Roman PUTANOWICZ*

GROUNDS FOR THE SELECTION OF SOFTWARE COMPONENTS FOR BUILDING FEM SIMULATION SYSTEMS FOR COUPLED PROBLEMS

SUMMARY

This paper discusses rationales for selection of software components for building scientific simulation tools. Nowadays no single research team has the resources or knowledge to build non-trivial simulation software form scratch. Sharing experience about the motives behind the choice of software components and the consequences of particular decisions seems a valuable knowledge as it can help to avoid some potential traps. In the paper we discuss software selection decisions for our problem solving environment for numerical modelling of coupled problems. For selected tools we discuss pros and cons of their use and mention potential alternatives. The detailed discussion concerns implementation of a solver for thermoelasticity problems.

Keywords*: software, coupled fields, numerical analysis, finite element methods*

ANALIZA WYBRANYCH KOMPONENTÓW OPROGRAMOWANIA DO BUDOWY SYSTEMÓW SYMULACJI MES PROBLEMÓW SPRZĘŻONYCH

Artykuł prezentuje analizę wybranych komponentów oprogramowania pod kątem ich zastosowania do budowy systemów symulacji MES dla problemów sprzężonych. Budowa takich systemów bez wykorzystania pewnych gotowych komponentów jest praktycznie niemożliwa, jednak decyzja o ich użyciu uwarunkowana jest szeregiem czynników, a wybór konkretnego komponentu wpływa na przebieg tworzenia całego systemu symulacji. Istotne jest jak najwcześniejsze rozpoznanie możliwości oraz ograniczeń, często ukrytych, konkretnych komponentów, stąd cenna jest wymiana doświadczeń związanych z ich zastosowaniem. Artykuł omawia wybrane biblioteki i programy na przykładzie tworzenia programu od analizy zagadnień termosprężystości.

Słowa kluczowe*: komponenty, analiza numeryczna, problemy sprzężone, MES*

1. INTRODUCTION

Building a new scientific simulation environment, for instance based on finite element method, is a daunting, complex task (Oldehoeft 2002). Beside knowledge and skills required to solve technical problems it also raises the issues of long time commitment, resources allocation and team management. Despite these difficulties, building own simulation tools gives us an opportunity to shape them according to the new ideas that appear in computational science such as meshless methods, XFEM, discrete exterior calculus, to name a few. It also gives us an unparalleled insight into the nature of those simulation systems.

In order to manage the complexity and the development costs of a simulation system, a common approach is to use ready components, either COTS (Commercial Of-The-Shelf) or OSS (Open Source Software). While the code sharing is nothing new, we are observing steady shift towards component programming, even if not all use of software libraries can be branded as such. This trend is fuelled by the appearance of many comprehensive, high quality software packages and improvement and spread of standard interfaces between components, both on middleware and application level.

While removing the burden of low level implementation, the component approach faces the developers with the question which components the system should be composed of. Because of the early stages of the component programming, the support of component's interoperability and exchangeability is still unsatisfactory. Thus selecting a component is often "one-shot decision" – after the component is tied to the rest of the system it is too costly to exchange it with another. The other issue with components is that while being functionally equivalent they can differ significantly in the costs of getting started and maintenance.

The aim of the paper is to present the first results of a research project aimed at the development of a software environment which would allow the analyst to consider various physical fields coupled to mechanics. The application domain are engineering materials, especially concrete, so eventually coupled processes such as the ones covered in monographs (Gawin 2010, Kuhl 2005) are supposed to be analysed.

The mechanical model considered in the paper is statics of nonlinearly elastic materials, but it is understood here as a prototype of a general nonlinear constitutive model for the solid skeleton of the material (the model should eventually incorporate damage, plasticity and discrete cracking). On the other hand, the heat conduction model, to which the paper is limited, is a prototype of many nonstationary diffusion processes occurring in engineering materials, including moisture transport and related chemomechanical processes. This is

* Institute for Computational Civil Engineering, Cracow University of Technology, ul. Warszawska 24, 31-155 Cracow, Poland; e-mail: R.Putanowicz@L5.pk.edu.pl

because the partial differential equations governing the processes are quite similar.

It is stressed that the aim is to incorporate arbitrary nonlinearity and coupling in the analysis. This means that the mechanical and thermal material properties (e.g. Young's modulus and/or conductivity) are considered as time and temperature-dependent.

In the paper the mathematical model is reviewed and general algorithmic aspects are covered. The developed simulation environment FEMDK is described and its (prospective) advantages are listed. An example solution of a coupled thermoelasticity problem of the temperature and stress distribution in a thick-walled cylinder is presented.

2. SOFTWARE SELECTION DRIVERS

The problem with selecting software components or actually any software technology is such that when it turns out that a particular choice causes problems, it is usually to late to modify it. Is there any systematic way to arrive at good decisions? To the author's experience there is none – of course one can make a list of features that the components must support and investigate potential candidates, or even make a detailed study on the impact of a particular choice. Such study is however expensive and time consuming. In the end what really counts is the experience. This is probably especially true for academic software projects where the software choice is done by scientists who would rather focus on their own field than investigate the software itself. In most cases, it probably looks this way that the person responsible for the software choice selects some most promising candidates and the first one which installs cleanly and just „feels right", and seems easy to learn is selected. Thus the process is often driven by the first impression which can be misleading. In the light of the above, it is most valuable to gather experience from other researchres about why they selected a particular component or software technology. It would be even more valuable to know how they see their decisions from the time perspective. Unfortunately such kind of wisdom is hard to find, especially comments about wrong decisions and their consequences. This is the main motivation behind this paper, to share some of our experience related to the selection of software components. This is the experience of the early stage of the project, so the outcome of some decisions is not yet fully defined.

3. FEMDK PROJECT

The material for this paper is based on our work on Finite Element Method Development Kit (FEMDK) project. The scope of this project is solving multi-field problems that appear during the analysis of degradation phenomena of engineering materials with special attention paid to concrete. One of the main goals is building a problem solving environment which would facilitate fast creation of tools for solving coupled problems. The main stress is on the ease of defining multi-field problems, a possibility of experimenting with new numerical algorithms, a potential to accommodate various requirements regarding data formats, geometric models, element types, FEM interpolation, solvers, etc.

3.1. Integration of tools in FEMDK

In order to meet its goals, it is necessary for FEMDK to integrate several components (libraries and programs) in such way that it is easy to build new tools using these components, and it is easy to freely pass data between them. The integration of components within FEMDK can conceptually be done on four different levels.

Integration on library level

In case of libraries, by integration we understand ensuring compatibility between data structures and function interfaces of two or more libraries. In the most simple case it boils down to a direct translation of data structure of library A into a data structure of library B, as shown in Figure 1.

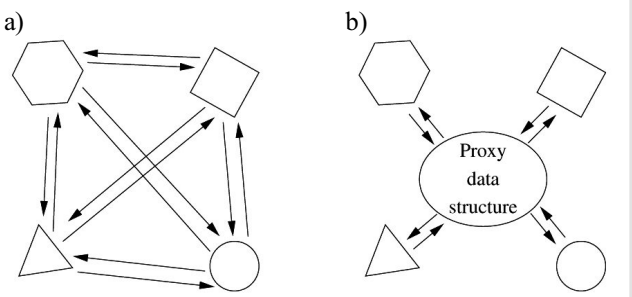

Fig. 1. Translation between data structures of different libraries: a) direct translation; b) translation via proxy

In the case of object oriented implementation of data structures, the translation layer can be built using such design patterns as Adapter shown in Figure 2, Bridge or Proxy. In some cases it can be more convenient to do the translation via the third, intermediate data structure as shown in Figure 1. It is worth to do so when one has potentially many different data structures, because this reduces the number of translation functions needed. In such case the main difficulty is to choose such intermediate structure that can be feasibly translated into the final data structures. In case of FEMDK this approach was taken when translating data structures for geometric model and finite element meshes. For the former we use CGM (Common Geometry Module)

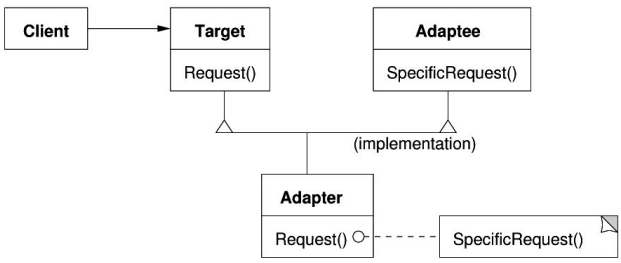

Fig. 2. Adapter design pattern, after (Gamma *et al.* 1995)

(Tautges 2001) library as the common denominator for geometric models. For meshes we use `moab::Core` class from MOAB library as the mesh proxy representation.

The translation between data structures can also be done by writing/reading them in some common data file format. In case of FEMDK we commonly use two such formats: file formats of VTK library for geometric and mesh data, and HDF5 data format for general data structures.

Integration on application level

In case of applications the integration can mean:

- Extending the application so that it can share data using some common data format.
- Providing batch processing facilities to interactive programs.
- Enabling the use of application in client–server architecture. Here various approaches can be taken: named pipes, threads, sockets, remote procedure call techniques, etc. The choice between them depends on one's ability to modify application source code.
- Rebuilding the application to turn it into a library (if possible).

Integration on user interface level

By integration on this level we mean providing common uniform user interface to components. In case of graphical user interface it means providing appropriate widgets for entering/displaying data. In case of command line interface it usually means providing appropriate command line options to configure and run components.

Integration on software build level

It should be possible to build and install all integrated components (as well as their dependencies) in some uniform and automated manner. This supports portability and is also important because FEMDK and some of its dependencies are meant to be used in source form to allow for their modifications and enhancements. To achieve this we use Dorsal tool developed for FEniCS (Logg and Wells 2010) project. Dorsal consists of a bash script and configuration data base, and it manages components download, configuration, building and installation – all this by delegating the job to commonly used programs like svn, cvs, make, CMake, autotools, etc.

Integration on documentation level

By this we understand providing a convenient way to access documentation of all components in question, and to navigate through this documentation.

3.2. Development of new tools for FEMDK

FEMDK makes it also necessary to design and build new software tools. Most of the tools are for special pre- and postprocessing tasks. As an example of such tool we can mention BGD (Basic Geometric Domain) library which is part of FEMDK. We have observed that many test data or common examples are based on simple geometric domains. The use of general geometric modelling tools to define such domains can be awkward, because many of the geometric and topological parameters are fixed, and the user wants a quick way to set the variable ones. To address this problem we have designed BGD library to provide descriptions of the most common geometric domains in 2D and 3D. Using this library we can quickly set up geometric model and convert it to suitable geometric data structures, for instance for tessellating it. Listing 1 shows the example of using BGD library to define and tessellate an I-type cross-section beam. In this example we use two classes from BGD library: `bgd::IShape` to define beam cross-section and `bgd::ExtrudedShape` to extrude the cross-section into a 3D object.

Listing 1: Code for generating unstructured mesh over extruded geometry.

```
1   getfem::mesh mesh;
2   femdk::GMSHMesher mesher;
3   femdk::bgd::IShape ishape;
4   ishape.SetVertexLC(0.1);
5   femdk::bgd::ExtrudedShape beam(&ishape);
6   ishape.SetVertexLC(0.1);
7   beam.SetParam(„dz", 1.0);
8   getfem::mesh beammesh;
9   mesher.generate(beam, beammesh);
10  getfem::vtk_export exportVTK(„ibeam.vtk", true);
11  exportVTK.exporting(beammesh);
12  exportVTK.write_mesh();
```

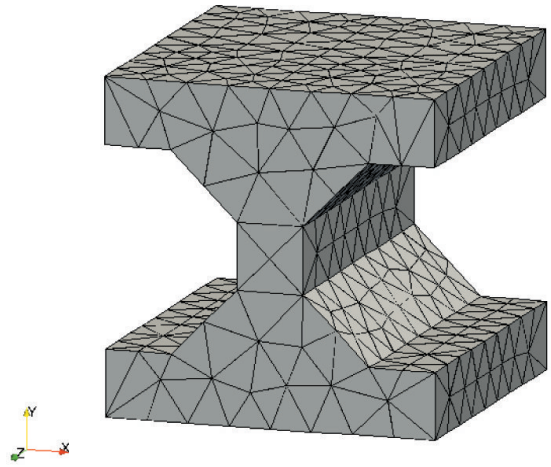

Fig. 3. Mesh resulting from code in Listing 1

It is also worth noticing that in line 8 we use external mesh generator gmsh for which an integration wrapper was build. This wrapper forks the main process and calls gmsh as a child process.

4. SOFTWARE SELECTION DECISIONS

In this section we present rationales for our choice of software components for FEMDK project. We comment on the tools we have chosen but also give some remarks on potential alternatives.

4.1. FEM library

The heart of FEMDK project is so-called FEM Kernel – a library which supports implementation of Finite Element Method. The main criteria for selecting FEM kernel library were: the support for automatic compilation of variational forms, the ability to handle simplicial and other meshes, support for multiple fields, support for XFEM methods. While there are at least a couple of libraries with these features we have chosen GetFEM++ (GetFEM++ 2010). Other candidates considered were Deal.II, libMesh, Dolfin. Compared to other libraries GetFEM++ seemed to put minimum restrictions on the kind of Finite Element method that can be implemented with it. For instance it supports simplicial, cubical and some non-standard families of finite elements while, otherwise very attractive, Deal.II library supports only cubical family of elements.

4.2. Visualisation library

The decision concerning selection of components for visualisation services is twofold. Firstly, one has to decide whether some very high level libraries should be used or one stays at relatively low level of OpenGL library and some simple wrappers for it. A high level visualisation toolkits like VTK provide almost all imaginable support but are complex – they generate overhead in terms of memory consumption and processing speed. To gain maximum efficiency they must be properly handled. For some types of applications it may make more sense to stick to relatively low level OpenGL library and some drivers for it, for instance QwtPlot3D or ifv++. For FEMDK however, we have decided to use VTK based solutions.

VTK library can be an excellent tool for data visualisation task but it might not be as handy for general 3D graphics, for instance for building a graphical preprocessor. For FEMDK we have decided to use HOOPS 3D library by TechSoft 3D (HOOPS 2010) which is the state-of-the-art 3D graphics system, proprietary, but available free of charge for research purposes.

4.3. Mesh handling library

For an environment such as FEMDK the crucial issue is to ensure support to exchange data between various mesh data structures in memory, as well as between various data file formats. To handle such tasks the MOAB (Mesh Oriented dAta Base) (MOAB 2010) library was selected. MOAB provides comprehensive, scalable, efficient solution for handling structured and unstructured meshes and for associating any data with mesh elements. What is more, it is an implementation of the iMesh interface developed by the Interoperable Technologies for Advanced Petascale Simulations (ITAPS) center, and can be used with other interfaces for geometry and filed information exchange. At one point as a viable alternative to MOAB the GrAL library was considered, however MOAB was chosen because of more vivid developers and users community.

4.4. GUI library

Although FEMDK is not exclusively graphical user interface oriented environment, GUI library plays important role in it. The choice of a GUI library is the hard one, as such library must be compatible with several other graphical components. It also determines to large extent the potential for portability of the whole environment. For FEMDK three candidates were seriously considered Qt, wx and FLTK.We have decided to use Qt, because beside GUI it supports development of non-GUI applications and provides many general purpose programming tools.

4.5. Scripting extension language

The tasks envisioned for FEMDK environment mandate the provision of a scripting interface. The choice of a scripting language is an important decision as the scripting language will contribute at large to how the users perceive the whole environment. In case of FEMDK three languages were considered: Tcl, Guile and Python. From the technical point of view there is not much difference in how interpreters for these languages are embedded in an application, nor how the extension for them can be written. For some time we have considered Guile as the main candidate. Guile is interpreter of Scheme which in turn is a functional language, and as such it has some attractive features different from the two other candidates. In the end, however, we have decided to use Python, because of already existing Python interfaces to components we would like to use in FEMDK. Python was also indicated as the preferred language by some FEMDK users.

4.6. Configuration tools

As FEMDK is more targeted at developers than at endusers the choice of software configuration and building tools is a decision that will affect potential users. Because of the use of Qt library its qmake tool was our first choice, but it has turned out that it is not flexible enough to our needs. Autotools (i.e. Automake, Autoconf, Libtool, etc) seem rather to be rooted in UNIX-like systems, and they demand somehow more experience from the users. In the end, we have decided to use CMake which is portable and able to support our various needs.

4.7. Other components used by FEMDK project

Because of constrained resources we could not select other components with the same amount of attention. The ones mentioned below were selected simply because we already have some experience in using them.

– Handling of scientific data – for persistent data storage we use HDF5. It is a data model, library and file format flexible enough to accommodate all our requirements. HDF5 is also supported by some of our other components.

– Automation of multi language programming – for this task we use `SWIG` (Simplified Wrapper and Interface Generator).
– Solver libraries – there are several candidates for this kind of components and we have decided to investigate the `Trilinos` framework (The Trilinos Project 2010).
– Mesh generation tools – here we are restricted by the available tools. We plan to closely integrate in our environment the following generators: `gmsh`, `geompack++`, `triangle`.
– Symbolic computing – the use of symbolic computing is in our "wish list", nevertheless we have started to investigate pros and cons of using Maxima computer algebra system.

5. GETFEM++ AS A FINITE ELEMENT KERNEL FOR FEMDK

As said in point 4.1 the main motivation for selecting `GetFEM++` library was its modular design and the very week assumptions about the nature of finite element method that can be implemented with `GetFEM++`. There is however a price to pay for that, and this price is obfuscation of otherwise straightforward trunk of FEM algorithm. This obfuscation is due to abstract design patterns and several indirection layers which constitute the mechanism that allows one to decouple components of the system, for instance providing a way to write generic algorithms independent on the concrete data structures they operate on. The problem is that this whole machinery is seldom discussed in presentation of FEM algorithms and grasping it (together with intricacies of C++ language) can be challenging.

The aim of this section is to comment on `GetFEM++` main components and provide a sort of succinct guide allowing readers to link `GetFEM++` classes with the mathematical or computational concepts the readers are familiar with.

In order to support implementation of Finite Element Method for a wide range of problems, `GetFEM++` provides the following facilities:

– support for matrix algebra on small, dense vectors and matrices,
– support for solving linear algebraic equations and eigenvalue problems,
– methods for interpolation, differentiation and integration over tessellated domains,
– support for generic description of mathematical models,
– support for integration of ordinary differential equations,
– support for equation assemblage,
– support for non-linear algebraic equations,
– support for level sets and XFEM methods,
– support for automatic interpolation between non-matching meshes,
– various post-processing utilities.

Presenting all the above subsystems is out of the limits of this paper, thus only two simple examples demonstrating some of them will be discussed. The examples are selected from the point of view of implementing non-stationary and non-linear models.

5.1. Implementing non-stationary models with GetFEM++

In this example we will consider a very simple mathematical model for projectile motion. Assuming that we neglect air resistance the set of equations that describe the motion can be written as:

$$\begin{aligned} \frac{dv_x}{dt} &= 0 \\ \frac{dv_y}{dt} &= -g \\ \frac{dx}{dt} - v_x &= 0 \\ \frac{dy}{dt} - v_y &= 0 \end{aligned} \tag{1}$$

with initial conditions $x(0) = x_0, y(0) = y_0, v_x(0) = v_{x0}$, $v_y(0) = v_{y0}$. Introducing the state vector $\mathbf{U}$, defined as

$$\mathbf{U} = \begin{bmatrix} v_x \\ v_y \\ x \\ y \end{bmatrix},$$

we can cast equation (1) into matrix form

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{K}\mathbf{U} = \mathbf{F} \tag{2}$$

where

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} 0 \\ -g \\ 0 \\ 0 \end{bmatrix}$$

and initial conditions are respectively

$$\mathbf{U}_0 = \begin{bmatrix} v_{x0} \\ v_{y0} \\ x_0 \\ y_0 \end{bmatrix}.$$

This matrix form will be the basis for the implementation with `GetFEM++` library.

The central GetFEM++ class for implementation of mathematical models is getfem::model. It allows one to declare model variables and data, build model state equation and solve it. Listing 2 shows a piece of the top level solution procedure.

Listing 2: Code for building mathematical model for projectile motion.

```
1   /* Explicit matrix and right hand side
2   * of the model tangent equation
3   */
4   gmm::dense_matrix<scalar_type> M(4,4);
5   gmm::clear(M);
6   M(0,0) = 1.0;
7   M(1,1) = 1.0;
8   M(2,2) = 1.0;
9   M(3,3) = 1.0;
10
11  gmm::dense_matrix<scalar_type> K(4,4);
12  gmm::clear(K);
13  K(2,0) = -1;
14  K(3,1) = -1;
15
16  getfem::scalar_type g = 9.81;
17
18  std::vector<scalar_type> F(4);
19  F[1] = -g;
20
21  getfem::model model;
22
23  /* specify model state variable and model parameters */
24  model.add_fixed_size_variable(„U", 4, 2);
25
26  scalar_type dt = 0.01 /* fixed timestep */;
27  model.add_initialized_scalar_data(„dt", dt);
28
29  add_explicit_d_on_dt_brick(model, „U", „dt", M);
30
31  getfem::size_type bi;
32  dal::bit_vector transient_bricks;
33
34  bi = add_explicit_matrix(model, „U", „U", K);
35  transient_bricks.add(bi);
36
37  bi = add_explicit_rhs(model, „U", F);
38  transient_bricks.add(bi);
39
40  scalar_type theta = 0.5;
41
42  model.add_initialized_scalar_data(„theta", theta);
43  getfem::add_theta_method_dispatcher(model,
44  transient_bricks, „theta");
```

In lines 1–19 we define matrices for our state equation. In line 21 we create an instance of getfem::model. In line 24 we say that the model will have fixed size (as opposed to mesh based) state variable. In lines 29–38 we add objects describing successive terms of the state equation (2). In line 43 we add time dispatcher object which manages time integration.

In case of non-stationary problems the main loop over time or any pseudo time variable should be provided by the user. In the case of the considered state equation it can be implemented as shown in Listing 3.

Listing 3: Implementation of time integration loop.

```
1   // State vector U
2   std::vector<getfem::scalar_type> U(4);
3   gmm::clear(U);
4   // projectile motion initial conditions
5   double angle = 45*M_PI/180;
6   double v0 = 1.0;
7   double x0 = 0.0;
8   double y0 = 0.0;
9   U[0] = v0 * cos(angle);
10  U[1] = v0 * sin(angle);
11  U[2] = x0;
12  U[3] = y0;
13
14  //Initialize model state variable at
15  //current(0) and previous(1) timestep
16  gmm::copy(U, model.set_real_variable(„U",0));
17  gmm::copy(U, model.set_real_variable(„U",1));
18
19  scalar_type residual = 1e-5;
20  gmm::iteration iter(residual, 0, 100);
21
22  scalar_type sol_t = 0.0;
23  model.first_iter();
24  bool go_ahead = true;
25  while(true == go_ahead) {
26  iter.init();
27  sol_t += dt; // advance current time
28
29  getfem::standard_solve(model, iter);
30  gmm::copy(model.real_variable(„U", 0), U);
31
32  /* Print solution */
33  std::cout << sol_t << „ „ << U << „\n";
34
35  if (U[3] < 0.0) { // hit the ground ?
36  go_ahead = false;
37  } else {
38  /* call time dispatcher */
39  model.next_iter();
40  }
41  }
```

Linear algebraic equations system resulting from discretisation of the state equation is solved in line 29. The update of the state variable on successive time steps is done in line 39 according to a time dispatcher added to each term of the state equation as shown in lines 32–44 in Listing 2.

For our simple equation it is enough to use getfem::theta_method_dispatcher. It allows to integrate the ODE system with forward Euler method ($\vartheta = 0.0$), backward Euler method ($\vartheta = 1.0$) or midpoint method ($\vartheta = 0.5$). The discretisation of time derivative for the term

$$\mathbf{M}\frac{d\mathbf{U}}{dt}$$

is provided by the class explicit_d_on_dt_brick for which the user can provide matrix M. This brick is not provided in GetFEM++ library but it is relatively easy to implement it taking as an example the class getfem::basic_d_on_dt_brick. This implementation is shown in

Listing 4. The most important lines in this code are lines 39, 40, 31. They are implementation of the following operation:

$$\mathbf{M}\frac{d\mathbf{U}}{dt} \rightarrow \mathbf{M}\frac{\mathbf{U}_{i+1}-\mathbf{U}_i}{\Delta t} \rightarrow \begin{cases} \frac{1}{\Delta t}\mathbf{M} & \text{to tangent matrix} \\ \frac{1}{\Delta t}\mathbf{M}\mathbf{U}_i & \text{to right hand size} \end{cases}$$

In line 39 we set the tangent matrix of the brick to $\mathbf{M}/\Delta t$. In line 40 we augment the right hand side of the brick with $1/\Delta t\mathbf{M}\mathbf{U}_i$ where $\mathbf{U}_i$ means the state variable on the already completed time step. These three lines should be self-explanatory. The remaining lines are related either to `GetFEM++` framework or can be seen as so-called syntactic noise of the programming language (in this case C++). The burden of these lines is the price to pay for having the same time code flexibility and generality on one hand and program efficiency on the other[1].

Listing 4: Implementation of time integration brick with explicit mass matrix.

```
1   namespace getfem {
2
3   struct have_private_data_brick :
4   public virtual_brick {
5   model_real_sparse_matrix rB;
6   model_complex_sparse_matrix cB;
7   model_real_plain_vector rL;
8   model_complex_plain_vector cL;
9   };
10
11  struct explicit_d_on_dt_brick :
12  public have_private_data_brick {
13  virtual void
14  asm_real_tangent_terms(const model &md,
15  size_type ib,
16  const model::varnamelist &vl,
17  const model::varnamelist &dl,
18  const model::mimlist &mims,
19  model::real_matlist &matl,
20  model::real_veclist &vecl,
21  model::real_veclist &,
22  size_type region,
23  build_version version) const {
24  GMM_ASSERT1(matl.size() == 1,
25  „Basic d/dt brick has one and only one term");
26  GMM_ASSERT1(mims.size() == 0,
27  „Explicit d/dt brick does not need mesh_im");
28  GMM_ASSERT1( vl.size() == 1
29  && dl.size() >= 2
30  && dl.size() <= 3,
31  „Wrong number of variables for basic d/dt brick");
32
33  const model_real_plain_vector &dt =
34  md.real_variable(dl[1]);
35  GMM_ASSERT1(gmm::vect_size(dt) == 1,
36  „Bad format for time step");
37
38  gmm::clear(matl[0]);
39  gmm::copy(rB, matl[0]);
40  gmm::scale(matl[0], scalar_type(1) / dt[0]);
41  gmm::mult(matl[0], md.real_variable(dl[0], 1), vecl[0]);
42  }
43
44  explicit_d_on_dt_brick(void) {
45  set_flags(„Basic d/dt brick", true /* is linear*/,
46  false /* is symmetric */, false /* is coercive */,
47  true /* is real */, true /* is complex */);
48  }
49
50  };
51
52  size_type add_explicit_d_on_dt_brick
53  (model &md, const std::string &varname,
54  const std::string &dataname_dt) {
55  pbrick pbr = new explicit_d_on_dt_brick;
56  model::termlist tl;
57  tl.push_back(model::term_description(varname,
58  varname, true));
59  model::varnamelist dl(1, varname);
60  dl.push_back(dataname_dt);
61  return md.add_brick(pbr, model::varnamelist(1, varname),
62  dl, tl, model::mimlist(), size_type(-1));
63  }
64
65  template <typename MAT>
66  size_type add_explicit_d_on_dt_brick(model &md,
67  const std::string &varname1,
68  const std::string &dataname,
69  const MAT &B) {
70  size_type ind = add_explicit_d_on_dt_brick(md,
71  varname1, dataname);
72  set_private_data_matrix(md, ind, B);
73  return ind;
74  }
75
76  } /* namespace getfem */
```

6. IMPLEMENTING COUPLED PROBLEMS WITH FEMDK

6.1. Summary of computational thermoelasticity

We start the analysis by assuming small strains and writing the standard equation of equilibrium (momentum balance) valid at each point of the considered isotropic solid in Voigt's notation:

$$\mathbf{L}^T\boldsymbol{\sigma} + \rho\mathbf{b} = 0 \tag{3}$$

where $\mathbf{L}$ is a suitable matrix of differential operators, $\boldsymbol{\sigma}$ is the stress tensor, ρ is the material density and $\mathbf{b}$ is the body force vector. The equation should be complemented by proper boundary conditions. It can be written in terms of displacements $\mathbf{u}$ which are usually the discretized fundamental unknown field.

We assume that the stress $\boldsymbol{\sigma}$ is derived from a certain free energy functional and is related to strain $\boldsymbol{\varepsilon}$ according to:

$$\boldsymbol{\sigma} = \mathbf{E}\left(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^{\theta}\right), \quad \mathbf{E} = \mathbf{E}(\boldsymbol{\varepsilon}, \theta), \quad \boldsymbol{\varepsilon}^{\theta} = \alpha\theta\boldsymbol{\Pi} \tag{4}$$

[1] There are some projects which solve this problem by providing special language for mathematical modelling and a compiler that translates abstract, high level constructions into efficient code. With maturing middle level tools for finite element method this approach will, in author's opinion, gain more widespread attention in future.

where $\boldsymbol{\varepsilon}^{\theta}$ are thermal strains, _ is the expansion coefficient, $\theta = T - T_0$ is the relative temperature, i.e. its increase with respect to strain-free (initial, reference) temperature T_0. The stiffness matrix **E** can depend on $\boldsymbol{\varepsilon}$ and θ, I is the unit tensor in vector form, $\boldsymbol{\Pi} = [1, 1, 1, 0, 0, 0]$. Note that for linear kinematics the strain is computed from the displacement vector as $\boldsymbol{\varepsilon} = \mathbf{L}\mathbf{u}$.

Then, for non-stationary heat conduction the balance equation reads:

$$\rho c\dot{\theta} + \nabla^{\mathrm{T}}\mathbf{q} = r \tag{5}$$

where c the specific heat capacity, **q** the heat flux density and r is the heat source density. Again, proper boundary conditions have to be specified. The heat flux density is usually expressed in terms of temperature gradient (Fourier's law):

$$\mathbf{q} = -\boldsymbol{\Lambda}\nabla\theta, \quad \boldsymbol{\Lambda} = \boldsymbol{\Lambda}(\theta) \tag{6}$$

where $\boldsymbol{\Lambda}$ is the conductivity matrix which can depend on θ.

The local mathematical model can be reworked into global weak formulation using the weighted residuum approach. The weak-form equilibrium equation, upon substitution of constitutive relation (4), is written as

$$\begin{aligned}&\int_V (\mathbf{L}\mathbf{v}_{\mathrm{u}})^{\mathrm{T}}\mathbf{E}\varepsilon \mathrm{d}V - \int_V (\mathbf{L}\mathbf{v}_{\mathrm{u}})^{\mathrm{T}}\mathbf{E}\boldsymbol{\Pi}\alpha\theta \mathrm{d}V = \\ &= \int_V \mathrm{v}_{\mathrm{u}}^{\mathrm{T}}\rho\mathbf{b}\mathrm{d}V + \int_S \mathrm{v}_{\mathrm{u}}^{\mathrm{T}}\mathbf{t}\mathrm{d}S \quad \forall\mathbf{v}_{\mathrm{u}}\end{aligned} \tag{7}$$

where **t** are the tractions. The weak form of the heat conduction equation, upon substitution of Fourier's equation (6), reads

$$\begin{aligned}&\int_V \mathbf{v}_{\theta}\rho c\dot{\theta}\mathrm{d}V + \int_V (\nabla\mathbf{v}_{\theta})^{\mathrm{T}}\boldsymbol{\Lambda}\nabla\theta \mathrm{d}V = \\ &= \int_V \mathbf{v}_{\theta} r \mathrm{d}V - \int_S \mathbf{v}_{\theta} q_n \mathrm{d}S \quad \forall\mathbf{v}_{\theta}\end{aligned} \tag{8}$$

where q_n is the heat flux normal to the body surface. The two weak-form equations, in which $\mathbf{v}_{\mathrm{u}}$ and $\mathbf{v}_{\theta}$ contain weight functions, are augmented by suitable essential boundary conditions. Equations (7) and (8) are assumed to be coupled one-way, i.e. the heat conduction induces thermal strains and stresses, see for instance (Nicholson 2008). For a general case of two-way coupling the Reader is referred for instance to (Maugin 1992). If the two balance equations are not coupled both ways, one can solve the problem of non-stationary heat flow first and then consider the temperature-dependent elasticity problem of stress evolution. This is the way thermomechanical coupling is often implemented in commercial codes.

Such one-way coupling of momentum balance to heat conduction is also considered here, but using a monolithic incremental-iterative scheme. The backward Euler scheme is employed to integrate over time. The weak-form equations (7) and (8) are linearized at the current time step $t + \Delta t$. The fundamental unknown fields of displacement and temperature are discretized using a suitable finite element interpolation. The following matrix equations are obtained:

$$\begin{bmatrix}\mathbf{K}_{uu} & \mathbf{K}_{u\theta} \\ \mathbf{0} & \mathbf{K}_{\theta\theta}\end{bmatrix}\begin{bmatrix}\mathrm{d}\breve{\mathbf{u}} \\ \mathrm{d}\breve{\boldsymbol{\theta}}\end{bmatrix} = \begin{bmatrix}\mathbf{f}_{ext} - \mathbf{f}_{int}^{i} \\ \mathbf{Q}_{ext} - \mathbf{Q}_{int}^{i}\end{bmatrix} \tag{9}$$

where $\mathbf{K}_{uu}$ is the tangent operator, $\mathbf{K}_{u\theta}$ is the coupling matrix, $\mathbf{K}_{\theta\theta}$ is the algorithmic matrix related to heat capacity and conduction, d denotes iterative correction of a quantity, $\breve{\mathbf{u}}$ and $\breve{\boldsymbol{\theta}}$ are nodal displacements and nodal temperatures, respectively, and finally the right-hand side contains out-of-balance forces and heat source terms.

A generalization of the mathematical formulation of multi-field problems can be found for instance in (Kuhl 2005), while the extension of the theory to multicomponent materials (together with its thermodynamic background) is presented for instance in (Kubik 2004).

6.2. Extending GetFEM++ bricks system

The equations outlined in the previous section can be directly used to implement thermo-mechanical coupling brick for `GetFEM++`. We should look at equation (9), especially at its left hand side. In `GetFEM++` there are already functions to add bricks that calculate terms $\mathbf{K}_{uu}$ and $\mathrm{K}_{\theta\theta}$, they are `add_isotropic_linearized_elasticity_brick` and `add_generic_elliptic_brick`, respectively. What is left is the implementation of the brick for the term $\mathbf{K}_{u\theta}$. While we will not show the full source code of the brick as it is too long and full of technical intricacies, we would like to show two functions which are at the core of the implementation of such brick. Both functions do the same – they assemble the coupling term, however the first function is for the case when material properties can vary within the domain, while the second is for the case of constant properties, that is constant elastic properties and constant thermal expansion coefficient. The code is given in Listing 5. The reader should note direct equivalence of the term

$$\int_V (\mathbf{L}\mathbf{v}_{\mathrm{u}})^{\mathrm{T}}\mathbf{E}\alpha\theta \mathrm{d}V$$

from equation (8) and the lines 19–24. The operator **L** is hidden in line 22. The stiffness matrix **E** is expressed with Lame's coefficients μ and λ. The operator `vGrad` means vectorized gradient of element shape functions, and the operator `Base` calculates the values of element shape functions. The numbers after # sign in the arguments to these operators denote finite element meshes where the operators are applied. Using this we can handle a situation where material properties, displacement field and temperature field are approximated with different finite elements or even on completely different meshes.

The textual description of terms assembly is be parsed by `GetFEM++` library and automatically converted into appropriate assembly code.

Listing 5: Implementation of thermo-mechanical coupling brick.

```
template<class MAT, class VECT>
void asm_stiffness_matrix_for_linear_thm
(const MAT &RM_, const mesh_im &mim,
const mesh_fem &mf_u,
const mesh_fem &mf_t,
const mesh_fem &mf_data,
const VECT &LAMBDA,
const VECT &MU,
const VECT &ALPHA,
const mesh_region &rg =
mesh_region::all_convexes()) {
MAT &RM = const_cast<MAT &>(RM_);
GMM_ASSERT1(mf_data.get_qdim() == 1,
„invalid data mesh fem (Qdim=1 required)”);

GMM_ASSERT1( mf_u.get_qdim()
== mf_u.linked_mesh().dim(),
„wrong qdim for the mesh_fem”);
generic_assembly assem(„lambda=data$1(#3)”
„mu=data$2(#3);”
„alpha=data$3(#3);”
„t=comp(vGrad(#1).Base(#2).Base(#3));”
„M(#1,#2)= -t(:,i,i,:,k).((3*lambda(k)
+2*mu(k))*alpha(k));”
);
assem.push_mi(mim);
assem.push_mf(mf_u);
assem.push_mf(mf_t);
assem.push_mf(mf_data);
assem.push_data(LAMBDA);
assem.push_data(MU);
assem.push_data(ALPHA);
assem.push_mat(RM);
assem.assembly(rg);
}

template<class MAT, class VECT>
void asm_stiffness_matrix_for_homogeneous_linear_thm
(const MAT &RM_, const mesh_im &mim,
const mesh_fem &mf_u,
const mesh_fem &mf_t,
const VECT &LAMBDA,
const VECT &MU,
const VECT &ALPHA,
const mesh_region &rg = mesh_region::all_convexes()) {
MAT &RM = const_cast<MAT &>(RM_);
GMM_ASSERT1(mf_u.get_qdim() == mf_u.linked_mesh().dim(),
„wrong qdim for the mesh_fem”);
generic_assembly assem(„coeff=data$1(1);”
„t=comp(vGrad(#1).Base(#2));”
„M(#1,#2)+= -t(:,i,i,:).coeff(1);”
);
assem.push_mi(mim);
assem.push_mf(mf_u);
assem.push_mf(mf_t);
VECT data(1);
data[0] = (3*LAMBDA[0] + 2*MU[0])*ALPHA[0];
assem.push_data(data);
assem.push_mat(RM);
assem.assembly(rg);
}
```

6.3. Thermo-mechanical analysis benchmark

In order to test our implementation we have considered problem for which an analytical solution can be found. In this problem an annulus geometry is loaded with internal and external pressure and with thermal load. Listing 6 shows declaration of `femdk::ExactAxisymThm` class which can calculate the exact solution. For temperature field Dirichlet, Neumann and Robin boundary conditions can be specified.

Listing 6: Class for calculating exact solution of a thermo-mechanical problem for annulus geometry.

```
namespace femdk {
typedef enum {
/** Dirichlet condition u = v */
BC_DIRICHLET = 0,
/** Neumann condition du/dn = v */
BC_NEUMANN = 1,
/** Robin condition a*du/dn + bu = v */
BC_ROBIN = 2,
} BCType_t;

using bgeot::base_small_vector;
using getfem::base_node;

class ExactAxisymThm {
public:
ExactAxisymThm();
void setGeometry(double innerRadius,
double outerRadius);
void setMechConditions(double innerPressure,
double outerPressure);
void setThermConditions(BCType_t inner,
BCType_t outer,
int size,
double *data);
void setRefTemperature(double Tref);
void setMaterial(double E,
double nu,
double alpha);

void setModeToEvalT(void);
void setModeToEvalU(void);
void setModeToEvalSigma(void);
void setEvalMode(char mode);

double evalTAt(double r);
double evalTAt(const base_node &p);
double evalUAt(double r);
double evalUAt(const base_node &p);
/** Compute stress tensor in cylindrical coordinates.
*/
void evalSigmaAt(double r, base_small_vector &sigma);
void evalSigmaAt(const base_node &p,
base_small_vector &sigma);
bgeot::base_small_vector operator ()(double r);
bgeot::base_small_vector operator ()(const base_node &p);
};
```

Figure 4 shows the problem setup where constant temperature and pressure are applied on internal and external boundary. Plane strain conditions are assumed. The load and material parameters are given in Table 1.

Table 1
Material and load parameters for thermoelasticity benchmark

Property	Symbol	Value	Unit
Density	ρ	2300	kg/m^3
Poisson's ratio	ν	0.21	
Modulus of elasticity	E	32 000	MPa
Coeff. of ther. expansion	α	10^{-5}	1/°C
Specific heat capacity	c_p	0.75	kJ/(kg·K)
Thermal conductivity	k	1.7	W/(m·K)
Reference temperature	T_0	0	°C
Inner wall temperature	T_i	20	°C
Outer wall temperature	T_o	200	°C
Inner wall pressure	p_i	1100	Pa
Outer wall pressure	p_o	1700	Pa

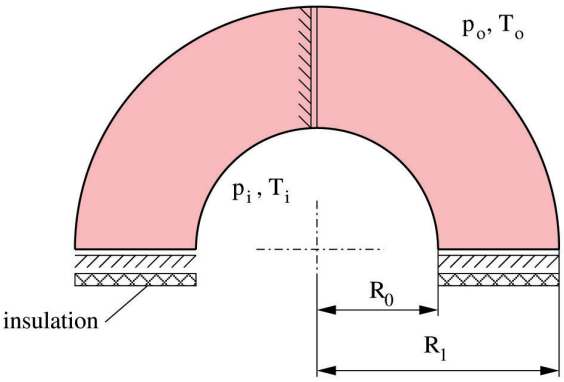

Fig. 4. Geometry and load setup for the benchmark

The results obtained from our code are in good agreement with analytical results as shown in Figures 6 and 7. Thanks to the flexibility of GetFEM++ kernel we were able to solve the problem in several configurations, for instance setting different meshes for temperature and displacement fields.

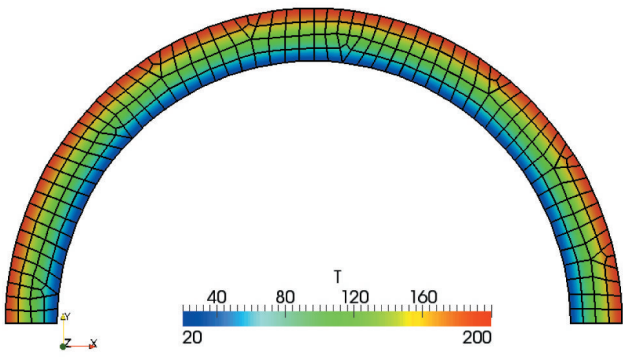

Fig. 5. Finite element mesh and temperature field

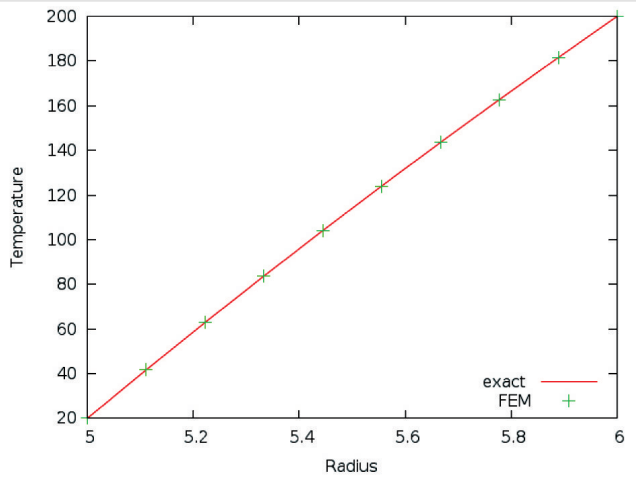

Fig. 6. Comparison of analytical and FEM results for temperature

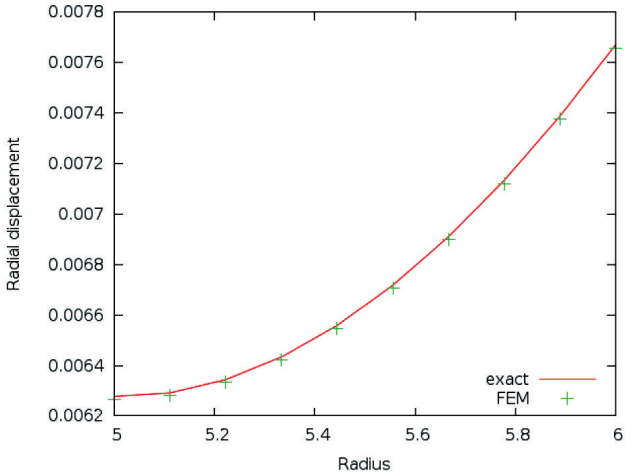

Fig. 7. Comparison of analytical and FEM results for radial displacement

7. SUMMARY

With this paper we would like to share our experience with building FEM tools for analysis of coupled problems. Using an example of thermoelasticity we have shown how the mathematical formulation can be cast into implementation based on `GetFEM++` library as the finite element kernel of our FEMDK project. We hope that the examples and comments attached to them can help to make more concious decisions regarding the selection of software components for FEM analysis. As for FEMDK project we are developing it in two directions. One is implementation of more complex material models for elastoplasticity, damage and fracture mechanics. The other is building more user friendly interface that will lower the barriers related to complexity of Get FEM++ kernel and the burden of C++. One possible way to do it is to enable some computation to be done via Python scripts. Surely this will lower the performance of programs but we will gain in terms of performance of programmers, which is more important for our research on finite element method itself.

Acknowledgments

Scientific research has been carried out as a part of the project „Innovative recourses and effective methods of safety improvement and durability of buildings and transport infrastructure in the sustainable development" financed by the European Union from the European Fund of Regional Development based on the Operational Program of the Innovative Economy.

References

Gamma E., Helm R., Johnson R.E., Vlissides J. 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Gawin D. 2010, *Degradation processes in microstructure of cement composites at high temperature*. Number Engineering studies no 69. Polish Academy of Sciences, Warsaw (in Polish).

GetFEM++ 2010, *Homepage*, http://download.gna. org/getfem/html/homepage/index.html.

HOOPS 2010, 3D *Framework*, http://developer. techsoft3d.com/hoops/index.html.

Kuhl D. 2005, *Modellierung und Simulation von Mehrfeldproblemen der Strukturmechanik*. Ph.D. dissertation, Ruhr University of Bochum, Bochum.

Kubik J. 2004, *Elements of the thermomechanics.* Politechnika Opolska, Opole (in Polish).

Logg A., Wells G.N. 2010, *DOLFIN: Automated finite element computing*. ACM Transactions on Mathematical Software, 37(2), pp. 417–444.

Maugin G.A. 1992, *The Thermomechanics of Plasticity and Fracture*. Cambridge University Press, Cambridge.

MOAB: 2010, *A Mesh-Oriented datABase*, http://trac. mcs.anl.gov/projects/ITAPS/wiki/MOAB.

Nicholson D.W. 2008, *Finite element analysis. Theromechanics of solids*. CRC Press, Boca Raton.

Oldehoeft R. 2002, *Taming complexity in high performance computing*. [in:] Computational science, mathematics and software, Ronald F. Boisvert and Elias N. Houstis (Eds.). Purdue University Press, West Lafayette, IN, USA, pp. 57–77.

Tautges T.J. 2001, *The Common Geometry Module (CGM): a Generic, Extensible Geometry Interface*, Engineering with Computers, 17(3), pp. 299–314.

The Trilinos Project 2010, http://trilinos.sandia. gov.