years of active development, during which the code has been extended from a tool that was originally oriented toward solid mechanics to become a truly multiphysics modeling tool with adaptivity and distributed parallel processing support, without any large-scale redesign. The OOFEM code was originally developed by the author at the Czech Technical University. At present, the code consists of more than 200k lines of source code in C++. It is freely available under General Public License, and it is being developed by an international community (visit the project web pages [1] for further information). The design is based on traditional object-oriented paradigms, such as encapsulation, abstraction, and inheritance. On top of these fundamental concepts, a hierarchy of classes is designed to map the mathematical problem described by a set of partial differential equations in space and time into flexible and cooperating software objects that solve the problem. Initially, the object-oriented design may seem complex in comparison with traditional finite element codes. However, when it is structured properly and in agreement with object-oriented philosophy, many benefits can be achieved. The encapsulation of attributes and methods into a class can hide the implementation details, provided that the object state is requested and manipulated by the corresponding services. Inheritance allows specialization and extension of existing classes, and when combined with abstract methods defined by parent classes it allows for extremely modular and extensible design. The concept of abstract classes allows an abstract interface to be designed in terms of the service specification. The abstract interface has to be implemented by the derived classes by implementing the required services. This allows all derived classes to be treated using the same high-level interface, without regarding the particular details of each derived class.

In the implementation, single inheritance has been preferred wherever the parent class defines a compulsory basic interface. However, in many cases, optional functionality may also be implemented by derived classes. As an example, consider the implementation of a particular finite element. The compulsory part of the element interface (involving methods for evaluating characteristic matrices and vectors, etc.) is defined by the parent *Element* class. Additional functionality, e.g. an error estimation capability, may be supported only by some elements. It is not a smart idea to extend the basic interface by including optional functionality, as this will result in a very complex interface specification, with many methods that will prevent implementation of simple elements without optional functionality. A possible remedy is then based on using multiple inheritance, where a particular element is derived from the base class and optionally from classes defining the optional interfaces. This solution may seem natural, but it has one important drawback: if a class is based on (derived from) another one, it is automatically derived from all classes as its parent. This prevents the selective application of optional interfaces. This problem has been solved in some object-oriented languages, e.g. in Java, where only single inheritance is supported, but a class can implement several interfaces. Derived classes do not inherit interfaces automatically; interface implementation has to be declared by each class explicitly. As the presented code has been implemented in C++,

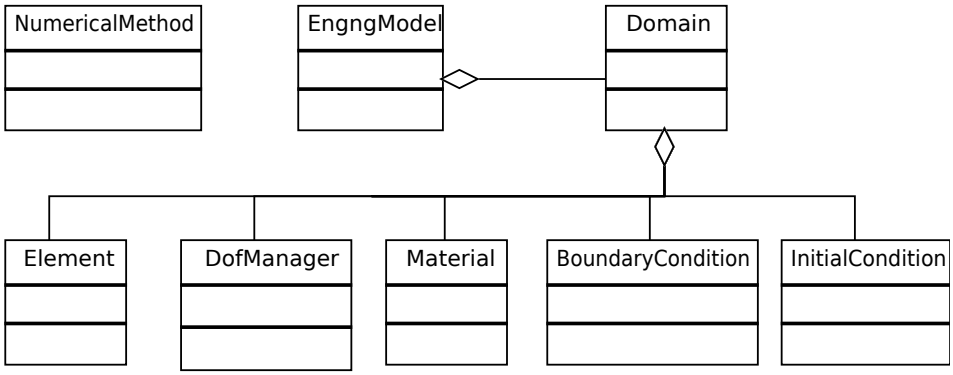

Figure 1: Problem representation in OOFEM.

an additional concept has been defined to support an optional interface. It is based on multiple inheritance, but supplemented by an abstract interface request service, which allows selective decisions on implemented interfaces by particular elements.

2 Overall design

This section presents the general structure of the OOFEM code using uniform modeling language (UML) notation (see [2] for details). The abstract classes are represented by rectangles. The lines marked with a triangle represent the generalization/specialization relation (inheritance), where the triangle mark points to the parent class. The lines marked with a diamond represent the whole/part relation, in which diamond mark points to the "whole" class possessing the "part" class, and an association is represented by a solid line drawn between the classes.

The problem under consideration is represented by a class derived from the *EngngModel* class, see Fig. 1. Its role is to assemble the governing equation(s) and use a suitable numerical method (represented by the class derived from the *NumericalMethod* class), to solve the resulting system of equations. The discretization of the problem domain is represented by the *Domain* class, which maintains the lists of objects representing nodes, elements, material models, boundary conditions, etc. The *Domain* class is an attribute of the *EngngModel*. For each solution step, the *EngngModel* instance assembles the governing equation(s) by summing up the contributions requested from the domain components (nodes, elements, etc.). This abstract approach is supported by suitable abstract services at the element and nodal classes for getting code numbers and evaluating characteristic components.

Since the governing equations are typically represented numerically in matrix form, implementation is based on vector and sparse matrix representations, see Fig. 2. The parent *SparseMatrix* class declares the abstract interface, allowing manipulation of different sparse matrix formats using the same interface. Derived classes represent different sparse storage schemes and implement, for example, the skyline, a compressed row or compressed column formats, as well as classes implementing the *SparseMatrix* interface and representing an interface to third-party libraries, e.g. IML[3], Spooles[4], DSS[5], or Petsc[6]. The advantage of the design described here is its modular design with decoupled problem formulation, numerical solution and sparse storage. It allows a particular problem to be implemented in a form that will work with all suitable numerical solvers and sparse matrix storage schemes, even those added in the future.

The individual finite elements are represented by classes derived from *ElementGeometry* and one or more classes derived from *ElementEvaluator*, see Fig. 3. The *ElementGeometry* class defines the element geometry and keeps a list of element nodes. The *ElementEvaluator* (or, more precisely, the problem related class derived from the parent *ElementEvaluator*) defines the problem-specific methods (e.g. methods for evaluating the stiffness matrix or element load vector) required by the corresponding problem. The implementation of individual elements is further facilitated by the library of classes representing integration points, integration rules, and finite element interpolation spaces. The *ElementEvaluator* problem specific methods can be implemented by the evaluator itself, without regarding the details of the particular element, because the evaluator has access to representations of element geometry, element interpolation and integration rules, and each of these can be manipulated by an abstract interface. Multiple integration rules can be created by elements to implement reduced or selective integration schemes, and multiple interpolation can also be created. Many Lagrange-based interpolation schemes are provided, as well as B-spline, NURBS, and T-spline based interpolations. Thanks to T-spline based interpolations, the implementation of isogeometric analysis [7] has been pretty straightforward (see [8, 9] for more details).

The essential feature is decoupling of the element

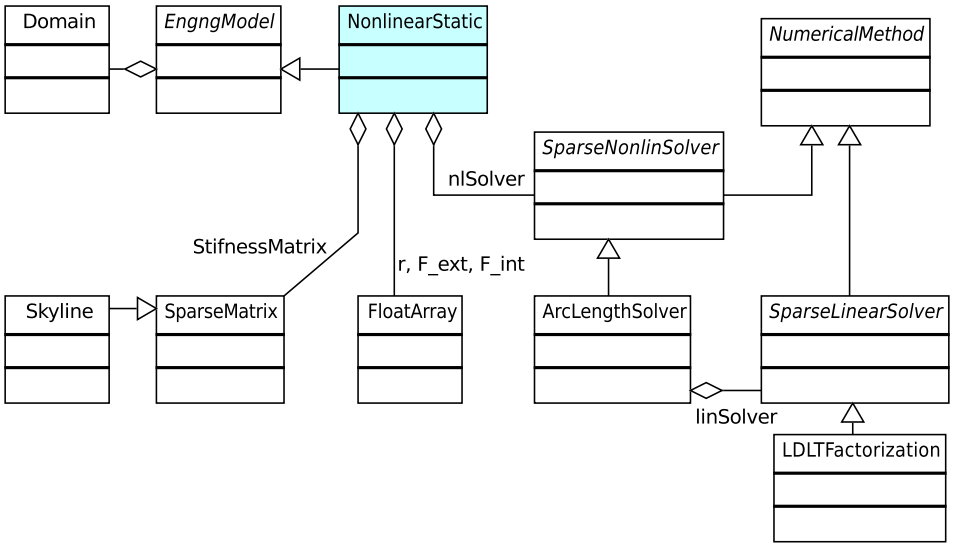

Figure 2: Nonlinear static problem and corresponding classes.

geometry description (represented by the *ElementGeometry* class) and problem-specific functionality (represented by the classes derived from the *ElementEvaluator* class), which allows natural implementation of elements for coupled problems, where one needs to combine functionality from individual subproblems into a single element (represented by corresponding classes derived from *ElementEvaluator*). This is a better solution than deriving the coupled element from several sub-problem elements, leading to duplication of element geometry data. Introducing the *CoupledEvaluator* class makes it possible to combine individual evaluators. *CoupledEvaluator* is derived from the base *ElementEvaluator* class, and comes with the capability to group individual low-level evaluators together (complemented by evaluators for coupling terms) by performing local assembly from individual contributions. When a problem-specific evaluator is available, the definition of a particular element is straightforward. It consists in:

1. defining a new class, derived from *ElementGeometry* and class(es) representing problem-specific evaluator(s);

2. setting up its interpolation(s) and integration rules.

No additional coding is necessary.

Nonlinear, path-dependent material models require keeping a loading history in each integration point, described by a set of internal variables. Naturally, these variables should be stored in the corresponding integration point. However, efficient implementation is not straightforward, as the amount and the type of internal variables vary for different material models. Instead, each material model defines an associated status, a class derived from *MaterialStatus*, which serves as a container for material model specific internal variables. The integration point comes with the possibility to store a unique material status instance, which is created by the corresponding material. As the integration point is the parameter to all services provided by a material model, the associated status and the corresponding internal variables are conveniently accessible. The elements do not communicate directly with the associated material model, as an additional layer is inserted between them representing the cross section model. The role of the cross section is to integrate the response (in terms of stress and strain, for example) over the cross section geometry composed of possibly different materials. This is especially helpful in the case of shell and beam elements, where the introduction of the cross section allows cross section details to be hidden by decoupling finite element and cross section formulations. This approach allows the use of the same beam element formulation with integral or layered cross section descriptions, for example. For problems where the cross section model is irrelevant, a simple dummy cross section model is provided, routing all requests directly to the underlying material model.

3 Parallelization Strategy

The code provides support for parallel and distributed computations. The design of parallel algorithms requires partitioning of the problem into a set of tasks, the number of which is greater than or equal to the number of available processors. The partitioning of the problem can be fixed (static load balancing) or can change during the solution (dynamic load balancing). The latter option is often necessary in order to achieve good load balancing of the work between processors, and thus optimal scalability. The adopted

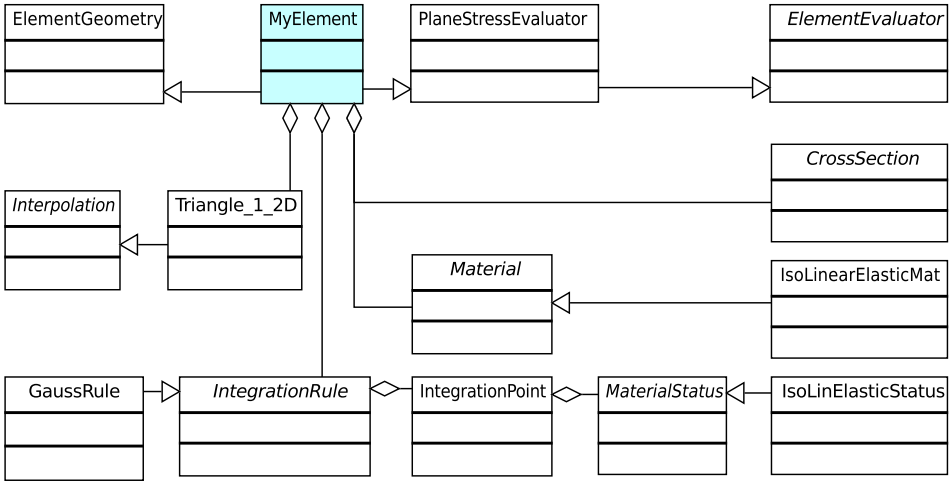

Figure 3: Element implementation.

parallelization strategy is based on the domain decomposition paradigm, where the computational mesh is divided into partitions assigned to individual processing units. The node cut approach is used. This approach is based on unique assignment of individual elements to partitions. A node is then either assigned to a partition (local node), if it is surrounded exclusively by elements assigned to that partition, or is shared by several partitions (shared node), if it is incident to elements owned by different partitions. The communication model is based on the message passing paradigm, which is available on most hardware configurations. This allows different parallel architectures to be supported, ranging from systems with shared memory to massively parallel systems with distributed memory. The Message Passing Interface (MPI) [10, 11] is used.

Efficient parallelization requires that all steps in the solution procedure can be processed in parallel. In a typical problem, this involves assembling the characteristic problem, solving it, and postprocessing it. High-level communication services were developed on the top of the message passing library, providing transparent data streams for parallel, non-blocking communication between partitions. Provided that the partitioning of the problem is given, each processor can assemble its local contribution to the global characteristic equation(s). The rows of global matrices and vectors are distributed on individual processors according to the distribution of nodes and corresponding DOFs. This part is identical to the serial assembly process. After local assembly is finished, the shared node contributions are exchanged. This is integrated into general purpose assembly services, so that the users can use any sparse matrix representations and the exchange of shared contributions is performed transparently. After the assembly process is finished, the global set of (linearized) equations is solved in parallel. After the solution, the local solution vectors on each partition are collected, and local postprocessing (stress and strain evaluation, for example) is performed on each partition in parallel, typically without the need for further communication.

The load balance recovery is achieved by repartitioning the problem domain and transferring the work (typically represented by finite elements) from one sub-domain to another. There are in general two basic factors causing load imbalance between individual subdomains:

1. one coming from the nature of the application, e.g. switching from a linear response to a nonlinear response in certain regions or local adaptive refinement;

2. external factors, caused by resource reallocation, typical in non-dedicated cluster environments, where individual processors are shared by different applications and users, leading to variation in allocated processing power.

Repartitioning is an optimization problem with multiple constraints. The optimal algorithm should balance the work while minimizing the work transfer and keeping the sub-domain interfaces as small as possible. Other constraints can reflect the differences in the processing power of individual processors, or may be induced by the topology of the network. The load balancing layer is transparently integrated into the computational kernel. Details on implementing dynamic load balancing in OOFEM can be found in [12, 13].

4 Multiphysics and adaptive capabilities

The framework supports both fully and weakly (staggered) coupled multiphysics simulations. The development of multiphysics solution schemes is relatively

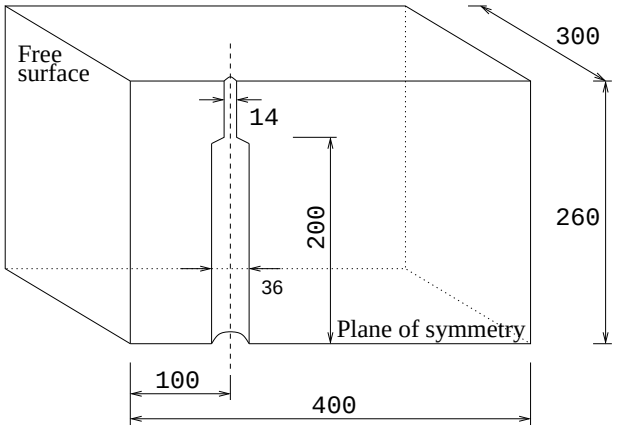

Figure 4: Geometry of an Anchor Pullout Test.

straightforward, as the individual elements can easily be constructed from existing single physics formulations by supplying only a definition of corresponding coupling terms, by implementing the corresponding element evaluator. Concurrent multiscale simulations have also been developed, using a microscale model of the representative volume element in each integration point to obtain a macroscale material response by means of homogenizations. Additionally, any primary or secondary variable can be represented as a field. Fields can represent any scalar, vector or tensorial quantity and provide services to evaluate the field at any point of interest. This feature simplifies mutual field exchange in staggered simulations to a large extent, naturally allowing the possibility to have different discretizations on each subproblem. The individual subproblems can be arbitrarily assembled into a staggered solution scheme, and data is transparently exchanged. The field mapping is also essential in h-adaptive, nonlinear solution procedures, enabling the solution state to be mapped from old to updated discretization. The sequence of meshes is generated on the basis of spatial error distribution, estimated by a suitable a-posteriori error estimator, represented by a class derived from the abstract *ErrorEstimator* class. The mesh can be refined by a built-in, fully parallel, subdivision-based remeshing algorithm or by an external application. At present, an interface to the T3D mesh generator [14, 15] is supported.

5 Examples

5.1 Parallel analysis of 3D anchor test

The capabilities and the performance of the parallel adaptive load-balancing framework are illustrated on a three-dimensional adaptive analysis of an anchor pullout test. In this example, h-adaptive analysis is used together with a heuristic error indicator based on the

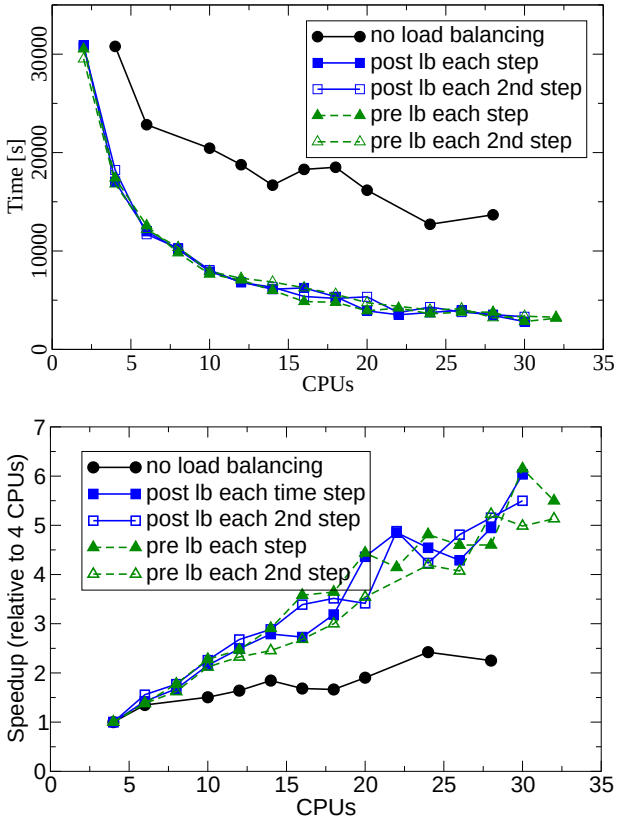

Figure 5: Anchor pullout: timing and speedups.

attained damage level. In order to assess the behavior and performance of the proposed methodology, the case study analyses were run without dynamic load balancing (static partitioning was employed, marked as "nolb"), and with dynamic load balancing performed before error assessment ("prelb"), or after error assessment ("postlb").

The geometry and setup of the test are shown in Fig. 4. The anchor is located close to the boundary, requiring full 3D analysis with only one plane of symmetry. As the steel anchor is pulled out of the concrete, a crack surface is initiated at the anchor head and starts to propagate towards the boundary as the loading increases. An anisotropic, non-local damage based model has been used for modeling the concrete fracture. The original mesh consists of 16772 linear tetrahedral elements and 1456 nodes, which was subsequently refined in 20 steps into a final mesh with 125400 elements and 22441 nodes. The problem was solved on an SGI Altix 4700 machine installed at the CTU computing center. The obtained solution times (averaged over two or three analysis runs) and the corresponding speedups (relative to 4 CPUs) are summarized in Fig. 5.

The results reveal that the effect of dynamic load balancing is quite substantial. When no load balancing is applied, the solution times decrease only slightly with the number of CPUs. This is a direct

Figure 6: Oparno Bridge.

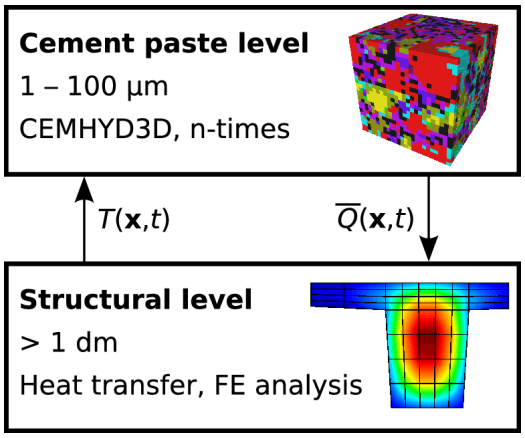

Figure 7: Multiscale computational scheme.

consequence of the heavy imbalance due to the localized refinement resulting in a dramatic increase in the number of elements in one subdomain or in a small number of subdomains. With dynamic load balancing, this effect is alleviated. The obtained speedup of the load-balanced computation shows a nice linear trend, indicating very good scalability of the parallel algorithm.

5.2 Multiscale Heat Transport

This example illustrates multiscale simulation, which helped to find the optimal position for cooling pipes and a cooling regime on an arch bridge over the Oparno Valley, Czech Republic. The bridge was built between 2008 and 2010, with the arches spanning 135 m, see Fig. 6. Hydrating concrete produces a significant amount of hydration heat, which causes several problems in massive concrete elements. There were concerns that significant tensile stresses might appear during the cooling of a massive cross-section of the arch. These tensile stresses can lead to cracking, which can negatively affect the durability of the final

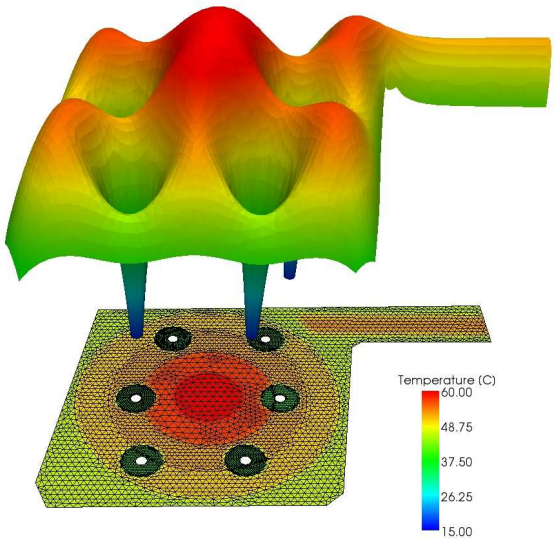

Figure 8: Oparno Bridge: Temperature profile.

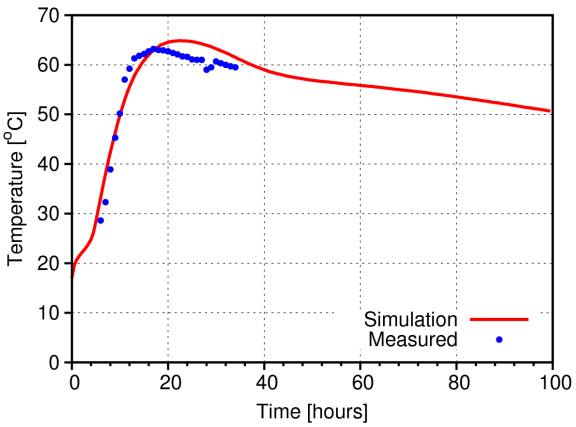

Figure 9: Oparno Bridge: comparison with measurements.

structure. Another negative effect can appear when the temperature exceeds 70 degrees Celsius, when the formation of monosulphate can be followed by ettringite formation (so-called delayed ettringite formation, DEF). Two computational scales were involved (see Fig. 7):

1. the level of cement paste, where the CEMHYD3D [16] material model predicts the evolution of a discrete microstructure on the scale of micrometers and returns the liberated heat;

2. the structural level, where the heat balance equation is solved with finite elements.

The details can be found in [17].

Fig. 8 shows the temperature evolution during concrete hardening and the induced out-of-plane stress.

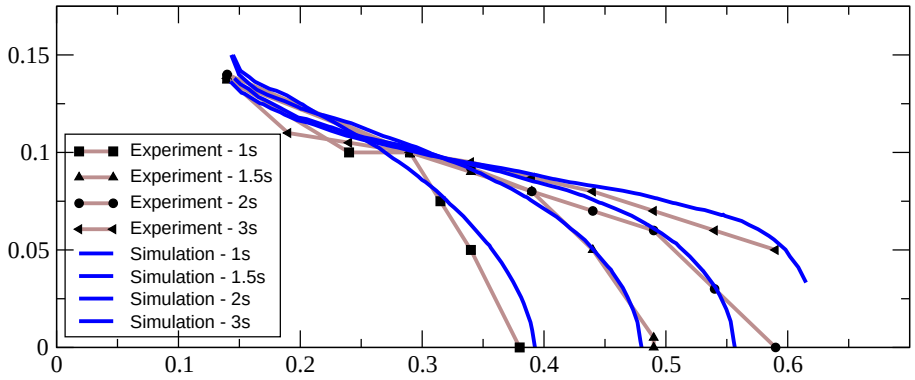

Figure 12: LBOX test: profiles of the spreading concrete at different times.

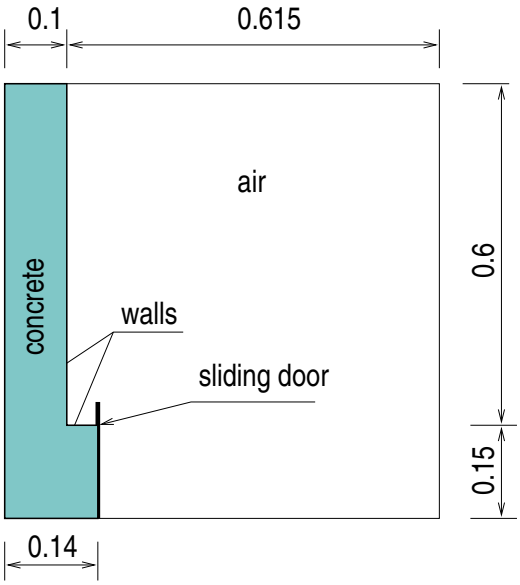

Figure 10: LBOX test: geometry.

The simulation runs on the left symmetric part of the arch cross-section. Fig. 9 compares the multiscale simulation with the temperature in the core of the cross-section. The temperature measured in the concrete remained below 65°C during summer casting, which was found acceptable.

5.3 Casting Simulation

The last example illustrates the simulation of fresh concrete casting, where the fresh concrete has been modeled as a homogeneous, non-Newtonian fluid. The simulation is based on an incompressible flow model of two immiscible fluids (concrete and air), where the interface tracking technique is used to track the position of the interface between the fluids. The geometry and the setting of the experimental setup, known as the L-Box test, is presented in Fig. 10.

The concrete is confined in an L-shaped reservoir with a vertical gate. After the removal of the gate, the concrete starts spreading into the form-work. Frictionless boundary conditions are assumed on the walls

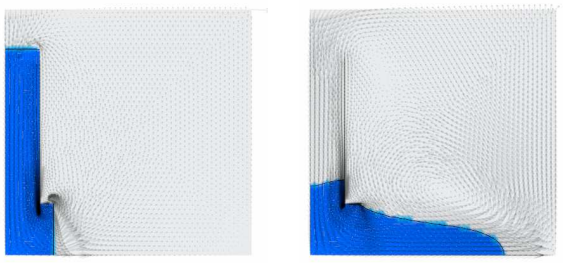

Figure 11: LBOX test: spreading conrete profiles.

forming the reservoir, and also on the formwork. The numerical simulations use the Bingham model for the concrete suspension. The unstructured, triangular grid consists of 3652 nodes and 6927 triangles. Realistic modeling of the gradual gate opening is particularly important, as the results, especially in the initial phase, are very sensitive to the gate opening speed. The profiles of the fresh concrete at different times (see Fig. 11) are compared with experimental observations in Fig. 12. Very good agreement has been obtained.

6 Conclusions

The paper has focused on a description of the object-oriented structure of a finite element based simulation framework. The fundamental design patterns and their consequences have been discussed. The paper has also described the design and implementation of some advanced features, including parallel and distributed support and multiphysics. In the final part, selected examples have been used to demonstrate the capabilities of the code.

Acknowledgments

This work was supported by the Ministry of Education of the Czech Republic, under project MSM 6840770003.

References

[1] B. Patzák. OOFEM project home page, 2012. http://www.oofem.org.

[2] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc.; 2nd edition, 2005.

[3] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington. IML++ (Iterative Methods Library) project page, 2012. http://math.nist.gov/iml++.

[4] C. Ashcraft et al. SPOOLES: SParse Object Oriented Linear Equations Solver, 2012. ww.netlib.org/linalg/spooles/spooles.2.2.html.

[5] R. Vondráček. Use of a Sparse Direct Solver in Engineering Applications of the Finite Element Method. Česká technika – nakladatelství ČVUT, ISBN: 978-80-01-04245-8, 2008.

[6] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[7] T.J.R. Hughes, J.A. Cottrell, Y. Bazilevs. Isogeometric Analysis: CAD, Finite Elements, NURBS, Exact Geometry and Mesh Refinement. Computer Methods in Applied Mechanics and Engineering, 194, 4135-4195, 2005.

[8] D. Rypl, B. Patzák. From the Finite Element Analysis to the Isogeometric Analysis in an Object Oriented Computing Environment. Advances in Engineering Software, 44 (1), pp. 116-125, 2012.

[9] D. Rypl, B. Patzák. Object Oriented Implementation of the T-spline Based Isogeometric Analysis. Advances in Engineering Software, 50, pp. 137–149, 2012.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, 1995.

[11] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference. MIT Press, Boston, 1996.

[12] B. Patzák, D. Rypl, and Z. Bittnar. Parallel explicit finite element dynamics with nonlocal constitutive models. *Computers and Structures*, 79(26-28):2287–2297, 2001.

[13] B. Patzák, D. Rypl. Object-oriented, parallel finite element framework with dynamic load balancing. Advances in Engineering Software, 47(1):35 – 50, 2012.

[14] D. Rypl. T3D project home page, 2012. http://www.t3d.info.

[15] D. Rypl. Sweeping of Unstructured Meshes over Generalized Extruded Volumes, Finite Elements in Analysis and Design, 46 (1–2), 203–215, 2010.

[16] V. Šmilauer, T. Krejčí. Multiscale Model for Temperature Distribution in Hydrating Concrete. International Journal for Multiscale Computational Engineering, 7(2), 1543–1649, 2009.

[17] V. Šmilauer, J. L. Vítek, B. Patzák, Z. Bittnar. Optimalizace chlazení oblouku Oparenského mostu. Cooling optimization in Oparno bridge's arch. Časopis Beton TKS. 2011, vol. 11, no. 4, p. 62-65.