Available online at www.sciencedirect.com

ScienceDirect

Advances in Engineering Software 39 (2008) 1010–1022

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

Asynchronous interface between a finite element commercial software *ABAQUS* and an academic research code *HEREZH++*

G. Rio [a], H. Laurent [a,*], G. Blès [b]

[a] *Laboratoire Génie Mécanique et Matériaux, Université de Bretagne-Sud, BP 92116, 56321 LORIENT cedex, France*
[b] *Laboratoire Mécanique des Structures Navales, ENSIETA, 2 rue François Verny, 29806 BREST cedex 09, France*

Received 19 April 2007; accepted 18 January 2008
Available online 7 March 2008

Abstract

The aim of this paper is to describe an efficient method to connect two independent softwares so as to jointly use best qualities of each software around a complex problem solved by the finite element method (FEM). This connection makes it possible to extend quickly and easily the applicability of new models developed in academic softwares, by their simultaneous use with commercial softwares. This is particularly interesting when these models are very difficult to implement directly in commercial softwares.

Most of the commercial FEM applications allow users to add additional features, physical models or boundary conditions via a programming interface. Within these user routines, access to internal data structures is possible, either through subroutine parameters and global variables, or via internal modules for reading and storing data. We use these capabilities to link the commercial software *ABAQUS* and an academic object-oriented *C++* software *HEREZH++*, via the user-defined mechanical material behaviour (*Umat*). In this interface, *HEREZH++* computes the mechanical behaviour of material and the code coupling performs a communication procedure between *ABAQUS* and *HEREZH++*. This paper describes this architecture which allows to implement easily original behaviour law in the commercial *ABAQUS* code. The asynchronous code coupling is made with a named piped interprocess communication method and an interface written in *c/C++*. Several test samples are presented to show the efficiency and accuracy of the proposed implementations concerning the computational time. In particular, an industrial test is carried out with an original behaviour model of elasto–visco-hysteresis which would have been very difficult to implement directly in *ABAQUS*.
© 2008 Elsevier Ltd. All rights reserved.

Keywords: Code coupling; Interprocess communication; *ABAQUS*; *HEREZH++*; *Umat*; Object-oriented programming; Finite element analysis

1. Introduction

Finite element analysis is an extremely powerful numerical technique for the solution of a complex linear/non-linear system equation representing physical/engineering processes. In this field, correctly describing the behaviour of the material is extra challenging when it comes to the processing of materials and manufacturing processes. Material models must account for varying strain, strain-rate, temperature, etc.

Such constitutive models could then be implemented by researchers as *User-defined Material* models within some of the commercial finite element codes. This approach allows researchers to focus on the material response modeling rather than the entire finite element software system. This approach is uncoupled from the commercial software vendor's business-driven priorities and permits material model development in unison with material characterization testing (e.g. [1–4]). Then, once the material model is matured and verified, discussion can take place with the commercial software vendor to integrate the new material model into their code as a fully functional option, including appropriate documentation and testing.

* Corresponding author.
E-mail address: herve.laurent@univ-ubs.fr (H. Laurent).

Usually, introduction of a new constitutive model in FE code is made with a user subroutine, named "*Umat*" in this paper, by reference to the user interface of Abaqus [5][1] similar interfaces can be found for instance in *LS-DYNA* [6], *CASTEM* [7], *ZEBULON* [8], etc. This subroutine is usually written in *FORTRAN*, with a fixed list of input/output data arguments. *Umat* has two major functions: it updates the stresses at the end of each time increment and it provides the material Jacobian matrix for the mechanical constitutive model, and is also executed for each material integration point at each iteration in each time increment. Tangent modulus of each integration point is calculated based on the stress level at the beginning of each time increment. The accuracy of the results depends on the size of the time increment. A smaller time increment will yield better results, but also results in longer computation time.

With this method, in most cases, numerical implementation of the models into finite element codes is straightforward, but this method can also cause some difficulties. The first difficulty appears during the crucial stage of testing ("debug"). Generally, this stage is a major step for industrial software because this code is optimized for its high speed time execution and not for debugging. A first solution would be to employ another external software, in using the same subroutine *Umat* but in simulating only the behaviour at only one integration point (e.g. software code *SiDoLo* [9]). In this case, however, it is not really possible to do any finite element computation. Another solution is to employ a second finite element software, which is perfectly controllable in debug mode, to develop the *Umat* (e.g. [8]).

Another difficulty in using subroutine *Umat* proceeds from the *FORTRAN* programming environment. Indeed, nowadays, several scientific developments are made in object-oriented language, for example in Smalltalk [10] or *C*++ [11,12], or in using object-oriented finite element framework (e.g. Femlab [13,14]). The object-oriented programming technique can greatly improve the implementation efficiency, extendibility, and ease of maintenance of large engineering software. But this technique is also difficultly introduced in a *Umat*, classically proposed in *FORTRAN 77* in *ABAQUS*, even if the *FORTRAN 90 or 95* norms have brought about some object-oriented method improvements.

In this context, we have tried to define a new form of software interface to link two codes: a commercial FE software *ABAQUS* and an object-oriented FE software, developed in *C*++, named *HEREZH*++ [15]. This software provides material behaviour computing, and transfers data to *ABAQUS* using *Umat* at each integration point and at each iteration of incremental step.

We use interprocess communication by named pipes to define the interface between the two different software initially operating asynchronously. *HEREZH*++ is managed by an independent process which can, in particular, define its self-memory location sizes. With this proposed technique, the development of new material behaviour and its debugging can be carried out with any programming language frame which assures final connection with commercial software without modifying the latter.

This paper is organized as follows. In Section 2, the User subroutine *Umat* to define a material's mechanical behaviour is introduced. Section 3 presents the in-house code *HEREZH*++ and its implementation in *C*++. In Section 4, the implementation of the code coupling between *ABAQUS* and *HEREZH*++ is discussed. The new code coupling is validated through some numerical examples using an elastic behaviour law in Section 5. Finally, to show the interest and the limitations of the code coupling, we use an original behaviour model, named elasto–visco-hysteresis [16–18] in an industrial test. With this material model, the hysteretic part normally needs larger dynamic memory storage and we show the advantages of this code coupling in this example.

2. *Umat*: User subroutine to define a material's mechanical behaviour

User-defined mechanical material behaviour in *ABAQUS* is provided by means of an interface whereby any mechanical constitutive model can be added to the library. It requires that a constitutive model (or a library of models) is programmed in *User subroutine Material UMAT* (*ABAQUS/Standard*) or *VUMAT* (*ABAQUS/Explicit*). User subroutines can be included in a model by using the user option on the *ABAQUS* execution procedure to specify the name of a FORTRAN source or object file that contains the subroutines. In *ABAQUS*'s documentation [5], some examples of application of *Umat* are proposed. The *Umat*:

- can be used to define the mechanical constitutive behaviour of a material;
- can be used with any procedure that includes mechanical behaviour;
- can use solution-dependent state variables;
- must update the stresses and solution-dependent state variables to their values at the end of the increment for which it is called;
- must provide the material Jacobian matrix, $\frac{\partial \Delta\sigma}{\partial \Delta\varepsilon}$, for the mechanical constitutive model.

The *ABAQUS* code calculates the strain increments for a time step and transmits them to the *Umat* subroutine at the beginning of each time equilibrium. This subroutine is called at all material calculation Gauss points of elements, with the call listed in Table 1.

According to the complexity of the user-defined material, some or all the parameters can be used. One can refer to the *ABAQUS*'s documentation [5], for the precise

[1] In this paper, we refer mainly to subroutine *Umat* used in the commercial software *ABAQUS*.

Table 1
Call of the user subroutine material *Umat*, which in our case, calls a *C* function, allowing the communication by named pipes

```
SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,SCD,
   1 RPL,DDSDDT,DRPLDE,DRPLDT,
   2 STRAN,DSTRAN,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CMNAME,
   3 NDI,NSHR,NTENS,NSTATV,PROPS,NPROPS,COORDS,DROT,PNEWDT,
   4 CELENT,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
C
     INCLUDE 'ABA_PARAM.INC'
C
     CHARACTER*80 CMNAME
     DIMENSION STRESS(NTENS),STATEV(NSTATV),
   1 DDSDDE(NTENS,NTENS),DDSDDT(NTENS),DRPLDE(NTENS),
   2 STRAN(NTENS),DSTRAN(NTENS),TIME(2),PREDEF(1),DPRED(1),
   3 PROPS(NPROPS),COORDS(3),DROT(3,3),DFGRD0(3,3),DFGRD1(3,3)
C
C  C function call for the communication by named pipe
     call appelc(%ref(STRESS), %ref(DDSDDE), %ref(SSE),
   1 %ref(SPD), %ref(SCD), %ref(RPL), %ref(DDSDDT),
   2 %ref(DRPLDE), %ref(DRPLDT), %ref(STRAN), %ref(DSTRAN),
   3 %ref(TIME), %ref(DTIME), %ref(TEMP), %ref(DTEMP),
   4 %ref(MATERL// char(0)), %ref(NDI), %ref(NSHR),
   5 %ref(NTENS), %ref(NSTATV), %ref(PROPS), %ref(NPROPS),
   6 %ref(COORDS), %ref(DROT), %ref(PNEWDT), %ref(CELENT),
   7 %ref(DFGRD0), %ref(DFGRD1), %ref(NOEL), %ref(NPT),
   8 %ref(KSLAY), %ref(KSPT), %ref(KSTEP), %ref(KINC))
     RETURN
     END
```

meaning of all these input–output parameters. But in all situations, variables to be defined are at least:

- $DDSDDE(NTENS, NTENS) = \frac{\partial \Delta\sigma}{\partial \Delta\varepsilon}$: Jacobian matrix of the constitutive model, where $\Delta\boldsymbol{\sigma}$ are the stress increments and $\Delta\boldsymbol{\varepsilon}$ are the strain increments. The size of this array depends on the value of *NTENS* (Size of the stress or strain component array). $DDSDDE(I, J)$ defines the change in the Ith stress component at the end of the time increment caused by an infinitesimal perturbation of the Jth component of the strain increment array.
- $STRESS(NTENS)$: This array is passed in as the stress tensor at the beginning of the increment and must be updated in this routine to be the stress tensor at the end of the increment. In finite-strain problems the stress tensor has already been rotated to account for rigid body motion in the increment before *Umat* is called, so that only the co-rotational part of the stress integration should be done in *Umat*. The measure of stress used is "true" (Cauchy) stress.

An accurate Jacobian matrix $\frac{\partial \Delta\sigma}{\partial \Delta\varepsilon}$ is essential to achieve fast quadratic convergence in the global Newton–Raphson iterations.

3. The finite element code *HEREZH++*

The object-oriented framework of *HEREZH++* [15] described here was implemented in *C++* [19]. *HEREZH++* is dedicated to research in the field of mechanics in large transformations. Its objective is to be sufficiently flexible to easily integrate new concepts. The various interesting concepts of object-oriented language made profitable in finite element software are: encapsulate data, template, static and dynamic polymorphism with, in particular, the overload of operator, for classic classes such as vectors, tensors, matrices... but also for advanced classes with virtual class, such as the finite elements, the behaviours, the algorithms of resolution... Standard Template Library (STL) is intensively used concerning containers: lists, adaptable arrays, stacks, associative containers, as well as the basic algorithms associated such as sorting, merging and suppression of redundancy. For more details about *HEREZH++* project one can refer to [15].

The development and the suppression of bugs are more easily carried out with only *HEREZH++*, compared with the use of the industrial code whose first vocation is not correction of bugs. For instance, the developers can use a symbolic debugger, a tool for finding memory leaks, and others "profiling tools" useful to improve the resulting code. In addition, two processes *HEREZH++* can work in an asynchronous way, while dialoguing to simulate the final behaviour of the interface between *ABAQUS* and *HEREZH++*.

All the existing material behaviors in *HEREZH++* can be used with the asynchronous interface. Each material behavior corresponds to a separated class, which inherits a common pure virtual class. This pure virtual class constitutes then the only interface for the rest of *HEREZH++*, for the constitutive law purpose. In this context, according to a classical way of oriented-object development, introduction of a new material behavior consists only to defining a set of functions, which inherit the virtual functions of the parent's.

4. Interface technology between *ABAQUS* and *HEREZH++*

Starting from an equilibrium at a given time t_n of the non-linear procedure and at each Gauss integration point, the input dataflow:

- time increment Δt,
- stress vector $\{\sigma(t_n)\}$,
- total mechanical strain vector $\{\varepsilon(t_n)\}$
- and an initial guess for total mechanical strain increment vector $\{\Delta\varepsilon(t_n)\}$ calculated from current displacement increments,

are switched to user subroutine *Umat*. These data are then transferred between the *Umat* and *HEREZH++* via an interface (see Section 4.2).

HEREZH++ updates the stress vector $\{\sigma(t_{n+1})\}$ and the consistent tangent operator $\frac{\partial\Delta\sigma}{\partial\Delta\varepsilon}$, according to the constitutive laws. Finally, these data are supplied to *ABAQUS* via the interface and the *Umat*. Their flow and the task of the user-defined subroutine are schematically shown in Fig. 1.

4.1. Solution for the interface process communication

In the interface between the two processes, *Umat* and *HEREZH++*, there must be fast exchange of data. Interprocess communication (IPC) provides a mechanism for exchanging data between processes (either on the same or different networked computers) and enables communication between applications even though they may be written in different languages for different target operating systems. There are various forms of IPC [20]. All of these have the objective of moving data from one address space to another.

In Unix system, two mechanisms of primary communication exist:

- via anonymous pipe or via named pipe,
- via a structured and ordered list of memory segments where processes store or retrieve data (named queue IPC).

There is also the communication of interprocess of a more sophisticated interprocess, for instance:

- Unix sockets method,
- CORBA (Common Object Request Broker Architecture) technology,
- communication with parallel computing in using, for example MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) libraries (see for example [21]).

These last three high-level technologies are an on-layer of the communication by pipes.

The queue IPC is also a fast and interesting method, but it comes with some difficulties and risks: processes are deeply linked by common segments and synchronization mechanism is needed, which is opposed to the classical basic rule of encapsulation of data. Nevertheless, we can notice that these risks can be overcome with no difficulty.

In our case, we suppose only communication using computer memory and no transfers are made with memory mapped files. Due to the small complexity and dataflow and to optimize the duration of data transfers we have adopted the first primary IPC solutions. We use "named pipe method" which allows two unrelated processes to communicate with each other.

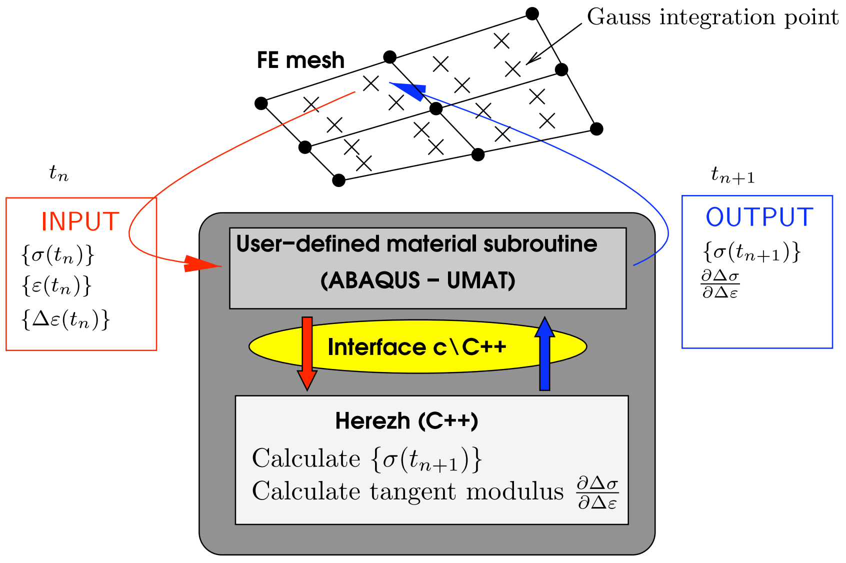

FE mesh
Gauss integration point
t_n
INPUT
$\{\sigma(t_n)\}$
$\{\varepsilon(t_n)\}$
$\{\Delta\varepsilon(t_n)\}$
t_{n+1}
OUTPUT
$\{\sigma(t_{n+1})\}$
$\frac{\partial\Delta\sigma}{\partial\Delta\varepsilon}$
User−defined material subroutine
(ABAQUS − UMAT)
Interface c\C++
Herezh (C++)
Calculate $\{\sigma(t_{n+1})\}$
Calculate tangent modulus $\frac{\partial\Delta\sigma}{\partial\Delta\varepsilon}$

Fig. 1. Interface between the two codes *ABAQUS* and *HEREZH++* via the user subroutine material *Umat*.

Named pipes are also known as FIFOs (first-in, first-out) and can be used to establish a one-way (half-duplex) flow of data. They are identified by their access point, which is basically in a file kept on the file system. Because named pipes have the pathname of a file associated with them, it is possible for unrelated processes to communicate with each other; in other words, two unrelated processes can open the file associated with the named pipe and begin communication. Unlike anonymous pipes, which are process-persistent objects, named pipes are file system-persistent objects, that is, they exist beyond the life of the process. In order to communicate by means of a named pipe, the processes have to open the file associated with the named pipe. By opening the file for reading, the process has access to the reading end of the pipe, and by opening the file for writing, the process has access to the writing end of the pipe. A named pipe supports blocked read and write operations by default: if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa. This operation is used to simply synchronize the two processes without calls to the unix system of synchronization per flag for example.

As well as considerations of simplicity and efficiency, we have adopted the named pipe IPC method for the following reasons:

- Named pipes are very efficient.
- The blocking I/O operation allows simple synchronization mechanism.
- Write (using write function call) to a named pipe is guaranteed to be atomic.
- Named pipes have permissions (read and write) associated with them, unlike anonymous pipes. These permissions can be used to enforce secure communication.

But this method also has limitations:

- Named pipes can only be used for communication among processes on the same host machine.
- Named pipes can be created only in the local file system of the host, that is, you cannot create a named pipe on the NFS file system.
- Due to the basic blocking nature of pipes, careful programming is required for the client and server, in order to avoid deadlocks.
- Named pipe data is a byte stream, and no record identification exists.

4.2. Implementation of the interface

With the named pipe method, several structures have been implemented. First, a *Umat* subroutine is developed: its goal is to transfer information variables to a *c*/*C*++ routine, only by pointer. The interprocess communication by named pipe is then provided by this *c*/*C*++ routine (see Fig. 2). An "input named pipe" is used to transfer information to *Umat* (*ABAQUS*) at *HEREZH*++ and an "output named pipe" for the other way. The information transfer by named pipe is byte stream tabular.

For each transfer, the objective is to group the whole of the data so as to limit the number of calls. The parameters transmitted by Abaqus to the subroutine *c*/*C*++ by addresses can be stored in memory in very different places. Also, to group them, they are copied in a data structure *c*/*C*++ "union" which establishes an equivalence between the arrays of real and integer, and the buffer of bytes (or characters) transmitted by named pipe.

To optimize the flow of data via named pipe (see Fig. 3), data structures are divided in the table into three parts:

- in the front part of the table, there is a zone dedicated to input data (e.g. tangent modulus $= \frac{\partial \Delta \sigma}{\partial \Delta \varepsilon}$),
- in the middle of the table, a second zone is dedicated to input/output data (e.g. stress $\{\boldsymbol{\sigma}(t_n)\}$ and $\{\boldsymbol{\sigma}(t_{n+1})\}$),
- in the other end, a last zone is dedicated to data input (e.g. total mechanical strain $\{\varepsilon(t_n)\}$ and $\{\Delta\varepsilon(t_n)\}$).

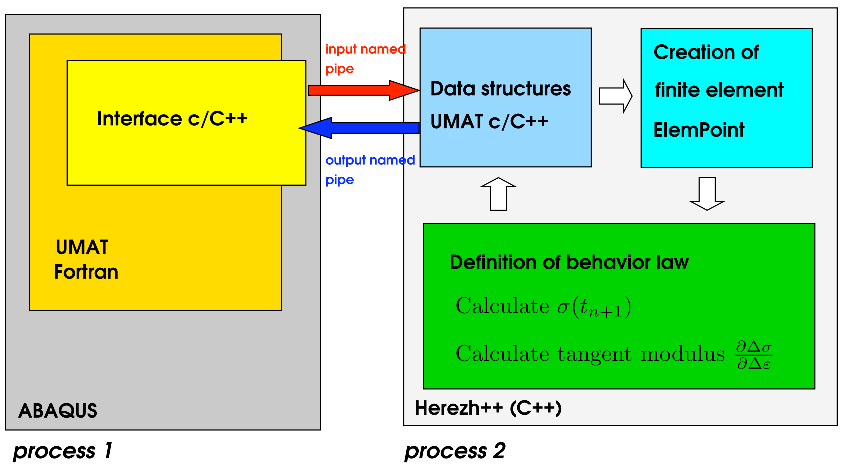

input named pipe
Interface c/C++
Data structures UMAT c/C++
Creation of finite element ElemPoint
output named pipe
UMAT Fortran
Definition of behavior law
Calculate $\sigma(t_{n+1})$
Calculate tangent modulus $\frac{\partial \Delta \sigma}{\partial \Delta \varepsilon}$
ABAQUS
Herezh++ (C++)
process 1
process 2

Fig. 2. Named pipe between *Umat* and *HEREZH*++.

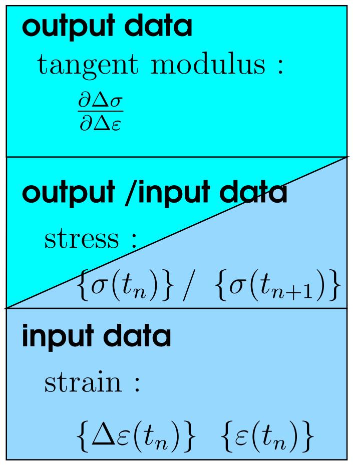

output data
tangent modulus :
$\frac{\partial \Delta\sigma}{\partial \Delta\varepsilon}$
output /input data
stress :
$\{\sigma(t_n)\}$ / $\{\sigma(t_{n+1})\}$
input data
strain :
$\{\Delta\varepsilon(t_n)\}$ $\{\varepsilon(t_n)\}$

Fig. 3. Dataflow distribution in structure *c*/*C*++.

In input data transfer to *HEREZH*++, the buffer really used contains only the last two parts of the table. For the other way, in output data, the buffer is composed only of the first two parts. We use the named pipe supports blocked read and write operations by default: if a process opens the file for reading, it is blocked until another process opens the file for writing, and vice versa.

Appendix shows details about the *C* functions, used to interface the *FORTRAN* subroutine with the named pipes.

The same mechanism of data exchange is employed for the process *HEREZH*++. However, the object-oriented structures strongly typed of *HEREZH*++ lead to some differences. All data structures are encapsulated in a dedicated class, which includes all the methods necessary for the various operations. In particular, the operations when *HEREZH*++ calculates the mechanical behaviour or when *HEREZH*++ uses a *Umat* calculated by another *HEREZH*++ process, during the deserialisation phase (i.e. transformation of the information read in the pipe towards information usable by the code), the various data are transformed into instances (objects) of the classes used by *HEREZH*++.

Lastly, in our case, *HEREZH*++ must store the data relative to the behaviour (see Section 5.3.2) for all the integration points of all the finite elements. This data storage dynamically increases during calculation which is difficult to managed with *ABAQUS* and therefore emphasizes the interest of the use of an external program for that purpose.

However, at the beginning of calculation, *HEREZH*++ knows neither the number of elements nor the number of integration points. The solution adopted to solve this difficulty was to create in *HEREZH*++ a particular class of finite elements designed to store the data of all the integration points of the same element. The geometrical support is the *point*. During the first iteration of the first increment, these elements, *points*, are built dynamically, as well as the containers relating to the integration points associated with them. At the end of this iteration, all *points* are assembled in a mesh structure which is created at this stage. The interest of this method is that once the mesh is created, all the existing mechanisms and operations currently in *HEREZH*++ are available for this mesh and the associated elements, in particular, back-up on hard disk postprocessing.

We can note, in Fig. 2, that the *ABAQUS* process, can be advantageously replaced by a second *HEREZH*++ process during the debug step. Also, let us notice that the *Umat* subroutine integrated in *ABAQUS* is identical for all behaviour laws and information transfer. In particular, the introduction of new external behaviour does not involve any modification on this part.

5. Numerical results and discussion

The quality and the reliability of the implementation of the interface are assessed here through the resolution of several test cases. In order to investigate the effect of information transfer on the simulation run times, all the numerical test have been run on a *Dell Precision Workstation 450* under *Linux version 2.6.11.11, Debian 1:3.3.5-8ubuntu2*. This machine has two *Intel(R) Xeon(TM) CPU 2.40 GHz* processors and 3 GB of memory. All computational examples are carried out with version 6.5 of the commercial code *ABAQUS* [5]. The different components of time, used during these tests, are defined as follows:

- user time (noted u) refers to the CPU time spent executing *ABAQUS*,
- system time (noted s) refers to the amount of OS kernel CPU time spent by the operating system doing work on behalf of the *ABAQUS* process,
- total CPU time (noted *t*) is the sum of these two numbers,
- wall clock time (or elapsed time) (noted *w*) refers to the actual physical time spent for the analysis process to complete.

If the analysis job is running on a single CPU, and the job has exclusive access to that CPU, the difference between total CPU time and wall clock time is largely the time taken to perform all I/O requests. If the job is run across several CPUs, the user time, system time, and total CPU time reported is the sum across all the CPUs. All the presented tests are carried out with two CPUs, but we have observed that the use of a single CPU reduces the computational time in half. We should also note that we do not take the pre-compilation time of the Umat in *ABAQUS* into account.

5.1. Uniaxial tension of cube with simplified mesh with elastic behaviour law

For validation purposes, we conducted direct FE computations on an elastic behaviour law, with material parameters: Young's modulus $E = 1 \times 10^{11}$ MPa and Poisson's ratio $\nu = 0.3$. This law is the simplest and least expensive in computational time. The goal of these tests is to estimate the CPU additional time produced by the code coupling. Three cases are compared:

- intern elastic's law of *ABAQUS* (referred as *Abq*),
- elastic's law with *FORTRAN77*'s *Umat* in *ABAQUS* (referred as *Umat*),
- elastic's law with the interface procedure proposed in this paper (referred as *Abq-Hz++*).

This test is a case of homogeneous deformation of a cube of unit dimension. This is a static non-linear analysis consisting of a cube in uniaxial tension. The cube is meshed with only one 8-node brick element of type *C3D8*. The prescribed displacement is 0.01 mm with isostatic boundary conditions. The computation is performed with 200 fixed of 0.005 time increments. Due to the small displacement, the global problem is mainly linear, but we have enforced a non-linear execution with small steps to obtain an overall CPU time not too small. So, at each step, only one iteration suffices to converge. Thereby, in a total account, the 200 imposed steps, lead to nearly the same global number of iterations: 202.

The results obtained with the model that uses the *Umat* user subroutine and the interface between *HEREZH++* and *ABAQUS* (*Abq-Hz++*) are identical to those obtained using the built-in *ABAQUS* material model. We verify that the convergence is identical between the three cases. The details of analysis time obtained for the three cases are described in Table 2.

For this one element test, we can notice that the computational time is very similar for the three cases.

5.2. Uniaxial tension of cube with more and more refined mesh and elastic behaviour law

This second test is the same as the previous test but in this case the non-linear simulation is made with 100 increments of 0.01. To show the evolution of the CPU time in function of the number of degrees of freedom, we use a more and more refined mesh (elements of type *C3D8* (see Fig. 4)) as follows:

- mesh with $1 \times 1 \times 1$ elements (one-element widthwise, one in height and one in thickness) which includes 24 degrees of freedom (*DOF*),
- mesh with $2 \times 2 \times 2$ elements: 81 *DOF*,
- mesh with $5 \times 5 \times 5$ elements: 648 *DOF*,

Table 2
Computational time (in s) for uniaxial tension with only one element and elastic's law

	u	*s*	*t*	*w*
Abq	4.92	1.91	6.83	7
Umat	5.13	1.81	6.94	7
Abq-Hz++	5.50	2.30	7.8	8

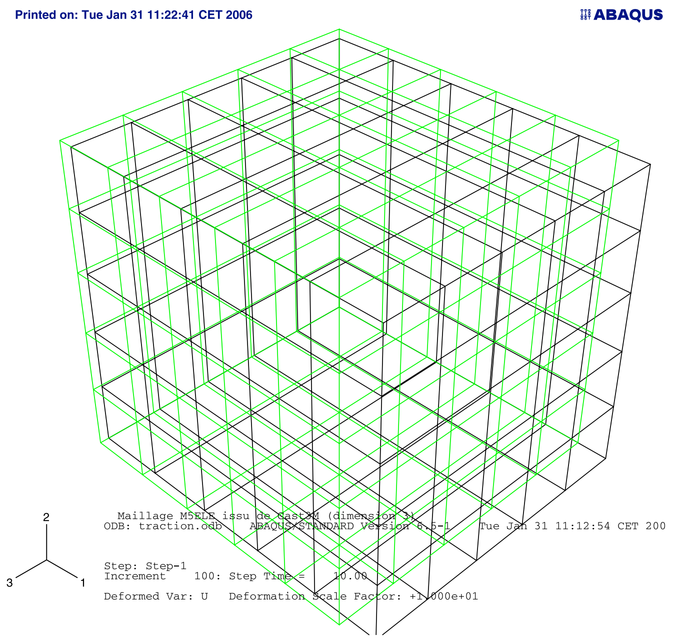

Printed on: Tue Jan 31 11:22:41 CET 2006
ABAQUS
Maillage M5ELE issu de Cast3M (dimension 3)
ODB: traction.odb ABAQUS/STANDARD Version 6.5-1 Tue Jan 31 11:12:54 CET 200
Step: Step-1
Increment 100: Step Time = 10.00
Deformed Var: U Deformation Scale Factor: +1.000e+01
2
3
1

Fig. 4. Initial and deformed mesh of $5 \times 5 \times 5$ elements *C3D8* for cube in tension with elastic law (amplification factor of 10).

- mesh with $20 \times 20 \times 20$ elements: 27,783 *DOF*
- mesh with $30 \times 30 \times 30$ elements: 89,373 *DOF*.

Now, to evaluate the time for information transfer, we use a particular behaviour law in *HEREZH++* with any value calculated but we use the *Umat* in elasticity. In this case, calculations between *Umat* and *Abq-Hz++* are identical, only the time to IPC and transformations by *HEREZH++* are added. All the analysis time for the five meshes are listed in the Table 3.

With the $20 \times 20 \times 20$ elements (see Table 3 and Fig. 5), the total time of *Umat* is 1712 s, the transfer time is then: $4433 - 1712 = 2721$ s. The user and system times, used by *HEREZH++* processor, give 1568 s and 1437 s respectively which gives total time near 3005 s. The calculated total time is then of $1712 + 2721 + 3005 = 7438$ s which seems coherent with the time given by *Abq-Hz++* of 7174 s. The computational time $7174 - 1712 = 5462$ s is near factor $3 \simeq 5462/1712$ between *Umat* and *Abq-Hz++* which is reasonably good. Fig. 6 shows the evolution of wall clock time in function of the number of degrees of freedom.

5.3. Analysis of an automotive boot seal

This industrial test is provided in the documentation of *ABAQUS* [5]. Boot seals are used to protect constant velocity joints and steering mechanisms in automobiles. These flexible components must accommodate the motions associated with the angulation of the steering mechanism. Some parts of the boot seal are in constant contact with an internal metal shaft, while other areas come into contact with

Table 3
Computational time (in s) for uniaxial tension of cube with elastic's law in function of mesh

Mesh	DOF		*u*	*s*	*t*	*w*	% CPU
1 × 1 × 1	24	*Abq*	2.51	0.88	3.39	3	88.7
		Umat	2.61	1.06	3.67	5	63
		Abq-Hz++	2.77	0.98	3.75	6	55
2 × 2 × 2	81	*Abq*	3.23	0.95	4.18	4	89.6
		Umat	3.57	1.13	4.70	5	86.4
		Abq-Hz++	4.09	1.62	5.71	9	65
5 × 5 × 5	648	*Abq*	13.07	1.48	14.55	16	88.2
		Umat	16.07	1.59	17.66	19	92
		Abq-Hz++	16.88	9.95	26.83	39	67
20 × 20 × 20	27,783	*Abq*	1497.1	105.14	1602.2	4392	36.5
		Umat	1599.5	113.37	1712	4801	35.8
		Abq-Hz++	3120.5	1312.5	4433	7174	61
30 × 30 × 30	89,373	*Abq*	9491.9	640.84	10,133	25,715	39.2
		Umat	9770.	699.84	10,470	27,164	38.3
		Abq-Hz++	16,322	5526.0	21,848	35,047	62.3

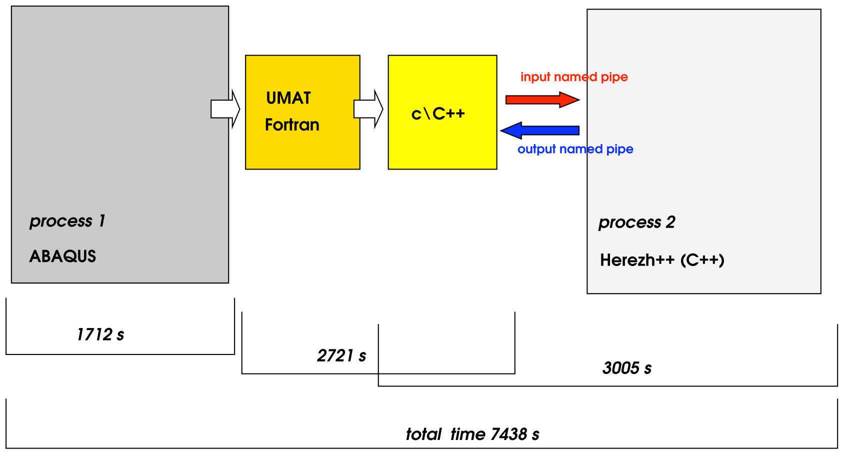


Fig. 5. Evolution of time during the information transfer for 20 × 20 × 20 elements.

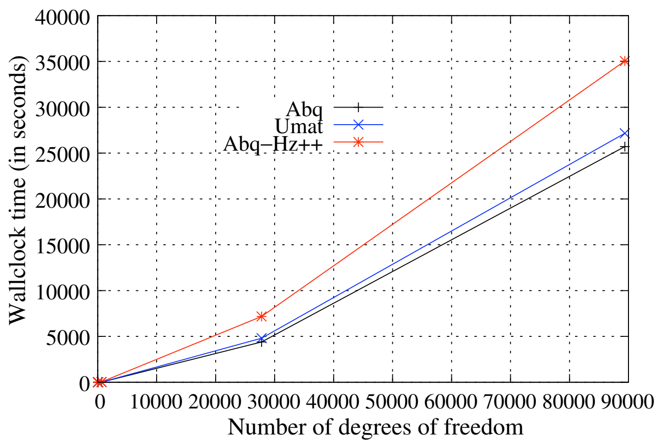


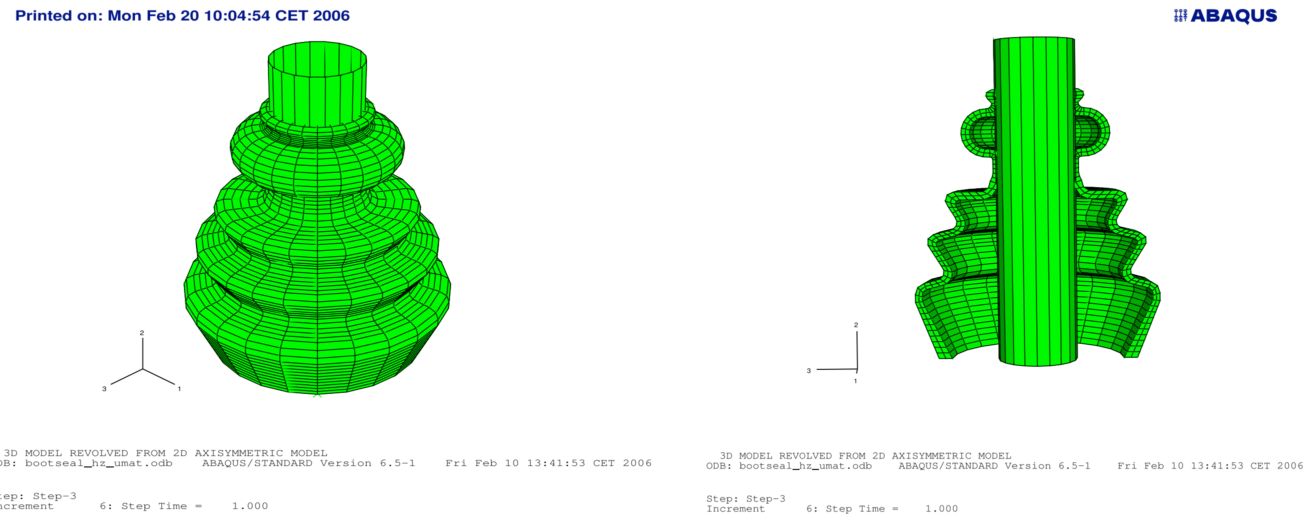
Fig. 6. Evolution of computational time for uniaxial tension test in function of *DOF*.

the metal shaft during angulation. In addition, the boot seal may also come into contact with itself, both internally and externally. The contacting regions affect its performance and longevity.

5.3.1. Geometry, model, and loading

In this example the deformation of the boot seal, caused by a typical angular movement of the shaft, is studied. The boot seal with the internal shaft is shown in Fig. 7. The corrugated shape of the boot seal tightly grips the steering shaft at one end, while the other end is fixed. The rubber seal is modeled with first-order, hybrid brick elements with two elements through the thickness using symmetric model generation. The seal has a non-uniform thickness varying from a minimum of 3.0 mm to a maximum of 4.75 mm at the fixed end. The number of elements is 8713 and 19,230 nodes with 35,010 degrees of freedom.

The internal shaft is considered to be rigid and is modeled as an analytical rigid surface; the radius of the shaft is 14 mm. The rigid body reference node is located precisely in the center of the constant velocity joint. Contact is



```
3D MODEL REVOLVED FROM 2D AXISYMMETRIC MODEL
ODB: bootseal_hz_umat.odb     ABAQUS/STANDARD Version 6.5-1     Fri Feb 10 13:41:53 CET 2006

Step: Step-3
Increment      6: Step Time =     1.000
```

```
3D MODEL REVOLVED FROM 2D AXISYMMETRIC MODEL
ODB: bootseal_hz_umat.odb     ABAQUS/STANDARD Version 6.5-1     Fri Feb 10 13:41:53 CET 2006

Step: Step-3
Increment      6: Step Time =     1.000
```

Fig. 7. Undeformed model of bootseal.

specified between the rigid shaft and the inner surface of the seal. Self-contact is specified on the inner and outer surfaces of the seal.

The inner radius at the neck of the boot seal is smaller than the radius of the shaft so as to provide a tight fit between the seal and the shaft. In the first step the initial interference fit is resolved, corresponding to the assembly process of mounting the boot seal onto the shaft. The second step simulates the angulation of the shaft by specifying a finite rotation of 12° at the rigid body reference node of the shaft. During the third step, the angulated shaft travels of 12° in the opposite direction. These three steps give non-radial loading.

5.3.2. Elasto–visco-hysteresis behaviour law

To show the interest of this code coupling in the introduction of a new constitutive model in *ABAQUS*'s code, the rubber is modeled with an original behaviour law, named elasto–visco-hysteresis law.

In general, the theory of elasto–visco-plasticity takes into consideration the rate-dependent material behaviour with equilibrium hysteresis and incorporates all macroscopically observable phenomena. The present model of elasto–visco-hysteresis uses a different approach. Instead of using a partition of the strain, a superposition of stress contributions is performed. This has already been used successfully in the elasto-hysteresis model [16,17] for modeling the behaviour of shape memory alloys, the ferroelectrical and ferromagnetical materials [22] and the PA66 solid polymer [23].

According to Guélin [18], the elasto–visco-hysteresis model is based on the superposition of hyperelastic $\boldsymbol{\sigma}_{\mathrm{e}}$, viscous stress $\boldsymbol{\sigma}_{\mathrm{v}}$ (with two Maxwell branches) and pure hysteresis $\boldsymbol{\sigma}_{\mathrm{h}}$ contributions. The superposition of stresses states that the Cauchy stress tensor $\boldsymbol{\sigma}$ is expressed from the decomposition given by the relation:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}_{\mathrm{e}} + \boldsymbol{\sigma}_{\mathrm{v}} + \boldsymbol{\sigma}_{\mathrm{h}} \tag{1}$$

The first two contribution behaviours are classical. For the original hysteresis behaviour, we shall refer to Guélin [18] for more information.

The integration of the constitutive equation:

$$\dot{\boldsymbol{S}} = 2\mu\bar{\boldsymbol{D}} + \beta\Phi(\Delta_r^t\boldsymbol{S},\bar{\boldsymbol{D}})\Delta_r^t\boldsymbol{S} \tag{2}$$

describes the implicit evolution of the deviatoric stress tensor $\boldsymbol{S}$ in function of the deviatoric strain-rate tensor $\bar{\boldsymbol{D}}$, the finite variation of deviatoric stress $\Delta_r^t S$ between a reference time r and the current time t, the non-reversible intrinsic dissipated rate Φ function of $\boldsymbol{S}$ and $\bar{\boldsymbol{D}}$ and two material parameters μ and β.

A numerical algorithm is defined for the management of reference points of space stress tensor. The management of these inversion and crossing points is carried out using the intrinsic dissipation rate function Φ previously defined. These points are discrete "memorization" of the loading path, which can be different in each material point of the structure. The hysteresis behaviour defined here is close to the behaviour of hardening plastic.

One of the difficulties introduced by the hysteresis is then the management and the dynamic increasing in the data storage for the stress tensors. The developed solution uses the doubly linked list containers of the "Standard Template Library" associated with C++. The use of abstract class of tensors also allows for a simple transposition of the analytical expressions towards the data-processing coding.

The material parameters used in this study are:

- Mooney–Rivlin's hyperelastic parameters:

  $C_{01} = 0.05\,\mathrm{MPa}, \quad C_{10} = 0.474\,\mathrm{MPa} \quad \text{and} \quad K = 17\,\mathrm{MPa}$

- viscoelastic parameters with two Maxwell contributions (a dashpot and a spring in series):
  - $E = 0.7$ MPa, $\nu = 0.3$ and $\mu = 23.4$ MPa s
  - $E = 0.6 \times 10^{-1}$ MPa, $\nu = 0.3$ and $\mu = 104.6$ MPa s

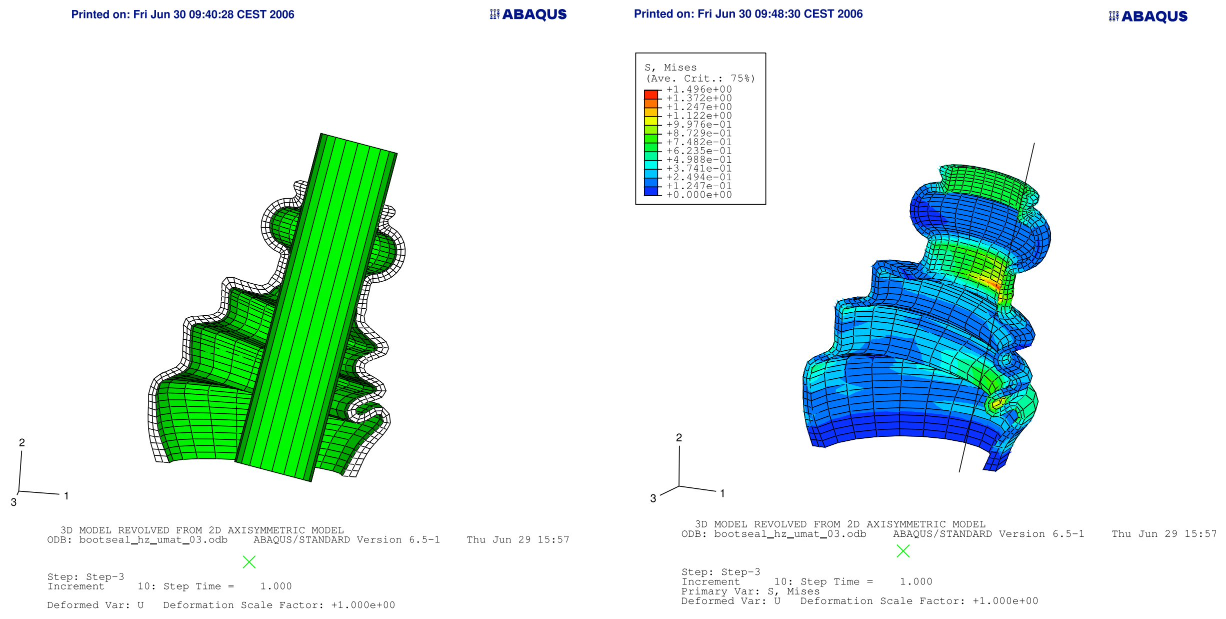

Printed on: Fri Jun 30 09:40:28 CEST 2006
ABAQUS
3D MODEL REVOLVED FROM 2D AXISYMMETRIC MODEL
ODB: bootseal_hz_umat_03.odb ABAQUS/STANDARD Version 6.5-1 Thu Jun 29 15:57
Step: Step-3
Increment 10: Step Time = 1.000
Deformed Var: U Deformation Scale Factor: +1.000e+00
Printed on: Fri Jun 30 09:48:30 CEST 2006
ABAQUS
S, Mises
(Ave. Crit.: 75%)
+1.496e+00
+1.372e+00
+1.247e+00
+1.122e+00
+9.976e-01
+8.729e-01
+7.482e-01
+6.235e-01
+4.988e-01
+3.741e-01
+2.494e-01
+1.247e-01
+0.000e+00
3D MODEL REVOLVED FROM 2D AXISYMMETRIC MODEL
ODB: bootseal_hz_umat_03.odb ABAQUS/STANDARD Version 6.5-1 Thu Jun 29 15:57
Step: Step-3
Increment 10: Step Time = 1.000
Primary Var: S, Mises
Deformed Var: U Deformation Scale Factor: +1.000e+00

Fig. 8. Deformed configuration of half the model and contours of maximum principal stress in the seal.

- hysteretic parameters:

$$np = 0.9, \quad \mu = 0.6 \times 10^{-2}\ \text{MPa} \quad \text{and} \quad Q_0 = 39\ \text{MPa}$$

5.3.3. Results

With interface between *ABAQUS* and *HEREZH++* and elasto–visco-hysteresis behaviour law, we obtained the contours of maximum principal stresses in the bootseal according to Fig. 8. The rotation of the shaft causes the stretching of one side and compression on the other side of the boot seal. The surfaces have come into self-contact on the compressed side. Comparison of the analysis times gives a total time of 15,339 s with 14,116 s for computational time for behaviour law by *HEREZH++*. The transfer time is near 270 s which shows that in this test, it is the elasto–visco-hysteretic behaviour law which takes more time.

6. Conclusion

In this paper, we have proposed a technique to couple an object-oriented "in-house" finite element code *HEREZH++*, with a commercial software *ABAQUS* via the user-defined mechanical material behaviour (*Umat*). The proposed code coupling is composed by named pipes inter-process communication and *c*/*C*++ interface procedure. We have shown in several tests that this code coupling has some advantages for the development of mechanical behaviour law but also some limitations, due to the time to execute the transfer of informations between the two codes. On an industrial test, we have used a new behaviour law named elasto–visco-hysteresis which needs large memory data. This test demonstrates the accuracy, reliability and efficiency of this code coupling which can be use in an industrial framework.

We can note that this methodology is quite general, and could be used, to connect an academic tool to any commercial software, which includes the possibility of implementing a user-defined material model. The academic tool can use any language, provided that a call to the *C* functions can be done. It is the case of the majority of the modern platforms for the numerical developments.

Acknowledgement

The authors are grateful for the support and the financial participation of "Barre Thomas Société des Polymères".

Appendix. Details of the *C* functions used for the communication by named pipes

See Tables A1–A4.

Table A1
C upper functions call for input and output operation with the named pipes

```
/*include of the header's*/
#include ⟨sys/types.h⟩
#include ⟨sys/stat.h⟩
#include ⟨ctype.h⟩
#include ⟨stdio.h⟩
#include ⟨sys/fcntl.h⟩
```

(continued on next page)

Table A1 (*continued*)

```
#include ⟨unistd.h⟩
extern ''C"
/*—————————————————————————————————————————————————*/
void appelc_(
double* STRESS,double* DDSDDE,double* SSE,double* SPD,double* SCD,
double* RPL,double* DDSDDT,double* DRPLDE,double* DRPLDT,double* STRAN,
double* DSTRAN,double* TIME,double* DTIME,double* TEMP,double* DTEMP,
char* CMNAME,int* NDI,int* NSHR,int* NTENS,int* NSTATV,double* PROPS,
int* NPROPS,double* COORDS,double* DROT,double* PNEWDT,double* CELENT,
double* DFGRD0,double* DFGRD1,int* NOEL,int* NPT,int* LAYER,int* KSPT,
int* KSTEP,int* KINC)
{
/*—————————————————————————————————————————————————*/
/* C function for sending data in the pipe */
/*—————————————————————————————————————————————————*/
EcritureDonneesUmat(STRESS,SSE,SPD,SCD,STRAN,DSTRAN,TIME,DTIME,TEMP
,DTEMP,CMNAME,NDI,NSHR,NTENS,NSTATV,COORDS,DROT,PNEWDT,CELENT
,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,KSTEP,KINC);
/*—————————————————————————————————————————————————*/
/* C function for reading data in the pipe */
/*—————————————————————————————————————————————————*/
LectureDonneesUmat(STRESS,DDSDDE,SSE,SPD,SCD,NDI,NSHR,NTENS,PNEWDT
,RPL,DDSDDT,DRPLDE,DRPLDT);
}
```

Table A2
Definition of the global variables

```
/*======definition of global variables========*/
/*definition of a union structure
which links arrays data of char, double and int*/
  union Tab_car_double_int
    {char tampon[928]; double x[116]; int n[232];};
    Tab_car_double_int t_car_x_n;/* buffer container for input–output */
    char* envoi = "../Umat_envoi_Hz";/*named pipe for sending data*/
    char* reception = "../Umat_reception_Hz";/*named pipe for getting data*/
/*—links between the buffer memory and
of the more comprehensible variables*/
      /*—only used in output*/
double*u_herezh_DDSDDT = &t_car_x_n.x[0];
double*u_herezh_DRPLDE = &t_car_x_n.x[6];
double*u_herezh_DDSDDE = &t_car_x_n.x[12];
      /*—used in input and output */
double*u_herezh_RPL = &t_car_x_n.x[48];
double*u_herezh_STRESS = &t_car_x_n.x[49];
double*u_herezh_SSE = &t_car_x_n.x[55];
double*u_herezh_SPD = &t_car_x_n.x[56];
double*u_herezh_SCD = &t_car_x_n.x[57];
double*u_herezh_DRPLDT = &t_car_x_n.x[58];
double*u_herezh_PNEWDT = &t_car_x_n.x[59];
      /*—only used in input */
double* u_herezh_STRAN = &t_car_x_n.x[60];
double* u_herezh_DSTRAN = &t_car_x_n.x[66];
double* u_herezh_TIME = &t_car_x_n.x[72];
double* u_herezh_DTIME = &t_car_x_n.x[74];
double* u_herezh_TEMP = &t_car_x_n.x[75];
double* u_herezh_DTEMP = &t_car_x_n.x[76];
double* u_herezh_COORDS = &t_car_x_n.x[77];
double* u_herezh_DROT = &t_car_x_n.x[80];
double* u_herezh_CELENT = &t_car_x_n.x[89];
double* u_herezh_DFGRD0 = &t_car_x_n.x[90];
double* u_herezh_DFGRD1 = &t_car_x_n.x[99];
int*u_herezh_NDI = &t_car_x_n.n[216];
int*u_herezh_NSHR = &t_car_x_n.n[217];
int*u_herezh_NTENS = &t_car_x_n.n[218];
```

Table A2 (*continued*)

```
int*u_herezh_NSTATV = &t_car_x_n.n[219];
int*u_herezh_NOEL = &t_car_x_n.n[220];
int*u_herezh_NPT = &t_car_x_n.n[221];
int*u_herezh_LAYER = &t_car_x_n.n[222];
int*u_herezh_KSPT = &t_car_x_n.n[223];
int*u_herezh_KSTEP = &t_car_x_n.n[224];
int*u_herezh_KINC = &t_car_x_n.n[225];
char*u_herezh_CMNAME = &t_car_x_n.tampon[904];
```

Table A3
C function for reading data in the pipe

```
/*function for reading data in the pipe*/
void LectureDonneesUmat(double* STRESS,double* DDSDDE,double* SSE
  ,double* SPD,double* SCD,const int* NDI,const int* NSHR
  ,const int* NTENS,double* PNEWDT,double* RPL,double* DDSDDT
  ,double* DRPLDE,double* DRPLDT)
  {int tub; tub = open(envoi,O_RDONLY);/* open the pipe for reading */
    /* read in the pipe and put in the lower part of the buffer */
    read (tub,t_car_x_n.tampon,480);
    close (tub);/* close the connection to the pipe */
    /* transfert between the function parameters and the buffer */
    int ij;
    for (ij = 0; ij < 6; ij++)
      {STRESS[ij] = u_herezh_STRESS[ij]; DDSDDT[ij] = u_herezh_DDSDDT[ij];
      DRPLDE[ij] = u_herezh_DRPLDE[ij];
      int kl;
      for (kl = 0; kl < 6; kl++)
        int r = ij*6 + kl; DDSDDE[r] = u_herezh_DDSDDE[r];
      }
  *SSE = *u_herezh_SSE; *SPD = *u_herezh_SPD; *SCD = *u_herezh_SCD;
  *PNEWDT = *u_herezh_PNEWDT; *RPL = *u_herezh_RPL;
  *DRPLDT = *u_herezh_DRPLDT;
  };
```

Table A4
C function for sending data in the pipe

```
/*function for sending data in the pipe*/
void EcritureDonneesUmat
  (double* STRESS,double* SSE,double* SPD,double* SCD
    ,const double* STRAN,const double* DSTRAN,const double* TIME
    ,const double* DTIME,const double* TEMP,const double* DTEMP
    ,const char* CMNAME,const int* NDI,const int* NSHR,const int* NTENS
    ,const int* NSTATV,const double* COORDS,const double* DROT
    ,double* PNEWDT,const double* CELENT,const double* DFGRD0
    ,const double* DFGRD1,const int* NOEL,const int* NPT
    ,const int* LAYER,const int* KSPT,const int* KSTEP,const int* KINC)
{int tab;
  /* transfert between the buffer and the function parameters */
  int ij;
  for (ij = 0; ij < 6; ij++)
      {u_herezh_STRESS[ij] = STRESS[ij]; u_herezh_STRAN[ij] = STRAN[ij];
      u_herezh_DSTRAN[ij] = DSTRAN[ij];};
    *u_herezh_SSE = *SSE; *u_herezh_SPD = *SPD; *u_herezh_SCD = *SCD;
    *u_herezh_PNEWDT = *PNEWDT;
    *u_herezh_TIME[0] = TIME[0]; u_herezh_TIME[1] = TIME[1];
    *u_herezh_DTIME = *DTIME;
    *u_herezh_DTEMP = *DTEMP;
    *u_herezh_NDI = *NDI;
    *u_herezh_NSHR = *NSHR;
    *u_herezh_NTENS = *NTENS;
    *u_herezh_NSTATV = *NSTATV;
    int i;
```

(*continued on next page*)

Table A4 (*continued*)

```
    for (i = 0; i < 3; i++)
      {u_herezh_COORDS[i] = COORDS[i];
      int j;
      for (j = 0; j < 3; j++)
        {int r = i * 3 + j;
        u_herezh_DROT[r] = DROT[r]; u_herezh_DFGRD0[r] = DFGRD0[r];
        u_herezh_DFGRD1[r] = DFGRD1[r];
        }
      };
    *u_herezh_CELENT = *CELENT;
    *u_herezh_NOEL = *NOEL; *u_herezh_NPT = *NPT;
    *u_herezh_LAYER = *LAYER; *u_herezh_KSPT = *KSPT;
    *u_herezh_KSTEP = *KSTEP; u_herezh_KINC = *KINC;
    for (i = 0; i < 19; i++)
      {if (CMNAME[i] !='') u_herezh_CMNAME[i] = CMNAME[i];
       else {u_herezh_CMNAME[i] = '\0';};
       };
    int tub = open(reception,O_WRONLY);/* open the pipe for sending */
    char* tampon_envoi = &(t_car_x_n.tampon[384]);
    /* put the higher part of the buffer in the pipe */
    write (tub,tampon_envoi,544);
    close (tub);/* close the connection to the pipe. */
    return;
};
```

References

[1] Saleeb A, Trowbridge D, Wilt T, Marks J, Vesely I. Dynamic preprocessing software for the hyperviscoelastic modeling of complex anisotropic biological tissue materials. Adv Eng Software 2006;37(9):609–23.

[2] Doghri I, Ouaar A. Homogenization of two-phase elasto-plastic composite materials and structures: Study of tangent operators, cyclic plasticity and numerical algorithms. Int J Solids Struct 2003;40(7): 1681–712.

[3] Kang G. Finite element implementation of visco-plastic constitutive model with strain-range-dependent cyclic hardening. Commun Numer Methods Eng 2006;22(2):137–53.

[4] Koric S, Thomas BG. Efficient thermo-mechanical model for solidification processes. Int J Numer Methods Eng 2006;66(12): 1955–89.

[5] Abaqus, Hibbit and Karlson and Sorensen Inc., Theory manual – version 6.5 Edition, 2005.

[6] LsDyna, Livermore Software Technology Corporation, Livermore, CA, Keyword user's manual version 940 Edition, 1997.

[7] Verpeaux P, Charras T, Millard A. Castem 2000, Une approche moderne du calcul des structures. In: Pluralis, Fouet JM, Ladevèze P, Oyahon, R. (editors). Conférence calcul des structures et intelligence artificielle, vol. 2. 1988.

[8] Besson J, Foerch R. Large scale object-oriented finite element code design. Comput Methods Appl Mech Eng 1997;142(1–2):165–87.

[9] SiDoLo, P Pilvin. Université de Bretagne-Sud, Lorient, France, user's manual in French – Edition, 2007. <http://web.univ-ubs.fr/lg2m/pilvin/>.

[10] Zimmermann T, Dubois-Pelerin Y, Bomme P. Object-oriented finite element programming: I. Governing principles. Comput Methods Appl Mech Eng 1992;98(2):291–303.

[11] Dubois-Pelerin Y, Zimmermann T. Object-oriented finite element programming: III. An efficient implementation in C++. Comput Methods Appl Mech Eng 1993;108(2):165–83.

[12] Kong XA, Chen D. An object-oriented design of fem programs. Comput Struct 1995;57(1):157–66.

[13] FEMLAB, Natick MA, Comsol AB. term reference manual Edition, 2001.

[14] Gil L, Bugeda G. A C++ object-oriented programming strategy for the implementation of the finite element sensitivity analysis for a nonlinear structural material model. Adv Eng Software 2001;32(12): 927–35.

[15] HEREZH++, Rio G. Université de Bretagne-Sud, Lorient, France, certification iddn-fr-010-0106078-000-r-p-2006-035-20600, user's manual in French – version 6.390 Edition, 2007. <http://web.univ-ubs.fr/lg2m/rio/>.

[16] Manach PY, Favier D, Rio G. Finite element simulations of internal stresses generated during the pseudoelastic deformation of NiTi bodies. J de Phys 1996;C1(6):244–53.

[17] Rio G, Manach PY, Favier D. Finite element simulation of 3D mechanical behaviour of NiTi shape memory alloys. Arch Mech 1995;47(3):537–56.

[18] Guélin P. Remarques sur l'hystérésis mécanique. J de Mécanique Théorique et Appliquée 1980;19(2):217–45.

[19] Stroustrup B. The C++ programming language. Mass: Addison-Wesley; 1986.

[20] Immich PK, Bhagavatula RS, Pendse R. Performance analysis of five interprocess communication mechanisms across unix operating systems. J Syst Software 2003;68(1):27–43.

[21] Pantale O. Parallelization of an object-oriented fem dynamics code: influence of the strategies on the speedup. Adv Eng Software 2005;36(6):361–73.

[22] Tourabi A, Guélin P, Favier D. Towards modelling of deformable ferromagnets and ferroelectrics. Arch Mech 1995;47(3):437–83.

[23] Blès G, Gadaj SP, Nowacki WK, Tourabi A. Experimental study of a *pa*66 solid polymer in the case of shear cyclic loading. Arch Mech 2002;54(2):155–74.